DATA BOOK

TM-1300
Media Processor

Product Specification
Supersedes data of 1999 October 21
File under INTEGRATED CIRCUITS, TR1

2000 May 30

Philips
Semiconductors

PHILIPS

TERMS AND CONDITIONS

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or most work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or most work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS

Philips Semiconductors and Philips Electronics North America Corporation products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from improper use or sale.

Philips Semiconductors and Philips Electronics North America Corporation register eligible circuits under the Semiconductor Chips Protection Act.

DEFINITIONS

Data Sheet Identification	Product Status	Definition
Objective Specification	Formative or in Design	This data sheet contains the design target or goal specifications for product development. Specifications may change in any manner without notice.
Preliminary Specification	Preproduction Product	This data sheet contains preliminary data, and supplementary data will be published at a later date. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
Product Specification	Full Production	This data sheet contains Final Specifications. Philips Semiconductors reserves the right to make changes at any time without notice, in order to improve the design and supply the best possible product.

TriMedia Product Segment, 811 E. Arques Avenue, Sunnyvale, CA 94088

Foreword

The TriMedia™ TM-1300 is a higher speed, functionally enhanced version of the TM-1000 media processor.

TM-1300 contains an ultra-high performance Very Long Instruction Word processor, as well as a complete intelligent video and audio input/output subsystem. The processor has an instruction set that is optimized for processing audio, video and graphics. It includes powerful SIMD multimedia operators for eight- and 16-bit signal datatypes as well as a full complement of 32-bit IEEE compatible floating point operations.

TM-1300 is intended as a multi-standard programmable video, audio and graphics processor. It can either be used standalone, or as an accelerator to a general purpose processor.

The architecture of the TriMedia family came about as the result of many years of effort of many dedicated individuals. Going back in history, the origin of TriMedia was laid by the LIFE-1 VLIW processor, designed by Junien Labrousse and myself in 1987. Work continued afterwards in Philips Research Labs, Palo Alto. My special thanks go to the entire Palo Alto research team: Mike Ang, Uzi Bar-Gadda, Peter Donovan, Martin Freeman, Eino Jacobs, Beomsup Kim, Bob Law, Yen Lee, Vijay Mehra, Pieter van der Meulen, Ross Morley, Mariette Parekh, Bill Sommer, Artur Sorkin and Pierre Uszynski.

The Palo Alto period matured the architecture—we ported all video and audio algorithms that we could find to the compiler/simulator and refined the operation set. In addition, we learned how to give the architecture a market direction. In May 1994, Philips management—in particular Cees-Jan Koomen, Eddy Odijk, Theo Claasen and Doug Dunn—decided to develop TriMedia into a major Philips Semiconductors product line.

Under the guidance of Keith Flagler, the TriMedia team was built. All of them contributed to take this from a set of interesting ideas to a reliable and competitive product in a short period of time. The initial TriMedia team included Fuad Abu Nofal, Karel Allen, Mike Ang, Robert Aquino, Manju Asthana, Patrick de Bakker, Shiv Balakrishnan, Jai Bannur, Marc Berger, Sunil Bhandari, Rusty Biesele, Ahmet Bindal, David Blakely, Hans Bouwmeester, Steve Bowden, Robert Bradfield, Nancy Breede, Shawn Brown, Sujay Chari, Catherine Chen, Howen Chen, Yan-ming Chen, Yong Cho, Scott Clapper, Matthew Clayson, Paul Coelho, Richard Dodds, Marc Duranton, Darcia Eding, Aaron Emigh, Li Chi Feng, Keith Flagler, Jean Gobert, Sergio Golombek, Mike Grimwood, Yudi Halim, Hari Hampapuram, Carl Hartshorn, Judy Heider, Laura Hrenko, Jim Hsu, Eino Jacobs, Marcel Janssens, Patricia Jones, Hann-Hwan Ju, Jayne Keith, Bhushan Kerur, Ayub Khan, Keith Knowles, Mike Kong, Ashok Krishnamurti, Yen Lee, Patrick Leong, Bill Lin, Laura Ling, Chialun Lu, Naeem Maan, Nahid Mansipur, Mike Maynard, Vijay Mehra, Jun Mejia, Derek Meyer, Prabir Mohanty, Saed Muhssin, Chris Nelson, Stephen Ness, Keith Ngo, Francis Nguyen, Kathleen Nguyen, Derek Noonburg, Ciaran O'Donnel, Sang-Ju Park, Charles Peplinski, Gene Pinkston, Maryam Pirayou, Pardha Potana, Bill Price, Victor Ramamoorthy, Babu Rao Kandamilla, Ehsan Rashid, Selliah Rathnam, Margaret Redmond, Donna Richardson, Alan Rodgers, Tilakray Roychoudhury, Hani Salloum, Chris Salzmann, Bob Seltzer, Ravi Selvaraj, Jim Shimandle, Deepak Singh, Bill Sommer, Juul van der Spek, Manoj Srivastava, Renga Sundararajan, Ken-Sue Tan, Ray Ton, Steve Tran, Cynthia Tripp, Ching-Yih Tseng, Allan Tzeng, Barbara Vendelin, John Vivit, Rudy Wang, Rogier Wester, Wayne Wonchoba, Anthony Wong, Sara Wu, David Wyland, Ken Xie, Vincent Xie, Bettina Yeung, Robert Yin, Charles Young, Grace Yun, Elena Zelayeta and Vivian Zhu.

Expert help and feedback was received from many. In particular, I'd like to mention Kees van Zon of Philips Eindhoven for the help with filtering-related issues, and Craig Clapp of PictureTel for excellent feedback on all aspects of the architecture.

My special thanks go to Joe Kostelec. He made me understand that my ambitions could better be realized in California than in Europe. Furthermore, his vision and his wisdom are credited with keeping this project alive and growing until the 'investment decision.'

The vision of a universal media accelerator is credited to Jaap de Hoog. Jaap, I wish you were here to see it come to fruition.

–Gerrit Slavenburg

Table of Contents

3 DSPCPU Architecture

6 Video In

7 Enhanced Video Out

8 Audio In

9 Audio Out

12 SDRAM Memory System

13 System Boot

14 Image Coprocessor

15 Variable Length Decoder

16 I2C Interface

17 Synchronous Serial Interface

Pin List Chapter 1

by Muhammad Hafeez, Naeem Maan, Thorwald Rabeler, Luis Lucas, Gert Slavenburg

1.1 TM1300 VERSUS TM1100

The following summarizes pinout differences between TM1100 and TM1300:

- TM1300 uses a BGA 27x27 package and is hence not physically pin compatible with TM1100.
- TM1300 no longer has the MM_MATCHOUT and MM_MATCHIN pins, SDRAM read timing is now internally derived.
- TM1300 recommends different VDDQ/VSSQ board circuitry. Refer to VDDQ, VSSQ description.
- We recommend 50-ohm PCB traces for all SDRAM memory signal routing, with minimal wire lengths for 143-MHz SDRAM operation.
- We recommend 27-33 ohm series terminating resistors close to the TM1300 for all STRG3 and STRG5 I/O circuit pins used as outputs.
- TM1300 has one new memory address pin (MM_A13) to support 16-bit wide 64-Mbit SDRAM.
- TM1300 has 4 distinct serial stereo audio outputs (AO_SD1..4) instead of the single octal channel audio output (AO_SD) of TM1100.
- TM1300 introduces the SPDIF audio output pin, SPDO.
- TM1300 uses new I/O pad types, with different impedance/drive capabilities to ease board design.

1.2 BOUNDARY SCAN NOTICE

TM1300 implements full IEEE 1149.1 boundary scan. Any TM1300 pin designated “IN” only (from a functionality point of view) can become an output during boundary scan.

1.3 I/O CIRCUIT SUMMARY

TM1300 has a total of 169 functional pins, excluding VDDQ, VSSQ, VREF_PCI and VREF_PERIPH and digital power/ground. TM1300 uses the types of I/O circuits shown in the table below.

Pad Type	Pad Type Description
PCI	PCI2.1 compliant I/O, capable of using 3.3-V or 5-V PCI signaling conventions.
PCIOD	PCI2.1 compliant Open Drain I/O, capable of using 3.3-V or 5-V PCI signaling conventions.
IICOD	Open drain 3.3-V or 5-V I^2C I/O (for I^2C pins).
STRG3	3.3-V only low impedance I/O. Requires board level 27-33 ohm series terminator resistor to match 50 ohm PCB trace.
NORM3	3.3-V only I/O circuit with regular drive strength and board trace matched drive impedance.
STRG5	3.3-V low impedance output, combined with 5-V tolerant input. If used as output, it requires a board level 27-33 ohm series terminator resistor to match 50-ohm PCB trace.
WEAK5	3.3-V regular impedance output, with slow rise/fall, combined with 5-V tolerant input.

For the pins with 5-V input capability, the special pins VREF_PCI or VREF_PERIPH determine 3.3- or 5-V input tolerance, as per the table in Section 1.6. The above pad types are used in the modes listed in the following table.

Modes	Description
IN	Input only, except during boundary scan
OUT	Output only, except during boundary scan
OD	Open drain output - active pull low, no active drive high, requires external pull-up
I/O	Output or input
I/OD	Open drain output with input - active pull low, no active drive high, requires external pull-up

1.4 SIGNAL PIN LIST

In the table below, a pin name ending in a '#' designates an active-low signal (the active state of the signal is a low voltage level). All other signals have active-high polarity.

Pin Name	BGA Ball	Pad Type	Mode	Description
Main Clock Interface				
TRI_CLKIN	L20	NORM3	IN	Main input clock. The SDRAM clock outputs (MM_CLK0 and MM_CLK1) can be set to 2x or 3x this frequency. The on-chip DSPCPU clock (DSPCPU_CLK) can be set to 1x, 5/4, 4/3, 3/2 or 2x the SDRAM clock frequency. Maximum recommended ppm level is +/- 100 ppm or lower to improve jitter on generated clocks. Duty cycle should not exceed 30/70% asymmetry.
VDDQ	K20	N/A	PWR	Quiet VDD for the PLL subsystem. This pin should be supplied from VDD through a low-Q series inductor. It should be bypassed for AC to VSSQ, using a dual capacitor bypass (hi and low frequency AC bypass).
VSSQ	L19	N/A	GND	Quiet VSS for the PLL subsystem. **Should be AC bypassed to VDDQ, but should otherwise be left DC floating. It is connected on-chip to VSS. No external coil or other connection to board ground is needed, such connection would create a ground loop**.
Miscellaneous System Interface				
TRI_RESET#	G19	WEAK5	IN	TM1300 RESET input. This pin can be tied to the PCI RST# signal in PCI bus systems. Upon receiving RESET, TM1300 initiates its boot protocol.
BOOT_CLK	T20	NORM3	IN	Used for testing purposes. Must be connected to TRI_CLKIN for normal operation.
TESTMODE	P19	NORM3	IN	Used for testing purposes. Must be connected to VSS for normal operation.
SCANCPU	D20	NORM3	IN	Used for testing purposes. Must be connected to VSS for normal operation.
RESERVED1	E19	NORM3	I/O	Reserved pin. Has to be left unconnected for normal operation.
RESERVED2	D19	STRG5	I/O	Reserved pin. Has to be left unconnected for normal operation.
VREF_PCI	F2	N/A	PWR	VREF_PCI determines the mode of operation of the PCI pins listed in Section 1.6. VREF_PCI must be connected to 5V for use in a 5-V PCI signaling environment or to VSS (0 V) for use in 3.3-V PCI signaling environment. The supply to this pin should be AC bypassed and provide 40 mA of DC sink or source capability. **Note that this pin can not be directly connected to the PCI 'I/O designated power pins' in a dual voltage PCI plug-in card. Board level conversion circuitry is required.**
VREF_PERIPH	C18	N/A	PWR	VREF_PERIPH determines the mode of operation of the I/O pins listed in Section 1.6. VREF_PERIPH should be connected to 5V if any of the listed I/O pins provided should be 5-V input voltage capable. VREF_PERIPH should be connected to VSS (0-V) if all listed I/O pins are 3.3-V only inputs. The supply to this pin should be AC bypassed and provide 40 mA of DC sink or source capability.
TRI_USERIRQ	G20	WEAK5	IN	General purpose level/edge interrupt input. Vectored interrupt source number 4.
TRI_TIMER_CLK	H19	WEAK5	IN	External general purpose clock source for timers. Max. 40 MHz.
Main Memory Interface				
MM_CLK0 MM_CLK1	Y10 W10	STRG3	OUT	SDRAM output clock at 2x or 3x TRI_CLKIN frequency. Two identical outputs are provided to reliably drive several small memory configurations without external glue. A series terminating resistor close to TM1000 is recommended to reduce ringing. For driving a 50-ohm trace, a resistor of 27 to 33 ohm is recommended. We recommend against using higher impedance traces in the SDRAM signals.
MM_A00 MM_A01 MM_A02 MM_A03 MM_A04 MM_A05 MM_A06 MM_A07 MM_A08 MM_A09 MM_A10 MM_A11 MM_A12 MM_A13	W12 Y12 W11 Y11 Y9 W9 V9 Y8 W8 Y7 V12 Y13 W13 Y14	NORM3	OUT	Main memory address bus; used for row and column addresses (was 'RESERVED2' in TM1000 - also sometimes name MM_BA1) (new in TM1300 - also named MM_64M_11 in some documents)

Pin Name	BGA Ball	Pad Type	Mode	Description
MM_DQ00 MM_DQ01 MM_DQ02 MM_DQ03 MM_DQ04 MM_DQ05 MM_DQ06 MM_DQ07 MM_DQ08 MM_DQ09 MM_DQ10 MM_DQ11 MM_DQ12 MM_DQ13 MM_DQ14 MM_DQ15 MM_DQ16 MM_DQ17 MM_DQ18 MM_DQ19 MM_DQ20 MM_DQ21 MM_DQ22 MM_DQ23 MM_DQ24 MM_DQ25 MM_DQ26 MM_DQ27 MM_DQ28 MM_DQ29 MM_DQ30 MM_DQ31	Y20 V18 W19 W20 U18 V19 V20 T18 W18 V17 Y18 W17 Y17 W16 Y16 V15 W7 Y6 W6 V6 Y5 W5 Y4 W4 V2 V3 W1 W2 Y1 Y2 W3 Y3	NORM3	I/O	32-bit data I/O bus
MM_CKE0 MM_CKE1	Y19 U1	NORM3	OUT	Clock enable output to SDRAMs. Two identical outputs are provided in order to reliably drive several small memory configurations without external glue.
MM_CS0# MM_CS1# MM_CS2# MM_CS3#	U2 U20 U3 U19	NORM3	OUT	Chip select for DRAM rank n; active low
MM_RAS#	W14	NORM3	OUT	Row address strobe; active low
MM_CAS#	Y15	NORM3	OUT	Column address strobe; active low
MM_WE#	W15	NORM3	OUT	Write enable; active low
MM_DQM0 MM_DQM1 MM_DQM2 MM_DQM3	T19 R18 V1 V4	NORM3	OUT	MM_DQ Mask Enable; these are byte enable signals for the 32-bit MM_DQ bus
PCI Interface (Note: current buffer design allows drive/receive from either 3.3 or 5V PCI bus)				
PCI_CLK	T2	PCI	IN	All PCI input signals are sampled with respect to the rising edge of this clock. All PCI outputs are generated based on this clock. Clock is required for normal operation of the PCI block.

Pin Name	BGA Ball	Pad Type	Mode	Description
PCI_AD00 PCI_AD01 PCI_AD02 PCI_AD03 PCI_AD04 PCI_AD05 PCI_AD06 PCI_AD07 PCI_AD08 PCI_AD09 PCI_AD10 PCI_AD11 PCI_AD12 PCI_AD13 PCI_AD14 PCI_AD15 PCI_AD16 PCI_AD17 PCI_AD18 PCI_AD19 PCI_AD20 PCI_AD21 PCI_AD22 PCI_AD23 PCI_AD24 PCI_AD25 PCI_AD26 PCI_AD27 PCI_AD28 PCI_AD29 PCI_AD30 PCI_AD31	T1 R3 R2 R1 P2 P1 N2 N1 M2 M1 L2 L1 K1 K2 J1 J2 D1 D3 C1 B2 B1 C2 C3 A1 A3 C4 B4 A4 A5 C6 B6 A6	PCI	I/O	Multiplexed address and data.
PCI_C/BE#0 PCI_C/BE#1 PCI_C/BE#2 PCI_C/BE#3	M3 J3 D2 B3	PCI	I/O	Multiplexed bus commands and byte enables. High for command, low for byte enable.
PCI_PAR	H1	PCI	I/O	Even parity across AD and C/BE lines.
PCI_FRAME#	E2	PCI	I/O	Sustained tri-state. Frame is driven by a master to indicate the beginning and duration of an access.
PCI_IRDY#	E1	PCI	I/O	Sustained tri-state. Initiator Ready indicates that the bus master is ready to complete the current data phase.
PCI_TRDY#	F3	PCI	I/O	Sustained tri-state. Target Ready indicates that the bus target is ready to complete the current data phase.
PCI_STOP#	G2	PCI	I/O	Sustained tri-state. Indicates that the target is requesting that the master stop the current transaction.
PCI_IDSEL	A2	PCI	IN	Used as chip select during configuration read/write cycles.
PCI_DEVSEL#	F1	PCI	I/O	Sustained tri-state. Indicates whether any device on the bus has been selected.
PCI_REQ#	B7	PCI	I/O	Driven by TM1300 as PCI bus master to request use of the PCI bus.
PCI_GNT#	B5	PCI	IN	Indicates to TM1300 that access to the bus has been granted.
PCI_PERR#	G1	PCI	I/O	Sustained tri-state. Parity error generated/received by TM1300.
PCI_SERR#	H2	PCI	OD	System error. This signal is asserted when operating as target and detecting an address parity error.

Pin Name	BGA Ball	Pad Type	Mode	Description
PCI_INTA# PCI_INTB# PCI_INTC# PCI_INTD#	C9 A8 B8 A7	PCIOD PCI PCIOD PCIOD	I/OD I/O/OD I/OD I/OD	• Can operate as input (power up default) or output, as determined by direction control bits in PCI MMIO register INT_CTL. • As input, a PCI_INT# pin can be used to receive PCI interrupt requests (normal PCI use is active low, level sensitive mode, but the VIC can be set to treat these as positive edge triggered mode). As input, a PCI_INT# pin can also be used as a general interrupt request pin if not needed for PCI. • As output, the value of a PCI_INT# can be programmed through PCI MMIO registers to generate interrupts for other PCI masters. • Whenever XIO bus functionality is active, PCI_INTB# is a push-pull CMOS I/O pin. When the XIO bus is not active and regular PCI bus functionality is activated, then PCI_INTB# has a PCI compatible open drain output.
JTAG Interface (debug access port and 1149.1 boundary scan port)				
JTAG_TDI	F20	WEAK5	IN	JTAG test data input
JTAG_TDO	F18	WEAK5	I/O	JTAG test data output. This pin can either drive active low, high or float.
JTAG_TCK	F19	WEAK5	IN	JTAG test clock input
JTAG_TMS	E20	WEAK5	IN	JTAG test mode select input
Video In				
VI_CLK	C20	STRG5	I/O	• If configured as input (power up default): a positive transition on this incoming video clock pin samples all other VI_DATA input signals below if VI_DVALID is HIGH. If VI_DVALID is LOW, VI_DATA is ignored. Clock and data rates of up to 81 MHz are supported. • If configured as output: programmable output clock to drive an external video A/D converter. Can be programmed to emit integral dividers of DSPCPU_CLK. If used as output, a board level 27-33 ohm series resistor is recommended to reduce ringing.
VI_DVALID	A17	WEAK5	IN	VI_DVALID indicates that valid data is present on the VI_DATA lines. If HIGH, VI_DATA will be accepted on the next VI_CLK positive edge. If LOW, no VI_DATA will be sampled.
VI_DATA0 VI_DATA1 VI_DATA2 VI_DATA3 VI_DATA4 VI_DATA5 VI_DATA6 VI_DATA7	D18 C19 B20 B19 A20 A19 C17 B18	WEAK5	IN	CCIR656 style YUV 4:2:2 data from a digital camera, or general purpose high speed data input pins. Sampled on VI_CLK if VI_DVALID HIGH.
VI_DATA8 VI_DATA9	A18 B17	WEAK5	IN	Extension high speed data input bits to allow use of 10 bit video A/D converters in raw10 modes. VI_DATA[8] serves as START and VI_DATA[9] as END message input in message passing mode.Sampled on positive transitions of VI_CLK if VI_DVALID HIGH.
I^2C Interface				
IIC_SDA	R19	IICOD	I/OD	I^2C serial data
IIC_SCL	R20	IICOD	I/OD	I^2C clock
Video Out				
VO_DATA0 VO_DATA1 VO_DATA2 VO_DATA3 VO_DATA4 VO_DATA5 VO_DATA6 VO_DATA7	P20 N19 N20 M18 M19 M20 K19 J20	WEAK5	OUT	CCIR656 style YUV 4:2:2 digital output data, or general purpose high speed data output channel. Output changes on positive edge of VO_CLK.
VO_IO1	J18	WEAK5	I/O	This pin can function as HS output or as STMSG (Start Message) output. • If set as HS output, it outputs the horizontal sync signal • In message passing mode, this pin acts as STMSG output.

Pin Name	BGA Ball	Pad Type	Mode	Description
VO_IO2	H20	WEAK5	I/O	This pin can function as FS (frame sync) input, FS output or as ENDMSG output. • If set as FS input, it can be set to respond to positive or negative edge transitions. • If the Video Out (VO) unit operates in external sync mode and the selected transition occurs, the VO unit sends two fields of video data. Note: this works only once after a reset. • In message passing mode, this pin acts as ENDMSG output.
VO_CLK	J19	STRG5	I/O	The VO unit emits VO_DATA on a positive edge of VO_CLK. VO_CLK can be configured as input (reset default) or output. • If configured as input: VO_CLK is received from external display clock master circuitry. • If configured as output, TM1300 emits a programmable clock frequency. The emitted frequency can be set between approx. 4 and 81 MHz with a sub-Hertz resolution. The clock generated is frequency accurate and has low jitter properties due to a combination of an on-chip DDS (Direct Digital Synthesizer) and VCO/PLL. If used as output, a board level 27-33 ohm series resistor is recommended to reduce ringing.
Audio In (always acts as receiver, but can be master or slave for A/D timing)				
AI_OSCLK	B15	STRG3	OUT	Over-sampling clock. This output can be programmed to emit any frequency up to 40 MHz with a sub-Hertz resolution. It is intended for use as the $256f_s$ or $384f_s$ over sampling clock by external A/D subsystem. A board level 27-33 ohm series resistor is recommended to reduce ringing.
AI_SCK	A16	STRG5	I/O	• When the Audio In (AI) unit is programmed as a serial-interface timing slave (power-up default), AI_SCK is an input. AI_SCK receives the serial bit clock from the external A/D subsystem. This clock is treated as fully asynchronous to the TM1300 main clock. • When the AI unit is programmed as the serial-interface timing master, AI_SCK is an output. AI_SCK drives the serial clock for the external A/D subsystem. The frequency is a programmable integral divisors of the AI_OSCLK frequency. AI_SCK is limited to 22 MHz. The sample rate of valid samples embedded within the serial stream is variable. If used as output, a board level 27-33 ohm series resistor is recommended to reduce ringing.
AI_SD	C15	WEAK5	IN	Serial data from external A/D subsystem. Data on this pin is sampled on positive or negative edges of AI_SCK as determined by the CLOCK_EDGE bit in the AI_SERIAL register.
AI_WS	B16	WEAK5	I/O	• When the AI unit is programmed as the serial-interface timing slave (power-up default), AI_WS acts as an input. AI_WS is sampled on the same edge as selected for AI_SD. • When Audio In is programmed as the serial-interface timing master, AI_WS acts as an output. It is asserted on the opposite edge of the AI_SD sampling edge. AI_WS is the word-select or frame-synchronization signal from/to the external A/D subsystem.

Pin Name	BGA Ball	Pad Type	Mode	Description
Audio Out (always acts as sender, but can be master or slave for D/A timing)				
AO_OSCLK	B14	STRG3	OUT	Over sampling clock. This output can be programmed to emit any frequency up to 40 MHz, with a sub-Hertz resolution. It is intended for use as the 256 or 384f_s over sampling clock by the external D/A conversion subsystem. A board level 27-33 ohm series resistor is recommended to reduce ringing.
AO_SCK	A14	STRG5	I/O	• When the Audio Out (AO) unit is programmed to act as the serial interface timing slave (power up default), AO_SCK acts as input. It receives the Serial Clock from the external audio D/A subsystem. The clock is treated as fully asynchronous to the TM1300 main clock. • When the AO unit is programmed to act as serial interface timing master, AO_SCK acts as output. It drives the serial clock for the external audio D/A subsystem. The clock frequency is a programmable integral divisor of the AO_OSCLK frequency. AO_SCK is limited to 22 MHz. The sample rate of valid samples embedded within the serial stream is variable. If used as output, a board level 27-33 ohm series resistor is recommended to reduce ringing.
AO_SD1	B13	WEAK5	OUT	Serial data to external stereo audio D/A subsystem for first 2 of 8 channels. The timing of transitions on this output is determined by the CLOCK_EDGE bit in the AO_SERIAL register, and can be on positive or negative AO_SCK edges.
AO_SD2	A13	WEAK5	OUT	Serial data.
AO_SD3	C12	WEAK5	OUT	Serial data.
AO_SD4	B12	WEAK5	OUT	Serial data.
AO_WS	A15	WEAK5	I/O	• When the AO unit is programmed as the serial-interface timing slave (power-up default), AO_WS acts as an input. AO_WS is sampled on the opposite AO_SCK edge at which AO_SDx are asserted. • When the AO unit is programmed as serial-interface timing master, AO_WS acts as an output. AO_WS is asserted on the same AO_SCK edge as AO_SDx. AO_WS is the word-select or frame-synchronization signal from/to the external D/A subsystem. Each audio channel receives 1 sample for every WS period.
S/PDIF Output (Output)				
SPDO	A12	STRG3	OUT	Self clocking serial data stream as per IEC958, with 1937 extensions. Note that the low impedance output buffer requires a 27 to 33 ohm series terminator close to TM1300 in order to match the board trace impedance. This series terminator can be/must be part of the voltage divider needed to create the coaxial output through the AC isolation transformer.
Synchronous Serial Interface (SSI) to an off-chip modem front-end				
SSI_CLK	B11	WEAK5	IN	Clock signal of the synchronous serial interface to an off-chip modem analog frontend or ISDN terminal adapter; provided by the receive channel of an external communication device.
SSI_RXFSX	A11	WEAK5	IN	Receive frame sync reference of the synchronous serial interface, provided by the receive channel of an external communication device.
SSI_RXDATA	A10	WEAK5	IN	Receive serial data input; provided by the receive channel of an external communication device.
SSI_TXDATA	B10	WEAK5	OUT	Transmit serial data output; sent to the transmit channel of the external communication device.
SSI_IO1	A9	WEAK5	I/O	General purpose programmable I/O. Set to input on power up.
SSI_IO2	B9	WEAK5	I/O	General purpose programmable I/O. Set to input on power up. Can also be programmed to function as the transmit channel frame synchronization reference output.

1.5 POWER PIN LIST

VSS (ground)				VCC (3.3V I/O supply)				VDD (2.5V core supply)		
C5	H8	L9		C7	G17	R4		C8	H17	N17
C16	H9	L10		C10	G18	R17		C13	H18	N18
D4	H10	L11		C11	K3	U6		D8	J4	U8
D5	H11	L12		C14	K4	U7		D9	J17	U9
D16	H12	L13		D6	K17	U10		D12	M4	U12
D17	H13	M8		D7	K18	U11		D13	M17	U13
E3	J8	M9		D10	L3	U14		H3	N3	V8
E4	J9	M10		D11	L4	U15		H4	N4	V13
E17	J10	M11		D14	L17	V7				
E18	J11	M12		D15	L18	V10				
T3	J12	M13		F4	P3	V11				
T4	J13	N8		F17	P4	V14				
T17	K8	N9		G3	P17					
U4	K9	N10		G4	P18					
U5	K10	N11								
U16	K11	N12								
U17	K12	N13								
V5	K13									
V16	L8									

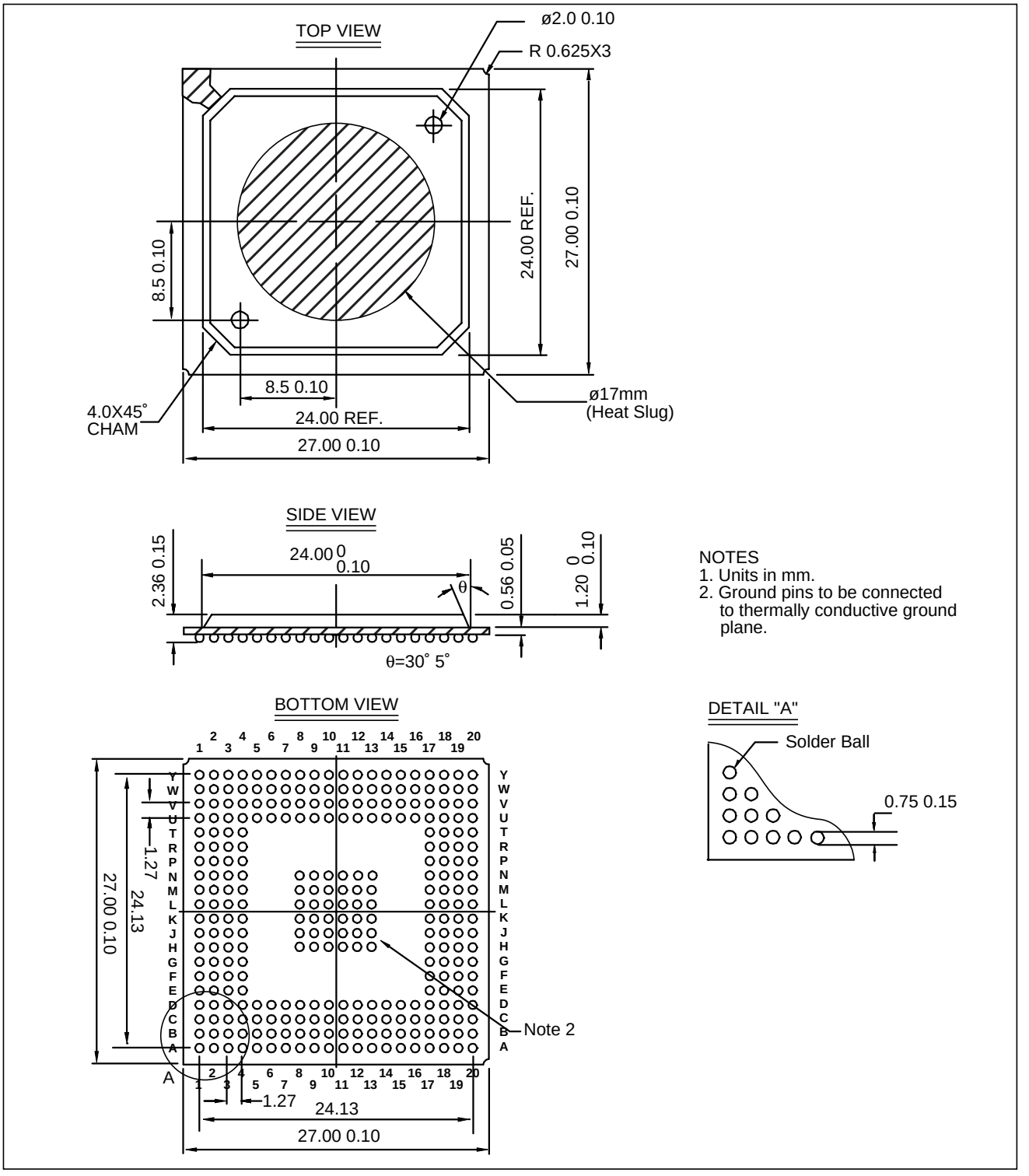
1.6 PIN REFERENCE VOLTAGE

With the exception of Open Drain mode outputs, outputs always drive to a level determined by the 3.3-V I/O voltage. VREF_PERIPH and VREF_PCI purely determine input voltage clamping, not input signal thresholds or output levels.

			Inputs always in 3.3-V mode			**Output only pins**	
			TRI_CLKIN BOOT_CLK TESTMODE SCANCPU RESERVED1			VO_DATA0 VO_DATA1 VO_DATA2 VO_DATA3 VO_DATA4 VO_DATA5 VO_DATA6 VO_DATA7	AI_OSCLK AO_OSCLK AO_SD1 AO_SD2 AO_SD3 AO_SD4 SSI_TXDATA SPDO
VREF_PCI determined mode			**VREF_PERIPH determined mode**			**SDRAM i/f (always 3.3-Volt mode)**	
PCI_AD00 PCI_AD01 PCI_AD02 PCI_AD03 PCI_AD04 PCI_AD05 PCI_AD06 PCI_AD07 PCI_AD08 PCI_AD09 PCI_AD10 PCI_AD11 PCI_AD12 PCI_AD13 PCI_AD14 PCI_AD15 PCI_AD16 PCI_AD17 PCI_AD18 PCI_AD19 PCI_AD20 PCI_AD21 PCI_AD22 PCI_AD23 PCI_AD24 PCI_AD25 PCI_AD26	PCI_AD27 PCI_AD28 PCI_AD29 PCI_AD30 PCI_AD31 PCI_CLK PCI_C/BE#0 PCI_C/BE#1 PCI_C/BE#2 PCI_C/BE#3 PCI_PAR PCI_FRAME# PCI_IRDY# PCI_TRDY# PCI_STOP# PCI_IDSEL PCI_DEVSEL# PCI_REQ# PCI_GNT# PCI_PERR# PCI_SERR# PCI_INTA# PCI_INTB# PCI_INTC# PCI_INTD# TRI_RESET#		TRI_USERIRQ TRI_TIMER_CLK JTAG_TDI JTAG_TDO JTAG_TCK JTAG_TMS VI_CLK VI_DVALID VI_DATA0 VI_DATA1 VI_DATA2 VI_DATA3 VI_DATA4 VI_DATA5 VI_DATA6 VI_DATA7 VI_DATA8 VI_DATA9 IIC_SDA IIC_SCL VO_IO1 VO_IO2 VO_CLK	AI_SCK AI_SD AI_WS AO_SCK AO_WS SSI_CLK SSI_RXFSX SSI_RXDATA SSI_IO1 SSI_IO2 RESERVED2		MM_CLK0 MM_CLK1 MM_A00 MM_A01 MM_A02 MM_A03 MM_A04 MM_A05 MM_A06 MM_A07 MM_A08 MM_A09 MM_A10 MM_A11 MM_A12 MM_A13 MM_DQ00 MM_DQ01 MM_DQ02 MM_DQ03 MM_DQ04 MM_DQ05 MM_DQ06 MM_DQ07 MM_DQ08 MM_DQ09 MM_DQ10 MM_DQ11 MM_DQ12 MM_DQM1 MM_DQM2 MM_DQM3	MM_DQ13 MM_DQ14 MM_DQ15 MM_DQ16 MM_DQ17 MM_DQ18 MM_DQ19 MM_DQ20 MM_DQ21 MM_DQ22 MM_DQ23 MM_DQ24 MM_DQ25 MM_DQ26 MM_DQ27 MM_DQ28 MM_DQ29 MM_DQ30 MM_DQ31 MM_CKE0 MM_CKE1 MM_CS0# MM_CS1# MM_CS2# MM_CS3# MM_RAS# MM_CAS# MM_WE# MM_DQM0

1.7 PACKAGE

BGA292 ball grid array package; 256 balls + 36 center ground & thermal balls; body 27 x 27 x 1.55 mm.



1.8 ORDERING INFORMATION

To order 143-MHz v1.2 TM-1300 parts, refer to part number ‘PTM1300AEBEA’, 12 nc product code 9352 6691 7557.

To order 166-MHz v1.2 TM-1300 parts, refer to part number ‘PTM1300FBEA’, 12 nc product code 9352 6687 1557.

SOT number is 553AA1.

1.9 PARAMETRIC CHARACTERISTICS

1.9.1 Operating Range and Thermal Characteristics

Functional operation, long-term reliability and AC/DC characteristics are guaranteed for the operating conditions below.

Symbol	Parameter	Minimum	Typical	Maximum	Units
V_{DD}	Core supply voltage	2.375	2.50	2.625	V
V_{CC}	I/O supply voltage	3.135	3.30	3.465	V
T_{case}	Operating case temperature range	0		85	˚C
Ψ_{jt}	junction to case thermal resistance		3.8		˚C/W
ϑ_{ja}	junction to ambient thermal resistance (natural convection)		15		˚C/W

1.9.2 Absolute Maximum Ratings

Permanent damage may occur if these conditions are exceeded

Symbol	Parameter	Min.	Max	Units	Notes
V_{DD}	2.5-V core supply voltage	-0.5	3.5	V	
V_{CC}	3.3-V I/O supply voltage	-0.5	4.6	V	
$V_{I\text{-}5V}$	DC input voltage on all 5-V pins	-0.5	VX+0.5	V	1
$V_{I\text{-}3.3V}$	DC input voltage on all 3.3-V pins	-0.5	VDD+0.3	V	
T_{stg}	Storage temperature range	-65	150	Deg. C	
T_{case}	Operating case temperature range	0	120	Deg. C	
V_{ESD}	Electrostatic handling for all pins	-	$\pm$1500	V	2

Notes: 1. VX for a 5V mode pin is either VREF_PCI or VREF_PERIPH, see Section 1.6.
2. Equivalent to discharging a 150-pF capacitor through a 1.5-Kohm series resistor. $\pm$1000 V for TM1300 v 1.15 .

1.9.3 Power Supply Sequencing

Power application and power removal should obey the following rules:

- V_{DD} should never exceed V_{CC} by more than 0.5 V
- V_{CC} should never exceed V_{DD} by more than 1.2 V

Permanent damage may occur if these rules are not observed.

1.9.4 DC/AC Characteristics

Symbol	Parameter	Condition/Notes	Min.	Max	Units
V_{DD}	Core supply voltage		2.375	2.625	V
V_{CC}	I/O supply voltage		3.135	3.465	V
I_{DD}	Core supply current	166 MHz CPU operation (typ. application)		1200	mA
I_{CC}	I/O supply current	143 MHz SDRAM operation (typ. app.)		170	mA
$I_{DD\text{-}pdn}$	Core supply current	CPU power down mode; 166 MHz		250	mA
$I_{CC\text{-}pdn}$	I/O supply current	CPU power down mode; 143 MHz		50	mA
$V_{IH\text{-}5v}$	Input HIGH voltage for I/O-5 V	Note 1. All I/O's except IICOD	2.0	VX+ 0.5	V
$V_{IH\text{-}3.3v}$	Input HIGH voltage for I/O-3.3 V	All I/Os except IICOD	2.0	V_{CC} + 0.3	V
$V_{IL\text{-}5v}$	Input LOW voltage for I/O-5 V	All I/Os except IICOD	-0.5	0.8	V
$V_{IL\text{-}3.3v}$	Input LOW voltage for I/O-3.3 V	All I/Os except IICOD	-0.3	0.8	V
$I_{IL\text{-}5v}$	Input leakage current for I/O-5 V	$0 < V_{IN} < 2.7V$	-70	70	uA
$I_{IL\text{--}3.3v}$	Input leakage current for I/O-3.3 V	$0 < V_{IN} < 2.7V$	-0	10	uA
C_{IN}	Input pin capacitance			8	pF

Notes: 1. VX for a 5V mode pin is either VREF_PCI or VREF_PERIPH, see Section 1.6.

1.9.4.1 TM-1300 and DSPCPU Core Current and Power Consumption Details

		TM1300-100:100			TM1300-143:143			TM1300-166:133			TM1300-180:144			
Symbol	**Current/Notes**	**Pwd**	**Typ**	**Max**	**Pwd**	**Typ**	**Max**	**Pwd**	**Typ**	**Max**	**Pwd**	**Typ**	**Max**	**Units**
TM-1300 (note 1)	I_{DD}	170	800	850	220	1100	1200	250	1200	1350	280	1300	1450	mA
	I_{CC}	40	140	130	50	170	190	45	165	185	50	170	190	mA
	Total Power Dissipation	0.5	2.3	2.5	0.7	3.3	3.5	0.7	3.5	3.8	0.8	3.8	4.1	W
	I_{DD} , DSPCPU Only	-	590	660	-	825	950	-	900	1050	-	1000	1150	mA
	I_{CC} , DSPCPU Only	-	50	40	-	65	50	-	63	50	-	65	50	mA
	Power DSPCPU Only	-	1.6	1.7	-	2.2	2.5	-	2.4	2.7	-	2.7	3.0	W
TM-1300 (note 1,2)	I_{DD} , Standby	-	400	-	-	575	-	-	630	-	-	690	-	mA
	Power Standby	-	1.1	-	-	1.5	-	-	1.7	-	-	1.8	-	W
	I_{DD} , Standby + bpwd	-	300	-	-	425	-	-	450	-	-	510	-	mA
	Power Standby + bpwd	-	0.8	-	-	1.2	-	-	1.3	-	-	1.4	-	W

Notes:
1. Consumption for TM-1300 is organized in several categories. Typ. column presents current consumption for a typical application with a CPI (Clocks Per Instruction) of 1.4 with all the peripherals units turned on (peripherals run on a random data pattern and running at the specified frequencies, VO runs at 27 MHz). Max. column provides current consumption for an application with a CPI of 1.1 with all the peripherals units turned on (peripherals run on a random data pattern and running at the specified frequencies, VO runs at 27 MHz). This is a dedicated application that heavily uses the DSPCPU and that does not reflect a real application but does indicate peak currents. Typ. measurements reflect real applications. Pwd. column indicates current consumption when Global Powerdown mode is activated. See Chapter 21, “Power Management.”
2. TM-1300 Standby rows indicate current consumption when DSPCPU is maintained under RESET (See Section 11.7.5, “BIU_CTL Register”) all peripherals turned off (i.e. not enabled) and all peripherals powered down (+ bpwd row).
3. Measurements accuracy is +/- 5%. Measurements are done with Vdd set to 2.5V and Vcc set to 3.3V.
4. Currents do not scale with frequency if the CPU:SDRAM ratio are different. Same ratio must be used.

1.9.4.2 TM-1300 Peripheral Current Consumption Details

		TM1300-100:100			TM1300-143:143			TM1300-166:133			TM1300-180:144			
Symbol	**Current/Notes**	**Pwd**	**Typ**	**Max**	**Pwd**	**Typ**	**Max**	**Pwd**	**Typ**	**Max**	**Pwd**	**Typ**	**Max**	**Units**
VO 27 MHz	I_{DD} , running raw mode	45	10	21	50	14	27	56	17	30	60	19	33	mA
	I_{CC} , running raw mode	-	15	20	-	23	34	-	22	32	-	25	35	mA
VO 81 MHz	I_{DD} , running raw mode	-	21	51	-	28	68	-	35	73	-	37	75	mA
	I_{CC} , running raw mode	-	43	62	-	56	81	-	57	87	-	60	90	mA
VI 27 MHz	I_{DD} , running raw mode	4	7	9	4	9	15	5	12	21	7	13	23	mA
	I_{CC} , running raw mode	-	12	21	-	18	33	-	17	32	-	19	35	mA
AO 44 KHz	I_{DD} , stereo 16-bit	1	2	2	2	2	2	3	2	3	3	3	4	mA
	I_{CC} , stereo 16-bit	-	1	1	-	1	1	-	1	1	-	1	1	mA
AI 44 KHz	I_{DD} , stereo 16-bit	1	1	2	2	2	2	3	2	3	3	2	3	mA
	I_{CC} , stereo 16-bit	-	1	1	-	1	1	-	1	1	-	1	1	mA
SPDIF 48 KHz	I_{DD} running PCM audio	1	2	2	2	2	2	3	3	4	3	4	4	mA
	I_{CC} running PCM audio	-	1	1	-	2	2	-	2	2	-	2	2	mA
ICP	I_{DD} , mem. block move	50	64	122	60	89	170	65	94	165	72	100	180	mA
	I_{CC} , mem. block move	-	33	72	-	45	101	-	45	96	-	50	105	mA
PCI 33 MHz	I_{DD} , DMA transfer	-	20	75	-	22	86	-	21	97	-	22	95	mA
	I_{CC} , DMA transfer	-	67	145	-	75	155	-	72	160	-	75	155	mA
VLD	I_{DD}	2	-	-	3	-	-	4	-	-	6	-	-	mA
	I_{CC}	-	-	-	-	-	-	-	-	-	-	-	-	mA
SSI 10 MHz	I_{DD}	2	-	-	3	-	-	5	-	-	7	-	-	mA
	I_{CC}	-	-	-	-	-	-	-	-	-	-	-	-	mA
DVDD	I_{DD}	15	-	-	20	-	-	22	-	-	25	-	-	mA
	I_{CC}	-	-	-	-	-	-	-	-	-	-	-	-	mA

Notes:
1. Pwd. column for peripheral units indicates current savings when block powerdown is activated compared to when it is idle. See Chapter 21, “Power Management” for block powerdown activation.
2. Typ. column for peripheral units indicates current required when data pattern is random. The Max. column indicates current ratings when data is switching from high to low level each cycle. Again that Max. column is to show peak current and does not represent a real application. For both columns the current reported is the current required by the peripheral as well as the internal bus and MMI to transfer the data to/from the peripheral unit.
3. Some currents are not reported due to the difficulty to measure it or because they are not relevant. For example SSI current is difficult to measure because it heavily involves the DSPCPU and thus makes it almost impossible to separate the current consumed by the SSI or the DSPCPU.
4. Measurements accuracy is +/- 5%. Measurements are done with Vdd set to 2.5V and Vcc set to 3.3V.
5. Currents do not scale with frequency if the CPU:SDRAM ratio are different. Same ratio must be used.

1.9.4.3 STRG3, STRG5 type I/O circuit

			TM1300-143			TM1300-166/180			
Symbol	**Parameter**	**Condition/Notes**	**Min.**	**Nominal**	**Max**	**Min.**	**Nominal**	**Max.**	**Units**
V_{OH}	Output HIGH voltage	I_{OUT} = 16.0 mA	$0.9V_{CC}$			$0.9V_{CC}$			V
V_{OL}	Output LOW voltage	I_{OUT} = -16.0 mA			$0.1V_{CC}$			$0.1V_{CC}$	V
Z_{OH}	Output AC impedance	HIGH level output state		11			8.5		ohm
Z_{OL}	Output AC impedance	LOW level output state		11			8.5		ohm
t_r	Output rise time	Test load of Figure 1-1.			2.0			1.6	ns
t_r	Output fall time	Test load of Figure 1-1.			2.0			1.6	ns

1.9.4.4 NORM3 type I/O circuit

			TM1300-143			TM1300-166/180			
Symbol	**Parameter**	**Condition/Notes**	**Min.**	**Nominal**	**Max.**	**Min.**	**Nominal**	**Max.**	**Units**
V_{OH}	Output HIGH voltage	I_{OUT} = 8.0 mA	$0.9V_{CC}$			$0.9V_{CC}$			V
V_{OL}	Output LOW voltage	I_{OUT} = -8.0 mA			$0.1V_{CC}$			$0.1V_{CC}$	V
Z_{OH}	Output AC impedance	HIGH level output state		23			17		ohm
Z_{OL}	Output AC impedance	LOW level output state		23			17		ohm
t_r	Output rise time	Test load of Figure 1-2.			4.0			3.0	ns
t_r	Output fall time	Test load of Figure 1-2.			4.0			3.0	ns

1.9.4.5 WEAK5 type I/O circuit

			TM1300-143			TM1300-166/180			
Symbol	**Parameter**	**Condition/Notes**	**Min.**	**Nominal**	**Max.**	**Min.**	**Nominal**	**Max.**	**Units**
V_{OH}	Output HIGH voltage	I_{OUT} = 6.0 mA	$0.9V_{CC}$			$0.9V_{CC}$			V
V_{OL}	Output LOW voltage	I_{OUT} = -6.0 mA			$0.1V_{CC}$			$0.1V_{CC}$	V
Z_{OH}	Output AC impedance	HIGH level output state		33			25		ohm
Z_{OL}	Output AC impedance	LOW level output state		33			25		ohm
t_r	Output rise time	Test load of Figure 1-3.			4.0			3.0	ns
t_r	Output fall time	Test load of Figure 1-3.			4.0			3.0	ns

1.9.4.6 IICOD (I^2c) type I/O circuit

Symbol	**Parameter**	**Condition/Notes**	**Min.**	**Nominal**	**Max.**	**Units**
V_{IL-IIC}	Input LOW voltage		-0.5		1.0	V
V_{IH-IIC}	Input HIGH voltage		2.3		3.6	V
V_{HYS}	Input Schmitt trigger hysteresis		0.25			V
V_{OL}	Output LOW voltage	I_{OUT} = -6.0 mA			0.6	V
t_f	Output fall time	10 - 400 pF load	1.5		250	ns

1.9.4.7 SDRAM interface timing

		TM1300-143		TM1300-166/180			
Symbol	**Parameter**	**Min.**	**Max**	**Min.**	**Max**	**Units**	**Notes**
f_{SDRAM}	MM_CLK frequency		143		143	MHz	1
T_{CS}	Skew between MM_CLK0, CLK1		0.1		0.1	ns	2
T_{PD}	Propagation delay of data, address, control		5.0		4.5	ns	3
T_{OH}	Output hold time of data, address and control	1.5		1.5		ns	3
T_{SU}	Input data setup time	1.0		0.4		ns	4
T_{IH}	Input data hold time	1.5		1.5		ns	4

Notes:
1. For best high speed SDRAM operation, 50-ohm matched PCB traces are recommended for all MM_xxx signals. Use 27-33 ohm series terminator resistors close to TM1300 in the MM_CLK0 and MM_CLK1 line only.
2. Equal load circuit. **MM_CLK0** and **MM_CLK1** are matched output buffers.
3. The center of the two rising edges on **MM_CLK0**, **MM_CLK1** are used as the clock reference point. Propagation delay guarantee is defined from 50% point of clock edge to 50% level on D/A/C. Output hold time guarantee is defined from 50% point of clock edge to 50% level on D/A/C.
4. **MM_CLK0** is used as a reference clock. Input setup time requirement is defined as data value 50% complete to 50% level on clock. Input hold time requirement is defined as minimum time from 50% level on clock to 50% change on data.

1.9.4.8 PCI Bus timing

The following specifications meet the PCI Specifications, Rev. 2.1 for 33-MHz bus operation.

Symbol	**Parameter**	**Min.**	**Max**	**Units**	**Notes**
$T_{val\text{-}PCI\ (Bus)}$	Clk to signal valid delay, bused signals	2	11	ns	1,2,3
$T_{val\text{-}PCI\ (ptp)}$	Clk to signal valid delay, point-to-point signals	2	12	ns	1,2,3
$T_{on\text{-}PCI}$	Float to active delay	2		ns	1
$T_{Off\text{-}PCI}$	Active to float delay		28	ns	1,7
$T_{su\text{-}PCI}$	Input setup time to CLK - bused signals	7		ns	3,4
$T_{su\text{-}PCI\ (ptp)}$	Input setup time to CLK - point-to-point signals	12		ns	3,4
$T_{h\text{-}PCI}$	Input hold time from CLK	0.2[a]		ns	4
$T_{rst\text{-}PCI}$	Reset active time after power stable	1		ms	5
$T_{rst\text{-}clk\text{-}PCI}$	Reset active time after CLK stable	100		µs	5
$T_{rst\text{-}off\text{-}PCI}$	Reset active to output float delay		40	ns	5,6,7

a. PCI Clock skew between two PCI devices must be lower than 1.8ns instead of 2ns as specified in PCI 2.1.

Notes:
1. See the timing measurement conditions in Figure 1-4.
2. Minimum times are measured at the package pin with the load circuit shown in Figure 1-8. Maximum times are measured with the load circuit shown in Figure 1-6 and Figure 1-7.
3. **REG#** and **GNT#** are point-to-point signals and have different input setup times. All other signals are bused.
4. See the timing measurement conditions in Figure 1-5.
5. **RST#** is asserted and de-asserted asynchronously with respect to CLK.
6. All output drivers are floated when **RST#** is active.
7. For the purpose of Active/Float timing measurements, the Hi-Z or ‘off’ state is defined to be when the total current delivered through the component pin is less than or equal to the leakage current specification.

1.9.4.9 JTAG I/O timing

Symbol	Parameter	Min.	Max	Units	Notes
$f_{JTAG-CLK}$	JTAG clock frequency		20	MHz	
$T_{clk-TDO}$	JTAG_TCK to JTAG_TDO valid delay	2	10	ns	1
T_{su-TCK}	Input setup time to JTAG_TCK	3		ns	2
T_{h-TCK}	Input hold time from JTAG_TCK	7		ns	2

Notes: 1. See the timing measurement conditions in Figure 1-10.
2. See the timing measurement conditions in Figure 1-9.

1.9.4.10 I^2C I/O timing

Symbol	Parameter	Min.	Max	Units	Notes
f_{SCL}	SCL clock frequency		400	kHz	1
T_{BUF}	Bus free time	1		us	2
T_{su-STA}	Start condition set up time	1		us	3
T_{h-STA}	Start condition hold time	1		us	3
T_{LOW}	SCL LOW time	1		us	1
T_{HIGH}	SCL HIGH time	1		us	1
T_f	SCL and SDA fall time (Cb = 10-400 pF, from V_{IH-IIC} to V_{IL-IIC})	20+0.1Cb	250	ns	1
T_{su-SDA}	Data setup time	100		ns	4
T_{h-SDA}	Data hold time	0		ns	4
T_{dv-SDA}	SCL LOW to data out valid		0.5	us	5
T_{dv-STO}	SCL HIGH to data out	1		ns	5

Notes: 1. See the timing measurement conditions in Figure 1-11.
2. See the timing measurement conditions in Figure 1-12.
3. See the timing measurement conditions in Figure 1-13.
4. See the timing measurement conditions in Figure 1-14.
5. See the timing measurement conditions in Figure 1-15.

1.9.4.11 Video In I/O Timing

Symbol	Parameter	Min.	Max	Units	Notes
f_{VI-CLK}	Video In clock frequency		81	MHz	
T_{su-CLK}	Input setup time to VI_CLK	2		ns	1
T_{h-CLK}	Input hold time from VI_CLK	2		ns	1

Notes: 1. See the timing measurement conditions in Figure 1-16.

1.9.4.12 Video Out I/O Timing

Symbol	Parameter	Min.	Max	Units	Notes
f_{VO-CLK}	Video Out clock frequency		81	MHz	
T_{CLK-DV}	VO_CLK to VO_DATA (or VO_IO*) out	3	8.5	ns	1,3
T_{CLK-DV}	VO_CLK to VO_DATA (or VO_IO*) out	3	8.5	ns	1,4
T_{su-CLK}	VO_IO* setup time to VO_CLK	10		ns	2
T_{h-CLK}	VO_IO* hold time from VO_CLK	3		ns	2

Notes: 1. See the timing measurement conditions in Figure 1-17.
2. See the timing measurement conditions in Figure 1-18.
3. CLKOUT asserted, i.e. the VO unit is the source of VO_CLK
4. CLKOUT negated, i.e. the external world is the source of VO_CLK

1.9.4.13 AudioIn I/O timing

Symbol	Parameter	Min.	Max	Units	Notes
$f_{AI\text{-}SCK}$	Audio In AI_SCK clock frequency		22	MHz	
$T_{su\text{-}SCK}$	Input setup time to AI_SCK	3		ns	1,2
$T_{h\text{-}SCK}$	Input hold time from AI_SCK	2		ns	1,2
$T_{SCK\text{-}WS}$	AI_SCK to AI_WS		10	ns	3

Notes: 1. See the timing measurement conditions in Figure 1-19.
2. The timing measurements are done with respect to the clock edge according to CLOCK_EDGE
3. SER_MASTER asserted, i.e. Audio In is the source of AI_WS. See the timing measurement condition in Figure 1-20.

1.9.4.14 Audio Out I/O timing

Symbol	Parameter	Min.	Max	Units	Notes
$f_{AO\text{-}SCK}$	Audio Out AO_SCK clock frequency		22	MHz	
$T_{SCK\text{-}DV}$	AO_SCK to AO_SDx valid	2	12	ns	1,3,4
$T_{SCK\text{-}DV}$	AO_SCK to AO_SDx valid	2	12	ns	1,3,5
$T_{su\text{-}SCK}$	Input setup time to AO_SCK	4		ns	2,3,5
$T_{h\text{-}SCK}$	Input hold time from AO_SCK	2		ns	2,3,5
$T_{SCK\text{-}WS}$	AO_SCK to AO_WS		10	ns	3,4,6

Notes: 1. See the timing measurement conditions in Figure 1-21.
2. See the timing measurement conditions in Figure 1-23.
3. The timing measurements are done with respect to the AO_SCK clock edge according to CLOCK_EDGE
4. TM-1 is the serial interface master, i.e. AO_SCK, AO_WS are outputs
5. TM-1 is serial interface slave, i.e. AO_SCK, AO_WS are inputs
6. See the timing measurement conditions in Figure 1-22.

1.9.4.15 SSI I/O timing

Symbol	Parameter	Min.	Max	Units	Notes
$f_{SSI\text{-}CLK}$	SSI_CLK clock frequency		20	MHz	1
$T_{CLK\text{-}DV}$	SSI_CLK to data valid	2	12	ns	2
$T_{su\text{-}CLK}$	Input setup time to SSI_CLK	3		ns	3
$T_{h\text{-}CLK}$	Input hold time from SSI_CLK	2		ns	3

Notes: 1. Interrupt latency limits SSI to a practical use at a bit rate of 1.5 Mbit/sec.
2. See the timing measurement conditions in Figure 1-24.
3. See the timing measurement conditions in Figure 1-25.

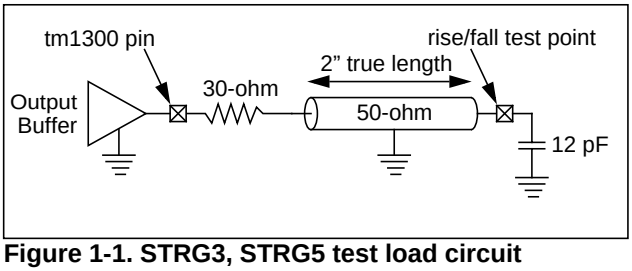


Figure 1-1. STRG3, STRG5 test load circuit

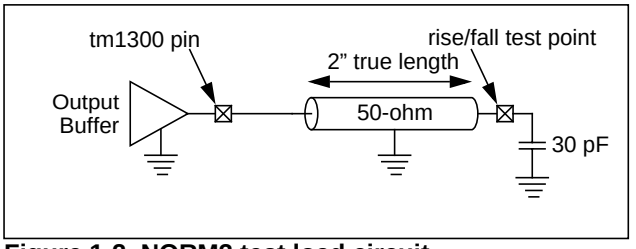


Figure 1-2. NORM3 test load circuit

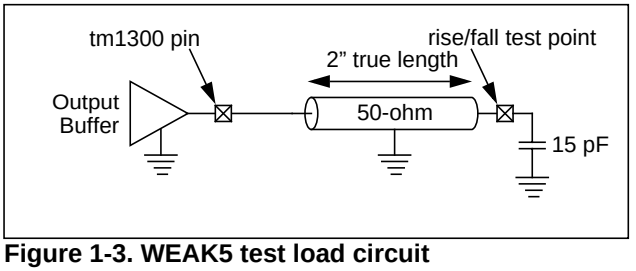


Figure 1-3. WEAK5 test load circuit

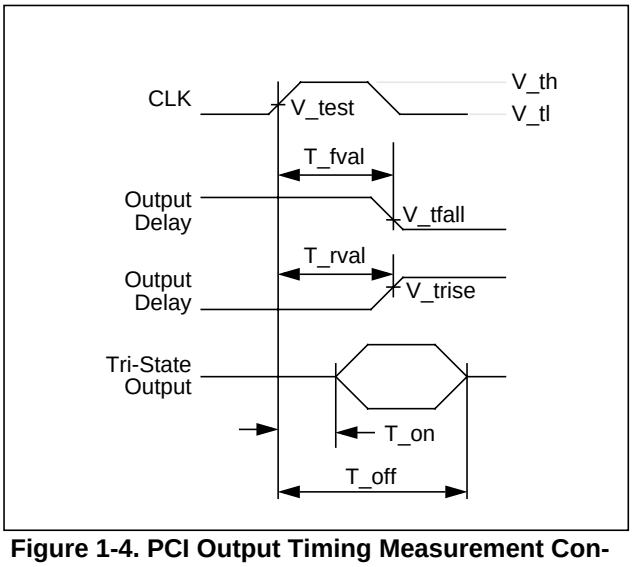


Figure 1-4. PCI Output Timing Measurement Conditions

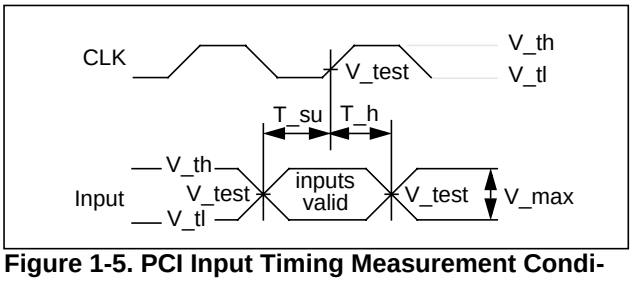


Figure 1-5. PCI Input Timing Measurement Conditions

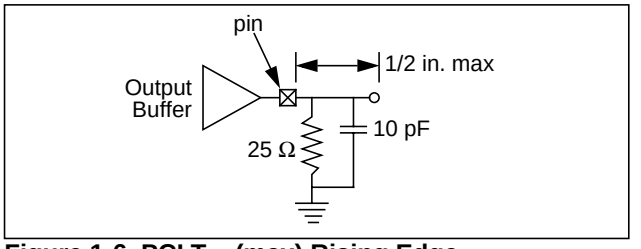


Figure 1-6. PCI T_{val}(max) Rising Edge

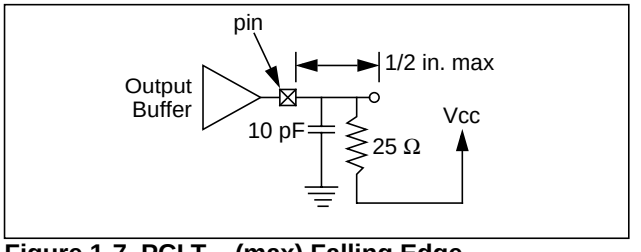


Figure 1-7. PCI T_{val}(max) Falling Edge

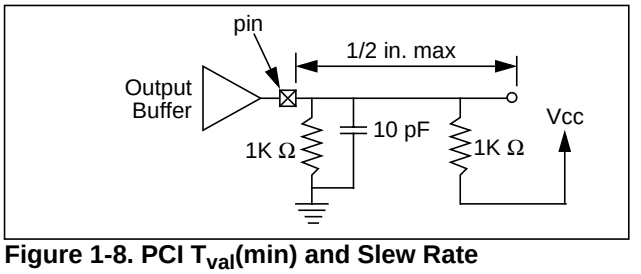


Figure 1-8. PCI T_{val}(min) and Slew Rate

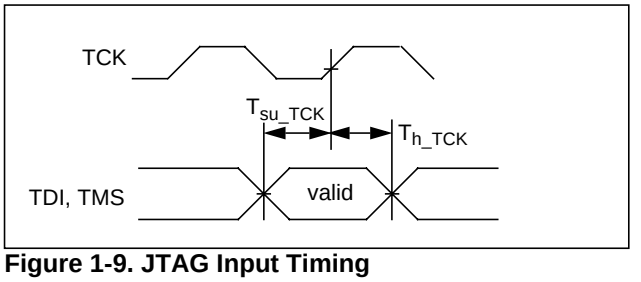


Figure 1-9. JTAG Input Timing

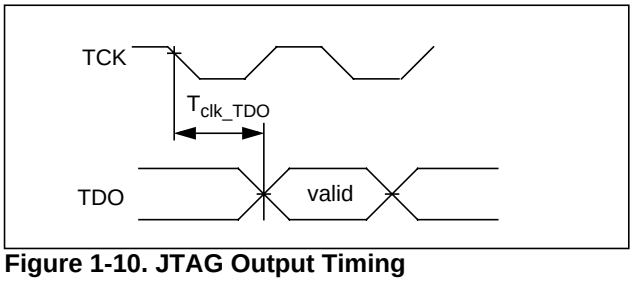


Figure 1-10. JTAG Output Timing

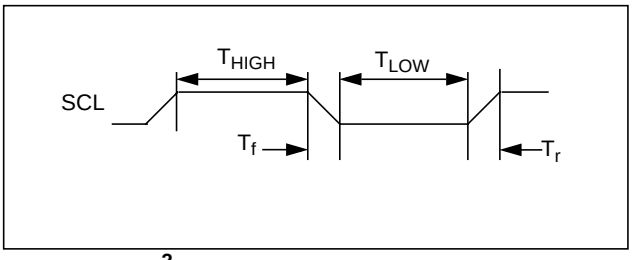


Figure 1-11. I²C I/O Timing

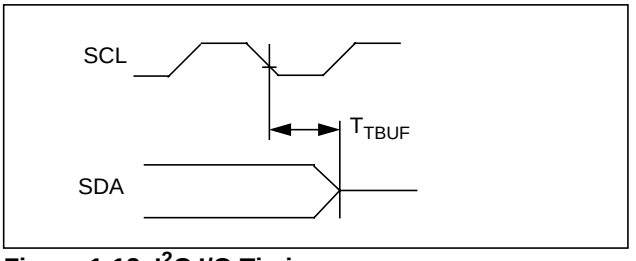


Figure 1-12. I²C I/O Timing

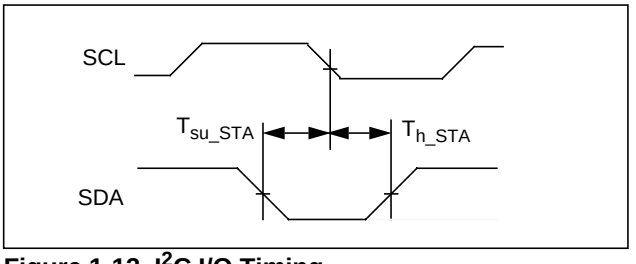


Figure 1-13. I²C I/O Timing

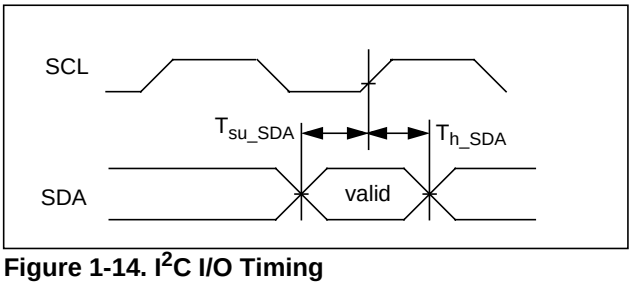


Figure 1-14. I²C I/O Timing

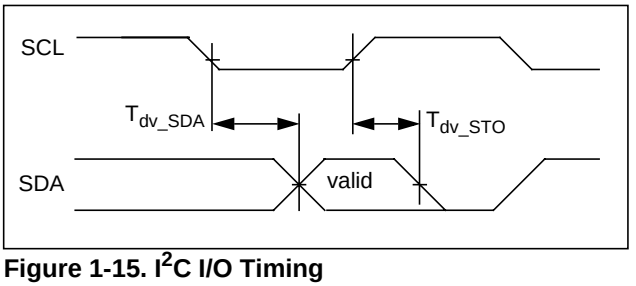


Figure 1-15. I²C I/O Timing

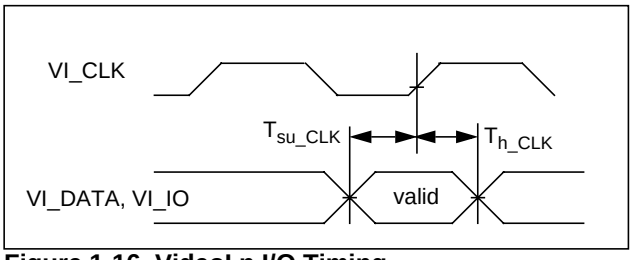


Figure 1-16. VideoI n I/O Timing

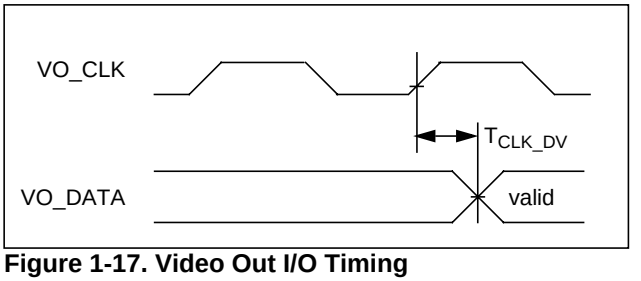


Figure 1-17. Video Out I/O Timing

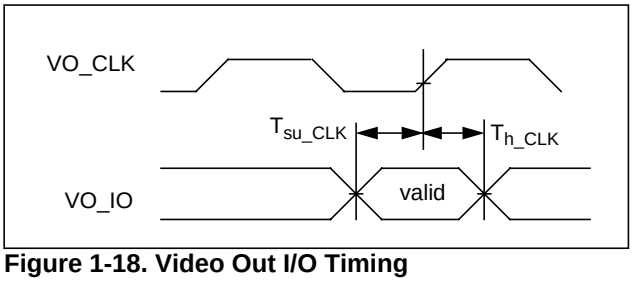


Figure 1-18. Video Out I/O Timing

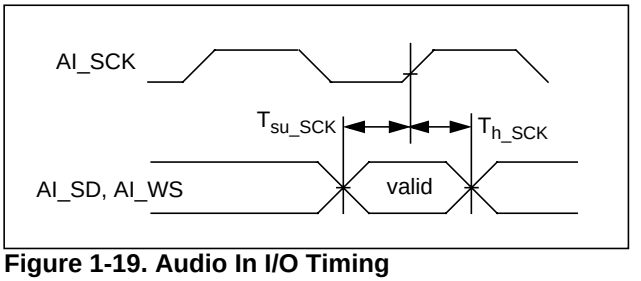


Figure 1-19. Audio In I/O Timing

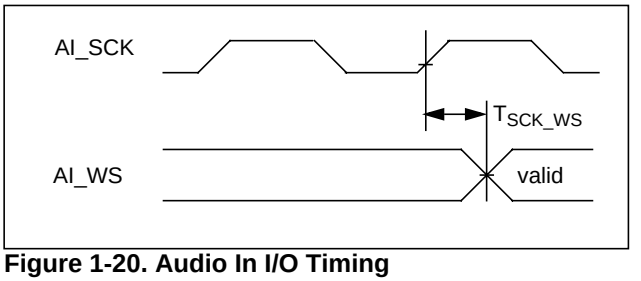


Figure 1-20. Audio In I/O Timing

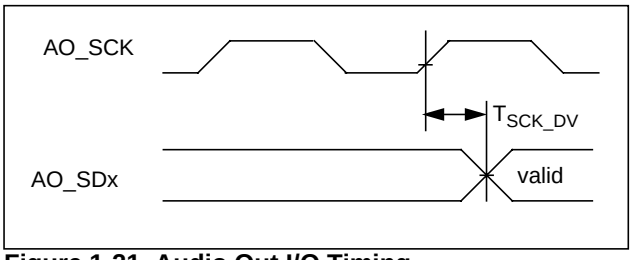


Figure 1-21. Audio Out I/O Timing

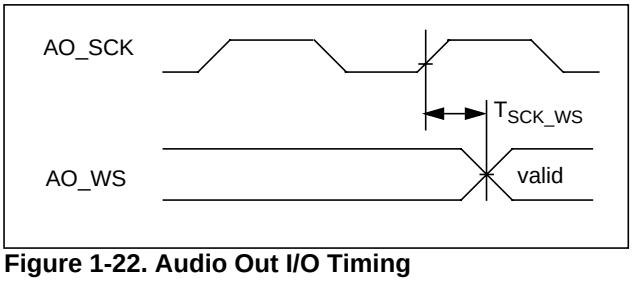


Figure 1-22. Audio Out I/O Timing

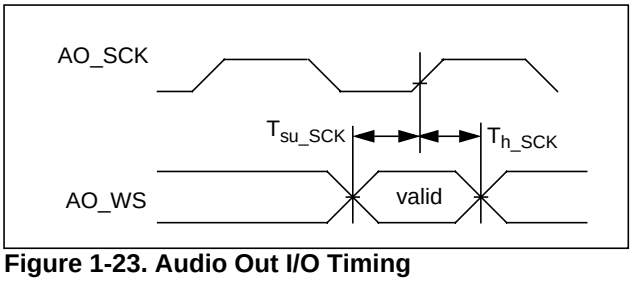


Figure 1-23. Audio Out I/O Timing

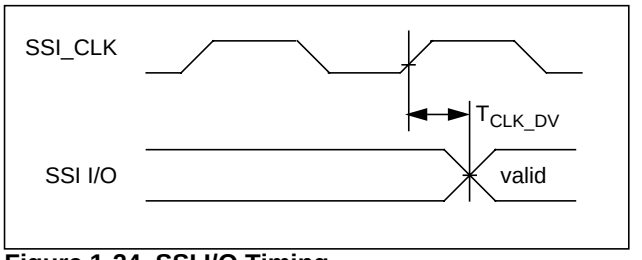


Figure 1-24. SSI I/O Timing

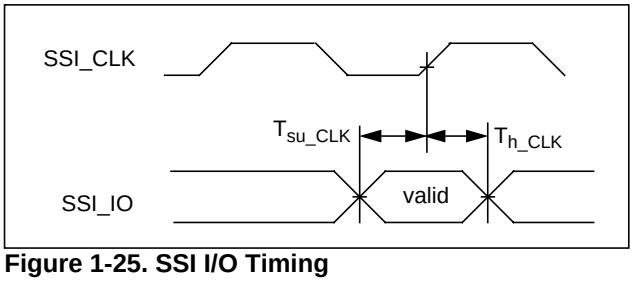


Figure 1-25. SSI I/O Timing

Overview Chapter 2

by Gert Slavenburg

2.1 INTRODUCTION

TM1300 is a successor to the TM1100 and TM1000 media processors. For those familiar with the TM1100, the new features specific to the TM1300 are summarized in Section 2.6. For those familiar with the TM1000, new features for the TM1300 are summarized in Section 2.7.

2.2 TM1300 FUNDAMENTALS

TM1300 is a media processor for high-performance multimedia applications that deal with high-quality video and audio. These applications can range from low-cost, dedicated systems such as video phones, video editing, digital television, security systems or set-top boxes to reprogrammable, multipurpose plug-in cards for personal computers. TM1300 easily implements popular multimedia standards such as MPEG-1 and MPEG-2, but its orientation around a powerful general-purpose CPU (called the DSPCPU) makes it capable of implementing a variety of multimedia algorithms, both open and proprietary. TM1300 is also easily configured in multiple processor configurations for very high-end applications.

More than just an integrated microprocessor with unusual peripherals, the TM1300 is a fluid computer system controlled by a small real-time OS kernel running on a very-long instruction word (VLIW) processor core. TM1300 contains a DSPCPU, a high-bandwidth internal bus, and internal bus-mastering DMA peripherals.

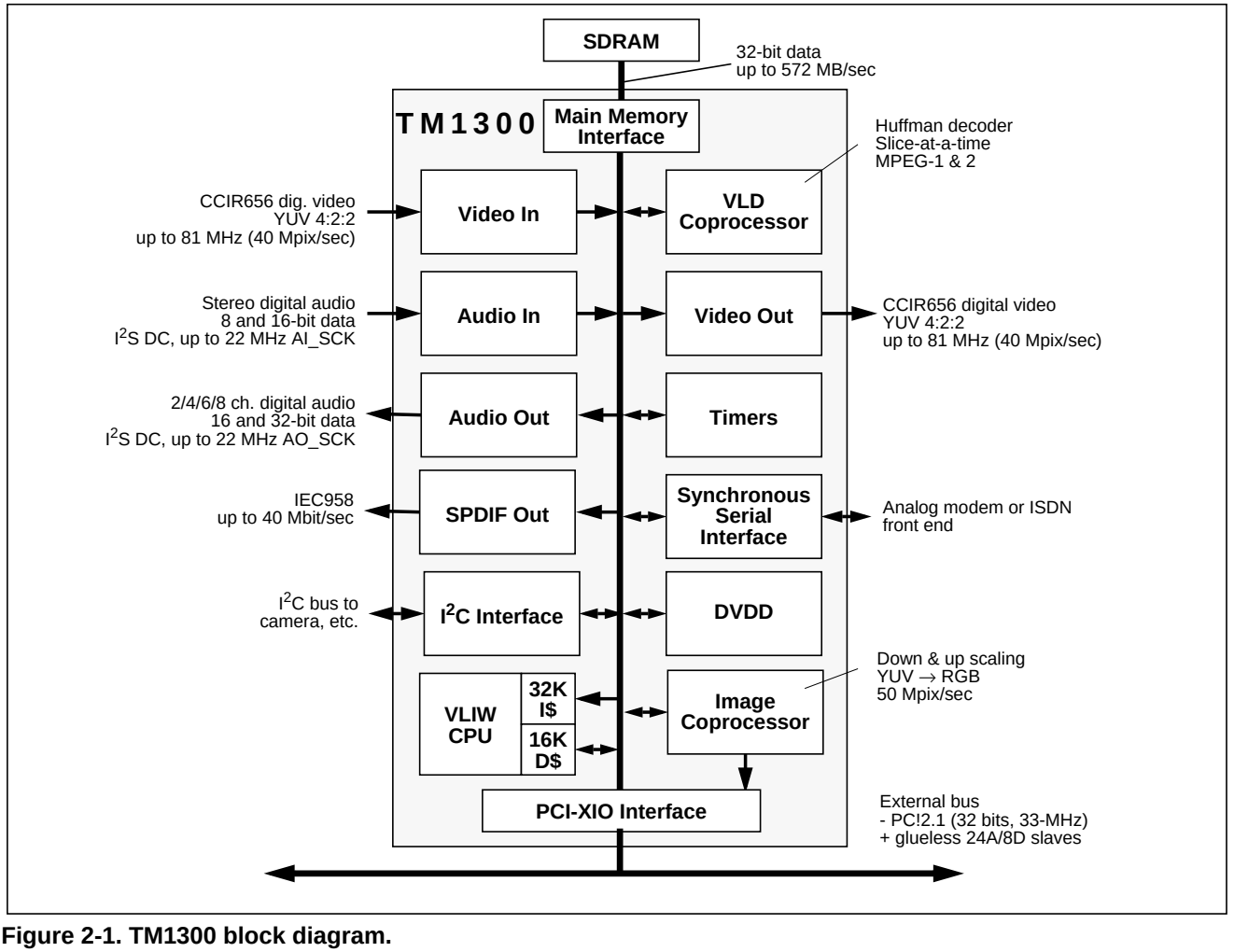


Figure 2-1. TM1300 block diagram.

Software compatibility between current and future Trimedia processor family members is at the source-code and library API level; binary compatibility between family members is not guaranteed.

Defining software compatibility at the source-code level gives Philips the freedom to strike the optimum balance between cost and performance for all chips in the family. A powerful compiler and software development environment ensure that programmers never need to resort to non-portable assembler programming. Programmers use the library APIs and multimedia operations from C and C++ source code.

TM1300 is designed both for use as an accelerator in a PC environment or as the sole CPU in cost-effective standalone systems. In standalone system applications, the TM1300 external bus allows for glueless connection of 8-bit wide ROM, EEPROM, or Flash memory for code storage. The external bus also allows intermixing of PCI2.1 master/slave peripherals and 8-bit simple peripherals, such as UARTs and other 8-bit microprocessor peripherals. This powerful external bus architecture gives system designers a variety of options to configure low-cost, high-performance system solutions.

Because it is based on a general-purpose CPU, TM1300 can also serve as a multifunctional PC enhancement vehicle. Typically, a PC must deal with multistandard video and audio streams; and applications require both decompression and compression. While the CPU chips used in PCs are becoming capable of low-resolution, real-time video decompression, high-quality decompression—not to mention compression—of studio-resolution video is still out of reach. Further, users expect their systems to handle live video and audio without sacrificing system responsiveness.

TM1300 enhances a PC system by providing real-time multimedia with the advantages of a special-purpose, embedded solution—low cost and chip count—*and* the advantages of a general-purpose processor—reprogrammability. For PC applications, TM1300 far surpasses the capabilities of fixed-function multimedia chips.

Future media processor family members will have different sets of interfaces appropriate for their intended use.

2.3 TM1300 CHIP OVERVIEW

Key features of TM1300 include:

- A very powerful, general-purpose VLIW processor core (the DSPCPU) that coordinates all on-chip activities. In addition to implementing the non-trivial parts of multimedia algorithms, the DSPCPU runs a small real-time operating system driven by interrupts from the other units.
- Independent DMA-driven multimedia I/O units that properly format data to make software media processing efficient.
- DMA-driven multimedia coprocessors that operate independently and in parallel with the DSPCPU to perform operations specific to important multimedia algorithms.

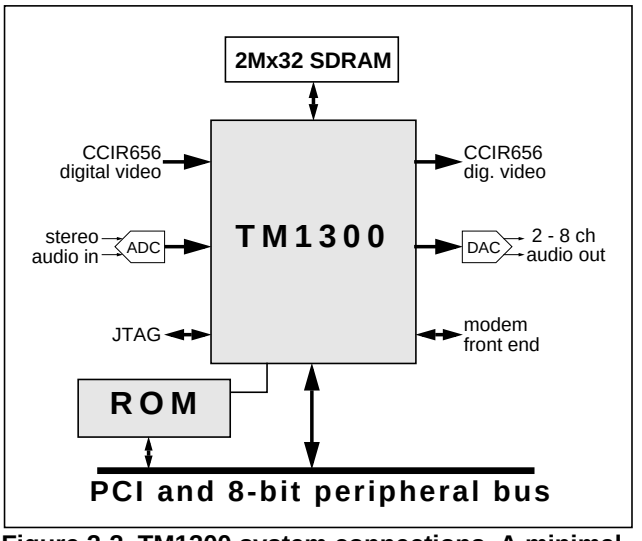


Figure 2-2. TM1300 system connections. A minimal TM1300 requires few supporting components.

- A high-performance bus and memory system that provide communication between TM1300's processing units.
- A flexible external bus interface.

Figure 2-1 shows a TM1300 block diagram. The bulk of a TM1300 system consists of the TM1300 microprocessor itself, external synchronous DRAM (SDRAM), and the external circuitry needed to interface to incoming and/or outgoing video and audio data streams and communication lines. TM1300's external peripheral bus can gluelessly interface to PC! 2.1 components and/or 8-bit microprocessor peripherals.

Figure 2-2 shows a possible minimally configured TM1300 system. A video input stream might come directly from a CCIR 656-compliant video camera chip in YUV 4:2:2 format through a glueless interface in this case. An analog camera can be connected via a CCIR 656 interface chip (such as the Philips SAA7113H). TM1300 outputs a CCIR656 video stream to drive a dedicated video monitor. Stereo audio input and up to 8-channel audio output require only low-cost external ADC and DAC. The operation of the video and audio interface units is highly customizable through programmable parameters.

The glueless PCI interface allows the TM1300 to display video in a host PC's video card. The Image Coprocessor (ICP) provides display support for live video input an arbitrary number of arbitrarily overlapped windows.

Finally, the Synchronous Serial Interface (SSI) requires only an external ISDN or analog modem front-end chip and phone line interface to provide remote communication support. It can be used to connect TM1300-based systems for video phone or videoconferencing applications, or it can be used for general-purpose data communication in PC systems.

The TM1300 JTAG port allows a debugger on a host system to access and control the state of a TM1300 in a target system. It also implements 1149.1 boundary scan functionality.

2.4 BRIEF EXAMPLES OF OPERATION

The key to understanding TM1300 operation is observing that the DSPCPU and peripherals are time-shared and that communication between units is through SDRAM memory. The DSPCPU switches from one task to the next; first it decompresses a video frame, then it decompresses a slice of the audio stream, then back to video, etc. As necessary, the DSPCPU issues commands to the peripheral function units to orchestrate their operation.

The DSPCPU can enlist the ICP and other coprocessors to help with some of the straightforward, tedious tasks associated with video processing. The ICP is very well suited for arbitrary size horizontal and vertical video resizing and color space conversion.

The DSPCPU can enlist the input/output peripherals to autonomously receive or transmit digital video and audio data with minimal CPU supervision. The I/O units have been designed to interface to the outside world through industry standard audio and video interfaces, while delivering or taking data in memory in formats suitable for software processing.

2.4.1 Video Decompression in a PC

An example TM-1300 implementation is as a video-decompression engine on a PCI card in a PC. In this case, the PC does not need to know the TM1300 has a powerful, general-purpose CPU; rather, the PC just treats the hardware on the PCI card as a ‘black-box’ engine.

Video decompression begins when the PC operating system hands the TM1300 a pointer to compressed video data in the PC’s memory (the details of the communication protocol are handled by the software driver installed in the PC’s operating system).

The DSPCPU fetches data from the compressed video stream via the PCI bus, decompresses frames from the video stream, and places them into local SDRAM. Decompression may be aided by the VLD (variable-length decoder) coprocessor unit, which implements Huffman decoding and is controlled by the DSPCPU.

When a frame is ready for display, the DSPCPU gives the ICP a display command. The ICP then autonomously fetches the decompressed frame data from SDRAM and transfers it over the PCI bus to the frame buffer in the PC’s video display card. Alternately, video can be sent to the graphics card using the VO unit.

2.4.2 Video Compression

Another typical application for TM1300 is in video compression. In this case, uncompressed video is usually supplied directly to the TM1300 system via the Video In (VI) unit. A camera chip connected directly to the VI unit supplies YUV data in 8-bit, 4:2:2 format. The VI unit samples the data from the camera chip and demultiplexes the raw video to SDRAM in three separate areas, one each for Y, U, and V.

When a complete video frame has been read from the camera chip by the VI unit, it interrupts the DSPCPU. The DSPCPU compresses the video data in software (using a set of powerful data-parallel multimedia operations) and writes the compressed data to a separate area of SDRAM.

The compressed video data can now be transmitted or stored in any of several ways. It can be sent to a host system over the PCI bus for archival on local mass storage, or the host can transfer the compressed video over a network. The data can also be sent to a remote system using the modem/ISDN interface to create, for example, a video phone or videoconferencing system.

Since the powerful, general-purpose DSPCPU is available, the compressed data can be encrypted before being transferred for security.

2.5 INTRODUCTION TO TM1300 BLOCKS

The remainder of this chapter provides a brief introduction to the internal components of TM1300.

2.5.1 Internal ‘Data Highway’ Bus

The internal bus (or data highway) connects all internal blocks together and provides access to internal control/status registers of each block, external SDRAM, and the external bus peripheral chips. The internal bus consists of separate 32-bit data and address buses. Transactions on the bus use a block-transfer protocol. On-chip peripheral units and coprocessors can be masters or slaves on the bus.

Access to the internal bus is controlled by a central arbiter, which has a request line from each potential bus master. The arbiter is programmable so that the arbitration algorithm can be tailored for different applications. Peripheral units make requests to the arbiter for bus access and, depending on the arbitration mode, bus bandwidth is allocated to the units in different amounts. Each mode allocates bandwidth differently, but each mode guarantees each unit a minimum bandwidth and maximum service latency. All unused bandwidth is allocated to the DSPCPU.

The bus allocation mechanism is one of the features of TM1300 that makes it a true real-time system instead of just a highly integrated microprocessor with unusual peripherals.

2.5.2 VLIW Processor Core

The heart of TM1300 is a powerful 32-bit DSPCPU core. The DSPCPU implements a 32-bit linear address space and 128, fully general-purpose 32-bit registers. The registers are not separated into banks; any operation can use any register for any operand.

The TM1300 core uses a VLIW instruction-set architecture and is fully general-purpose. The VLIW instruction length allows five simultaneous operations to be issued every clock cycle. These operations can target any five of the 27 functional units in the DSPCPU, including integer and floating-point arithmetic units and data-parallel multimedia operation units.

Although the processor core runs a real-time operating system to coordinate all activities in the TM1300 system, the core is not intended for true general-purpose computer use. For example, the TM1300 processor core does not implement demand-paged virtual memory, memory address translation, or 64-bit floating point - all essential features in a general-purpose computer system.

TM1300 uses a VLIW architecture to maximize processor throughput at the lowest possible cost. VLIW architectures have performance exceeding that of superscalar general-purpose CPUs without the cost and complexity of a superscalar CPU implementation. The hardware saved by eliminating superscalar logic reduces cost and allows the integration of multimedia-specific features that enhance the power of the processor core.

The TM1300 operation set includes all traditional microprocessor operations. In addition, multimedia operations are included that dramatically accelerate standard video and audio compression and decompression algorithms. As just one of the five operations issued in a single TM1300 instruction, a single ‘custom’ or ‘media’ operation can implement up to 11 traditional microprocessor operations. These multimedia operations combined with the VLIW architecture result in tremendous throughput for multimedia applications.

The DSPCPU core is supported by separate 16-KB data and 32-KB instruction caches. The data cache is dual-ported to allow two simultaneous accesses; both caches are 8-way set-associative with a 64-byte block size.

2.5.3 Video In Unit

The Video In (VI) unit interfaces directly to any CCIR 601/656-compliant device that outputs 8-bit parallel, 4:2:2 YUV time-multiplexed data. Such devices include direct digital camera systems, which can connect gluelessly to TM1300 or through the standard CCIR 656 connector with only the addition of ECL level converters. A single chip external device can be used to convert to/from serial D1 professional video. Non-CCIR-compliant devices can use a digital video decoder chip, such as the Philips SAA7113H, to interface to TM1300.

The VI unit demultiplexes the captured YUV data before writing it into local TM1300 SDRAM. Separate planar data structures are maintained for Y, U, and V.

The VI unit can be programmed to perform on-the-fly horizontal resolution subsampling by a factor of two if needed. Many camera systems capture a 640-pixel/line or 720-pixel/line image. With subsampling, direct conversion to a 320-pixel/line or a 360-pixel/line image can be performed with no DSPCPU intervention. Performing this function during video input reduces initial storage and bus bandwidth requirements for applications requiring reduced resolution.

2.5.4 Enhanced Video Out Unit

The Enhanced Video Out (EVO) unit essentially performs the inverse function of the VI unit. EVO generates an 8-bit, CCIR656 digital video data stream that contains a composited video and graphics overlay image. The video image is taken from separate Y, U, and V planar data structures in SDRAM. The graphics overlay is taken from a pixel-packed YUV data structure in SDRAM. Compositing allows both alpha-blending and chroma keying.

The EVO unit can also upscale the video image horizontally by a factor of two to convert from CIF/SIF to CCIR 601 resolution. The overlay image, if enabled, is always in full-pixel resolution.

The EVO unit is capable of pixel emission rates up to 40 Mpix/sec and allows full programming of a horizontal and vertical frame/field structure. It is thus capable of refreshing both interlaced and non-interlaced (‘two f_h’) video displays with 4:3 or 16:9 or other aspect ratios.

The sample rate for EVO unit pixels is independently and dynamically programmable. The high-quality, on-chip sample clock generator circuit allows the programmer subtle control over the sampling frequency so that audio and video synchronization can be achieved in any system configuration. When changing the sample frequency, the instantaneous phase does not change, which allows sample frequency manipulation without introducing audio or video distortion.

2.5.5 Image Coprocessor

The ICP off-loads common image scaling or filtering tasks from the DSPCPU. Although these tasks can be easily performed by the DSPCPU, they are a poor use of the relatively expensive CPU resource. When performed in parallel by the ICP, these tasks are performed efficiently by simple hardware, which allows the DSPCPU to continue with more complex tasks.

The ICP can operate as either a memory-to-memory or a memory-to-PCI coprocessor device.

In memory-to-memory mode, the ICP can perform either horizontal or vertical image filtering and resizing. A high quality algorithm is used (5-tap polyphase filter in each direction). Filtering or scaling is done in either the horizontal or vertical direction in one pass. Two invocations of the ICP are required to filter or resize in both directions.

In memory-to-PCI mode, the ICP can perform horizontal resizing followed by color-space conversion. For example, assume an $n \times m$ pixel array is to be displayed in a window on the PC video screen while the PC is running a graphical user interface. The first step (if necessary) would use the ICP in memory-to-memory mode to perform a vertical resizing. The second step would use the ICP in memory-to-PCI mode to perform horizontal resizing and optional colorspace conversion from YUV to RGB.

While sending the final, resampled and converted pixels over the PCI bus to the video frame buffer, the ICP uses a full, per-pixel occlusion bit mask—accessed in destination coordinates—to determine which pixels are actually written to the graphics card frame buffer for display. Conditioning the transfer with the bit mask allows TM1300 to accommodate an arbitrary arrangement of overlapping windows on the PC video screen.

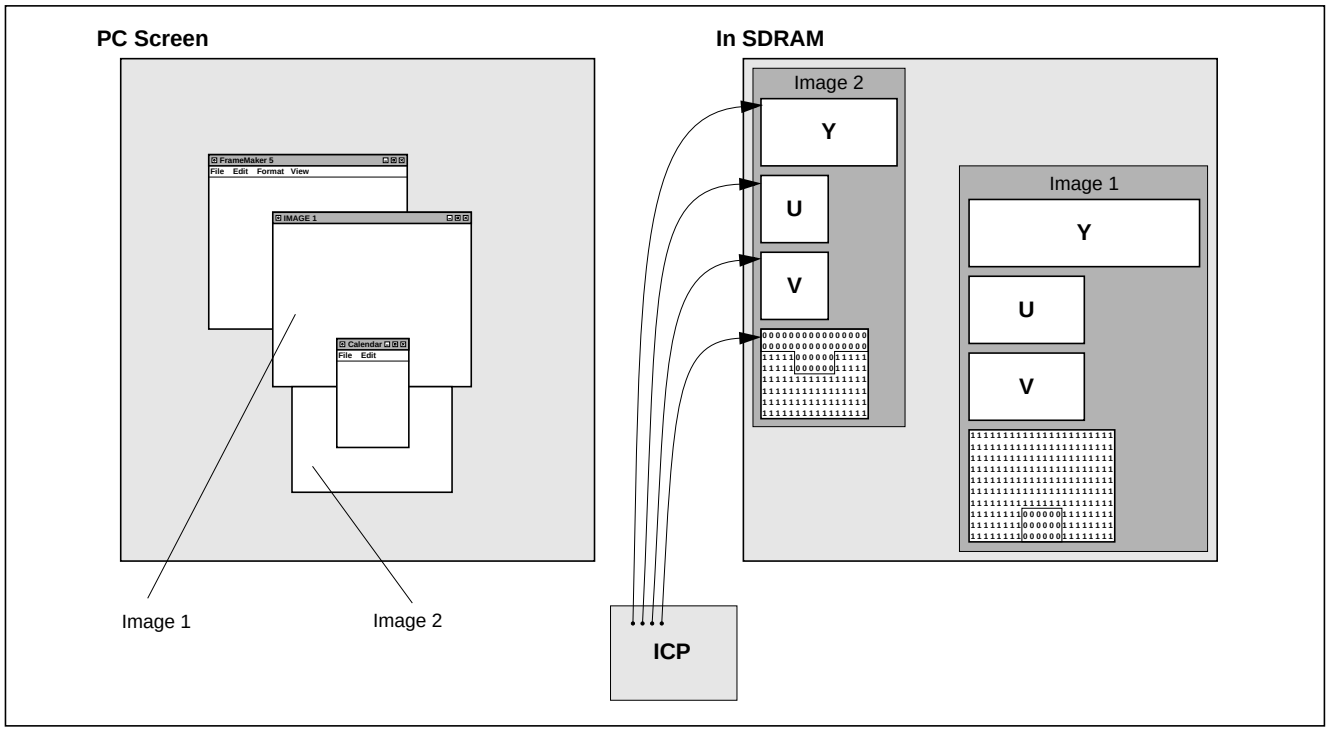


Figure 2-3. ICP - Windows on the PC screen and data structures in SDRAM for two live video windows.

Figure 2-3 illustrates a possible display situation and the data structures in SDRAM that support ICP operation. On the left, the PC video screen has four overlapping windows. Two, Image 1 and Image 2, are being used to display video generated by TM1300. The right side shows a conceptual view of SDRAM contents. Two data structures are present, one for Image 1 and the other for Image 2. Figure 2-3 represents a point in time during which the ICP is displaying Image 2.

When the ICP is displaying an image (i.e., copying it from SDRAM to a frame buffer), it maintains four pointers to the SDRAM data structures. Three pointers locate the Y, U, and V data arrays, the fourth locates the per-pixel occlusion bit map. The Y, U, and V arrays are indexed by source coordinates while the occlusion bit map is accessed with screen coordinates.

As the ICP generates pixels for display, it performs horizontal scaling and colorspace conversion. The final RGB pixel value is then copied to the destination address in the screen's frame buffer only if the corresponding bit in the occlusion bit map is a '1'.

As shown in the conceptual diagram, the occlusion bit map has a pattern of 1s and 0s corresponding to the shape of the visible area of the destination window in the frame buffer. When the arrangement of windows on the PC screen changes, modifications to the occlusion bit map is performed by TM1300 or host resident software.

It is important to note that there is no preset limit on the number and sizes of windows that can be handled by the ICP. The only limit is the available bandwidth. Thus, the ICP can handle a few large windows or many small windows. The ICP can sustain a transfer rate of 50 megapixels per second, which is more than enough to saturate PCI when transferring images to video frame buffers.

2.5.6 Variable-Length Decoder (VLD)

The variable-length decoder (VLD) relieves the DSPCPU of decoding Huffman-encoded video data streams. It can be used to help decode high bitrate MPEG-1 and MPEG-2 video streams. The lower bitrate of videoconferencing can be adequately handled by DSPCPU software without coprocessor.

The VLD is a memory-to-memory coprocessor. The DSPCPU hands the VLD a pointer to a Huffman-encoded bit stream, and the VLD produces a tokenized bit stream that is very convenient for the TM1300 image decompression software to use. The format of the output token stream is optimized for the MPEG-2 decompression software so that communication between the DSPCPU and VLD is minimized.

2.5.7 Audio In and Audio Out Units

The Audio In (AI) and Audio Out (AO) units are similar to the video units. They connect to most serial ADC and DAC chips, and are programmable enough to handle most serial bit protocols. These units can transfer MSB or LSB first and left or right channel first.

The audio sampling clock is driven by TM1300 and is software programmable within a wide range. Like the VO unit, AI and AO sample rates are separately and dynamically programmable. The high-quality on-chip sample clock generator circuits allows the programmer subtle control over the sampling frequency so that audio and video synchronization can be achieved in any system configuration. When changing the sample frequency, the instantaneous phase does not change, which allows sample frequency manipulation without introducing audio or video distortion.

As with the video units, the audio-in and audio-out units buffer incoming and outgoing audio data in SDRAM. The audio-in unit buffers samples in either 8- or 16-bit format, mono or stereo. The audio-out unit transfers 16- or 32-bit sample data for mono, stereo or up to 8 audio channels from memory to the external DACs. Any manipulation or mixing of sound data is performed by the DSPCPU since this processing will require only a small fraction of its processing capacity.

2.5.8 S/PDIF Out Unit

The Sony/Philips Digital Interface Out (SPDO) unit allows output of a 1-bit high-speed serial data stream. The primary application is output of digital audio data in Sony/Philips Digital Interface (S/PDIF) format to an external electrically isolated transformer. The SPDO unit can also be used as a general purpose high-speed data stream output device such as a UART.

The SPDO unit supports 2-channel PCM audio, one or more Dolby Digital six-channel data streams, or one or more MPEG-1 or MPEG-2 audio streams (embedded per Project 1937). It supports arbitrary programmable sample rates independent of and asynchronous to the AO unit sample rate.

2.5.9 Synchronous Serial Interface

The on-chip synchronous serial interface (SSI) is specially designed to interface to high integration analog modem frontends or ISDN frontend devices. In the analog modem case, all of the modem signal processing is performed in the TM1300 DSPCPU.

2.5.10 I^2C Interface

The I^2C bus is a 2-wire multi-master, multi-slave interface capable of transmitting up to 400 kbit/sec. TM1300 implements an I^2C master for use in single master environments only. This interface allows TM1300 to configure and inspect the status of I^2C peripheral devices, such as video decoders, video encoders and some camera types.

2.6 NEW IN TM1300 (VERSUS TM1100)

TM1300 offers significant improvements over the TM1100:

- DSPCPU and coprocessor speed of up to 166 MHz
- Support for 64-Mbit organized in x8 (limited to 32 MBytes), x16, x32 and 128 Mbit organized in x16 (limited to 32 MB). See Chapter 12, “SDRAM Memory System.”
- SDRAM speed up to 143 MHz and no external MATCHOUT to MATCHIN delay line.
- Video output speed improvement: up to 81 MHz.
- Video input speed improvement: up to 81 MHz.
- Prefetcheable SDRAM aperture to increase performance. See Chapter 11, “PCI Interface.”
- Individual powerdown capability for each coprocessor (e.g. ICP, EVO, etc.).
- New AO coprocessor with four separate channels and support of 16 or 32-bit samples. 8-bit samples are no longer supported.
- New SPDO coprocessor (for output of SPDIF and other 1-bit high-speed serial data streams)

2.7 NEW IN TM1300 (VERSUS TM1000)

Iin addition to the features described in Section 2.6 TM1300 offers also the following improvements over the TM1000:

- New DSPCPU instructions. See Appendix A, “DSPCPU Operations for TM1300.”
- Video Output unit improvements (8-bit alpha blending, chroma keying, genlock). See Chapter 7, “Enhanced Video Out.”
- Capability to intermix PCI2.1 and 8-bit peripherals or ROM/Flash memories on the external bus. See Chapter 22, “PCI-XIO External I/O Bus.”
- An on-chip DVD authentication/descrambling coprocessor. Information available to DVD product developers on special request.
- Full 1149.1 boundary scan.
- Improved PCI DMA read performance. See Section 11.1.
- Improved clock generation with new DDS blocks.

by Gert Slavenburg, Marcel Janssens

3.1 BASIC ARCHITECTURE CONCEPTS

This section documents the system programmer or ‘bare-machine’ view of the TM1300 CPU (or DSPCPU).

3.1.1 New in TM1300

Default reset value of PCSW register is 0x800. This new reset value allows Audio Out and SPDIF Out timestamp registers to be in phase with CCCOUNT (lower 32 bits).

3.1.2 Register Model

Figure 3-1 shows the DSPCPU’s 128 general purpose registers, r0...r127. In addition to the hardware program counter, PC, there are 4 user-accessible special purpose registers, PCSW, DPC (destination program counter), SPC (source program counter), and CCCOUNT. Table 3-1 lists the registers and their purposes.

Register r0 always contains the integer value '0', corresponding to the boolean value 'FALSE' or the single-precision floating point value +0.0. Register r1 always contains the integer value '1' ('TRUE'). The programmer is NOT allowed to write to r0 or r1.

> **Note:** Writing to r0 or r1 may cause reads from r0 or r1 scheduled in adjacent clock cycles to return unpredictable values. The standard assembler prevents/forbids the use of r0 or r1 as a destination register.

Registers r2 through r127 are true general purpose registers; the hardware does not imply their use in any way, though compiler or programmer conventions may assign particular roles to particular registers. The DPC and SPC relate to interrupt and exception handling and are treated in Section 3.1.5, “SPC and DPC—Source and Destination Program Counter.” The PCSW (Program Control and Status Word) register is treated in Section 3.1.4, “PCSW Overview.” CCCOUNT, the 64-bit clock cycle counter is treated in Section 3.1.6, “CCCOUNT—Clock Cycle Counter.”

Table 3-1. DSPCPU registers

Register	Size	Details
r0	32 bits	Always reads as 0x0; must not be used as destination of operations
r1	32 bits	Always reads as 0x1; must not be used as destination of operations
r2–r127	32 bits	126 general-purpose registers
PC	32 bits	Program counter
PCSW	32 bits	Program control & status word
DPC	32 bits	Destination program counter; latches target of taken branch that is interrupted
SPC	32 bits	Source program counter; latches target of taken branch that is not interrupted
CCCOUNT	64 bits	Counts clock cycles since reset

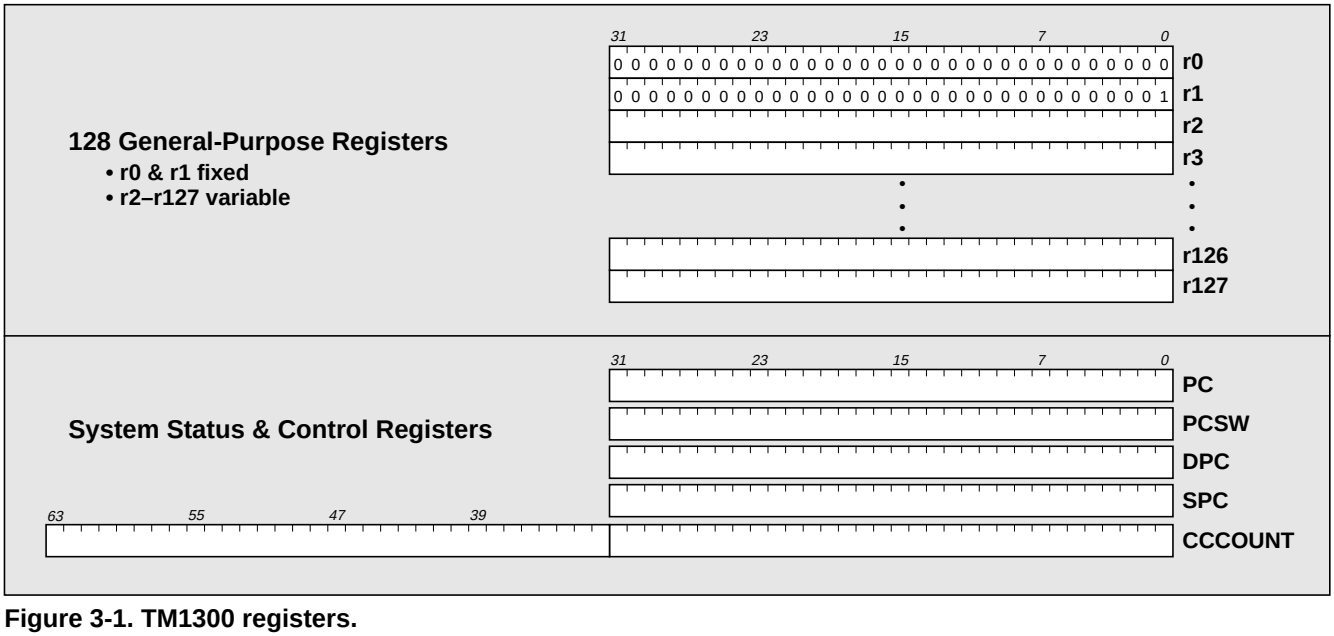


Figure 3-1. TM1300 registers.

3.1.3 Basic DSPCPU Execution Model

The DSPCPU issues one ‘long instruction’ every clock cycle. Each instruction consists of several operations (five operations for the TM1300 microprocessor). Each operation is comparable to a RISC machine instruction, except that the execution of an operation is conditional upon the content of a general purpose register. Examples of operations are:

```
IF r10 iadd r11 r12 → r13
```
(if r10 true, add r11 and r12 and write sum in r13)

```
IF r10 ld32d(4) r15 → r16
```
(if r10 true, load 32 bits from mem[r15+4] into r16)

```
IF r20 jmpf r21 r22
```
(if r20 true and r21 false, jump to address in r22)

Each operation has a specific, known execution latency in clock cycles. For example, iadd takes 1 cycle; thus the result of an iadd operation started in clock cycle *i* is available for use as an argument to operations issued in cycle *i+1* or later. The other operations issued in cycle *i* cannot use the result of iadd. The ld32d operation has a latency of 3 cycles. The result of an ld32d operation started in cycle *j* is available for use by other operations issued in cycle *j+3* or later. Branches, such as the jmpf example above have three delay slots. This means that if a branch operation in cycle *k* is taken, all operations in the instructions in cycle *k+1, k+2* and *k+3* are still executed.

In the above examples, r10 and r20 control conditional execution of the operations. Also known as ‘guarding’, here r10 and r20 contain the operation ‘guard’. See Section 3.2.1, “Guarding (Conditional Execution).”

Certain restrictions exist in the choice of what operations can be packed into an instruction. For example, the DSPCPU in TM1300 allows no more than two load/store class operations to be packed into a single instruction. Also, no more than five results (of previously started operations) can be written during any one cycle. The packing of operations is not normally done by the programmer. Instead, the *instruction scheduler* (See Philips TriMedia SDE Reference Manual) takes care of converting the parallel intermediate format code into packed instructions ready for the assembler. The rules are formally described in the *machine description file* used by the instruction scheduler and other tools.

3.1.4 PCSW Overview

Figure 3-2 shows the PCSW register. The TM1300 value of PCSW on reset is 0. For compatibility, any undefined PCSW fields should never be modified.

Note that the DSPCPU architecture has no condition codes or integer arithmetic status flags. Integer operations that generate out-of-range results deliver an operation specific bit pattern. For example, see dspiadd in Appendix A, “DSPCPU Operations for TM1300.” Predicate operations exist that take the place of integer status flags in a classical architecture. Multiword arithmetic is supported by the ‘carry’ operation which generates a ‘0’ or ‘1’ depending on the carry that would be generated if its arguments were summed.

FP-Related Fields. The IEEE mode field determines the IEEE rounding mode of all floating point operations, with the exception of a few floating point conversion operations that use fixed rounding mode. For example, see ifixrz, ifloatrz, ifixrz, ifloatrz in Appendix A, “DSPCPU Operations for TM1300.”

The *FP exception flags* are ‘sticky bits’ that are set as a side effect of floating-point computations. Each floating point operation can set one or more of the flags if it incurs the corresponding exception. The flags can only be reset by direct software manipulation of the PCSW (using the writepcsw operation). The bits have the meanings shown in Table 3-2.

The *FP exception trap enable bits* determine which FP exception flags invoke CPU exception handling. An exception is requested if the intersection of the exception flags and trap enable flags is non-zero. The acceptance and handling of exceptions is described in Section 3.5, “Special Event Handling.”

BSX (Bytesex). The DSPCPU has a switchable bytesex. The BSX flag in the PCSW can be written by software. Load/store operations observe little- or big-endian byte ordering based on the current setting of BSX.

IEN (Interrupt Enable). The IEN flag disables or enables interrupt processing for most interrupt sources. Only NMI (non-maskable interrupt) bypasses IEN. The acceptance and handling of interrupts is described in Section 3.5.3, “INT and NMI (Maskable and Non-Maskable Interrupts).”

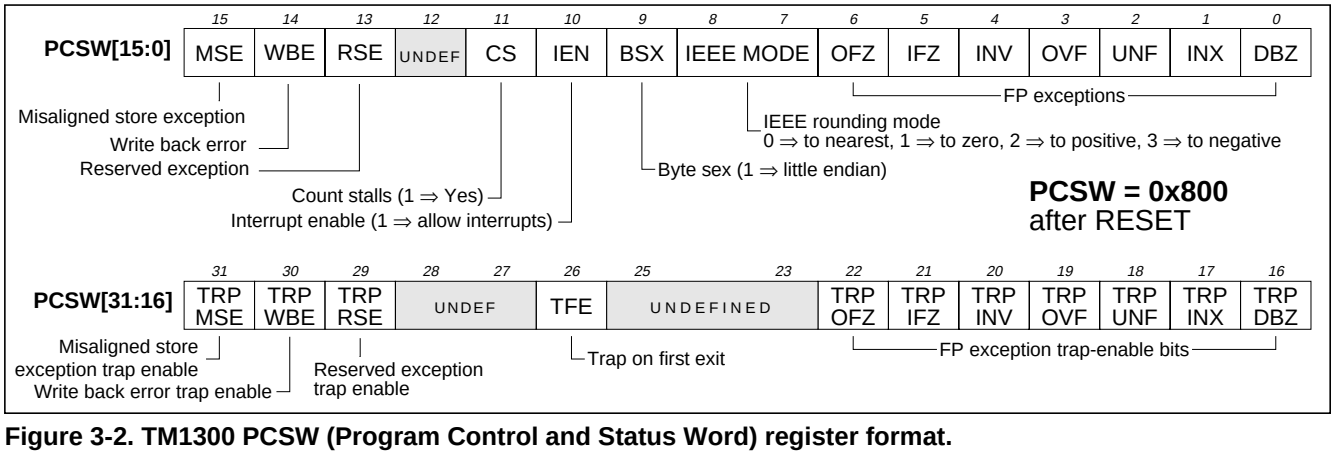


Figure 3-2. TM1300 PCSW (Program Control and Status Word) register format.

Table 3-2. PCSW FP exception flag definitions

Flag	Function
INV	Standard IEEE invalid flag
OVF	Standard IEEE overflow flag
UNF	Standard IEEE underflow flag
INX	Standard IEEE inexact flag
DBZ	Standard IEEE divide-by-zero flag
OFZ	'Output flushed to zero' set if an operation caused a denormalized result
IFZ	'Input flushed to zero' set if an operation was applied to one or more denormalized operands

CS (Count Stalls). The CS flag determines the mode of CCCOUNT, the 64-bit clock cycle counter. If CS = '1', the cycle counter increments on all clock cycles. If CS = '0', the clock cycle counter only increments on non-stall cycles. See also Section 3.1.6, "CCCOUNT—Clock Cycle Counter." After RESET, CS is set to '1'.

MSE and TRPMSE (Misaligned-Store Exception). The MSE bit will be set when the processor detects a store operation to an address that is not aligned. For example, a 32-bit store executed with an address that is not a multiple of four will cause MSE to be set. The TRPMSE bit enables the DSPCPU to raise misaligned address exceptions. An exception is requested if the intersection of MSE and TRPMSE is non-zero. The acceptance and handling of exceptions is described in Section 3.5, "Special Event Handling."

Unaligned load operations do not cause an exception, because load operations can be speculative (i.e. their result is thrown away).

When the DSPCPU generates an unaligned address, the low order address bit(s) (one bit in the case of a 16-bit load, two bits for a 32-bit load) are forced to zero and the load/store is executed from this aligned address.

WBE and TRPWBE (Write Back Error). The WBE flag will be set whenever a program attempts to write back more than 5 results simultaneously. This is indicative of a programming error, likely caused by the scheduler or assembler. The TRPWBE bit enables the corresponding exception.

RSE, TRPRSE (Reserved Exception). RSE and TRPRSE are reserved for diagnostic purposes and not described here.

TFE (Trap on First Exit). The TFE bit is a support bit for the debugger. The TFE bit is set by the debugger prior to taking a (non-interruptible) jump to the application program. On the next interruptible jump (the first interruptible jump in the application being debugged), an exception is requested because the TFE bit is set. The acceptance and handling of exception processing is described in Section 3.5, "Special Event Handling." It is the responsibility of the exception handler software to clear the TFE bit. The hardware does not clear or set TFE.

Corner-case note: Whenever a hardware update (e.g. an exception being raised) and a software update (through writepcsw) of the PCSW coincide, the new value of the PCSW will be the value that is written by the writepcsw instruction, except for those bits that the hardware is currently updating (which will reflect the hardware value).

3.1.5 SPC and DPC—Source and Destination Program Counter

The SPC and DPC registers are support registers for exception processing. The DPC is updated during every interruptible jump with the target address of that interruptible jump. If an exception is taken at an interruptible jump, the value in the DPC register can be used by the exception handling routine as the return address to resume the program at the place of interruption.

The SPC register is updated during every interruptible jump that is not interrupted by an exception. Thus on an interrupted interruptible jump, the SPC register is not updated. The SPC register allows the exception handling routine to determine the start address of the decision tree (a block of uninterruptible, scheduled TM1300 code) that was executing when the exception was taken (see also Section 3.5, "Special Event Handling").

Corner-case note: Whenever a hardware update (during an interruptible jump) and a software update (through writedpc or writespc) coincide, the software update takes precedence.

3.1.6 CCCOUNT—Clock Cycle Counter

CCCOUNT is a 64-bit counter that counts clock cycles since RESET. Cycle counting can occur in two modes, depending on PCSW.CS. If PCSW.CS = '1', the cycle count increments on every CPU clock cycle. If PCSW.CS = '0', the clock cycle count only increments on non-stall CPU cycles.

CCCOUNT is implemented as a master counter/slave register pair. The master 64-bit counter gets updated continuously. The value of the CCCOUNT slave register is updated with the current master cycle count during successful interruptible jumps only. The *cycles* and *hicycles* DSPCPU operations return the content of the 32 LSBs and 32 MSBs, respectively, of the slave register. This ensures that the value returned by hicycles and cycles is coherent, as long as there is no intervening interruptible jump, which makes these operations suitable for 64-bit high resolution timing from C source code programs. The *curcycles* DSPCPU operation returns the 32 LSBs of the master counter. The latter operation can be used for instruction cycle precise timing. When used, it must be precisely placed, probably at the assembly code level.

3.1.7 Boolean Representation

The bit pattern generated by boolean valued operations (ileq, fleq etc.) is '00...00' (FALSE) or '00...01' (TRUE). When interpreting a bit pattern as a boolean value, only the LSB is taken into account, i.e. 'xx..x0' is interpreted as FALSE and 'xx..x1' is interpreted as TRUE. In particular, wherever a general purpose register is used as a 'guard', the LSB determines whether execution of the guarded operation takes place.

3.1.8 Integer Representation

The architecture supports the notion of 'unsigned integers' and 'signed integers.' Signed integers use the standard two's-complement representation.

Arithmetic on integers does not generate traps. If a result is not representable, the bit pattern returned is operation specific, as defined in the individual operation description section. The typical cases are:

- Wrap around for regular add- and subtract-type operations.
- Clamping against the minimum or maximum representable value for DSP-type operations.
- Returning the least significant 32-bit value of a 64-bit result (e.g., integer/unsigned multiply).

3.1.9 Floating Point Representation

The TM1300 architecture supports single precision (32-bit) IEEE-754 floating point arithmetic.

All arithmetic conforms to the IEEE-754 standard in flush-to-zero mode.

All floating point compute operations round according to the current setting of the *PCSW IEEE mode* field. The current setting of the field determines result rounding (to nearest, to zero, to positive infinity, to negative infinity). Conversions from float to integer/unsigned are available in two forms: a PCSW rounding-mode-observing form and an ANSI-C-specific-rounding form. The ANSI-C-specific form forces round to zero regardless of the PCSW IEEE rounding mode. Conversion from integer/unsigned to float always observes the IEEE rounding mode.

Floating point exceptions are supported with two mechanisms. Each individual floating point operation (e.g. fadd) has a counterpart operation (faddflags) that computes the exception flag values. These operations can be used for precise exception identification[1]. The second mechanism uses the 'sticky' exception bits in the PCSW that collect aggregate exception events. The PCSW exception bits can selectively invoke CPU exception handling. See Section 3.5.2, "EXC (Exceptions)."

Table 3-3 shows the representation choices that were made in TM1300's floating point implementation.

3.1.10 Addressing Modes

The addressing modes shown in Table 3-4 are supported by the DSPCPU architecture (store operations allow only displacement mode).

1. This mechanism allows precise exception identification in the context of our multi-issue microprocessor core—where many floating point operations may issue simultaneously—at the expense of additional operations generated by the compiler. It also allows the compiler to issue compute operations speculatively and compute exceptions precisely.

Table 3-3. Special Float Value Representation

Item	Representation
+inf	0x7f800000
-inf	0xff800000
self generated qNaN	0xffffffff
result of operation on any NaN argument	argument | 0x00400000 (forcing the NaN to be quiet)
signalling NaN	never generated by TM1300, accepted as per IEEE-754

Table 3-4. Addressing Modes

Mode	Suffix	Applies to	Name
R[i] + scaled(#j)	d	Load & Store	Displacement
R[i] + R[k]	r	Load only	Index
R[i] + scaled(R[k])	x	Load only	Scaled index

In these addressing modes, R[i] indicates one of the general purpose registers. The scale factor applied (1/2/4) is equal to the size of the item loaded or stored, i.e. 1 for a byte operation, two for a 16-bit operation and four for a 32-bit operation. The range of valid 'i', 'j' and 'k' values may differ between implementations of the architecture; the minimum values for implementation-dependent characteristics are shown in Table 3-5.

Table 3-5. Minimum values for implementation-dependent addressing mode components

Parameter	Minimum Range
'i' and 'k'	0..127 (i.e., each implementation has at least 128 registers)
'j'	-64..63 (i.e., displacements will be at least 7 bits long and signed)

Note that the assembly code specifies the true displacement, and not the value to be scaled. For example, 'ld32d(–8) r3' loads a 32-bit value from address (r3 – 8). This is encoded in the binary operation pattern as a –2 in the seven-bit field by the assembler. At runtime, the scale factor four is applied to reconstruct the intended displacement of –8.

3.1.11 Software Compatibility

The DSPCPU architecture expressly does not support binary compatibility between family members. The ANSI C compiler ensures that all family members are compatible at the source-code level.

3.2 INSTRUCTION SET OVERVIEW

3.2.1 Guarding (Conditional Execution)

In the TM1300 architecture, all operations can be optionally 'guarded'. A guarded operation executes conditionally, depending on the value in the ‘guard' register. For example, a guarded add is written as:

```
IF R23 iadd R14 R10 → R13
```

This should be taken to mean

if R23 then R13 ← R14 + R10.

The ’if R23' clause controls the execution of the operation based on the LSB of R23. Hence, depending on the LSB of R23, R13 is either unchanged or set to contain the integer sum of R14 and R10.

Guarding applies to all DSPCPU operations, except iimm and uimm (load-immediate). It controls the effect on all programmer-visible states of the system, i.e. register values, memory content, exception raising and device state.

3.2.2 Load and Store Operations

Memory is byte addressable. Loads and stores must be ‘naturally aligned’, i.e. a 16-bit load or store must target an address that is a multiple of 2. A 32-bit load or store must target an address that is a multiple of 4. The BSX bit in the PCSW determines the byte order of loads and stores. For example, see ld32 and st32 in Appendix A, “DSPCPU Operations for TM1300.”

Only 32-bit load and store operations are allowed to access MMIO registers in the MMIO address aperture (see Section 3.4, “Memory and MMIO”). The results are undefined for other loads and stores. A load from a non-existent MMIO register returns an undefined result. A store to a non-existent MMIO register times out and then does not happen. There are no other side effects of an access to a nonexistent MMIO register. The state of the BSX bit has no effect on the result of MMIO accesses.

Loads are allowed to be issued speculatively. Loads outside the range of valid data memory addresses for the active process return an implementation-dependent value and do not generate an exception. Misaligned loads also return an implementation dependent value and do not generate an exception.

If a pair of memory operations involves one or more common bytes in memory, the effect on the common bytes is as defined in Table 3-6.

Table 3-4 shows the supported addressing modes. The minimum values of implementation-dependent addressing-mode components are shown in Table 3-5.

Note: The index and scaled-index modes are not allowed with store opcodes, due to the hardware restriction that each operation have at most 2 source operand registers and 1 condition register. Stores use 1 operand register for the value to be stored leaving only 1 register to form an address.

Table 3-6. Behavior of loads and stores with coincident addresses

Condition	Behavior
$T_{store} < T_{load}$	If a store is issued before a load, the value loaded contains the new bytes.
$T_{load} < T_{store}$	If a load is issued before a store, the value loaded contains the old bytes.
$T_{store1} < T_{store2}$	If store1 is issued before store2, the resulting value contains the bytes of store2.
$T_{store} = T_{load}$	If a load and store are issued in the same clock cycle, the result is UNDEFINED.
$T_{store1} = T_{store2}$	If two stores are issued in the same clock cycle, the resulting stored value is undefined.

The scale factor applied (1/2/4) in the scaled addressing modes is equal to the size of the item loaded or stored, i.e. 1 for a byte operation, 2 for a 16-bit operation and 4 for a 32-bit operation.

Table 3-7 lists the available load and store mnemonics for the three addressing modes.

Table 3-7. Load and store mnemonics

Operation	Displacement	Index	Scaled-Index
8-bit signed load	ild8d	ild8r	—
8-bit unsigned load	uld8d	uld8r	—
16-bit signed load	ild16d	ild16r	ild16x
16-bit unsigned load	uld16d	uld16r	uld16x
32-bit load	ld32d	ld32r	ld32x
8-bit store	st8d	—	—
16-bit store	st16d	—	—
32-bit store	st32d	—	—

Example usage of load and store operations:

```
IF r10 ild16d(12) r12 → r13
```

If the LSB of r10 is set, load 16 bits starting at address (r12+12) using the byte ordering indicated in PCSW.BSX, sign-extend the value to 32 bits and store the result in r13.

```
IF r10 st32d(40) r12 r13
```

If the LSB of r10 is set, store the 32-bit value from r13 to the address (r12+40) using the byte ordering indicated in PCSW.BSX.

3.2.3 Compute Operations

Compute operations are register-to-register operations. The specified operation is performed on one or two source registers and the result is written to the destination register.

Immediate Operations. Immediate operations load an immediate constant (specified in the opcode) and produce a result in the destination register.

Floating-Point Compute Operations. Floating-point compute operations are register-to-register operations. The specified operation is performed on one or two source registers and the result is written to the destination register. Unless otherwise mentioned all floating point operations observe the rounding mode bits defined in the PCSW register. All floating-point operations not ending in 'flags' update the PCSW exception flags. All operations ending in 'flags' compute the exception flags as if the operation were executed and return the flag values (in the same format as in the PCSW); the exception flags in the PCSW itself remain unchanged.

Multimedia Operations. These special compute operations are like normal compute operations, but the specified operations are not usually found in general purpose CPUs. These operations provide special support for multimedia applications.

3.2.4 Special-Register Operations

Special register operations operate on the special registers: PCSW, DPC, SPC and CCCOUNT.

3.2.5 Control-Flow Operations

Control-flow operations change the value of the program counter. Conditional jumps test the value in a register and, based on this value, change the program counter to the address contained in a second register or continue execution with the next instruction. Unconditional jumps always change the program counter to the specified immediate address.

Control-flow operations can be interruptible or non-interruptible. Execution of an interruptible jump is the only occasion where TM1300 allows special event handling to take place (see Section 3.5, "Special Event Handling").

3.3 TM1300 INSTRUCTION ISSUE RULES

The TM1300 VLIW CPU allows issue of 5 operations in each clock cycle according to a set of specific issue rules. The issue rules impose issue time constraints and a result writeback constraint. Any set of operations that meets all constraints constitutes a legal TM1300 instruction. A more extensive description and a few special case issue rules and limitations can be found in the Philips TriMedia SDE documentation.

Issue time constraints:

- an operation implies a need for a functional unit type (as documented in Appendix A, "DSPCPU Operations for TM1300.")
- each operation requires an issue slot that has an instance of the appropriate functional unit type attached

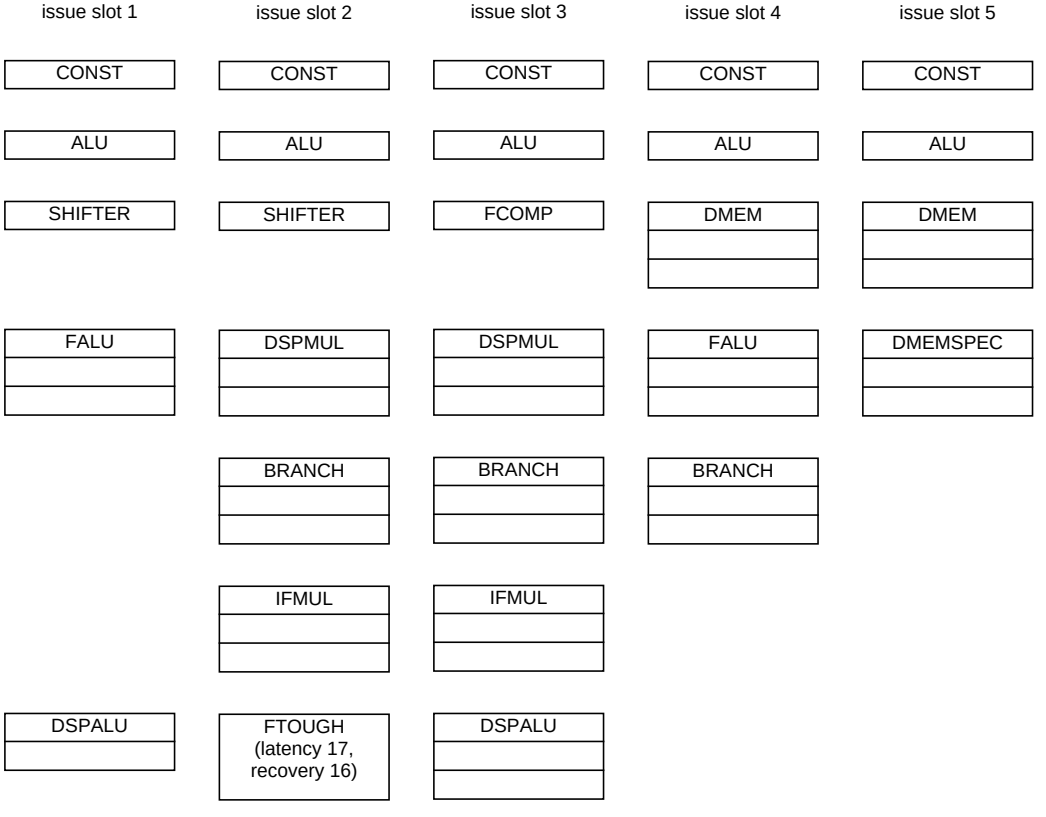


Figure 3-3. TM1300 issue slots, functional units, and latency.

- functional units should be ‘recovered’ from any prior operation issues

Writeback constraint:

- No more than 5 results should be simultaneously written to the register file at any point in time (write-back occurs ‘latency’ cycles after issue)

Figure 3-3 shows all functional units of TM1300, including the relation to issue slots, and each functional unit’s latency (e.g. 1 for CONST, 3 for FALU, etc.). With the exception of FTOUGH, each functional unit can accept an operation every clock cycle, i.e. has a recovery time of 1. The binding of operations to functional unit types is summarized in Table 3-8. In Appendix A, “DSPCPU Operations for TM1300”, each operation lists the precise functional unit and unit latency.

Table 3-8. Functional unit operations

unit type	operation category
const	immediate operations
alu	32-bit arithmetic, logical, pack/unpack
dspalu	dual 16-bit, quad 8-bit multimedia arithmetic
dspmul	dual 16-bit and quad 8-bit multimedia multiplies
dmem	loads/stores
dmemspec	cache coherency, cache control, prefetch
shifter	multi-bit shift
branch	control flow
falu	floating point arithmetic & conversions
ifmul	32-bit integer and floating point multiplies
fcomp	single cycle floating point compares
ftough	iterative floating point square root and division

3.4 MEMORY AND MMIO

TM1300 defines four apertures in its 32-bit address space: the memory hole, the DRAM aperture, the MMIO aperture and the PCI apertures (See Figure 3-4).The memory hole covers addresses 0..0xff. The DRAM and MMIO apertures are defined by the values in MMIO registers; the PCI apertures consist of every address that does not fall in the other three apertures.

3.4.1 Memory Map

DRAM is mapped into an aperture extending from the address in DRAM_BASE to the address in DRAM_LIMIT. The maximum DRAM aperture size is 64 MB.

The MMIO aperture is located at address MMIO_BASE and is a fixed 2-MB size.

In the default operating mode, all memory accesses not going to either the hole, DRAM or MMIO space are interpreted as PCI accesses. This behavior can be overridden as described in Section 5.3.8, “Memory Hole and PCI Aperture Disable.”

The MMIO aperture and the DRAM aperture can be at any naturally aligned location, in any order, but should not overlap; if they do, the consequences are undefined. The values of DRAM_BASE, DRAM_LIMIT, and MMIO_BASE are set during the boot process. In the case of a PCI host assisted boot, the values are determined by the host BIOS. In case of standalone boot (i.e., TM1300 is the PCI host), the values are taken from the boot ROM. Refer to Chapter 13, “System Boot” for details. DSPCPU update of DRAM_BASE and MMIO_BASE is possible, but not recommended, see Section 11.7.3, “MMIO/DRAM_BASE updates.”

3.4.2 The Memory Hole

The memory hole from address 0 to 0xff serves to protect the system from performance loss due to speculative loads. Due to the nature of C program references, most speculative loads issued by the DSPCPU fall in the range covered by the hole. Activated by default upon RESET, the hole serves to ensure that these speculative loads do NOT cause PCI read accesses and slow down the system. The value returned by any data load from the hole is 0. The hole only protects loads. Store operations in the hole do cause writes to PCI, SDRAM or MMIO as determined by the aperture base address values. If the SDRAM aperture overlaps the memory hole, the memory hole is ignored.

The hole can be temporarily disabled through the DC_LOCK_CTL register. This is described in Section 5.3.8, “Memory Hole and PCI Aperture Disable.”

3.4.3 MMIO Memory Map

Devices are controlled through memory-mapped device registers, referred to as MMIO registers. To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as ‘0’s. Some devices can autonomously access data memory (DMA) and most devices can cause CPU interrupts.

The 2-MB MMIO aperture is initially located at address 0xEFE00000 on RESET; it is relocated by the PCI BIOS

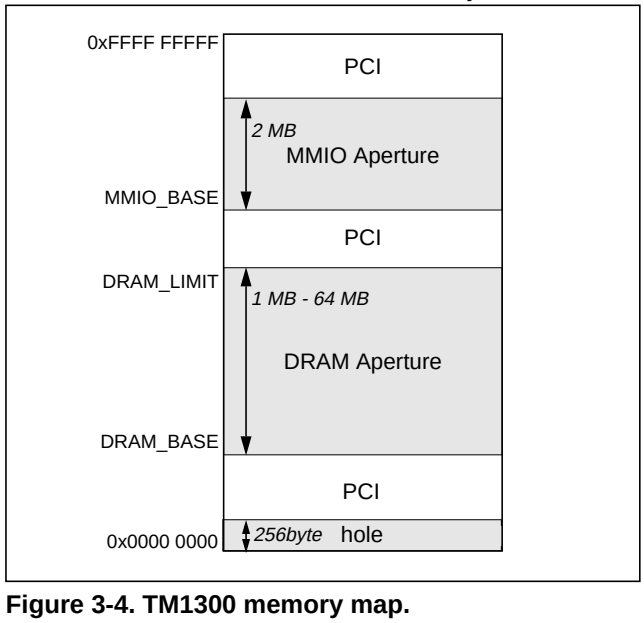


Figure 3-4. TM1300 memory map.

for PC-hosted TM1300 boards; its final location is determined by the boot EEPROM for standalone systems. See Chapter 13, “System Boot” for more information. Figure 3-5 gives a detailed overview of the MMIO memory map (addresses used are offsets with respect to the MMIO base). The operating system on TM1300 can change MMIO_BASE by writing to the MMIO_BASE MMIO location. User programs should not attempt this. Refer to the TriMedia SDE Reference Manual for the standard method to access the device registers from C language device drivers.

Only 32-bit load and store operations are allowed to access MMIO registers in the MMIO address aperture. The results are undefined for other loads and stores. Reads from non-existent MMIO registers return undefined values. Writes to nonexistent MMIO registers time out. There are no side effects of accesses to nonexistent MMIO registers. The state of the PCSW BSX bit has no effect on the result of MMIO accesses.

The Icache tag and LRU bit access aperture give the DSPCPU read-only access to the Icache status. Refer to Section 5.4.8, “Reading Tags and Cache Status” for details.

The EXCVEC MMIO location is explained in Section 3.5.2, “EXC (Exceptions).” Section 3.5.3, “INT and NMI (Maskable and Non-Maskable Interrupts),” describes the locations that deal with the setup and handling of interrupts: ISETTING, IPENDING, ICLEAR, IMASK and the interrupt vectors. The timer MMIO locations are described in Section 3.8, “Timers.” The instruction and data breakpoint are described in Section 3.9, “Debug Support.” The MMIO locations of each device are treated in the respective device chapters.

3.5 SPECIAL EVENT HANDLING

The TM1300 microprocessor responds to the special events shown in Table 3-9, ordered by priority.

With the exception of RESET, which is enabled at all times, the architecture of the DSPCPU allows special event handling to begin only during an *interruptible jump* operation (ijmpt, ijmpf or ijmpi) that succeeds (i.e., is a taken jump). EXC, NMI and INT handling can be initiated during handling of an EXC or an INT, but *only* during successful interruptible jumps.

Table 3-9. Special Events and Event Vectors

Event	Vector
RESET	(Highest priority) vector to DRAM_BASE
EXC	(All exceptions) vector to EXCVEC (programmable)
NMI, INT	(Non-maskable interrupt, maskable interrupt) use the programmed vector (one of 32 vectors depending on the interrupt source)

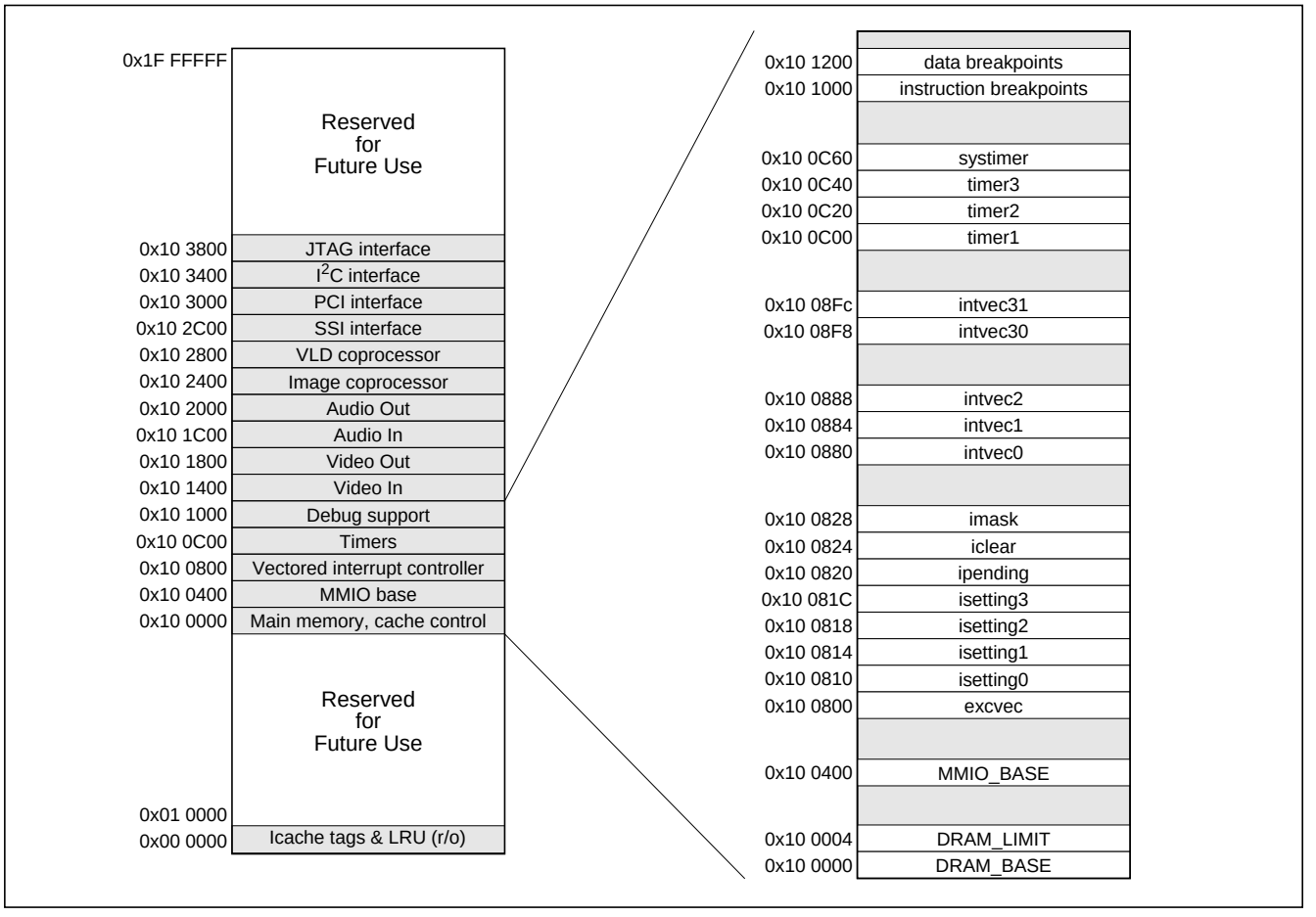


Figure 3-5. Memory map of MMIO address space (addresses are offset from MMIO_BASE).

The *instruction scheduler* uses interruptible jumps exclusively for inter-decision tree jumps. Hence, within a decision tree, no special-event processing can be initiated. If a tree-to-tree jump is taken, special-event processing is allowed. Since the only registers live at this point (i.e., that contain useful data) are the *global registers* allocated by the ANSI C compiler, only a subset of the registers needs to be preserved by the event handlers. Refer to the TriMedia SDE Reference Manual for details on which registers can be in use. The DSPCPU register state can be described by the contents of this subset of general purpose registers and the contents of the PCSW and the DPC value (the target of the inter-tree jump).

The priority resolution mechanism built into the DSPCPU hardware dispatches the highest-priority, non-masked special-event request at the time of a successful interruptible jump operation. In view of the simple, real-time-oriented nature of the mechanisms provided, only limited nesting of events should be allowed.

3.5.1 RESET

RESET is the highest priority special event. It is asserted by external hardware or by the host CPU. TM1300 will respond to it at any time.

External hardware reset through the TRI_RESET# pin initiates boot protocol execution as described in Chapter 13, “System Boot.” This causes the current PC value to be lost and instruction execution to start from address DRAM_BASE.

A PCI host CPU can perform a TM1300 DSPCPU-only reset by an MMIO write to the BIU_CTL.SR and CR bits. Such a reset does not cause a full boot, instead the DSPCPU resumes execution from DRAM_BASE.

3.5.2 EXC (Exceptions)

The DSPCPU enters EXC special-event processing under the following conditions:

1. RESET is de-asserted.
2. The intersection PCSW[15,6:0] & PCSW[31,22:16] is non-empty or PCSW.TFE is set.
3. A successful interruptible jump is in the final jump execution stage.

DSPCPU hardware takes the following actions on the initiation of EXC processing:

1. DPC is assigned the intended destination address of the successful jump.
2. Instruction processing starts at EXCVEC.

All other actions are the responsibility of the EXC handler software. Note that no other special event processing will take place until the handler decides to execute an interruptible jump that succeeds.

3.5.3 INT and NMI (Maskable and Non-Maskable Interrupts)

The on-chip Vectored Interrupt Controller (VIC) provides 32 INT request input hardware lines. The interrupt controller prioritizes and maps attention requests from several different peripherals onto successive INT requests to the DSPCPU.

INT special event processing will occur under the following conditions:

1. RESET is de-asserted.
2. The intersection PCSW[15,6:0] & PCSW[31,22:16] is empty and PCSW.TFE is not set.
3. The intersection of IPENDING and IMASK is non-empty.
4. The interrupt is at level NMI or PCSW.IEN = 1.
5. A successful interruptible jump is in the final jump execution stage.

DSPCPU hardware takes the following actions on the initiation of NMI or INT processing:

1. DPC gets assigned the intended destination address of the successful jump.
2. Instruction processing starts at the appropriate interrupt vector.

All other actions are the responsibility of the INT handler software. Note that no other special event processing will take place until the handler decides to execute an interruptible jump that succeeds.

3.5.3.1 Interrupt vectors

Each of the 32 interrupt sources can be assigned an arbitrary interrupt vector (the address of the first instruction of the interrupt handler). A vector is setup by writing the address to one of the MMIO locations shown in Figure 3-6. The state of the MMIO vector locations is undefined after RESET. (Addresses of the MMIO vector registers are offset with respect to MMIO_BASE.)

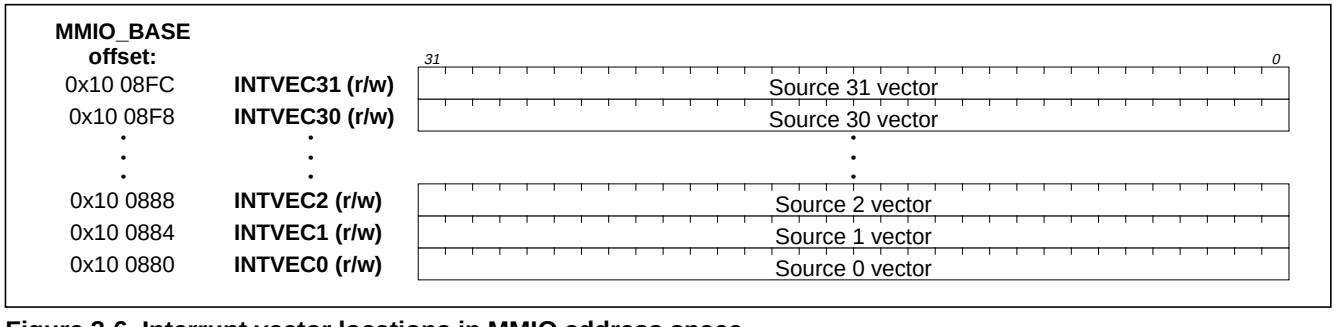


Figure 3-6. Interrupt vector locations in MMIO address space.

Programmer's note: See the *Philips TriMedia Cookbook* (Book 2 of TriMedia SDE documentation) for information on writing interrupt handlers.

3.5.3.2 Interrupt modes

DSPCPU interrupt sources can be programmed to operate in either *level-sensitive* or *edge-triggered* mode. Operation in edge-triggered or level-sensitive mode is determined by a bit in the ISETTING MMIO locations corresponding to the source, as defined in Figure 3-7. On RESET, all ISETTING registers are cleared.

In edge-triggered mode, the leading edge of the signal on the device interrupt request line causes the VIC (Vectored Interrupt Controller) to set the *interrupt pending* flag corresponding to the device source number. Note that, for active high signals, the leading edge is the positive edge, whereas for active low request signals (such as PCI INTA#), the negative edge is the leading edge. The interrupt remains pending until one of two events occurs:

- The VIC successfully dispatches the vector corresponding to the source to the TM1300 CPU, or
- TM1300 CPU software clears the interrupt-pending flag by a direct write to the ICLEAR location.

No interrupt acknowledge to ICLEAR is needed for devices operating in edge-triggered mode, since the vector dispatch clears the IPENDING request. The device itself may however need a device-specific interrupt acknowledge to clear the requesting condition. Edge-triggered mode is *not recommended* for devices that can signal multiple simultaneous interrupt conditions. The on-chip timers *must* be operated in edge triggered mode.

In level-sensitive mode, the device requests an interrupt by asserting the VIC source request line. The device holds the request until the device interrupt handler performs a device interrupt acknowledge. It is *highly recommended* that all off-chip and on-chip sources, with the exception of the timers, operate in level-sensitive mode.

3.5.3.3 Device interrupt acknowledge

All devices capable of generating level-triggered interrupts have interrupt acknowledge bits in their memory mapped control registers for this purpose. An interrupt acknowledge is performed by a store to such control register, with a '1' in the bit position(s) corresponding to the desired acknowledge flags.

Programmers note: the store operation that performs the interrupt acknowledge should be issued at least 2 cycles before the (interruptible) jump that ends an interrupt handler. This ensures that the same interrupt is not dispatched twice due to request de-assertion clock delays.

3.5.3.4 Interrupt priorities

Each interrupt source can be programmed to request one out of eight levels of priorities. The highest priority level (level 7) corresponds to requesting an NMI—an interrupt that cannot be masked by the DSPCPU PCSW.IEN bit. The other levels request regular interrupts, that can be masked as a group by the PCSW.IEN flag. Level six represents the highest priority normal interrupt level and level zero represents the lowest. Refer to Figure 3-7 for details of programming the priority level.

The VIC arbitrates the highest-priority pending interrupt requestor. Sources programmed to request at the same level are treated with a fixed priority, from source number 0 (highest) to 31 (lowest). At such time as the DSPCPU is willing to process special events, the vector of highest priority NMI source will be dispatched. If no NMI is pending, and the DSPCPU allows regular interrupts (PCSW.IEN is asserted), the vector of the highest priority regular source is dispatched. Once a vector is dispatched, the corresponding interrupt pending flag is deasserted (edge triggered mode sources only).

3.5.3.5 Interrupt masking

A single MMIO register (IMASK in Figure 3-8) allows masking of an arbitrary subset of the interrupt sources. Masking applies to both regular as well as NMI level requestors. Masking is used by software to disable unused devices and/or to implement nested interrupt handling. In the latter case, each interrupt handler can stack the old IMASK content for later restoration and insert a new mask that only allows the interrupts it is willing to handle. For level-triggered device handlers, IMASK should also exclude the device itself to prevent repeated handler activation.

Each interrupt source device typically has its own interrupt enable flag(s) that determine whether certain key

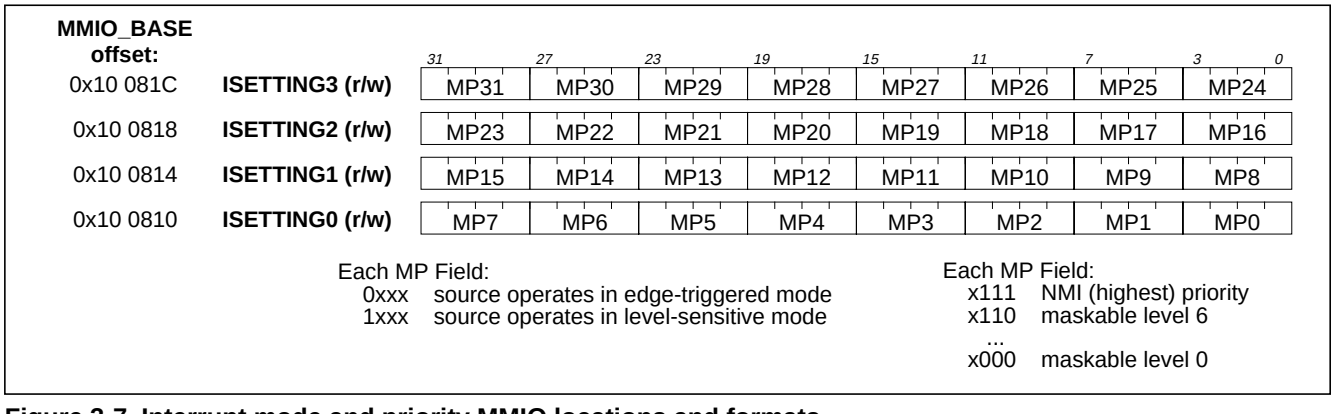


Figure 3-7. Interrupt mode and priority MMIO locations and formats.

device events lead to the request of an interrupt. In addition, the PCSW.IEN flag determines whether the DSPCPU is willing to handle regular interrupts. Non maskable interrupts ignore the state of this flag.

All three mechanisms are necessary: the PCSW.IEN flag is used to implement critical sections of code during which the RTOS (real-time operating system) is unable to handle regular interrupts. The IMASK is used to allow full control over interrupt handler nesting. The device interrupt flags set the operational mode of the device.

When RESET is asserted, IPENDING, ICLEAR, and IMASK are set to all zeroes. (MMIO register addresses shown in Figure 3-8 are offset addresses with respect to MMIO_BASE.)

3.5.3.6 Software interrupts and acknowledgment

The IPENDING register shown in Figure 3-8 can be read to observe the currently pending interrupts. Each bit read depends on the mode of the source:

- For a level-sensitive source, a bit value corresponds to the current state of the device interrupt request line.
- For an edge-triggered interrupt, a '1' is read if and only if an interrupt request occurred and the corresponding vector has not yet been dispatched.

Software can request an interrupt for sources operating in edge-triggered mode. Writes to the IPENDING register assert an interrupt request for all sources where a 1 occurred in the bit position of the written value. The state of sources where a 0 occurred in the written value is unchanged. Writes have no effect on level-sensitive mode sources. The interrupt request, if not masked, will occur at the next successful interruptible jump. This differs from the conventional software interrupt-like semantics of many architectures. Any of the 32 sources can be requested in software. In normal operation however, software-requested interrupts should be limited to source vectors not allocated for hardware devices. Note that another PCI master can request interrupts by manipulating the IPENDING location in the MMIO aperture. This is useful for inter-processor communication.

The ICLEAR register reads the same as the IPENDING register. Writes to the ICLEAR register serve to clear pending flags for edge-triggered mode sources. All IPENDING flags corresponding to bit positions in which '1's are written are cleared. IPENDING flags corresponding to bit positions in which '0's are written are not affected. Writes have no effect on level-sensitive mode sources. When a pending interrupt bit is being cleared through a write to the ICLEAR register at the same time that the hardware is trying to set that interrupt bit, the hardware takes precedence.

3.5.3.7 NMI sequentialization

In most applications, it is desirable not to nest NMIs. The NMI interrupt handler can accomplish this by saving the old IMASK content and clearing IMASK before the first interruptible jump is executed by the NMI handler.

3.5.3.8 Interrupt source assignment

Table 3-10 shows the assignment of devices to interrupt source numbers, as well as the recommended operating mode (edge or level triggered). Note that there are a total of 5 external pins available to assert interrupt requests. The PCI INTA to INTD requests are asserted by active low signal conventions, i.e. a zero level or a negative edge asserts a request. The USERIRQ pin operates with active high signalling conventions.

3.6 TM1300 TO HOST INTERRUPTS

In systems where TM1300 is operating in the presence of a host CPU on PCI, TM1300 can generate interrupts to the host, using any combination of the four PCI INTA# to INTD# pins. In a typical host system, only one of these pins needs to be wired to the PCI bus interrupt request lines. Any unused pins of this group are then available for use as software programmable I/O pins.

The INT_CTL register (see Figure 3-9) IEx bits, when set, enable the open collector driver of the four INTD#..INTA# pins. The INTx bits determine the output value generated (if enabled). A '1' in INTx causes the corresponding PCI interrupt pin to be asserted (low INTx# pin). The ISx bits are read-only and reflect the cur-

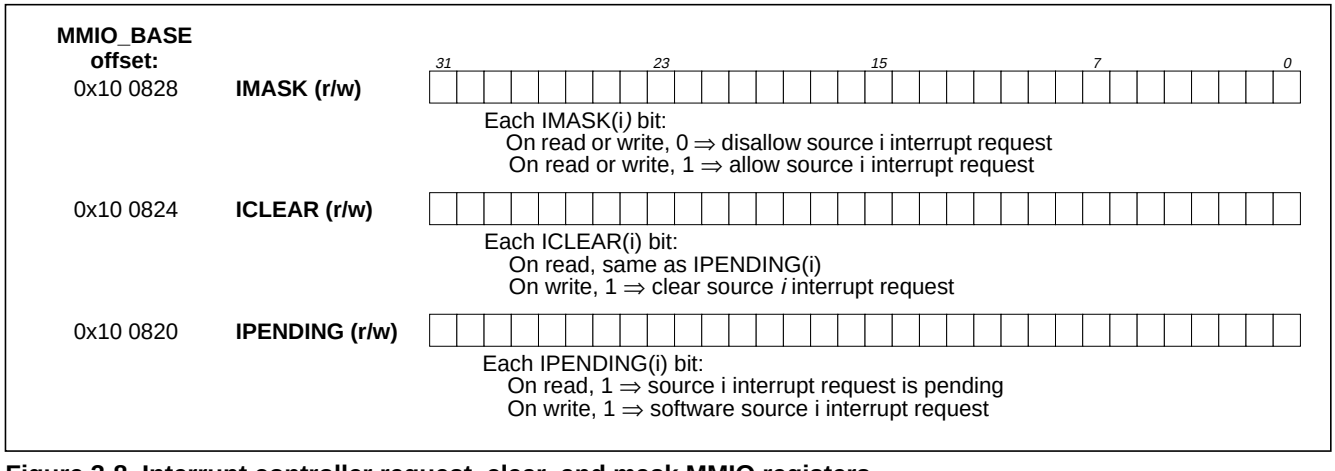


Figure 3-8. Interrupt controller request, clear, and mask MMIO registers.

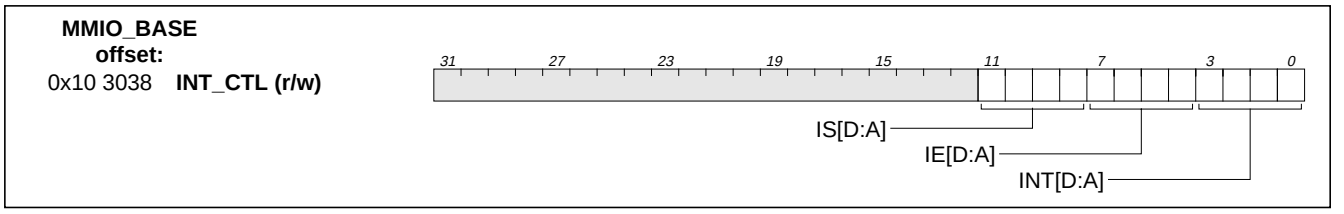


Figure 3-9. Host interrupt control register

Table 3-10. Interrupt source assignments

SOURCE NAME	SRC NUM	MODE	SOURCE DESCRIPTION
PCI INTA	0	level	PCI_INTA# pin signal
PCI INTB	1	level	PCI_INTB# pin signal
PCI INTC	2	level	PCI_INTC# pin signal
PCI INTD	3	level	PCI_INTD# pin signal
TRI_USERIRQ	4	either	external general-purpose pin
TIMER1	5	edge	general-purpose timer
TIMER2	6	edge	general-purpose timer
TIMER3	7	edge	general-purpose timer
SYSTIMER	8	edge	reserved for debugger
VIDEOIN	9	level	video in block
VIDEOOUT	10	level	video out block
AUDIOIN	11	level	audio in block
AUDIOOUT	12	level	audio out block
ICP	13	level	image coprocessor
VLD	14	level	VLD coprocessor
SSI	15	level	SSI interface
PCI	16	level	PCI BIU (DMA, etc.; see Table 11-14 for possible interrupt causes)
IIC	17	level	I^2C interface
JTAG	18	level	JTAG interface
t.b.d.	19..24		reserved for future devices
SPDO	25	level	SPDO block
t.b.d.	26..27		reserved for future devices
HOSTCOM	28	edge	(software) host communication
APP	29	edge	(software) application
DEBUGGER	30	edge	(software) debugger
RTOS	31	edge	(software) RTOS

rent actual state of the pins. Note that the pins have negative logic (active low) polarity and are of the open collector output type. Hence the pin voltage is low (active) when the logical value set or seen in the INT_CTL register is a ‘1’.

The assertion and de-assertion of host interrupts is the responsibility of TM1300 software.

See also Section 11.7.17, “INT_CTL Register.”

3.7 HOST TO TM1300 INTERRUPTS

A host CPU can generate an interrupt to TM1300 in several ways:

- by a PCI MMIO write to IPENDING to assert the HOSTCOMM interrupt (bit 28)
- by a hardware circuit that asserts one of the interrupt request pins TRI_USERIRQ, or INTA..INTD.

The first and most common method requires no circuitry and leaves the interrupt pins available for other purposes.

3.8 TIMERS

The DSPCPU contains four programmable timer/counters, all with the same function. The first three (TIMER1, TIMER2, TIMER3) are intended for general use. The fourth timer/counter (SYSTIMER) is reserved for use by the system software and should not be used by applications.

Each timer has three registers as shown in Figure 3-10. The MMIO register addresses shown are offset addresses with respect to the timer’s base address.

Each timer/counter can be set to count one of the event types specified in Table 3-12. Note that the DATABREAK event is special, in that the timer/counter may increment by zero, one or two in each clock cycle. For all other event types, increments are by zero or one. The CACHE1 and CACHE2 events serve as cache performance monitoring support. The actual event selected for CACHE1 and CACHE2 is determined by the MEM_EVENTS MMIO register, see Section 5.7, “Performance Evaluation Support.” If a TM1300 pin signal (VICLK, etc.) is selected as an event, positive-going edges on the signal are counted.

Each timer increments its value until the modulus is reached. On the clock cycle where the incremented value would equal or exceed the modulus, the value wraps around to zero or one (in the case of an increment by two), and an interrupt is generated as defined in Table 3-10. The timer interrupt source mode should be set as edge-sensitive. No software interrupt acknowledge to the timer device is necessary.

Counting starts and continues as long as the run bit is set.

Loading a new modulus does not affect the contents of the value register. If a store operation to either the modulus or value register results in value and modulus being the same, no interrupt will be generated. If the run bit is set, the next value will be modulus+1 or modulus+2, and

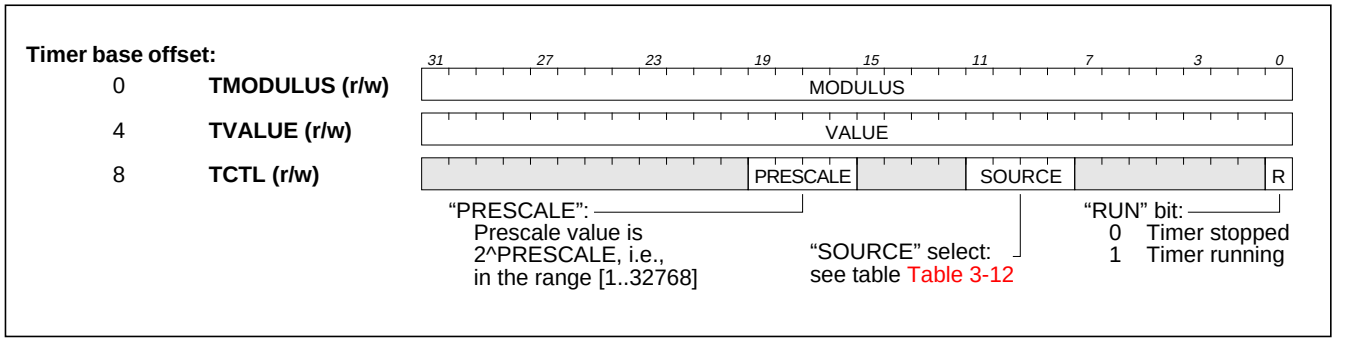


Figure 3-10. Timer register definitions.

Table 3-11. Timer base MMIO address

TIMER1	MMIO_BASE+0x10,0C00
TIMER2	MMIO_BASE+0x10,0C20
TIMER3	MMIO_BASE+0x10,0C40
SYSTIMER	MMIO_BASE+0x10,0C60

Table 3-12. Timer source selections

Source Name	Source Bits Value	Source Description
CLOCK	0	CPU clock
PRESCALE	1	prescaled CPU clock
TRI_TIMER_CLK	2	external clock pin
DATABREAK	3	data breakpoints
INSTBREAK	4	instruction breakpoints
CACHE1	5	cache event 1
CACHE2	6	cache event 2
VI_CLK	7	video in clock pin
VO_CLK	8	video out clock pin
AI_WS	9	audio in word strobe pin
AO_WS	10	audio out word strobe pin
SSI_RXFSX	11	SSI receive frame sync pin
SSI_IO2	12	SSI transmit frame sync pin
—	13-15	undefined

the counter will have to loop around before an interrupt is generated.

A modulus value of zero causes a wrap-around as if the modulus value was 2^{32}.

On RESET, the TCTL registers are cleared, and the value of the TMODULUS and TVALUE registers is undefined.

3.9 DEBUG SUPPORT

This section describes the special debug support offered by the DSPCPU. Instruction and data breakpoints can be defined through a set of registers in the MMIO register space. When a breakpoint is matched, an event is generated that can be used as a timer source (see Section 3.8, "Timers"). The timer TMODULUS has to be set to generate a DSPCPU interrupt after the desired number of breakpoint matches.

3.9.1 Instruction Breakpoints

The instruction-breakpoint control register is shown in Figure 3-11. On RESET, the BICTL register is cleared. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

The instruction-breakpoint address-range registers are shown in Figure 3-12. After RESET, the value of these registers is undefined. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

When the IC bit in the breakpoint control register is set to '1', instruction breakpoints are activated. Any instruction address issued by the TM1300 chip is compared against the low and high address-range values. The IAC bit in the breakpoint control register determines whether the instruction address needs to be inside or outside of the range defined by the low and high address-range registers. A successful comparison takes place when either:

- IAC = '0' and low ≤ iaddr ≤ high, or
- IAC = '1' and iaddr < low or iaddr > high.

On a successful comparison, an instruction breakpoint event is generated, which can be used as a clock input to a timer. After counting the programmed number of instruction breakpoint events, the timer will generate an interrupt request.

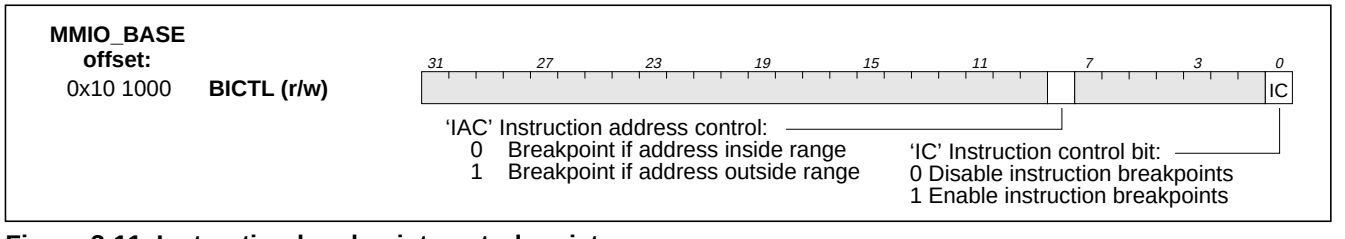


Figure 3-11. Instruction-breakpoint control register.

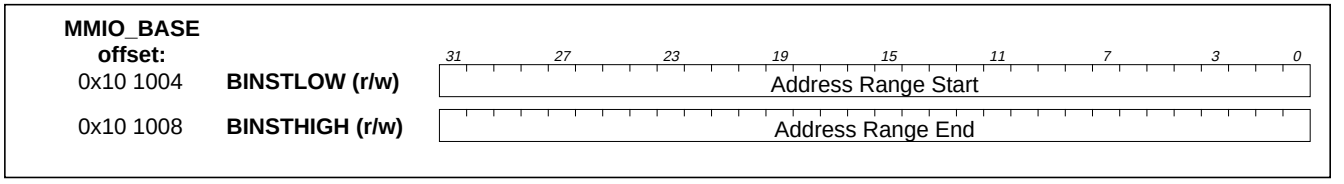


Figure 3-12. Instruction-breakpoint address-range registers.

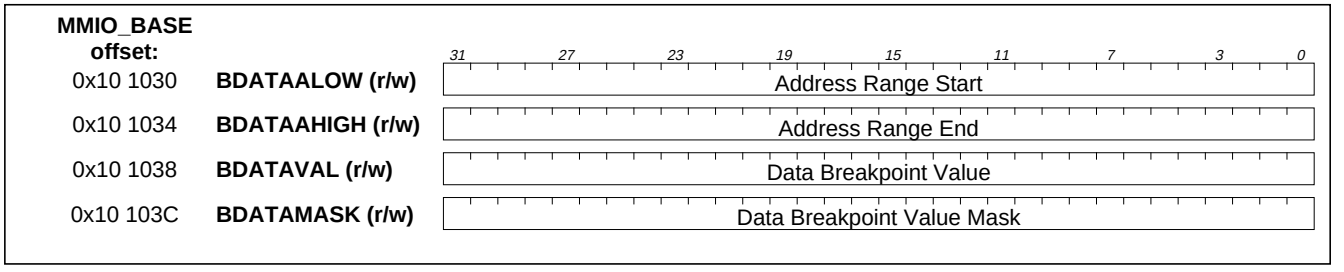


Figure 3-13. Data-breakpoint address-range and value-compare registers.

3.9.2 Data Breakpoints

The data-breakpoint address-range and compare-value registers are shown in Figure 3-13. After RESET, the value of the data breakpoint registers is undefined. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

The data-breakpoint control register is shown in Figure 3-14. On RESET, the BDCTL register is cleared. (The register address shown is offset with respect to MMIO_BASE.)

When the DC bits in the data breakpoint control register are not set to ‘0’, data breakpoints are activated. When the value of the DC bits is ‘1’ or ‘3’, any data address from load operations (if the BL bit is set) and/or store operations (if the BS bit is set) issued by the DSPCPU is compared against the low and high address-range values. The DAC bit in the breakpoint control register determines whether data addresses need to be inside or outside of the range defined by the low and high address-range registers. A successful comparison occurs when either:

- DAC = ‘0’ and low ≤ daddr ≤ high, or
- DAC = ‘1’ and daddr < low or daddr > high.

Note that this comparison works for all addresses regardless of the aperture to which they belong. When the value of the DC bits is ‘2’ or ‘3’, any data value from load operations (if the BL bit is set) and/or store operations (if the BS bit is set) issued by the TM1300 CPU is compared against the value in the BDATAVAL register. Only the bits for which the corresponding BDATAMASK register bits are set to ‘1’ will be used in the comparison. The DVC bit in the breakpoint control register determines whether the data value needs to be equal or not equal to the comparison value. A successful comparison occurs when either of the following are true:

- DVC = ‘0’ and (data & BDATAMASK) = (BDATAVAL & BDATAMASK).
- DVC = ‘1’ and (data & BDATAMASK) != (BDATAVAL & BDATAMASK).

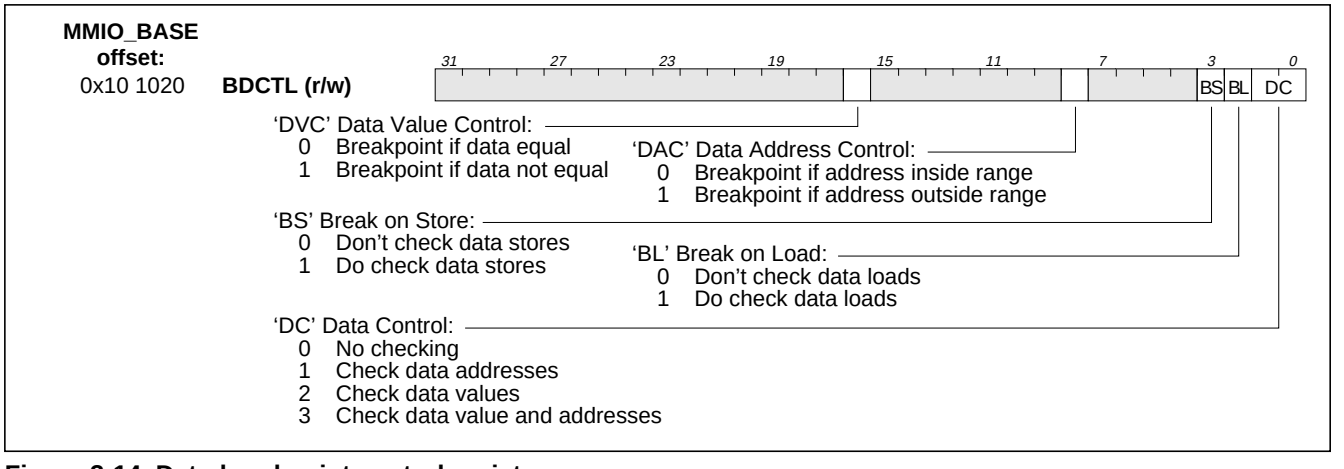


Figure 3-14. Data-breakpoint control register.

Note: use a nonzero datamask or the result is undefined.

When a successful comparison has taken place, a data breakpoint event is generated, which can be used as a clock input to a timer. After counting the set number of data breakpoint events, the timer will generate an interrupt request.

When the value of the DC bits is ‘3’, a data breakpoint event is generated if and only if a successful comparison occurs on both address and data simultaneously.

Note that up to two data breakpoint events can occur per clock cycle, due to the dual load/store capability of the CPU and data cache.

Custom Operations for Multimedia Chapter 4

by Gert Slavenburg, Pieter v.d. Meulen, Yong Cho, Sang-Ju Park

4.1 CUSTOM OPERATIONS OVERVIEW

Custom operations in the TM1300 DSPCPU architecture are specialized, high-function operations designed to dramatically improve performance in important multimedia applications. When properly incorporated into application source code, custom operations enable an application to take advantage of the highly parallel TM1300 microprocessor implementation. Achieving a similar performance increase through other means—e.g., executing a higher number of traditional microprocessor instructions per cycle—would be prohibitively expensive for TM1300's low-cost target applications.

Custom operations are simple to understand and consistent in their definition, but their unusual functions make it difficult for automatic code generation algorithms to use them effectively. Consequently, custom operations are inserted into source code by the programmer. To make this process as painless as possible, custom operation syntax is consistent with the C programming language, and, just as with all other operations generated by the compiler, the scheduler takes care of register allocation, operation packing, and flow analysis.

4.1.1 Custom Operation Motivation

For both general-purpose and embedded microprocessor-based applications, programming in a high-level language is desirable. To effectively support optimizing compilers and a simple programming model, certain microprocessor architecture features are needed, such as a large, linear address space, general-purpose registers, and register-to-register operations that directly support the manipulation of linear address pointers. A common choice in microprocessor architectures is 32-bit linear addresses, 32-bit registers, and 32-bit integer operations. TM1300 is such a microprocessor architecture.

For the data manipulation in many algorithms, however, 32-bit data and operations are wasteful of expensive silicon resources. Important multimedia applications, such as the decompression of MPEG video streams, spend significant amounts of execution time dealing with eight-bit data items. Using 32-bit operations to manipulate small data items makes inefficient use of 32-bit execution hardware in the implementation. If these 32-bit resources could be used instead to operate on four eight-bit data items simultaneously, performance would be improved by a significant factor with only a tiny increase in implementation cost.

Getting the highest execution rate from standard microprocessor resources is one of the motivations behind custom operations in TM1300. A range of custom operations is provided that each processes—simultaneously—four 8-bit or two 16-bit data items. There is little cost difference between a standard 32-bit ALU and one that can process either one pair of 32-bit operands or four pairs of eight-bit operands, but there is a big performance difference for TM1300's target applications.

TM1300's custom operations go beyond simply making the best use of standard resources. Some custom operations combine several simple operations. These combinations are tailored specifically to the needs of important multimedia applications. Some high-function custom operations eliminate conditional branches, which helps the scheduler make effective use of all five operation slots in each TM1300 instruction. Filling up all five slots is especially important in the inner loops of computational intensive multimedia applications.

In short, custom operations help TM1300 reach its goals of extremely high multimedia performance at the lowest possible cost.

4.1.2 Introduction to Custom Operations

Table 4-1 and Table 4-2 contain two listings of the custom operations available in the TM1300 architecture. Table 4-1 groups the custom operations by type of function while Table 4-2 lists the operations by operand size. For more detailed information about the custom operations, Appendix A, "DSPCPU Operations for TM1300."

Some operations exist in several versions that differ in the treatment of their operands and results, and the mnemonics for these versions make it easy to select the appropriate operation. For example, the sum of products operations all have "fir" in their mnemonics; the prefix and suffix of the mnemonic expresses the treatment of the operands and result. The ifir8ii operation treats both of its operands as signed (ifir8ii) and produces a signed result (ifir8ii). The ifir8iu operation treats its first operand as signed (ifir8iu), the second as unsigned (ifir8iu), and produces a signed result (ifir8iu). The ume8ii operation implements an eight-bit motion-estimation; it treats both operands as signed but produces an unsigned result.

The operations beginning with "dsp" implement a clipping (sometimes called saturating) function before storing the result(s) in the destination register. Otherwise, their naming follows the rules given above where appropriate. For example, the dspuquadaddui operation implements four 8-bit additions; it treats the first operand of

Table 4-1. Key Multimedia Custom Operations Listed by Function Type

Function	Custom Op	Description
DSP absolute value	dspiabs	Clipped signed 32-bit absolute value
	dspidualabs	Dual clipped absolute values of signed 16-bit halfwords
Shift	dualasr	dual-16 arithmetic shift right
Clip	dualiclipi	dual-16 clip signed to signed
	dualuclipi	dual-16 clip signed to unsigned
Min,max	quadumax	Unsigned bytewise quad max
	quadumin	Unsigned bytewise quad min
DSP add	dspiadd	Clipped signed 32-bit add
	dspuadd	Clipped unsigned 32-bit add
	dspidualadd	Dual clipped add of signed 16-bit halfwords
	dspuquadaddui	Quad clipped add of unsigned/ signed bytes
DSP multiply	dspimul	Clipped signed 32-bit multiply
	dspumul	Clipped unsigned 32-bit multiply
	dspidualmul	Dual clipped multiply of signed 16-bit halfwords
DSP subtract	dspisub	Clipped signed 32-bit subtract
	dspusub	Clipped unsigned 32-bit subtract
	dspidualsub	Dual clipped subtract of signed 16-bit halfwords
Sum of products	ifir16	Signed sum of products of signed 16-bit halfwords
	ifir8ii	Signed sum of products of signed bytes
	ifir8iu	Signed sum of products of signed/unsigned bytes
	ufir16	Unsigned sum of products of unsigned 16-bit halfwords
	ufir8uu	Unsigned sum of products of unsigned bytes
Merge, pack	mergedual16lsb	Merge dual-16 least-significant bytes
	mergelsb	Merge least-significant bytes
	mergemsb	Merge most-significant bytes
	pack16lsb	Pack least-significant 16-bit halfwords
	pack16msb	Pack most-significant 16-bit halfwords
	packbytes	Pack least-significant bytes
Byte averages	quadavg	Unsigned byte-wise quad average
Byte multiplies	quadumulmsb	Unsigned quad 8-bit multiply most significant
Motion estimation	ume8ii	Unsigned sum of absolute values of signed 8-bit differences
	ume8uu	Unsigned sum of absolute values of unsigned 8-bit differences

each addition as unsigned, the second operand as signed, and produces an unsigned result for each addition. Each result, which is computed with no loss of precision, is clipped into the representable range of a byte (0..255).

Table 4-2. Key Multimedia Custom Operations Listed by Operand Size

Op. Size	Custom Op	Description
32-bit	dspiabs	Clipped signed 32-bit abs value
	dspiadd	Clipped signed 32-bit add
	dspuadd	Clipped unsigned 32-bit add
	dspimul	Clipped signed 32-bit multiply
	dspumul	Clipped unsigned 32-bit multiply
	dspisub	Clipped signed 32-bit subtract
	dspusub	Clipped unsigned 32-bit subtract
16-bit	mergedual16lsb	Merge dual-16 least-significant bytes
	dualasr	dual-16 arithmetic shift right
	dualiclipi	dual-16 clip signed to signed
	dualuclipi	dual-16 clip signed to unsigned
	dspidualmul	Dual clipped multiply of signed 16-bit halfwords
	dspidualabs	Dual clipped absolute values of signed 16-bit halfwords
	dspidualadd	Dual clipped add of signed 16-bit halfwords
	dspidualsub	Dual clipped subtract of signed 16-bit halfwords
	ifir16	Signed sum of products of signed 16-bit halfwords
	ufir16	Unsigned sum of products of unsigned 16-bit halfwords
	pack16lsb	Pack least-significant 16-bit halfwords
	pack16msb	Pack most-significant 16-bit halfwords

Table 4-2. Key Multimedia Custom Operations Listed by Operand Size

Op. Size	Custom Op	Description
8-bit	quadumax	Unsigned bytewise quad max
	quadumin	Unsigned bytewise quad min
	dspuquadaddui	Quad clipped add of unsigned/ signed bytes
	ifir8ii	Signed sum of products of signed bytes
	ifir8iu	Signed sum of products of signed/unsigned bytes
	ufir8uu	Unsigned sum of products of unsigned bytes
	mergelsb	Merge least-significant bytes
	mergemsb	Merge most-significant bytes
	packbytes	Pack least-significant bytes
	quadavg	Unsigned byte-wise quad average
	quadumulmsb	Unsigned quad 8-bit multiply most significant
	ume8ii	Unsigned sum of absolute values of signed 8-bit differences
	ume8uu	Unsigned sum of absolute values of unsigned 8-bit differences

4.1.3 Example Uses of Custom Ops

The next three sections illustrate the advantages of using custom operations. Also, the more complex examples illustrate how custom operations can be integrated into application code by providing listings of C-language program fragments. The examples progress in complexity from simple to intricate; the most interesting examples are taken from actual multimedia codes, such as MPEG decompression.

4.2 EXAMPLE 1: BYTE-MATRIX TRANSPOSITION

The goal of this example is to provide a simple, introductory illustration of how custom operations can significantly increase processing speed in small kernels of applications. As in most uses of custom operations, the power of custom operations in this case comes from their ability to operate on multiple data items in parallel.

Imagine that our task is to transpose a packed, 4-by-4 matrix of bytes in memory; the matrix might, for example, contain 8-bit pixel values. Figure 4-1 illustrates both the organization of the matrix in memory and the task to be performed in standard mathematical notation.

Performing this operation with traditional microprocessor instructions is straight forward but time consuming. One way to perform the manipulation is to perform 12 load-byte instructions (since only 12 of the 16 bytes need to be repositioned) and 12 store-byte instructions that place the bytes back in memory in their new positions. Another way would be to perform four load-word instructions, reposition the bytes in registers, and then perform four store-word instructions. Unfortunately, repositioning the bytes in registers would require a large number of instructions to properly shift and mask the bytes. Performing the 24 loads and stores makes implicit use of the shifting and masking hardware in the load/store units and thus yields a shorter instruction sequence.

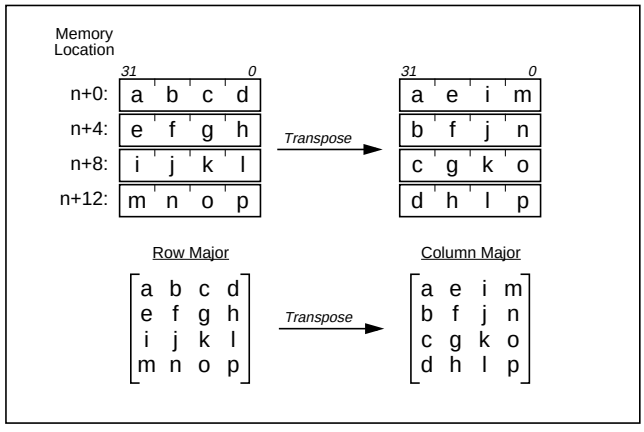


Figure 4-1. Byte-matrix transposition. Top shows byte matrices packed into memory words; bottom shows mathematical matrix representation.

The problem with performing 24 loads and stores is that loads and stores are inherently slow operations because they must access at least the cache and possibly slower layers in the memory hierarchy. Further, performing byte loads and stores when 32-bit word-wide accesses run just as fast wastes the power of the cache/memory interface. We would prefer a fast algorithm that takes full advantage of cache/memory bandwidth while not requiring an inordinate number of byte-manipulation instructions.

TM1300 has instructions that merge and pack bytes and 16-bit halfwords directly and in parallel. Four of these instructions can be applied in this case to speed up the manipulation of bytes that are packed into words.

Figure 4-2 shows the application of these instructions to the byte-matrix transposition problem, and the left side of Figure 4-3 shows a list of the operations needed to implement the matrix transpose. When assembled into actual TM1300 instructions, these custom operations would be packed as tightly as dependencies allow, up to five operations per instruction.

Note that a programmer would not need to program at this level (TM1300 assembler). The matrix transpose would be expressed just as efficiently in C-language source code, as shown on the right side of Figure 4-3. The low-level code is shown here for illustration purposes only.

The first sequence of four load-word operations in Figure 4-3 brings the packed words of the input matrix into registers R10, R11, R12, and R13. The next sequence of four merge operations produces intermediate results into registers R14, R15, R16, and R17. The next sequence of four pack operations could then replace the original operands or place the transposed matrix in separate registers if the original matrix operands were need-

```
ld32d(0) r100 → r10
ld32d(4) r100 → r11
ld32d(8) r100 → r12
ld32d(12) r100 → r13

mergemsb r10 r11 → r14
mergemsb r12 r13 → r15
mergelsb r10 r11 → r16
mergelsb r12 r13 → r17
pack16msb r14 r15 → r18
pack16lsb r14 r15 → r19
pack16msb r16 r17 → r20
pack16lsb r16 r17 → r21

st32d(0) r101 r18
st32d(4) r101 r19
st32d(8) r101 r20
st32d(12) r101 r21
```

```
char matrix[4][4];
        .
        .
        .
int *m = (int *) matrix;

temp0 = MERGEMSB(m[0], m[1]);
temp1 = MERGEMSB(m[2], m[3]);
temp2 = MERGELSB(m[0], m[1]);
temp3 = MERGELSB(m[2], m[3]);
m[0]  = PACK16MSB(temp0, temp1);
m[1]  = PACK16LSB(temp0, temp1);
m[2]  = PACK16MSB(temp2, temp3);
m[3]  = PACK16LSB(temp2, temp3);
        .
        .
        .
```

Figure 4-3. On the left is a complete list of operations to perform the byte-matrix transposition of Figure 4-1 and Figure 4-2. On the left is an equivalent C-language fragment.

ed for further computations (the TM1300 optimizing C compiler performs this analysis automatically). In this example, the transpose matrix is placed in registers R18, R19, R20, and R21. The final four store-word operations put the transposed matrix back into memory.

Thus, using the TM1300 custom operations, the byte-matrix transposition requires four load-word operations and four store-word operations (the minimum possible) and eight register-to-register data-manipulation operations. The result is 16 operations, or byte-matrix transposition at the rate of one operation per byte.

While the advantage of the custom-operation-based algorithm over the brute-force code that uses 24 load- and store-byte instruction seems to be only eight operations (a 33% reduction), the advantage is actually much greater. First, using custom operations, the number of memory references is reduced from 24 to eight (a factor of three). Since memory references are slower than register-to-register operations (such as the custom operations in this example), the reduction in memory references is significant.

Further, the ability of the TM1300 VLIW compilation system to exploit the performance potential of the TM1300 microprocessor hardware is enhanced by the custom-operation-based code. This is because it is easier for the compilation system to produce an optimal schedule (arrangement) of the code when the number of memory references is in balance with the number of register-to-register operations. The TM1300 CPU (like all high-performance microprocessors) has a limit on the number of memory references that can be processed in a single cycle (two is the current limit). A long sequence of code that contains only memory references can result in empty operation slots in the long TM1300 instructions. Empty operation slots waste the performance potential of the TM1300 hardware.

As this example has shown, careful use of custom operations has the potential to not only reduce the absolute number of operations needed to perform a computation but can also help the compilation system produce code that fully exploits the performance potential of the TM1300 CPU.

4.3 EXAMPLE 2: MPEG IMAGE RECONSTRUCTION

The complete MPEG video decoding algorithm is composed of many different phases, each with computational intensive kernels. One important kernel deals with reconstructing a single image frame given that the forward- and backward-predicted frames and the inverse discrete cosine transform (IDCT) results have already been computed. This kernel provides an excellent opportunity to illustrate of the power of TM1300's specialized custom operators.

In the code fragments that follow, the backward-predicted block is assumed to have been computed into an array back[], the forward-predicted block is assumed to have been computed into forward[], and the IDCT results are assumed to have been computed into idct[].

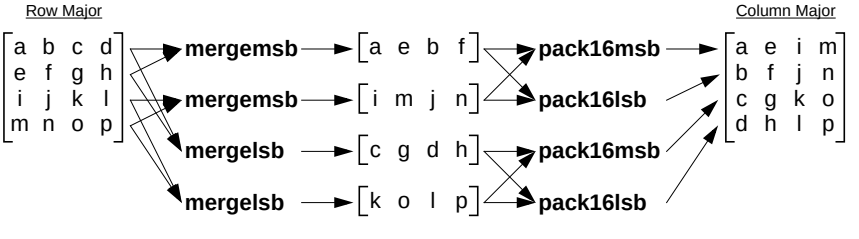


Figure 4-2. Application of merge and pack instructions to the byte-matrix transposition of Figure 4-1.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp;

    for (i = 0; i < 64; i += 1)
    {
        temp = ((back[i] + forward[i] + 1) >> 1) + idct[i];

        if (temp > 255)
            temp = 255;
        else if (temp < 0)
            temp = 0;

        destination[i] = temp;
    }
}
```

Figure 4-4. Straightforward code for MPEG frame reconstruction.

A straightforward coding of the reconstruction algorithm might look as shown in Figure 4-4. This implementation shares many of the undesirable properties of the first example of byte-matrix transposition. The code accesses memory a byte at a time instead of a word at a time, which wastes 75% of the available bandwidth. Also, in light of the many quad-byte-parallel operations introduced in Section 4.1.2, “Introduction to Custom Operations,” it seems inefficient to spend three separate additions and one shift to process a single eight-bit pixel. Perhaps even more unfortunate for a VLIW processor like TM1300 is the branch-intensive code that performs the saturation testing; eliminating these branches could reap a significant performance gain.

Since MPEG decoding is the kind of task for which TM1300 was created, there are two custom operations—quadavg and dspuquadaddui—that exactly fit this important MPEG kernel (and other kernels). These custom operations process four pairs of 8-bit pixel values in parallel. In addition, dspuquadaddui performs saturation tests in hardware, which eliminates any need to execute explicit tests and branches.

For readers familiar with the details of MPEG algorithms, the use of eight-bit IDCT values later in this example may be confusing. The standard MPEG implementation calls for nine-bit IDCT values, but extensive analysis has shown that values outside the range [–128..127] occur so rarely that they can be considered unimportant. Pursuant to this observation, the IDCT values are clipped into the eight-bit range [–128..127] with saturating arithmetic before the frame reconstruction code runs. The assumption that this saturation occurs permits some of TM1300’s custom operations to have clean, simple definitions.

The first step in seeing how custom operations can be of value in this case, is to unroll the loop by a factor of four. The unrolled code is shown in Figure 4-5. This creates code that is parallel with respect to the four pixel computations. As it is easily seen in the code, the four groups of computations (one group per pixel) do not depend on each other.

After some experience is gained with custom operations, it is not necessary to unroll loops to discover situations where custom operations are useful. Often, a good programmer with knowledge of the function of the custom operations can see by simple inspection opportunities to exploit custom operations.

To understand how quadavg and dspuquadaddui can be used in this code, we examine the function of these custom operations.

The quadavg custom operation performs pixel averaging on four pairs of pixels in parallel. Formally, the operation of quadavg is as follows:

```
quadavg rscr1 rsrc2 -> rdest
```

takes arguments in registers rsrc1 and rsrc2, and it computes a result into register rdest. rsrc1 = [abcd], rsrc2 = [wxyz], and rdest = [pqrs] where a, b, c, d, w, x, y, z, p, q, r, and s are all unsigned eight-bit values. Then, quadavg computes the output vector [pqrs] as follows:

```
p = (a + w + 1) >> 1
q = (b + x + 1) >> 1
r = (c + y + 1) >> 1
s = (d + z + 1) >> 1
```

The pixel averaging in Figure 4-5 is evident in the first statement of each of the four groups of statements. The rest of the code—adding idct[i] value and performing the saturation test—can be performed by the dspuquadaddui operation. Formally, its function is as follows:

```
dspuquadaddui rsrc1 rsrc2 -> rdest
```

takes arguments in registers rsrc1 and rsrc2, and it computes a result into register rdest. rsrc1 = [efgh], rsrc2 = [stuv], and rdest = [ijkl] where e, f, g, h, i, j, k, and l are unsigned 8-bit values; s, t, u, and v are signed 8-bit values. Then, dspuquadaddui computes the output vector [ijkl] as follows:

```
i = uclipi(e + s, 255)
j = uclipi(f + t, 255)
k = uclipi(g + u, 255)
l = uclipi(h + v, 255)
```

The uclipi operation is defined in this case as it is for the separate TM1300 operation of the same name described in Appendix A, “DSPCPU Operations for TM1300,”. Its definition is as follows:

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp;

    for (i = 0; i < 64; i += 4)
    {
        temp = ((back[i+0] + forward[i+0] + 1) >> 1) + idct[i+0];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+0] = temp;

        temp = ((back[i+1] + forward[i+1] + 1) >> 1) + idct[i+1];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+1] = temp;

        temp = ((back[i+2] + forward[i+2] + 1) >> 1) + idct[i+2];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+2] = temp;

        temp = ((back[i+3] + forward[i+3] + 1) >> 1) + idct[i+3];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+3] = temp;
    }
}
```

Figure 4-5. MPEG frame reconstruction code using TM1300 custom operations; compare with Figure 4-4.

```
uclipi (m, n)
{
    if (m < 0) return 0;
    else if (m > n) return n;
    else return m;
}
```

To make is easier to see how these operations can subsume all the code in Figure 4-5, Figure 4-6 shows the same code rearranged to group the related functions. Now it should be clear that the quadavg operation can replace the first four lines of the loop assuming that we can get the individual 8-bit elements of the back[] and forward[] arrays positioned correctly into the bytes of a 32-bit word. That, of course, is easy: simply align the byte arrays on word boundaries and access them with word (integer) pointers.

Similarly, it should now be clear that the dspuquadaddui operation can replace the remaining code (except, of course, for storing the result into the destination[] array) assuming, as above, that the 8-bit elements are aligned and packed into 32-bit words.

Figure 4-7 shows the new code. The arrays are now accessed in 32-bit (int-sized) chunks, the loop iteration control has been modified to reflect the ‘four-at-a-time’ operations, and the quadavg and dspuquadaddui operations have replaced the bulk of the loop code. Finally, Figure 4-8 shows a more compact expression of the loop code, eliminating the temporary variable. Note that TM100 C compiler does the optimization by itself.

Again, note that the code in Figure 4-7 and Figure 4-8 assumes that the character arrays are 32-bit word aligned and padded if necessary to fill an integral number of 32-bit words.

The original code required three additions, one shift, two tests, three loads, and one store per pixel. The new code using custom operations requires only two custom operations, three loads, and one store for *four* pixels, which is more than a factor of six improvement. The actual performance improvement can be even greater depending on how well the compiler is able to deal with the branches in the original version of the code, which depends in part on the surrounding code. Reducing the number of branches almost always improves the chances of realizing maximum performance on the TM1300 CPU.

The code in Figure 4-8 illustrates several aspects of using custom operations in C-language source code. First, the custom operations require no special declarations or syntax; they appear to be simple function calls. Second, there is no need to explicitly specify register assignments for sources, destinations, and intermediate results; the compiler and scheduler assign registers for custom operations just as they would for built-in language operations such as integer addition. Third, the scheduler packs custom operations into TM1300 VLIW instructions as effectively as it packs operations generated by the compiler for native language constructs.

Thus, although the burden of making effective use of custom operations falls on the programmer, that burden consists only of discovering the opportunities for exploiting the operations and then coding them using standard C-language notation. The compiler and scheduler take care of the rest.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp0, temp1, temp2, temp3;

    for (i = 0; i < 64; i += 4)
    {
        temp0 = ((back[i+0] + forward[i+0] + 1) >> 1);
        temp1 = ((back[i+1] + forward[i+1] + 1) >> 1);
        temp2 = ((back[i+2] + forward[i+2] + 1) >> 1);
        temp3 = ((back[i+3] + forward[i+3] + 1) >> 1);

        temp0 += idct[i+0];
        if (temp0 > 255) temp0 = 255;
        else if (temp0 < 0) temp0 = 0;

        temp1 += idct[i+1];
        if (temp1 > 255) temp1 = 255;
        else if (temp1 < 0) temp1 = 0;

        temp2 += idct[i+2];
        if (temp2 > 255) temp2 = 255;
        else if (temp2 < 0) temp2 = 0;

        temp3 += idct[i+3];
        if (temp3 > 255) temp3 = 255;
        else if (temp3 < 0) temp3 = 0;

        destination[i+0] = temp0;
        destination[i+1] = temp1;
        destination[i+2] = temp2;
        destination[i+3] = temp3;
    }
}
```

Figure 4-6. Re-grouped code of Figure 4-5.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp;

    int *i_back    = (int *) back;
    int *i_forward = (int *) forward;
    int *i_idct    = (int *) idct;
    int *i_dest    = (int *) destination;

    for (i = 0; i < 16; i += 1)
    {
        temp = QUADAVG(i_back[i], i_forward[i]);

        temp = DSPUQUADADDUI(temp, i_idct[i]);

        i_dest[i] = temp;
    }
}
```

Figure 4-7. Using the custom operation dspquadaddui to speed up the loop of Figure 4-6.

4.4 EXAMPLE 3: MOTION-ESTIMATION KERNEL

Another part of the MPEG coding algorithm is motion estimation. The purpose of motion estimation is to reduce the cost of storing a frame of video by expressing the contents of the frame in terms of adjacent frames. A given frame is reduced to small blocks, and a subsequent frame is represented by specifying how these small blocks change position and appearance; usually, storing the difference information is cheaper than storing a whole block. For example, in a video sequence where the camera pans across a static scene, some frames can be expressed simply as displaced versions of their predecessor frames. To create a subsequent frame, most blocks are simply displaced relative to the output screen.

The code in this example is for a match-cost calculation, a small kernel of the complete motion-estimation code. As with the previous example, this code provides an excellent example of how to transform source code to make the best use of TM1300's custom operations.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i;

    int *i_back    = (int *) back;
    int *i_forward = (int *) forward;
    int *i_idct    = (int *) idct;
    int *i_dest    = (int *) destination;

    for (i = 0; i < 16; i += 1)
        i_dest[i] = DSPUQUADADDUI(QUADAVG(i_back[i], i_forward[i]), i_idct[i]);
}
```

Figure 4-8. Final version of the frame-reconstruction code.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 1)
        cost += abs(A[row][col] – B[row][col]);
}
```

Figure 4-9. Match-cost loop for MPEG motion estimation.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 4)
    {
        cost += abs(A[row][col+0] – B[row][col+0]);
        cost += abs(A[row][col+1] – B[row][col+1]);
        cost += abs(A[row][col+2] – B[row][col+2]);
        cost += abs(A[row][col+3] – B[row][col+3]);
```

Figure 4-10. Unrolled, but not parallel, version of the loop from Figure 4-9.

Figure 4-9 shows the original source code for the match-cost loop. Unlike the previous example, the code is not a self-contained function. Somewhere early in the code, the arrays A[][] and B[][] are declared; somewhere between those declarations and the loop of interest, the arrays are filled with data.

4.4.1 A Simple Transformation

First, we will look at the simplest way to use a TM1300 custom operation.

We start by noticing that the computation in the loop of Figure 4-9 involves the absolute value of the difference of two unsigned characters (bytes). By now, we are familiar with the fact that TM1300 includes a number of operations that process all four bytes in a 32-bit word simultaneously. Since the match-cost calculation is fundamental to the MPEG algorithm, it is not surprising to find a custom operation—ume8uu—that implements this operation exactly.

To understand how ume8uu can be used in this case, we need to transform the code as in the previous example. Though the steps are presented here in detail, a programmer with a even a little experience can often perform these transformations by visual inspection.

To use a custom operation that processes 4 pixel values simultaneously, we first need to create 4 parallel pixel computations. Figure 4-10 shows the loop of Figure 4-9 unrolled by a factor of 4. Unfortunately, the code in the unrolled loop is not parallel because each line depends on the one above it. Figure 4-11 shows a more parallel version of the code from Figure 4-10. By simply giving each computation its own cost variable and then summing the costs all at once, each cost computation is completely independent.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 4)
    {
        cost0 = abs(A[row][col+0] - B[row][col+0]);
        cost1 = abs(A[row][col+1] - B[row][col+1]);
        cost2 = abs(A[row][col+2] - B[row][col+2]);
        cost3 = abs(A[row][col+3] - B[row][col+3]);

        cost += cost0 + cost1 + cost2 + cost3;
```

Figure 4-11. Parallel version of Figure 4-10.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
unsigned char *CA = A;
unsigned char *CB = B;

for (row = 0; row < 16; row += 1)
{
    int rowoffset = row * 16;

    for (col = 0; col < 16; col += 4)
    {
        cost0 = abs(CA[rowoffset + col+0] - CB[rowoffset + col+0]);
        cost1 = abs(CA[rowoffset + col+1] - CB[rowoffset + col+1]);
        cost2 = abs(CA[rowoffset + col+2] - CB[rowoffset + col+2]);
        cost3 = abs(CA[rowoffset + col+3] - CB[rowoffset + col+3]);

        cost += cost0 + cost1 + cost2 + cost3;
```

Figure 4-13. The loop of Figure 4-11 recoded with one-dimensional array accesses.

Excluding the array accesses, the loop body in Figure 4-11 is now recognizable as the function performed by the ume8uu custom operation: the sum of 4 absolute values of 4 differences. To use the ume8uu operation, however, the code must access the arrays with 32-bit word pointers instead of with 8-bit byte pointers.

Figure 4-13 shows the loop recoded to access A[][] and B[][] as one-dimensional instead of two-dimensional arrays. We take advantage of our knowledge of C-language array storage conventions to perform this code transformation. Recoding to use one-dimensional arrays prepares the code for transformation to 32-bit array accesses.

(From here on, until the final code is shown, the declarations of the A and B arrays will be omitted from the code fragments for the sake of brevity.)

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 1)
    cost += UME8UU(IA[i], IB[i]);
```

Figure 4-12. The loop of Figure 4-14 with the inner loop eliminated.

Figure 4-14 shows the loop of Figure 4-13 recoded to use ume8uu. Once again taking advantage of our knowledge of the C-language array storage conventions, the one-dimensional byte array is now accessed as a one-dimensional 32-bit-word array. The declarations of the pointers IA and IB as pointers to integers is the key, but also notice that the multiplier in the expression for row offset has been scaled from 16 to 4 to account for the fact that there are 4 bytes in a 32-bit word.

Of course, since we are now using one-dimensional arrays to access the pixel data, it is natural to use a single for loop instead of two. Figure 4-12 shows this streamlined version of the code without the inner loop. Since C-language arrays are stored as a linear vector of values, we can simply increase the number of iterations of the outer loop from 16 to 64 to traverse the entire array.

The recoding and use of the ume8uu operation has resulted in a substantial improvement in the performance of the match-cost loop. In the original version, the code executed 1280 operations (including loads, adds, subtracts, and absolute values); in the restructured version, there are only 256 operations—128 loads, 64 ume8uu operations, and 64 additions. This is a factor of five reduction in the number of operations executed. Also, the

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (row = 0; row < 16; row += 1)
{
    int rowoffset = row * 4;

    for (col4 = 0; col4 < 4; col4 += 1)
        cost += UME8UU(IA[rowoffset + col4], IB[rowoffset + col4]);
}
```

Figure 4-14. The loop of Figure 4-13 recoded with 32-bit array accesses and the ume8uu custom operation.

overhead of the inner loop has been eliminated, further increasing the performance advantage.

4.4.2 More Unrolling

The code transformations of the previous section achieved impressive performance improvements, but given the VLIW nature of the TM1300 CPU, more can be done to exploit TM1300's parallelism.

The code in Figure 4-12 has a loop containing only 4 operations (excluding loop overhead). Since TM1300's branches have a 3-instruction delay and each instruction can contain up to 5 operations, a fully utilized minimum-sized loop can contain 16 operations (20 minus loop overhead).

The TM1300 compilation system performs a wide variety of powerful code transformation and scheduling optimizations to ensure that the VLIW capabilities of the CPU are exploited. It is still wise, however, to make program parallelism explicit in source code when possible. Explicit parallelism can only help the compiler produce a fast running program.

To this end, we can unroll the loop of Figure 4-12 some number of times to create explicit parallelism and help the compiler create a fast running loop. In this case, where the number of iterations is a power-of-two, it makes sense to unroll by a factor that is a power-of-two to create clean code.

Figure 4-15 shows the loop unrolled by a factor of eight. The compiler can apply common sub-expression elimination and other optimizations to eliminate extraneous operations in the array indexing, but, again, improvements in the source code can only help the compiler produce the best possible code and fastest-running program.

Figure 4-16 shows one way to modify the code for simpler array indexing.

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 8)
{
    cost0 = UME8UU(IA[i+0], IB[i+0]);
    cost1 = UME8UU(IA[i+1], IB[i+1]);
    cost2 = UME8UU(IA[i+2], IB[i+2]);
    cost3 = UME8UU(IA[i+3], IB[i+3]);
    cost4 = UME8UU(IA[i+4], IB[i+4]);
    cost5 = UME8UU(IA[i+5], IB[i+5]);
    cost6 = UME8UU(IA[i+6], IB[i+6]);
    cost7 = UME8UU(IA[i+7], IB[i+7]);

    cost += cost0 + cost1 + cost2 +
            cost3 + cost4 + cost5 +
            cost6 + cost7;
}
```

Figure 4-15. Unrolled version of Figure 4-12. This code makes good use of TM1300's VLIW capabilities.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 8, IA += 8, IB +=
8)
{
    cost0 = UME8UU(IA[0], IB[0]);
    cost1 = UME8UU(IA[1], IB[1]);
    cost2 = UME8UU(IA[2], IB[2]);
    cost3 = UME8UU(IA[3], IB[3]);
    cost4 = UME8UU(IA[4], IB[4]);
    cost5 = UME8UU(IA[5], IB[5]);
    cost6 = UME8UU(IA[6], IB[6]);
    cost7 = UME8UU(IA[7], IB[7]);

    cost += cost0 + cost1 + cost2 +
            cost3 + cost4 + cost5 +
            cost6 + cost7;
}
```

Figure 4-16. Code from Figure 4-15 with simplified array index calculations.

Cache Architecture Chapter 5

by Eino Jacobs

5.1 MEMORY SYSTEM OVERVIEW

The high-performance video and audio throughput of TM1300 is implemented by its DSPCPU and autonomous I/O and co-processing units, but the foundation of this processing is the TM1300 memory hierarchy. To get the full potential of the chip's processing units, the memory hierarchy must read and write data (and DSP CPU instructions) fast enough to keep the units busy.

To meet the requirements of its target applications, TM1300's memory hierarchy must satisfy the conflicting goals of low cost, simple system design (e.g., low parts count), and high performance. Since multimedia video streams can require relatively large temporary storage, a significant amount of external DRAM is required. Minimizing the cost of bulk memory is important.

TM1300's memory system achieves a good compromise between cost and performance by coupling substantial on-chip caches with a glueless interface to synchronous DRAM (SDRAM). SDRAM provides higher bandwidth than standard DRAM for only a small cost premium. A block diagram of the memory system is shown in Figure 5-1. SDRAM permits TM1300 to use a narrower and simpler interface than would be required to achieve similar performance with standard DRAM.

The separate on-chip data and instruction caches serve only the DSPCPU since the data access patterns of the autonomous I/O and graphics units exhibit little or no locality of reference (they access each piece of the multimedia data stream only once in each operation).

Without the caches, the CPU would not be able to achieve its performance potential. SDRAM has enough bandwidth to handle serial streams of multimedia data, but its bandwidth and latency are insufficient to satisfy the CPU's high rate of random data accesses and repeated instruction accesses.

Table 5-1. 100-MHz TM1300 memory bandwidth parameters

Magnitude	Use
2800 MB/s	Instruction bandwidth (224 bits/instruction)
800 MB/s	Data bandwidth (two 32-bit memory ports)
400 MB/s	Main-memory bandwidth (one 32-bit port)

Table 5-1 shows bandwidth parameters for the TM1300 DSPCPU and the main-memory interface. Although 400 MB/s is a lot of bandwidth, it is clear that the SDRAM alone cannot keep up with the CPU's maximum requirements for instructions and data. Luckily, multimedia algorithms resemble other computer programs in terms of locality of reference, so the on-chip caches typically supply the majority of instructions and data to the DSPCPU. The wide paths to the caches are matched to the bandwidth requirements of the DSPCPU.

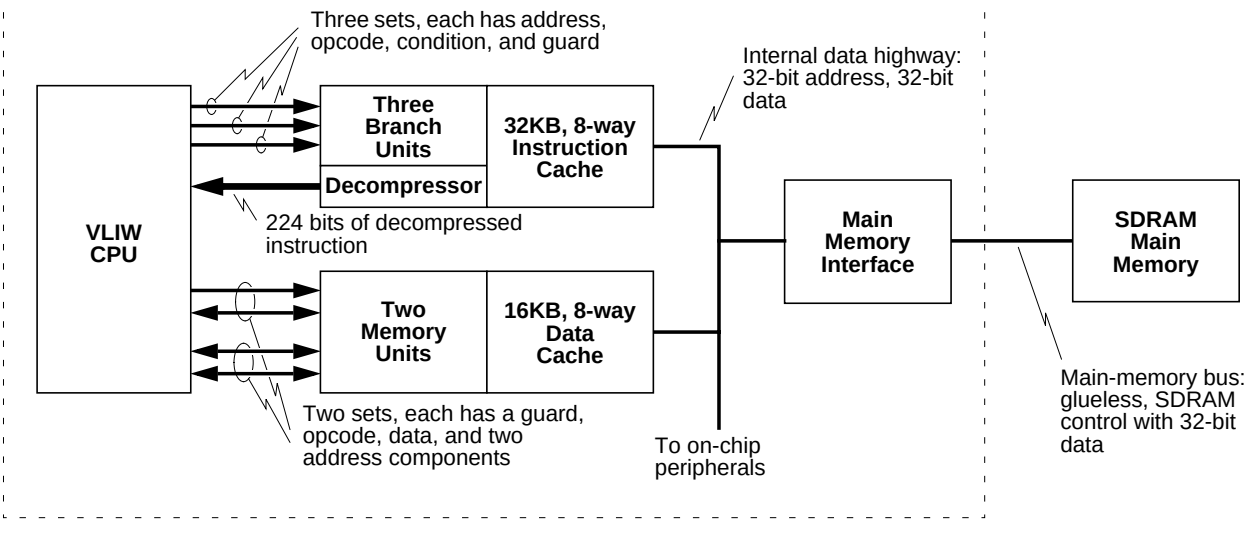


Figure 5-1. The main components of the TM1300 memory system.

Table 5-2. Summary of memory system characteristics

Unit	Description
Branch units	Branch units execute branch operations. Up to three branch operations can be executed in parallel, but the program must guarantee that only one branch is taken.
Decompression unit	Instructions are stored in memory and in the instruction cache in a space-saving, compressed format. The decompression unit expands instructions to their full, 28-byte size before they are issued to the CPU.
Instruction cache	The instruction cache holds 32 KB, is 8-way set-associative, and has a 64-byte block size. A miss in a block causes the entire block to be read from SDRAM. The cache can sustain an issue rate of one instruction per cycle on cache hits.
Memory units	Memory units execute load and store operations. The data cache is dual ported to allow the memory units to operate concurrently.
Data cache	The data cache holds 16 KB, is 8-way set-associative, has a 64-byte block size, and implements a copyback, allocate-on-write policy. A miss in a block causes the entire block to be read from SDRAM. The cache supports memory-mapped I/O through non-cacheable address regions.
Data highway	The on-chip data highway bus serves all on-chip units. The highway has separate 32-bit data and address buses. Bus bandwidth is allocated by the highway arbiter according to one of several modes.
Main-memory interface	The main-memory interface contains the data-highway access arbiter, the SDRAM controller, and MMIO logic.
SDRAM main memory	External SDRAM connects gluelessly to TM1300 over the 32-bit main-memory bus.

To improve cache behavior and thus program performance, the caches have a locking mechanism. In addition, the instruction cache is coupled with an instruction decompression unit. The compressed instruction format improves the cache hit rate and reduces the bus bandwidth required between main memory and cache. Instructions in main memory and cache use the compressed format.

TM1300's processing units access the external SDRAM through the on-chip central "data highway" bus. The highway consists of separate 32-bit address and data buses, and use of the bus is mediated by the main-memory interface unit. The main-memory interface contains the SDRAM controller and a central arbiter that determines how much of the available SDRAM memory bandwidth is allocated to each unit. Unused bandwidth is always made available to the VLIW CPU for cache refill and memory accesses that bypass the caches.

Table 5-2 gives a summary description of each component of TM1300's memory system.

5.2 DRAM APERTURE

TM1300 implements a 32-bit linear address space of bytes. Within that address space, TM1300 supports several different apertures for specific purposes. The DRAM aperture describes the part of the address space into which the external SDRAM is mapped. SDRAM must consist of a single, contiguous region of memory, which is the most practical configuration for TM1300 systems.

The location and size of the DRAM aperture is defined by two registers, DRAM_BASE and DRAM_LIMIT. These registers are both readable and writeable as MMIO registers and as PCI configuration space registers. The view of the registers in MMIO space is shown in Figure 5-2. The view of the registers in PCI configuration space is described in Chapter 11, "PCI Interface." In normal operation, the base address registers are assigned once during boot and not changed when the DSPCPU is running. Refer to Chapter 11, "PCI Interface," and Chapter 13, "System Boot," for a description of this process.

DRAM_LIMIT must be set equal to DRAM_BASE plus the actual size of SDRAM present. The amount of the SDRAM is not required to be a power of 2, but it must be a multiple of 64 KB. Note that the size of the aperture as set in the PCI configuration space can be larger, because it must be a power of 2.

A memory operation will access SDRAM if its address satisfies:

$$[\text{DRAM\_BASE}] \le \text{address} < [\text{DRAM\_LIMIT}]$$

Any address outside this range cannot access SDRAM.

When TM1300 is reset, DRAM_BASE_FIELD is set to 0x0 and DRAM_LIMIT is set to 0x0010 0000 (1-MB DRAM aperture starting at address 0x0). The boot process described in Chapter 13, "System Boot," overrides these initial settings.

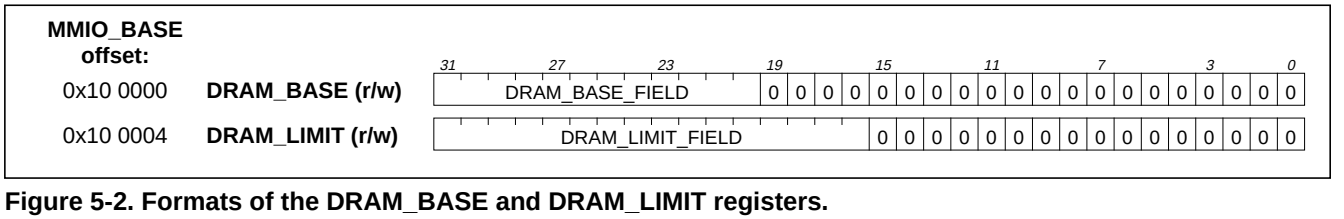


Figure 5-2. Formats of the DRAM_BASE and DRAM_LIMIT registers.

5.3 DATA CACHE

The data cache serves only the DSPCPU and is controlled by two memory units that execute the load and store operations issued by the DSPCPU. The following sections describe the data cache and its operation; Table 5-3 summarizes the important characteristics for easy reference.

Table 5-3. Summary of data cache characteristics

Characteristic	TM1300 Implementation
Cache size	16 KB
Cache associativity	8-way set-associative
Block size	64 bytes
Valid bits	One valid bit per 64-byte block
Dirty bits	One dirty bit per 64-byte block
Miss transfer order	Miss transfers begin with the critical word first
Replacement policies	Copyback, allocate on write, hierarchical LRU
Endianness	Either little- or big-endian, determined by PCSW bit
Ports	The cache is quasi dual ported; two accesses can proceed concurrently if they reference different banks (determined by bits [4:2] of the computed addresses)
Alignment	Access must be naturally aligned (32-bit words on 32-bit boundaries, 16-bit half-words on 16-bit boundaries); the appropriate number of LSBs of un-naturally aligned addresses are set to zero. For misaligned stores, PCSW.MSE is asserted to generate an exception
Partial word operations	The cache implements 8-bit and 16-bit accesses with the same performance as 32-bit accesses
Operation latency	Three cycles for both load and store operations
Coherency enforcement	Software uses special operations to enforce cache coherency
Cache locking	Up to 1/2 (four out of 8 blocks of each set) of the cache contents can be locked; granularity is 64-byte
Non-cacheable region	One non-cacheable aperture in the DRAM address space is supported.

5.3.1 General Cache Parameters

The TM1300 data cache is 16 KB in size with a 64-byte block size. Thus, it contains 256 blocks each with its own address tag. The cache is 8-way set-associative, so there are 32 sets, each containing 8 tags. A single valid bit is associated with a block, so each block and associated address tag is either entirely valid in the cache or invalid. On a cache miss, 64 bytes are read from SDRAM to make the entire block valid.

Each block also contains a dirty bit, which is set whenever a write to the block occurs. Each set contains 10 bits to support the hierarchical LRU replacement policy.

The geometry of the data cache is available to software by reading the MMIO register DC_PARAMS. Figure 5-3 shows the format of the DC_PARAMS register; Table 5-4 lists its field values. The product of block size, associativity, and number of sets gives the total cache size (16 KB in this case).

Table 5-4. DC_PARAMS field values

Field Name	Value
BLOCK SIZE	64
ASSOCIATIVITY	8
NUMBER_OF_SETS	32

5.3.2 Address Mapping

TM1300 data addresses are mapped onto the data cache storage structure as shown in Figure 5-4. A data address is partitioned into four fields as described in Table 5-5.

Table 5-5. Data address field partitioning

Field	Address Bits	Purpose
Byte	1..0	Byte offset within a word for byte or half-word accesses
Word	5..2	Selects one of the words in a set (one of 16 words in the case of TM1300)
Set	10..6	Selects one of the sets in the cache (one of 32 in the case of TM1300)
Tag	31..11	Compared against address tags of set members

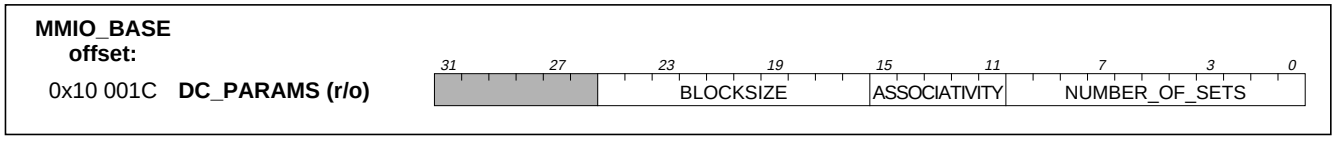


Figure 5-3. Format of the DC_PARAMS register.

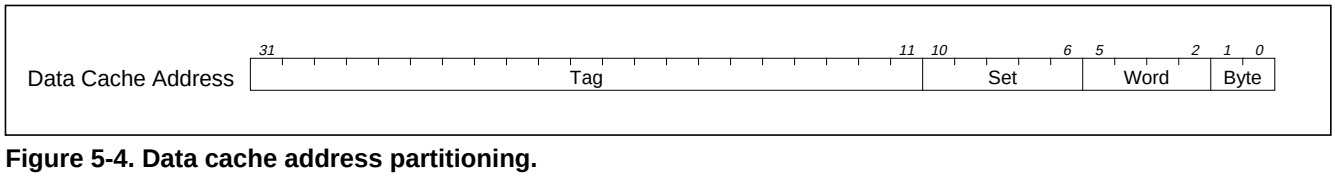


Figure 5-4. Data cache address partitioning.

5.3.3 Miss Processing Order

When a miss occurs, the data cache fills the block containing the requested word from the critical word first. The CPU is stalled until the first word is transferred. The block is then filled up while the CPU keeps running.

5.3.4 Replacement Policies, Coherency

The cache implements a copyback replacement policy with one dirty bit per 64-byte block. Thus, when a miss occurs and the block selected for replacement has its dirty bit set, the dirty block must be written to main memory to preserve its modified contents. On TM1300, the dirty block is written to memory before the needed block is fetched.

Coherency is not maintained in any way by hardware between the data cache, the instruction cache, and main memory. Special operations are available to implement cache coherency in software. See Section 5.6, “Cache Coherency,” for a discussion of coherency issues.

Write misses are handled with an allocate-on-write policy—the write that caused the miss stores its data in the cache after the missing block is fetched into the cache.

The cache implements a hierarchical LRU replacement algorithm to determine which of the eight elements (blocks) in a set is replaced. The algorithm partitions the eight set elements into four groups, each group with two elements. The hierarchical LRU replacement victim is determined by selecting the least-recently used group of two elements and then selecting the least-recently used element in that group. This hierarchical algorithm yields performance close to full LRU but is simpler to implement.

See Section 5.5, “LRU Algorithm,” for a full discussion of the LRU algorithm.

5.3.5 Alignment, Partial-Word Transfers, Endian-ness

The cache implements 32-bit word, 16-bit half-word, and 8-bit byte transfers. All transfers, however, must be to addresses that are naturally aligned; that is, 32-bit words must be aligned on 32-bit boundaries, and 16-bit half-words must be aligned on 16-bit boundaries.

Like other TM1300 processing units, the CPU has the capability to use either big- or little-endian byte order. It is recommended that all units and the CPU run with the same endian-ness. Detailed endian-ness description can be found in Appendix C, “Endian-ness.”

5.3.6 Dual Ports

To allow two accesses to proceed in parallel, the data cache is quasi-dual ported. The cache is implemented as eight banks of single-ported memory, but the hardware allows each bank to operate independently. Thus, when the addresses of two simultaneous accesses select two different banks, both accesses can complete simultaneously. Bank selection is determined by the three low-order address bits [4..2] of each address. Thus, the words in a 64-byte cache block are distributed among the eight blocks, which prevents conflicts between two simultaneously issued accesses to adjacent words in a cache block. The TM1300 compiling system attempts to avoid bank conflicts as much as possible.

The dual-ported cache can execute the load and store opcodes (ild8d, uld8d, ild16d, uld16d, ld32d, h_st8d, h_st16d, h_st32d, ild8r, uld8r, ild16r, uld16r, ld32r, ild16x, uld16x, ld32x) in either or both of the two ports.

The special opcodes alloc, dcb, dinvalid, pref, rdtag and rdstatus can only be executed in the second port, not in the first port. Whenever any of these special opcodes is issued in the second port, there should not be a concurrent load or store operation in the first. This is a special scheduling constraint.

5.3.7 Cache Locking

The data cache allows the contents of up to one-half of its blocks to be locked. Thus, on TM1300, up to 8 KB of the cache can be used as a high-speed local data memory. Only four out of eight blocks in any set can be locked.

A locked block is never chosen as a victim by the replacement algorithm; its contents remain undisturbed until either (1) the block’s locked status is changed explicitly by software, or (2) a dinvalid operation is executed that targets the locked block.

Cache locking occurs only for the data in the address range described by the MMIO registers DC_LOCK_ADDR and DC_LOCK_SIZE. The granularity of the address range is one 64-byte cache block. The MMIO register DC_LOCK_CTL contains the cache-locking enable bit DC_LOCK_ENABLE. Figure 5-5 shows the layout of the data-cache lock registers. Locking will occur for an address if locking is enabled and both of the following are true:

1. The address is greater than or equal to the value in DC_LOCK_ADDR.
2. The address is less than the sum of the values in DC_LOCK_ADDR and DC_LOCK_SIZE.

Programmers (or compilers) must combine all data that needs to be locked into this single linear address range.

Setting DC_LOCK_ENABLE to ‘1’ causes the following sequence of events:

1. All blocks that are in cache locations that will be used for locking are copied back to main memory (if they are dirty) and removed from the cache.
2. All blocks in the lock range are fetched from main memory into the cache. If any block in the lock range was already in the cache, it’s first copied back into main memory (if it’s dirty) and invalidated.
3. The LRU status of any set that contains locked blocks is set to the initialization value.
4. Cache locking is activated so that the locked blocks cannot be victims of the replacement algorithm.

This sequence of events is triggered by writing ‘1’ to DC_LOCK_ENABLE even if the enable is already set to

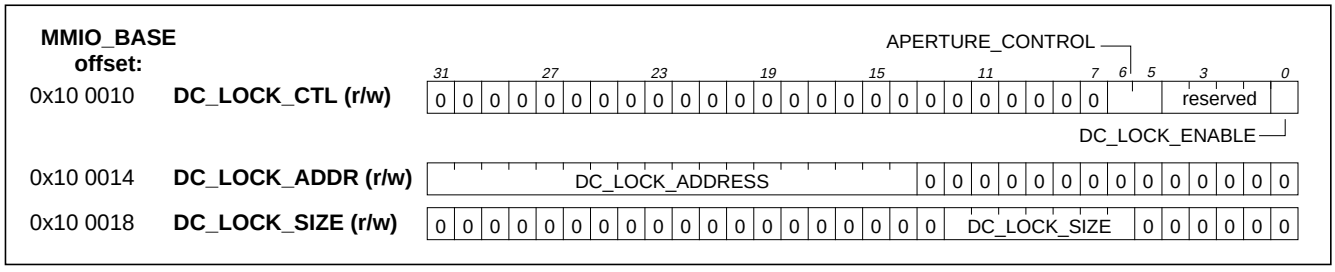


Figure 5-5. Formats of the registers in charge of data-cache locking.

'1'. Setting DC_LOCK_ENABLE to '0' causes no action except to allow the previously locked blocks to be replacement victims.

To program a new lock range, the following sequence of operations is used:

1. Disable cache locking by writing '0' to DC_LOCK_ENABLE.
2. Define a new lock range by writing to DC_LOCK_ADDR and DC_LOCK_SIZE.
3. Enable cache locking by writing '1' to DC_LOCK_ENABLE.

Dirty locked blocks can be written back to main memory while locking is enabled by executing copyback operations in software.

Programmer's note: Software should not execute dinvalid operations on a locked block. If it does, the block will be removed from the cache, creating a 'hole' in the lock range (and the data cache) that cannot be reused until locking is deactivated.

Cache locking is disabled by default when TM1300 is reset.

The RESERVED field in DC_LOCK_CTL should be ignored on reads and written as all zeroes.

Locking should not be enabled by PCI accesses to the MMIO registers.

5.3.8 Memory Hole and PCI Aperture Disable

Bits 6 and 5 in DC_LOCK_CTL comprise the APERTURE_CONTROL field. This field can be used to change the memory map as seen by the DSPCPU. The hardware RESET value of the field corresponds to the memory map as described in Section 3.4.1, "Memory Map."

Table 5-6. Aperture control field

Value	Memory map properties
00 (RESET)	Normal operation memory map (Section 3.4.1): • loads to 0..0xff always return 0 and cause no PCI read (memory hole is enabled) • PCI aperture(s) are enabled
01	• loads to address 0..0xff cause a PCI read, i.e. the memory hole is disabled • PCI aperture(s) are enabled
10	PCI apertures are disabled for loads • loads return a 0 and cause no PCI read
11	RESERVED for future extensions

5.3.9 Non-cacheable Region

The data cache supports one non-cacheable address region within the DRAM address space aperture. The base address of this region is determined by the value in the DRAM_CACHEABLE_LIMIT MMIO register, which is shown in Figure 5-6. Since uncached memory operations always incur many stall cycles, the non-cacheable region should be used sparingly.

A memory operation is non-cacheable if its target address satisfies:

[dram_cacheable_limit] <= address < [dram_limit]

Thus, the non-cacheable region is at the high end of the DRAM aperture. The format of the DRAM_CACHEABLE_LIMIT register forces the size of the non-cacheable region to be a multiple of 64 KB.

When TM1300 is reset, DRAM_CACHEABLE_LIMIT is set equal to DRAM_LIMIT, which results in a zero-length non-cacheable region.

Programmer's note: When DRAM_CACHEABLE_LIMIT is changed to enlarge the region that is non-cacheable, software must ensure coherency. This is accomplished by explicitly copying back dirty data (using dcb operations) and invalidating (using dinvalid operations) the cache blocks in the previously unlocked region.

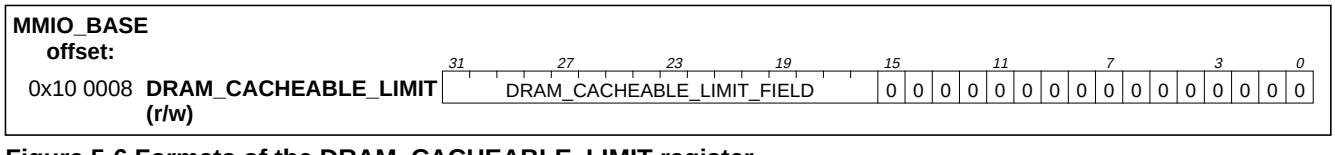


Figure 5-6 Formats of the DRAM_CACHEABLE_LIMIT register.

5.3.10 Special Data Cache Operations

A program can exercise some control over the operation of the data cache by executing special operations. The special operations can cause the data cache to initiate the copyback or invalidation of a block in the cache. These operations are typically used by software to keep the cache coherent with main memory.

In addition, there are special operations that allow a program to read tag and status information from the data cache.

Special data cache operations are always executed on the memory port associated with issue slot 5.

5.3.10.1 Copyback and invalidate operations

The data cache controller recognizes a copyback and an invalidate operation as shown in Table 5-7.

Table 5-7. Copyback and invalidate operations

Mnemonic	Description
dcb(offset) r*src1*	Data-cache copyback block. Causes the block that contains the target address to be copied back to main memory if the block is valid and dirty.
dinvalid(offset) r*src1*	Data-cache invalidate block. Causes the block that contains the target address to be invalidated. No copyback occurs even if the block is dirty.

The dcb and dinvalid operations both compute a target word address that is the sum of a register and seven-bit offset. The offset can be in the range [–256..252] and must be divisible by four.

dcb operation. The dcb operation computes the target address, and if the block containing the address is found in the data cache, its contents are written back to main memory if the block is both valid and dirty. If the block is not present, not valid, or not dirty, no action results from the dcb operation. If the dcb causes a copyback to occur, the CPU is stalled until the copyback completes. If the block is not in cache, the operation causes no stall cycles. If the block is in cache but not dirty, the operation causes 4 stall cycles. If the block is dirty, the dcb operation causes a writeback and takes at least 19 stall cycles.

The dcb operation clears the dirty bit but leaves a valid copy of the written-back block in the cache.

dinvalid operation. The dinvalid operation computes the target address, and if the block containing the address is found in the data cache, its valid and dirty bits are cleared. No copyback operation will occur even if the block is valid and dirty prior to executing the dinvalid operation. The CPU is stalled for 2 cycles, if the target block is in the cache; otherwise, no stall cycles occur.

A dinvalid or dcb operation updates the LRU information to least recently used in its set.

Programmer's note: Software should not execute dinvalid operations on locked blocks; otherwise, a 'hole' is created that cannot be reused until locking is deactivated.

5.3.10.2 Data cache tag and status operations

The data cache controller recognizes two DSPCPU operations for reading cache status as shown in Table 5-8.

The rdtag and rdstatus operations both compute a target word address that is the sum of a register and scaled seven-bit offset. The offset must be divisible by four and in the range [–256..252].

Table 5-8. Cache read-status operations

Mnemonic	Description
rdtag(offset) r*src1*	Read data-cache tag. The target address selects a data-cache block directly; the operation returns a 32-bit result containing the 21-bit cache tag and the valid bit.
rdstatus(offset) r*src1*	Read data-cache status. The target address selects a data-cache set directly; the operation returns a 32-bit result containing the set's eight dirty bits and ten LRU bits.

rdtag operation. The target address computed by rdtag selects the data cache block by specifying the cache set and set element directly. Address bits [10..6] specify the cache set (one of 32), and bits [13..11] specify the set element (one of eight). All other target address bits are ignored. This operation causes no CPU stall cycles.

The result of the rdtag operation is a full 32-bit word with the format shown in Figure 5-7.

rdstatus operation. The target address computed by rdstatus selects the data cache set by specifying the set number directly. Address bits [10..6] specify the cache set (one of 32); all other target address bits are ignored. This operation causes 1 CPU stall cycle.

The result of the rdstatus operation is a full 32-bit word with the format shown in Figure 5-7. See Section 5.6.7, "LRU Bit Definitions," for a description of the LRU bits.

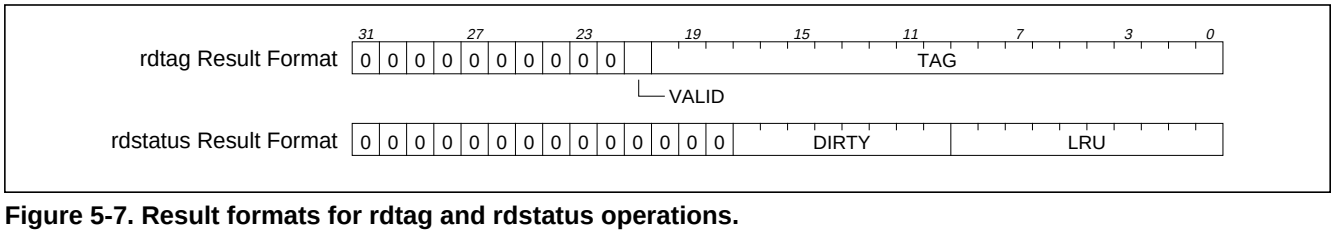


Figure 5-7. Result formats for rdtag and rdstatus operations.

5.3.10.3 Data cache allocation operation

The data cache controller recognizes allocation operations as shown in Table 5-9. The allocation operations allocate a block and set the status of this block to valid. No data is fetched from main memory. The allocated block is undefined after this operation. The programmer has to fill it with valid data by store operations. Allocation operations to apertures other than cacheable DRAM will be discarded. Allocation of a non-dirty block causes 3 stall cycles. Allocation of a dirty block will cause writeback of this block to the SDRAM and take at least 11 stall cycles.

Table 5-9. Data cache allocation operations

Mnemonic	Description
allocd(offset) r*src1*	Data-cache allocate block with displacement. Causes the block with address (rsrc1+offset) & (~(cache_block_size - 1)) to be allocated and set valid.
allocr r*src1 rsrc2*	Data-cache allocate block with index. Causes the block with address (rsrc1+rsrc2) & (~(cache_block_size - 1)) to be allocated and set valid.
allocx r*src1 rsrc2*	Data-cache allocate block with scaled index. Causes the block with address (rsrc1 + 4 * rsrc2) & (~(cache_block_size - 1)) to be allocated and set valid.

5.3.10.4 Data cache prefetch operation

The data cache controller recognizes prefetch operations as shown in Table 5-10. The prefetch operations load a full cache block from memory concurrently with other computation. If the prefetched block is already in cache, no data is fetched from main memory. Prefetch operations to other apertures than cacheable DRAM are discarded. This operation is not guaranteed to execute, it will not execute if the cache is already occupied with two cache misses when the operation is issued. The prefetch operations cause 3 stall cycles if there is no copyback of a dirty block. If a dirty block is the target of the prefetch, the dirty block will be written back to SDRAM, and at least 11 stall cycles are taken.

5.3.11 Memory Operation Ordering

The TM1300 memory system implements traditional ordering for memory operations that are issued in different clock cycles. That is, the effects of a memory operation issued in cycle j occur before the effects of a memory operation issued in cycle j+1.

For memory operations issued in the same cycle, however, it is not possible to execute memory operations in a traditional order. So long as the simultaneous memory operations access different addresses (aliasing is not possible in TM1300), no problems can occur. If two simultaneous operations do access the same address, however, TM1300 behavior is undefined. Specifically, two cases are possible:

Table 5-10. Data cache prefetch operations

Mnemonic	Description
prefd(offset) r*src1*	Data-cache prefetch block with displacement. Causes the block with address (rsrc1+offset) & (~(cache_block_size - 1)) to be prefetched
prefr r*src1 rsrc2*	Data-cache prefetch block with index. Causes the block with address (rsrc1+rsrc2) & (~(cache_block_size - 1)) to be prefetched.
pref16x r*src1 rsrc2*	Data-cache prefetch block with scaled 16-bit index. Causes the block with address (rsrc1 + 2 * rsrc2) & (~(cache_block_size - 1)) to be prefetched.
pref32x r*src1 rsrc2*	Data-cache prefetch block with scaled 32-bit index. Causes the block with address (rsrc1 + 4 * rsrc2) & (~(cache_block_size - 1)) to be prefetched.

1. When multiple values are written to the same address in the same cycle, the resulting value in memory is undefined.
2. When a read and a write occur to the same address in the same clock cycle, the value returned by the read is undefined.

The behavior of simultaneous accesses to the same address is undefined regardless of whether one or both memory operations hit in the cache.

Hidden Memory System Concurrency. Some cache operations may be overlapped with CPU execution. In general, a program cannot determine in what order cache misses will complete nor can a program determine when and in what order copyback operations will complete. A program can, however, enforce the completion of copyback transactions to main memory because copyback and invalidate operations can complete only if pending copyback transactions for the same block have completed. Thus, a program can synchronize to the completion of a copyback operation by dirtying a block, issuing a copyback operation for the block, and then issuing an invalidate operation for the block.

Ordering Of Special Memory Operations. The following are special memory operations:

1. Loads or stores to MMIO addresses.
2. Non-cached loads or stores.
3. Any copyback or invalidate operation.
4. Loads or stores that cause a PCI-bus access.

The CPU is stalled until these special memory operations are completed; there is no overlap of CPU execution with these special memory operations. Thus, a programmer can assume that traditional memory operation ordering applies to special memory operations. Note, however, that ordering is undefined for two special memory operations issued in the same cycle.

5.3.12 Operation Latency

Load and store operations have an operation latency of three cycles, regardless of the size of the data transfer.

5.3.13 MMIO Register References

Memory operations that reference MMIO registers are not cached, and the CPU is stalled until the MMIO reference completes. A MMIO register reference occurs when an address is in the range:

[MMIO_BASE] ≤ address < ([MMIO_BASE] + 0x200000)

The size of the MMIO aperture is hardwired at 2 MB.

5.3.14 PCI Bus References

Any CPU memory operation that references an address outside the SDRAM and MMIO address apertures is assumed to reference a device or memory on the PCI bus. PCI-bus data transfers are not cached, and the CPU is stalled until the PCI transfer completes.

5.3.15 CPU Stall Conditions

The data cache causes the CPU to stall when:

1. Any cache miss occurs.
2. Two simultaneously issued, cacheable memory operations need to access the same cache bank (bank conflict).
3. An access that references an address in the MMIO aperture is issued.
4. An access to the PCI bus is issued.
5. A non-trivial copyback or invalidate operation is issued.
6. An access to the non-cacheable region in the DRAM aperture is issued.

5.3.16 Data Cache Initialization

When TM1300 is reset, the data cache executes an initialization sequence. The cache asserts the CPU stall signal while it sequentially resets all valid and dirty bits. The cache de-asserts the stall signal after completing the initialization sequence.

5.4 INSTRUCTION CACHE

The instruction cache stores compressed CPU instructions; instructions are decompressed before being delivered to the CPU. The following sections describe the instruction cache and its operation; Table 5-11 summarizes instruction-cache characteristics.

Table 5-11. Instruction cache characteristics

Characteristic	TM1300 Implementation
Cache size	32 KB
Cache associativity	8-way set-associative
Block size	64 bytes
Valid bits	One valid bit per 64-byte block
Replacement policy	Hierarchical LRU (least-recently used) among the eight blocks in a set
Operation latency	Branch delay is three cycles
Coherency enforcement	Software uses a special operation to enforce cache coherency
Cache locking	Up to 1/2 (four out of eight blocks of each set) of the cache contents can be locked; granularity is 64 bytes

5.4.1 General Cache Parameters

The TM1300 instruction cache is 32 KB in size with a 64-byte block size. Thus, the cache contains 512 blocks each with its own address tag. The cache is 8-way set-associative, so there are 64 sets, each containing 8 tags. A single valid bit is associated with a block, so each block and associated address tag is either entirely valid or invalid; on a cache miss, 64 bytes are read from SDRAM to make the entire block valid.

The geometry of the instruction cache is available to software by reading the MMIO register IC_PARAMS. Figure 5-8 shows the format of the IC_PARAMS register; Table 5-12 lists its field values.

The product of the block size, associativity, and number of sets gives the total cache size (32 KB in this case).

Table 5-12. IC_PARAMS field values

Field Name	Value
BLOCKSIZE	64
ASSOCIATIVITY	8
NUMBER_OF_SETS	64

5.4.2 Address Mapping

TM1300 instruction addresses are mapped onto the data cache storage structure as shown in Figure 5-9. An instruction address is partitioned into three fields as described in Table 5-13

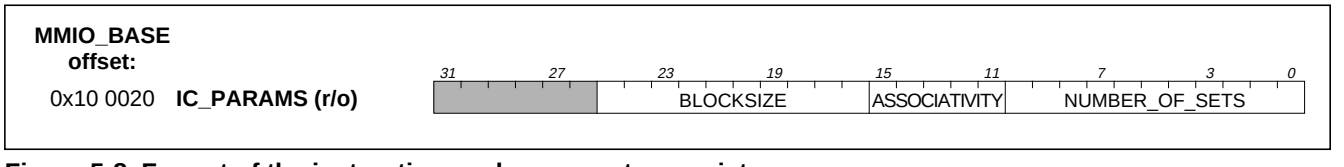


Figure 5-8. Format of the instruction-cache parameters register.

Table 5-13. Instruction Address Field Partitioning

Field	Address Bits	Purpose
Offset	5..0	Byte offset into a set
Set	11..6	Selects one of the sets in the cache (one of 64 in the case of TM1300)
Tag	31..12	Compared against address tags of set members

5.4.3 Miss Processing Order

When a miss occurs, the instruction cache starts filling the requested block from the beginning of the block. The DSPCPU is stalled until the entire block is fetched and stored in the cache.

5.4.4 Replacement Policy

The hierarchical LRU replacement policy implemented by the instruction cache is identical to that implemented by the data cache. See Section 5.3.4, “Replacement Policies, Coherency,” for a description of the hierarchical LRU algorithm.

5.4.5 Location of Program Code

All program code must first be loaded into SDRAM. The instruction cache cannot fetch instructions from other memories or devices. In particular, the cache cannot fetch code from on-chip devices or over the PCI bus.

5.4.6 Branch Units

The instruction cache is closely coupled to three branch units. Each unit can accept a branch independently, so three branches can be processed simultaneously in the same cycle.

Branches in TM1300 are called ‘delayed branches’ because the effect of a successful (taken) branch is not seen in the flow of control until some number of cycles after the successful branch is executed. The number of cycles of latency is called the branch delay. On TM1300, the branch delay is three cycles.

Although three branches can be executed simultaneously, correct operation of the DSPCPU requires that only one branch be successful (taken) in any one cycle. DSPCPU operation is undefined if more than one concurrent branch operation is successful.

Each branch unit takes four inputs from the DSPCPU: the branch opcode, a guard bit, a branch condition, and a branch target address. A branch is deemed successful if and only if the opcode is a branch opcode, the guard bit is TRUE (i.e., = 1), and the condition (determined by the opcode) is satisfied.

5.4.7 Coherency: Special iclr Operation

A program can exercise some control over the operation of the instruction cache by executing the special iclr operation. This operation causes the instruction cache to clear the valid bits for all blocks in the cache, including locked blocks. The LRU replacement status of all blocks is reset to its initial value. The CPU is stalled while iclr is executing.

See Section 5.6, “Cache Coherency,” for further discussion of coherency issues.

5.4.8 Reading Tags and Cache Status

The instruction cache supports read access to its tag and status bits, but not through special operations as with the data cache. Since the instruction cache and branch units can execute only resultless operations, access to the instruction-cache tags and status bits is implemented using normal load operations executed by the DSPCPU that reference a special region in the MMIO address aperture. The region is 64 KB long and starts at MMIO_BASE. Instruction cache tags and status bits are read-only; store operations to this region have no effect. MMIO operations to this special region are only allowed by the DSPCPU, not by any other masters of the on-chip data highway, such as external PCI initiators.

Programmer’s note: Tag and status information cannot be read by PCI access, but only by DSPCPU access. Tag and status read cannot be scheduled in the same cycle with or one cycle after an iclr operation.

Reading A Tag And Valid Bit. To read the tag and valid bit for a block in the instruction cache, a program can execute a ld32 operation directed at the instruction-cache region in the MMIO aperture. The top of Figure 5-10 shows the required format for the target address. The most-significant 16 bits must be equal to MMIO_BASE, the least-significant 15 bits select the block (by naming the set and set member), and bit 15 must be set to zero to perform a tag read. Note that in TM1300, valid set numbers range from 0 to 63. Space to encode set numbers 64 to 511 is provided for future extensions.

A ld32 with an address as specified above returns a 32-bit result with the format shown at the top of Figure 5-11. Bit 20 contains the state of the valid bit, and the least-significant 20 bits contain the tag for the block addressed by the ld32.

Reading The LRU Bits. To read the LRU bits for a set in the instruction cache, a program can execute a ld32 operation as above but using the address format shown at the bottom of Figure 5-10. In this format, bit 15 is set to one to perform the read of the LRU bits, and the tag_i_mux field is set to zeros because it is not needed.

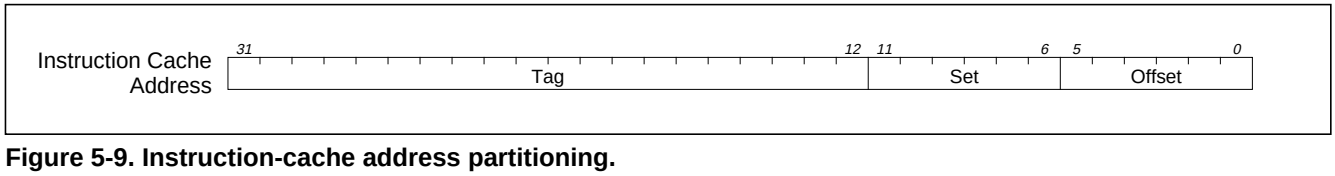


Figure 5-9. Instruction-cache address partitioning.

Reading the LRU bits produces a 32-bit result with the format shown at the bottom of Figure 5-11. The least-significant ten bits contain the state of the LRU bits when the ld32 was executed. See Section 5.6.7, “LRU Bit Definitions,” for a description of the LRU bits.

Note that the tag_i_mux and set fields in the address formats of Figure 5-10 are larger than necessary for the instruction cache in TM1300. These fields will allow future implementations with larger instruction caches to use a compatible mechanism for reading instruction cache information. The tag_i_mux field can accommodate a cache of up to 16-way set-associativity, and the set field can accommodate a cache with up to 512 sets. For TM1300, the following constraints of the values of these fields must be observed:

1. $0 \le$ tag_i_mux ≤ 7
2. $0 \le$ set ≤ 63

5.4.9 Cache Locking

Like the data cache, the instruction cache allows up to one-half of its blocks to be locked. A locked block is never chosen as a victim by the replacement algorithm; its contents remain undisturbed until the locked status is changed explicitly by software. Thus, on TM1300, up to 16 KB of the cache can be used as a high-speed instruction ‘ROM.’ Only four out of eight blocks in any set can be locked.

The MMIO registers IC_LOCK_ADDR, IC_LOCK_SIZE, and IC_LOCK_CTL—shown in Figure 5-12—are used to define and enable instruction locking in the same way that the similarly named data-cache locking registers are used. Section 5.3.7, “Cache Locking,” describes the details of cache locking; they are not repeated here.

Setting the IC_LOCK_ENABLE bit (in IC_LOCK_CTL) to ‘1’ causes the following sequence of events:

1. The instruction cache invalidates all blocks in the cache.
2. The instruction cache fetches all blocks in the lock range (defined by IC_LOCK_ADDR and IC_LOCK_SIZE) from main memory into the cache.
3. Cache locking is activated so that the locked blocks cannot be victims of the replacement algorithm.

The only difference between this sequence and the initialization sequence for data-cache locking is that dirty blocks (which cannot exist in the instruction cache) are not written back first.

Programmer’s note: Programmers (or compilers) must combine all instructions that need to be locked into the single linear instruction-locking address range.

The special iclr operation also removes locked blocks from the cache. If blocks are locked in the instruction cache, then instruction cache locking should be disabled in software (by writing ‘0’ to IC_LOCK_CTL) before an iclr operation is issued.

Locking should not be enabled by PCI accesses to the MMIO register.

5.4.10 Instruction Cache Initialization and Boot Sequence

When TM1300 is reset, the instruction cache executes an initialization and processor boot sequence. While reset is asserted, the instruction cache forces NOP operation to the DSPCPU, and the program counter is set to the default value reset_vector. When reset is deasserted, the initialization and boot sequence is as follows.

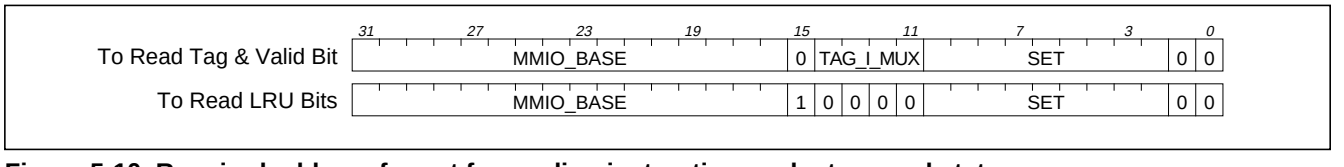


Figure 5-10. Required address format for reading instruction-cache tags and status.

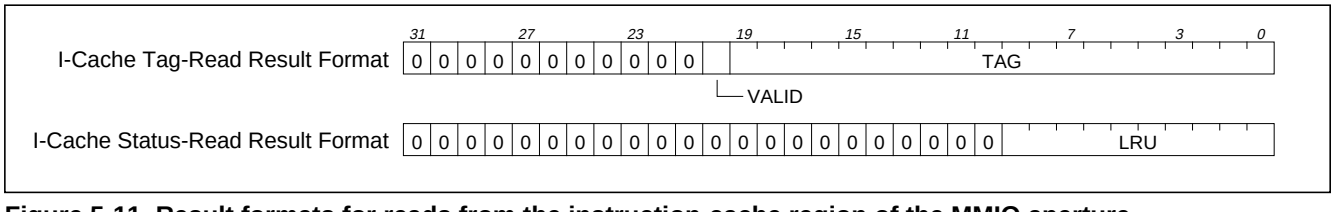


Figure 5-11. Result formats for reads from the instruction-cache region of the MMIO aperture.

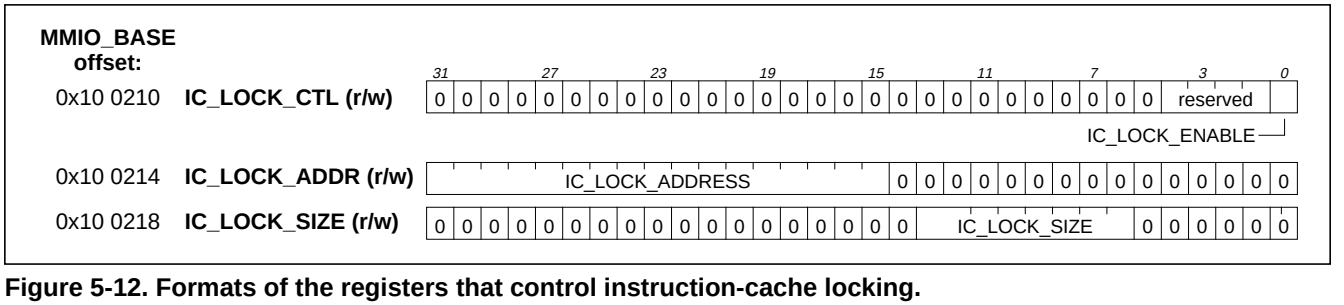


Figure 5-12. Formats of the registers that control instruction-cache locking.

1. The stall signal is asserted to prevent activity in the DSPCPU and data cache.
2. The valid bits for all blocks in the instruction cache are reset.
3. At the completion of the block invalidation scan, the stall signal to the DSPCPU and data cache are deasserted.
4. The DSPCPU begins normal operation with an instruction fetch from the address reset_vector.

The initialization process takes 512 clock cycles. Reset sets reset_vector equal to DRAM_BASE so that program execution starts at the initial value of DRAM_BASE. The initial value of DRAM_BASE is determined as described in Section 5.2, “DRAM Aperture.”

5.5 LRU ALGORITHM

When a cache miss occurs, the block containing the requested data must be brought into the cache to replace an existing cache block. The LRU algorithm is responsible for selecting the replacement victim by selecting the least-recently-used block.

The 8-way set-associative caches implement a hierarchical LRU replacement algorithm as follows. Eight sets are partitioned into four groups of two elements each. To select the LRU element:

- First, the LRU pair is selected out of the four pairs using a four-way LRU algorithm.
- Second, the LRU element of the pair is selected using a two-way LRU algorithm.

5.5.1 Two-Way Algorithm

The two-way LRU requires an administration of one bit per pair of elements. On every cache hit to one of the two blocks, the cache writes once to this bit (just a write, not a read-modify-write). If the even-numbered block is accessed, the LRU bit is set to ‘1’; if the odd-numbered block is accessed, the LRU bit is set to ‘0’. On a miss, the cache replaces the LRU element, i.e. if the LRU bit is ‘0’, the even numbered element will be replaced; if the LRU bit is ‘1’, the odd numbered element will be replaced.

5.6 CACHE COHERENCY

The TM1300 hardware does not implement coherency between the caches and main memory. Generalized coherency is the responsibility of software, which can use the special operations dcb, dinvalid, and iclr to enforce cache/memory synchronization.

5.6.1 Example 1: Data-Cache/Input-Unit Coherency

Before the CPU commands the video-in unit to capture a video frame, the CPU must be sure that the data cache contains no blocks that are in the address region that the video-in unit will use to store the input frame. If the video-in unit performs its input function to an address region and the data cache does hold one or more blocks from that region, any of the following may happen:

- A miss in the data cache may cause a dirty block to be copied back to the address region being used by the video-in unit. If the video-in unit already stored data in the block, the write-back will corrupt the frame data.
- The CPU will read stale data from the cache instead of from the block in main memory. Even though the video-in unit stored new video data in the block in main memory, the cache contents will be used instead because it is still valid in the cache.

To prevent erroneous copybacks or the use of stale data, the CPU must use dinvalid operations to invalidate all blocks in the address region that will be used by the VI unit.

5.6.2 Example 2: Data-Cache/Output-Unit Coherency

Before the CPU commands the video-out unit to send a frame of video, the CPU must be sure that all the data for the frame has been written from the data cache to the region of main memory that the video-out unit will output. Explicit action is necessary because the data cache—with its copyback write policy—will hold an exclusive copy of the data until it is either replaced by the LRU algorithm or the CPU explicitly forces it to be copied back to main memory.

Before an output command is issued to the video-out unit, the CPU must execute dcb operations to force coherency between cache contents and main memory.

5.6.3 Example 3: Instruction-Cache/Data-Cache Coherency

If code prepared by a program running on the CPU must be subsequently executed, coherency between the instruction and data caches must be enforced. This is accomplished by a two-step process:

1. Coherency between the data cache and main memory must be enforced since the instruction cache can fetch instructions only from main memory.
2. Coherency between the instruction cache and main memory is enforced by executing an iclr operation.

The CPU will now be able to fetch and execute the new instructions.

5.6.4 Example 4: Instruction-Cache/Input-Unit Coherency

When an input unit is used to load program code into main memory, the iclr operation must be issued before attempting to execute the new code.

5.6.5 Four-Way Algorithm

For administration of the four-way algorithm, the cache maintains an upper-left triangular matrix ‘R’ of 1-bit elements without the diagonal. R contains six bits (in gener-

LRU bit 9	LRU bit 8	LRU bit 7	LRU bit 6	LRU bit 5	LRU bit 4	LRU bit 3	LRU bit 2	LRU bit 1	LRU bit 0
2_way[3]	2_way[2]	2_way[1]	2_way[0]	R[1,0]	R[2,1]	R[2,0]	R[3,2]	R[3,1]	R[3,0]

Figure 5-13. LRU bit definitions; 2_way[k] is the two-way LRU bit of pair k = (j div 2) for set element j.

MMIO_BASE offset:

0x10 000C **MEM_EVENTS (r/w)**

31	27	23	19	15	11	7	3	0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0	Event2	Event1		

Figure 5-14. Format of the memory_events MMIO register.

al, n×(n–1)/2 bits for n-way LRU). If set element k is referenced, the cache sets row k to ‘1’ and column k to ‘0’:

R[k, 0..n–1] ← 1,
R[0..n–1, k] ← 0

The LRU element is the one for which the entire row is ‘0’ (or empty) and the entire column is ‘1’ (or empty):

R[k, 0..n–1] = 0 *and* R[0..n–1, k] = 1

For a 4-way set-associative cache, this algorithm requires six bits per set of four cache blocks. On every cache hit, the LRU info is updated by setting three of the six bits to ‘0’ or ‘1’, depending on the set element that was accessed. The bits need only be written, no read-modify-write is necessary. On a miss, the cache reads the six LRU bits to determine the replacement block.

TM1300 combines the two-way and four-way algorithms into an 8-way hierarchical LRU algorithm. A total of ten administration bits are required: six to maintain the four-way LRU plus four bits maintain the four two-way LRUs.

The hierarchical algorithm has performance close to full eight-way LRU, but it requires far fewer bits—ten instead of 28 bits—and is much simpler to implement.

To update the LRU bits on a cache hit to element j (with 0 <= j <= 7), the cache applies m = (j div 2) to the four-way LRU administration and (j mod 2) is applied to the two-way administration of pair m. To select a replacement victim, the cache first determines the pair p from the four-way LRU and then retrieves the LRU bit q of pair p. The overall LRU element is the p×2+q.

5.6.6 LRU Initialization

Reset causes the LRU administration bits to initialized to a legal state:

R[1,0] ← R[2,0] ← R[3,0] ← 1
R[2,1] ← R[3,1] ← R[3,2] ← 0
2_way[3] ← 2_way[2] ← 2_way[1] ← 2_way[0] ← 0

5.6.7 LRU Bit Definitions

The ten LRU bits per set are mapped as shown in Figure 5-13. This is the format of the LRU field as returned by the special operation rdstatus for the data cache and a ld32 from MMIO space (see Section 5.4.8, “Reading Tags and Cache Status”) for the instruction cache.

5.6.8 LRU for the Dual-Ported Cache

For the TM1300 dual-ported data cache, two memory operations to the same set are possible in a single clock cycle. To support this concurrency, two updates of the LRU bits of a single set must be possible.

The following rules are used by TM1300:

1. LRU bits that are changed by exactly one port receive the value according to the algorithm described above.
2. LRU bits that are changed by both ports receive a value as if the algorithm were first applied for the access in port zero and then for the access in port one.

5.7 PERFORMANCE EVALUATION SUPPORT

The caches implement support for performance evaluation. Several events that occur in the caches can be counted using the TM1300 timer/counters, by selecting the source CACHE1 and/or CACHE2, as described in Section 3.8, “Timers.” Two different events can be tracked simultaneously by using 2 timers.

The MMIO register MEM_EVENTS determines which events are counted. See Figure 5-14 for the format of MEM_EVENTS. Table 5-14 lists the events that can be tracked and the corresponding values for the MEM_EVENTS fields. Event1 selects the actual source

for the TIMER CACHE1 source. Event2 selects the source for TIMER CACHE2.

Table 5-14. Trackable cache-performance events

Encoding	Event
0	No event counted
1	Instruction-cache misses
2	Instruction-cache stall cycles (including data-cache stall cycles if both instruction-cache and data-cache are stalled simultaneously)
3	Data-cache bank conflicts
4	Data-cache read misses
5	Data-cache write misses
6	Data-cache stall cycles (that are not also instruction-cache stall cycles)
7	Data-cache copyback to SDRAM
8	Copyback buffer full
9	Data-cache write miss with all fetch units occupied
10	Data cache stream miss
11	Prefetch operation started and not discarded
12	Prefetch operation discarded (because it hits in the cache or there is no fetch unit available)
13	Prefetch operation discarded (because it hits in the cache)
14–15	Reserved

If the memory bus is available:

- On read data cache miss the minimum waiting time is 12 SDRAM clock cycles, if critical word first is granted by the Main Memory Interface (MMI). If not, then data cache waits from 12 to 18 SDRAM cycles (16 SDRAM cycles are required to fetch 64 bytes from SDRAM.
- On write data cache miss, the missing line needs to be fetched, thus it implies the same SDRAM cycles as a read data cache miss. If the victimized cache line is dirty, the cache line is copied back to memory after the read of the missing line is done and thus does not add extra stall cycles.
- Prefetch delay is the same as read data cache if memory bus is available. As a reminder the prefetch may be discarded if the data cache state machine is “full”, and there is a 3 stall cycle penalty when the prefetch is issued.

5.8 MMIO REGISTER SUMMARY

Table 5-15 lists the MMIO registers that pertain to the operation of TM1300’s instruction and data caches.

Table 5-15. MMIO register summary

Name	Description
DRAM_BASE	Sets location of the DRAM aperture
DRAM_LIMIT	Sets size of the DRAM aperture
DRAM_CACHEABLE_LIMIT	Divides DRAM aperture into cacheable and non-cacheable portions
MEM_EVENTS	Selects which two events will be counted by timer/counters
DC_LOCK_CTL	Data-cache locking enable and aperture control
DC_LOCK_ADDR	Sets low address of the data-cache address lock aperture
DC_LOCK_SIZE	Sets size of the data-cache address lock aperture
DC_PARAMS	Read-only register with data-cache parameter information
IC_PARAMS	Read-only register with instruction-cache parameter information
IC_LOCK_CTL	Instruction-cache locking enable
IC_LOCK_ADDR	Sets low address of the instruction-cache address lock aperture
IC_LOCK_SIZE	Sets size of the instruction-cache address lock aperture
MMIO_BASE	Sets location of the MMIO aperture

Video In Chapter 6

by Gert Slavenburg

6.1 VIDEO IN OVERVIEW

The Video In (VI) unit provides the following functions:

- Digital video input from a digital camera or analog camera (using a video decoder).
- High-bandwidth (81 MB/sec) raw input data channel.
- Direct 8-10 bit interface for video A/D converters at up to 81-MHz sample rate.
- Receiver port for TM1300-to-TM1300 unidirectional message passing

The VI unit operates in one of the modes per Table 6-1.

Table 6-1. VI unit mode selection.

Mode	Function	Explanation
0000	fullres capture	YUV 4:2:2 capture, no decimation
0001	halfres capture	YUV 4:2:2 capture, decimate by 2
0010	raw8 capture	raw 8-bit data capture, pack 4 bytes to a word
0011	raw10s capture	raw 10-bit data capture, sign extend to 16 bits, pack 2 to a word
0100	raw10u capture	raw 10-bit data capture, zero-extend to 16 bits, pack 2 to a word
0101	message passing	message reception from EVO
0110 .. 1111	Reserved	

Digital video input is in YUV 4:2:2 with 8-bit resolution multiplexed in CCIR656 format[1] from a digital camera or CCIR656-capable video decoder (such as the Philips SAA7111 or SAA7113), across an 8-bit-wide interface. Resolutions up to CCIR601 are accepted at 50 or 60 fields per second. A programmable rectangular image is captured from a video frame and written in *planar format* to TM1300 SDRAM. The video camera or decoder can be programmed using the TM1300 I^2C bus. In *fullres capture* mode, luminance (Y) and chrominance (U, V) pass unmodified. In *halfres capture* mode, luminance and chrominance are horizontally decimated by a factor of two to convert to CIF-like resolution with YUV 4:2:2 or MPEG sampling rules. If vertical subsampling on chrominance is desired, it can be performed by software on the DSPCPU or by the on-chip image coprocessor (ICP).

When operating as raw input data channel, VI accepts 8-bit-wide data. The operation mode is *raw8 capture.* No data selection or data interpretation is done. Data is written in packed form, four bytes to a word, to local SDRAM. There is no hardware control over the rate at which the source sends data. Instead, VI maintains two pointer/counter registers to ensure that no data is lost when the local SDRAM memory buffer fills. Data is accepted at the clock of the sender. If desired, VI_CLK can be programmed as an output to drive the data transfer at a programmable rate.

VI can accept raw data from up to 10-bit A/D converters, at sampling rates up to 81 MHz. VI can operate in *raw8, raw10u,* or *raw10s capture* mode for eight-bit, unsigned 10-bit or signed 10-bit data. In the 10-bit modes, data is zero- or sign-extended to 16 bits and stored in packed form in local SDRAM. As with the *raw8-capture* mode, VI maintains two pointer/counter registers to ensure that no data is lost when the local SDRAM memory buffer fills. Data is accepted at the externally set sampling rate. If desired, VI_CLK can be programmed as an output to serve as a programmable sampling clock.

VI can act as receiver from the Enhanced Video Out (EVO) unit of another TM1300. One EVO unit can broadcast to multiple receiving VIs. In this *message passing* mode, no data selection or data interpretation is done. Each message of the sender is written as byte-packed data to a separate local SDRAM memory buffer. Message start and end is indicated by the sender. The receiving VI will accept data until the sender indicates message end or until the current memory buffer is full. If the memory buffer fills before message end is encountered, the received data is truncated and an error condition is raised.

6.1.1 Interface

Besides the VI-specific pins in Table 6-2, the TM1300 I^2C interface is typically used to control the external camera or video decoder.

Figure 6-1 through Figure 6-4 illustrate typical connections for commonly used external sources. Note that VI_DVALID is only used in special circumstances, e.g. when sending data through a channel that results in clock periods both with and without data transfers.

1. Refer to CCIR recommendation 656: interfaces for digital component video signals in 525-line and 625-line television systems. Recommendation 656 is included in the Philips Desktop Video Data Handbook.

Table 6-2. VI unit interface pins

VI_CLK	I/O-5	• If configured as input (power up default): a positive transition on this incoming video clock pin samples all other VI_DATA input signals below if VI_DVALID is HIGH. If VI_DVALID is LOW, VI_DATA is ignored. Clock and data rates of up to 81 MHz are supported. • If configured as output: programmable output clock to drive an external video A/D converter. Can be programmed to emit integral dividers of DSPCPU_CLK. • See Section 6.2 for clock programming details.
VI_DVALID	IN-5	VI_DVALID indicates that valid data is present on the VI_DATA lines. If HIGH, VI_DATA will be accepted on the next VI_CLK positive edge. If LOW, no VI_DATA will be sampled.
VI_DATA[7:0]	IN-5	CCIR656 style YUV 4:2:2 data from a digital camera, or general purpose high speed data input pins. Sampled on positive transitions of VI_CLK if VI_DVALID HIGH.
VI_DATA[9:8]	IN-5	Extension high speed data input bits to allow use of 10-bit video A/D converters in raw10 modes. VI_DATA[8] serves as START and VI_DATA[9] as END message input in message passing mode.Sampled on positive transitions of VI_CLK if VI_DVALID HIGH.

6.1.2 Diagnostic Mode

The VI logic can be set to operate in diagnostic mode, which connects the inputs of VI to the outputs of the EVO unit. This mode provides boot diagnostics with the ability to verify major operational aspects of the chip before handing control to an operating system.

Diagnostic mode is entered by writing a control word with a ‘1’ in the DIAGMODE bit position to the VI_CTL register (see Figure 6-11). The EVO unit has to be setup to provide a clock before starting DIAGMODE. After a VI software reset, the DIAGMODE bit has to be set back to ‘1’. In diagnostic mode, the VI signals are exactly as shown in Figure 6-2, except that the inputs come from the on-chip EVO unit. Note that the inputs are truly taken from the TM1300 EVO external pins, i.e. if an external (board level) source is driving EVO pins, diagnostic mode is not capable of testing the EVO unit.

Note that the diagnostic mode only controls an input multiplexer. VI can be programmed and operated in all usual modes. The *raw* modes are particularly attractive for diagnostics purposes, since they allow VI to operate almost as an on-chip logic analyzer.

6.1.3 Power Down and Sleepless

The VI unit enters power down state whenever TM1300 is put in global power down mode, except if the SLEEPLESS bit in VI_CTL is set. In the latter case, the block continues DMA operation and will wake up the DSPCPU whenever an interrupt is generated.

The EVO block can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. Refer to Chapter 21, “Power Management.”

It is recommended that the EVO unit be stopped (by negating VI_CTL.CAPTURE_ENABLE) before block-level power down is started, or that SLEEPLESS mode be used when global power down is activated.

6.1.4 Hardware and Software Reset

Video In is reset by a TM1300 hardware reset (pin TRI_RESET#) or by a VI software reset. The latter is accomplished by writing a control word of 0x00080000 to the VI_CTL register. After a software reset, allow for 5 video clock cycles delay before enabling VI capture. Upon hardware or software reset, the VI_CTL, VI_STATUS, and VI_CLOCK registers are set to all ’0’s. The state of the other registers after RESET is undefined. Note that the VI clock has to be present while applying the software reset.

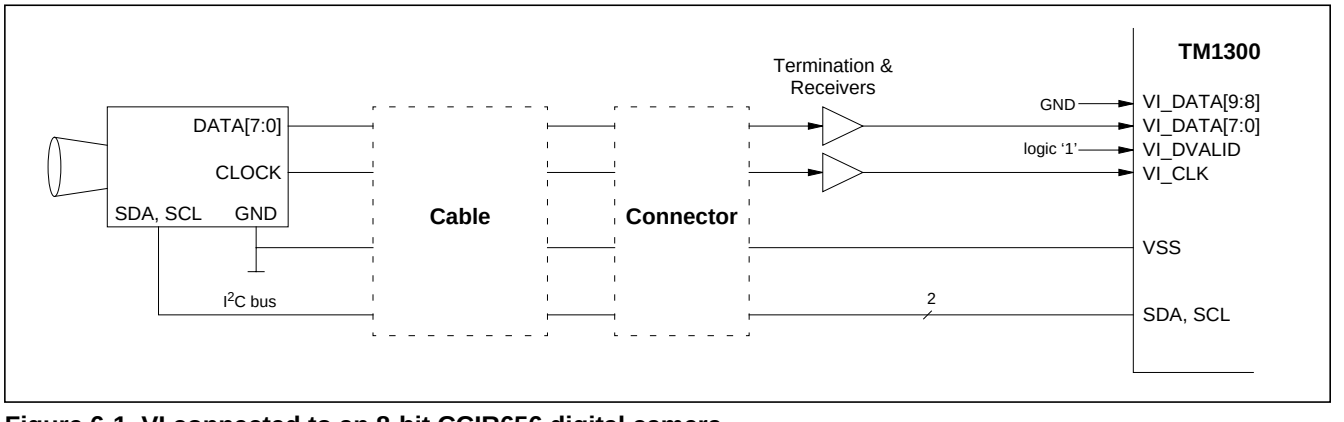


Figure 6-1. VI connected to an 8-bit CCIR656 digital camera.

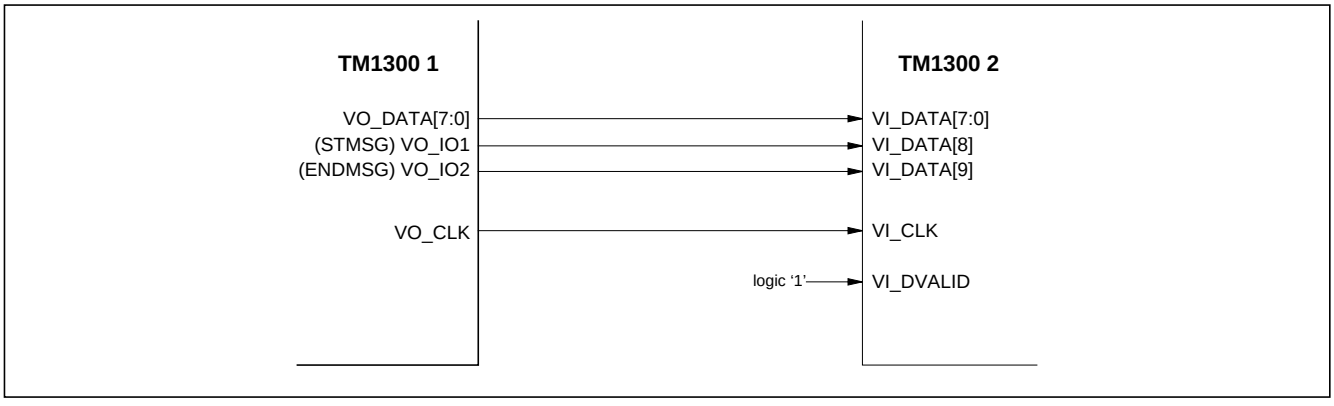


Figure 6-2. VI unit connected to an EVO unit of another TM1300.

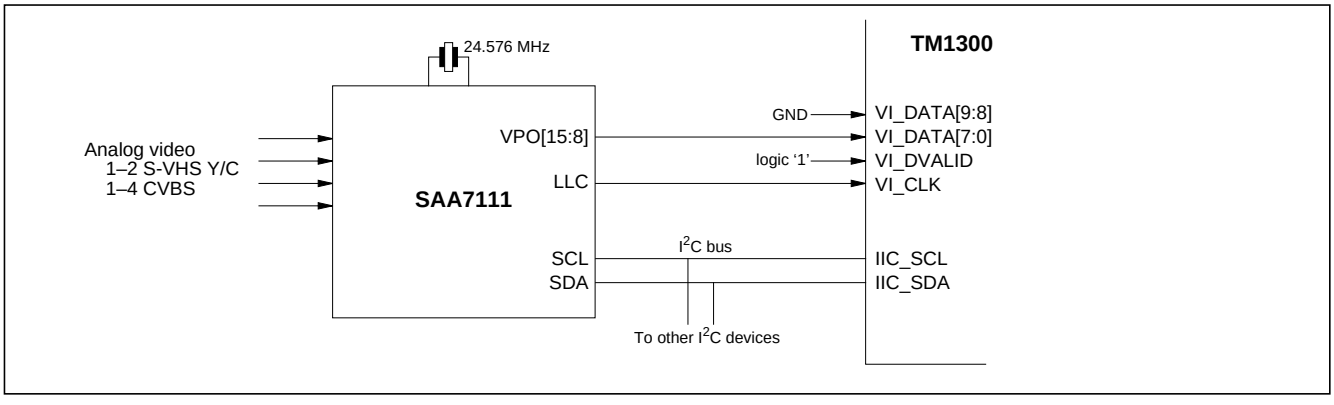


Figure 6-3. VI unit connected to a video decoder.

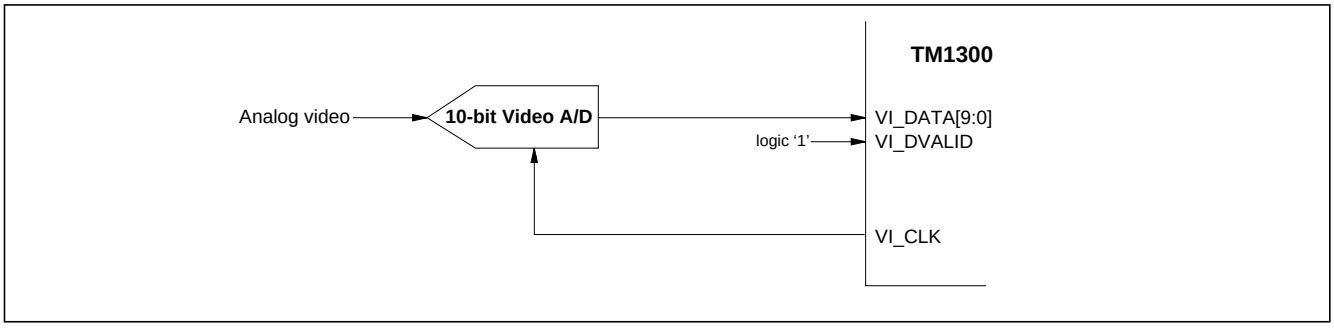


Figure 6-4. VI connected to a 10-bit video A/D converter.

6.2 CLOCK GENERATOR

The VI block can operate in two distinct clocking modes, as controlled by the VI_CLOCK control register (see Figure 6-11).

SELFCLOCK = 0: 'External clocking mode'. This is the most common mode of operation. In this mode, the VI_CLK pin is an asynchronous clock input. All other inputs are sampled on positive edges of the VI_CLK clock signal. On-chip synchronizers ensure reliable asynchronous capture. This mode can be combined with DIAGMODE, in which case the EVO clock acts as the asynchronous clock source. In external clocking mode, the value of DIVIDER is ignored.

SELFCLOCK = 1: 'Internal clocking mode". This mode is typically intended for use with external A/D converters or other sources that require a clock. In this mode, VI_CLK is an output pin. Positive edges of VI_CLK are used to sample all other inputs. The generated clock frequency can be programmed using the DIVIDER field in the VI_CLOCK register.

$$f_{VICLK} = \frac{f_{DSPCPU}}{DIVIDER}$$

On RESET, VI_CLOCK is set to zero, i.e. external clocking mode is the default with DIVIDER ignored.

6.3 FULLRES CAPTURE MODE

In *fullres capture* mode, the VI unit receives all three video components Y, U, and V, as well as synchronization information (SAV and EAV codes) on the VI_DATA[7:0] pins in CCIR656 format. See Figure 6-8. The three video components Y, U, and V are separated into three different streams. Each component is written in packed form into separate Y, U, and V buffers in the SDRAM. This is commonly called a *planar* format[1] (see Figure 6-10).

The CCIR656 standard specifies that the camera has to obey the sampling rules illustrated in Figure 6-5. VI is capable of chrominance resampling, and can produce samples in memory in two ways:

VI_CTL.SC=0. 'Co-sited sampling' places luminance and chrominance samples in memory without any modification. Hence, a planar format results with sampling positions as per co-sited luminance and chrominance YUV 4:2:2 convention.

1. The *planar format* is most suitable as input to software compression algorithms.

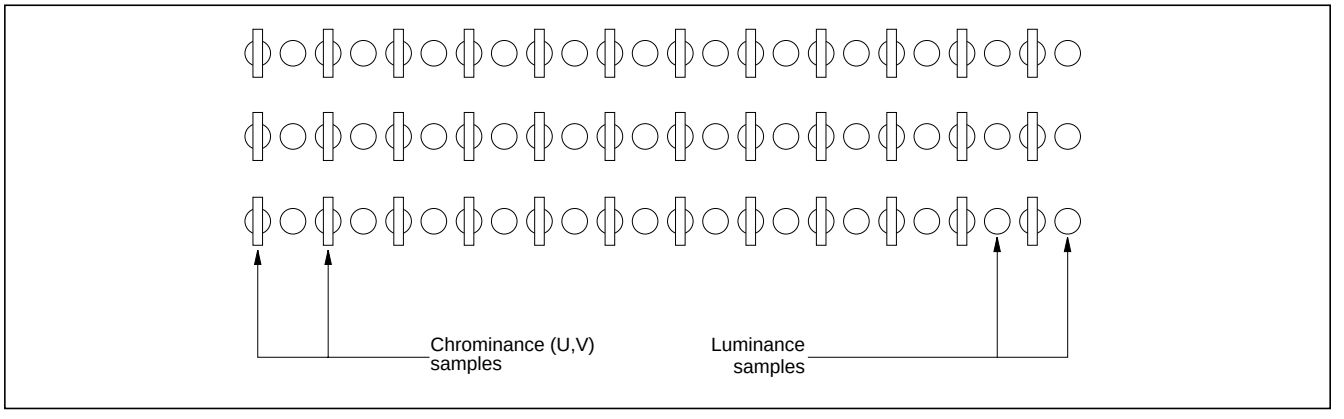


Figure 6-5. Camera YUV 4:2:2 sampling (co-sited luminance/chrominance).

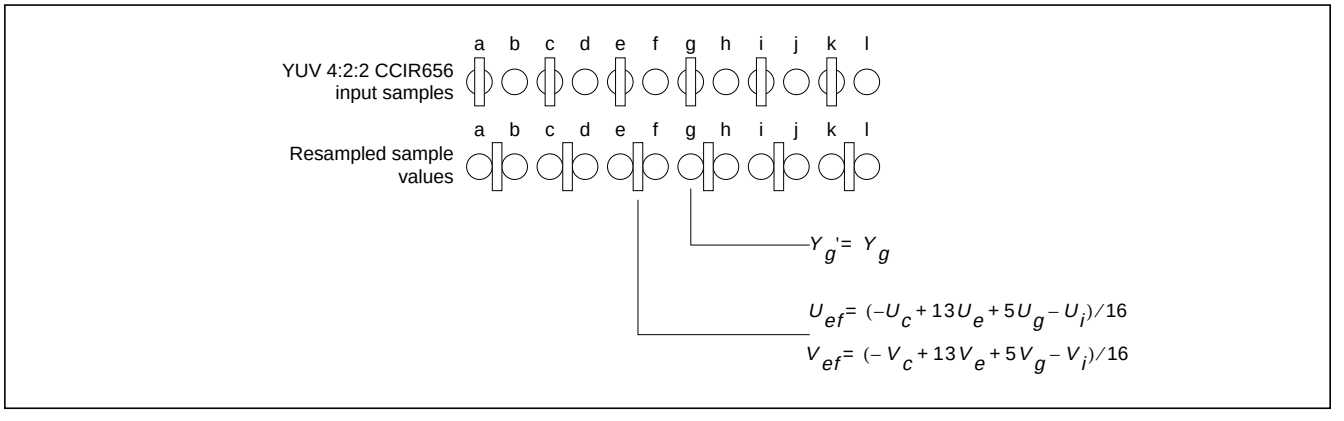


Figure 6-6. Chrominance re-sampling to achieve interspersed sampling.

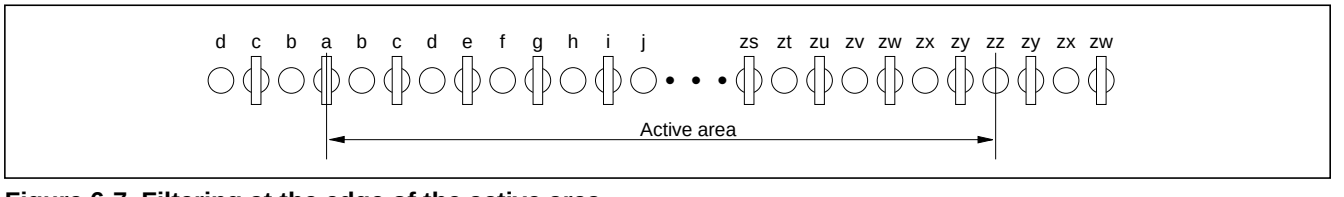


Figure 6-7. Filtering at the edge of the active area.

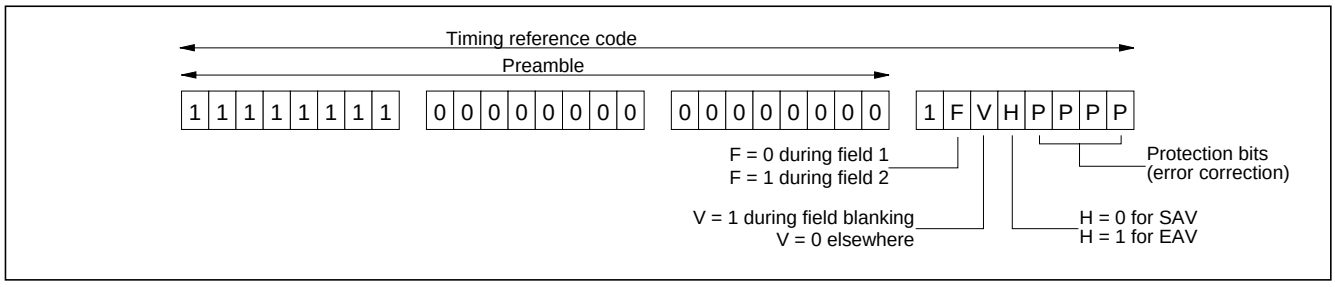


Figure 6-8. Format of CCIR656 SAV and EAV timing reference codes.

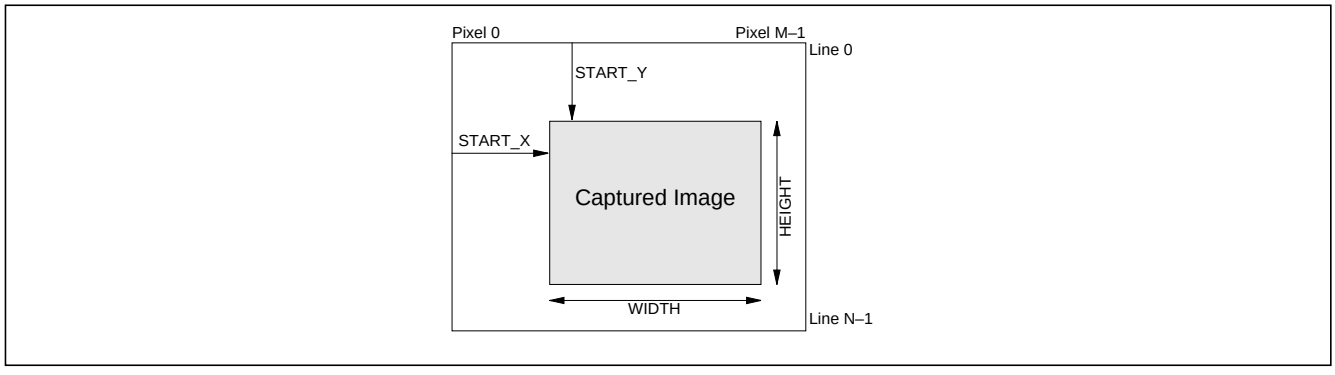


Figure 6-9. VI capture parameters.

VI_CTL.SC=1: 'Interspersed sampling' serves to generate a sampling structure in memory where chrominance samples are spatially midway between luminance samples, as shown in Figure 6-6. This 'interspersed' format is suitable for use in MPEG-1 encoding.

The VI hardware applies a (–1 13 5 –1)/16 filter as illustrated in Figure 6-6 to the chrominance samples before writing them to memory. This filter computes chrominance values at sample points midway between luminance samples[1]. Computed video data is clamped to 01h if the filter result is less than 01h and clamped to FFh if greater than FFh. Interspersed data format is preferred by some video compression standards. The MPEG-1 standard, for example, requires YUV 4:2:0 data with chrominance sampling positions horizontally and vertically midway between luminance samples. This can be achieved from the horizontally interspersed sampling format by vertical subsampling with a (1 1) / 2 or more sophisticated filter. Vertical filtering can be performed in software using the DSPCPU's efficient multimedia operations or by hardware in the on-chip ICP.

The filtering process exercises special care at the left and right edges of the active area of the CCIR656 data stream, as defined by the SAV, EAV code positions. See Figure 6-7. Since no pixels exist to the left of the first pixel or to the right of the last pixel, filtering can result in artifacts. To minimize artifacts, the image is extended by mirroring pixels around the left-most and right-most pixel. Note that the image is mirrored around pixel 'a', the first pixel after the SAV code and around pixel 'zz', the last pixel before the EAV[2] code. Pixel 'a' in Figure 6-7 is the (chroma, luma) pair defined by the first three camera bytes of the UYVYUYVY... stream after SAV.

Refer to Figure 6-11 for an overview of the memory mapped I/O (MMIO) registers that are used to control and observe the operation of VI in fullres capture mode. To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read and written as'0's.

Upon hardware or software reset (Section 6.1.4, "Hardware and Software Reset"), the VI_CTL, VI_STATUS, and VI_CLOCK registers are set to all zeros.

At any point in time, the VI_STATUS register fields (see Figure 6-11) indicate the current camera status:

- CUR_X: The pixel index (0 to M–1) of the most recently received camera pixel. CUR_X gets set to zero for the first pixel following receipt of a SAV code[3], and incremented on every valid Y sample received thereafter.
- CUR_Y: The line index (0 to N–1) within the current field of the camera line that is currently being received. CUR_Y gets set to zero upon receipt of a negative edge of V, i.e., upon the first SAV code containing V=0 after one or more SAV codes containing V=1. This is equivalent to the first line after the end of vertical retrace. CUR_Y gets incremented upon every successive SAV code.

1. All filters perform full precision intermediate computations and saturation upon generating the result bits.

2. EAV codes with multiple bit errors are accepted and enable the mirroring function.

3. Note that VI uses the SAV protection bits to implement single error correction and double error detection. An SAV code with double error is ignored.

- FIELD2: Indicates whether the field currently being received is a field1 or 2. This flag gets updated based on the F field of every received SAV code. Note that field1 is the 'top' field, i.e. the field containing the top-most visible line. Field1 contains lines 1,3,5 etc. Field2 contains lines 2,4,6,8 etc.

Table 6-3 illustrates common digital camera standards and the number of active pixels per line, lines per field, and fields per second. Note that any source is acceptable to VI, as long as the maximum VI_CLK rate is not exceeded.

Table 6-3. Common video source parameters.

Video Source	M (# active pixels)	N (# active lines)	Field Rate (Hz)
CCIR601 50 Hz/625 lines	720	288	50
CCIR601 60 Hz/525 lines	720	240	60
square pixel 50 Hz/625 lines	768	288	50
square pixel 60 Hz/525 lines	640	240	60

Figure 6-9 shows the details of an incoming field and the captured image. The incoming field consists of N horizontal lines, each line having M pixels labeled 0 through M–1. Lines are numbered from 0 through N–1. The captured image is a subset of the incoming image. It is defined by the capture parameters (START_X, START_Y, WIDTH, HEIGHT) held in the VI_CAP_START and VI_CAP_SIZE MMIO registers (see Figure 6-11).

- START_X: defines the starting pixel number (X-coordinate of the starting pixel). START_X must be even, and greater than or equal to '0'.
- START_Y: defines the starting line number (Y-coordinate of the starting pixel). START_Y must be greater than or equal to '0'.
- WIDTH: Defines the width of the captured image in pixels. WIDTH must be even.
- HEIGHT: Defines the height of the captured image in lines.

Image capture starts after the following conditions are met:

- VI_CTL.CAPTURE ENABLE is asserted.
- VI_STATUS.CAPTURE COMPLETE is de-asserted, indicating that any previously captured image has been acknowledged.
- CUR_Y = START_Y occurs.

Once image capture is started, HEIGHT 'lines' are captured. Each line capture starts if:

- The previous line capture, if any, is completed.
- CUR_X = START_X

Once line capture starts, it continues for 2*WIDTH pixel clocks[1] in which VI_DVALID is asserted, irrespective of the presence of one or more EAV codes.

Note that capture continues regardless of any horizontal or vertical retrace and associated CUR_Y or CUR_X reset. This provides special applications with the ability to capture information embedded inside the horizontal or vertical blanking interval. If it is desirable to capture pixels in the horizontal blanking interval, a minimum time separation of 1 µs is required between the last pixel captured on line *y* and the first pixel captured on line *y+1*. An exception to this rule is allowed if and only if the storage parameters below are chosen such that the last and first pixel end up in adjacent memory locations. Note that blanking information capture only makes sense in fullres mode with co-sited sampling. All other modes apply filtering, which will distort the numeric sample values.

The captured image is stored in SDRAM at a location defined by the storage parameters in MMIO registers (Y_BASE_ADR, Y_DELTA, U_BASE_ADR, U_DELTA, V_BASE_ADR, V_DELTA). Note that the base-address registers force alignment to 64-byte boundaries (six LSBs are always zero). The default memory packing is big-endian although little-endian packing is also supported by setting the LITTLE_ENDIAN bit in the VI_CTL register.

- Y_BASE_ADR: The desired starting (byte) address in SDRAM memory where the first Y (luminance) sample of the captured image will be stored. This address is forced to be 64-byte aligned (six LSBs always '0').
- Y_DELTA: The desired address difference between the last sample of a line and the address of the first sample on the next line. Note that the value of Y_DELTA must be chosen so that all line-start addresses are 64-byte aligned.
- U_BASE_ADR, U_DELTA, V_BASE_ADR, V_DELTA: Same functions and alignment restrictions as above, but for chrominance-component samples.

Horizontally-adjacent samples are stored at successive byte addresses, resulting in a packed form (four 8-bit samples are packed into one 32-bit word). Upon horizontal retrace, pixel storage addresses are incremented by the corresponding DELTA to compute the starting byte address for the next line. Note that DELTA is a 16-bit unsigned quantity. This process continues until HEIGHT lines of WIDTH samples have been stored in memory for luminance (Y). For chrominance, HEIGHT lines of half the WIDTH are stored[2]. See Figure 6-10.

Modifications to Y_BASE_ADR, U_BASE_ADR and V_BASE_ADR have no effect until the start of next capture, i.e. VI hardware maintains a separate pointer to track the current address. Modifications to Y_DELTA,

1. Four clocks for each C_b,Y,C_r,Y group representing two luminance pixels
2. Note that consecutive pixel components of each line are stored in consecutive memory addresses but consecutive lines need not be in consecutive memory addresses

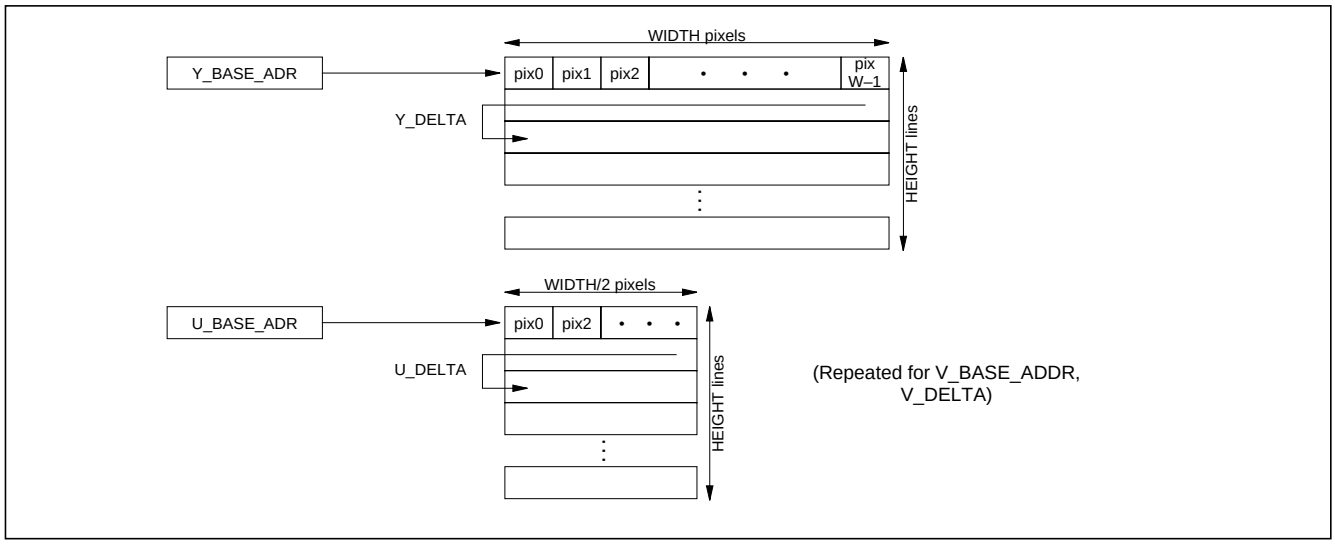


Figure 6-10. VI YUV 4:2:2 planar memory format.

U_DELTA and V_DELTA do affect the next horizontal retrace. Hence, under normal circumstances, the DELTA variables should not be changed during capture.

When capture is complete, i.e. any internal VI buffers have been flushed and the entire captured image is in local SDRAM, VI raises the STATUS register flag CAPTURE COMPLETE. If enabled in the VI_CTL register, this event causes a DSPCPU interrupt to be requested.

The programmer can determine whether the captured image is a field1 or field2 by inspection of the FIELD2 flag in VI_STATUS. Note that the FIELD2 flag changes at the start of the vertical blanking interval of the next field.

The CAPTURE COMPLETE flag is cleared by writing a word to VI_CTL with a '1' in the CAPTURE COMPLETE ACK bit position. This action has the following effect:

- it tells the hardware that a new Y,U, and V DMA buffer is available (or the old one has been copied)
- it clears the CAPTURE COMPLETE flag
- it tells VI to capture the next image

The user can program the Y_THRESHOLD field to generate pre-completion (or post-completion) interrupts. Whenever CUR_Y reaches Y_THRESHOLD, the THRESHOLD REACHED flag in the STATUS register is set. If enabled in the VI_CTL register, this event causes a DSPCPU interrupt request. The THRESHOLD REACHED flag is cleared by writing a word to VI_CTL with a '1' in the THRESHOLD REACHED ACK bit position. Note that, due to internal buffering in the VI unit, it is NOT guaranteed that all samples from lines up to and including CUR_Y have been written to local SDRAM upon THRESHOLD REACHED. The implementation guarantees a fixed maximum time of 2 µs between raising the interrupt and completion of all writes to SDRAM. The THRESHOLD interrupt mechanism works regardless of CAPTURE ENABLE. Hence, it can also be used to skip a desired number of fields without constant DSPCPU polling of VI_STATUS.

If VI internal buffers overflow due to insufficient internal data-highway bandwidth allocation, the HIGHWAY BANDWIDTH ERROR condition is raised in the VI_STATUS register. If enabled, this causes assertion of a VI interrupt request. Capture continues at the correct memory address as soon as the internal buffers can be written to memory, but one or more pixels may have been lost, and the corresponding memory locations are not written. The HBE condition can be cleared by writing a '1' to the HIGHWAY BANDWIDTH ERROR ACK bit in VI_CTL. Refer to Section 6.7, "Highway Latency and HBE" for more information.

Any interrupt event of VI (CAPTURE COMPLETE, THRESHOLD REACHED, HIGHWAY BANDWIDTH ERROR) leads to the assertion of a single VI interrupt (SOURCE 9) to the TM1300 Vectored Interrupt Controller. The interrupt handler routine should check the STATUS register to determine the set of VI events associated with the request. The vectored interrupt controller should always be set to have VI (SOURCE 9) operate in level sensitive mode. This ensures that each event is handled.

VI asserts the interrupt request line as long as one or more enabled events are asserted. The interrupt handler clears one or more selected events by writing a '1' to the corresponding ACK field in VI_CTL. The clearing of the last event leads to immediate (next DSPCPU clock edge) de-assertion of the interrupt request line to the Vectored Interrupt Controller. See Section 3.5.3, "INT and NMI (Maskable and Non-Maskable Interrupts)," for information on how to program interrupt handler routines.

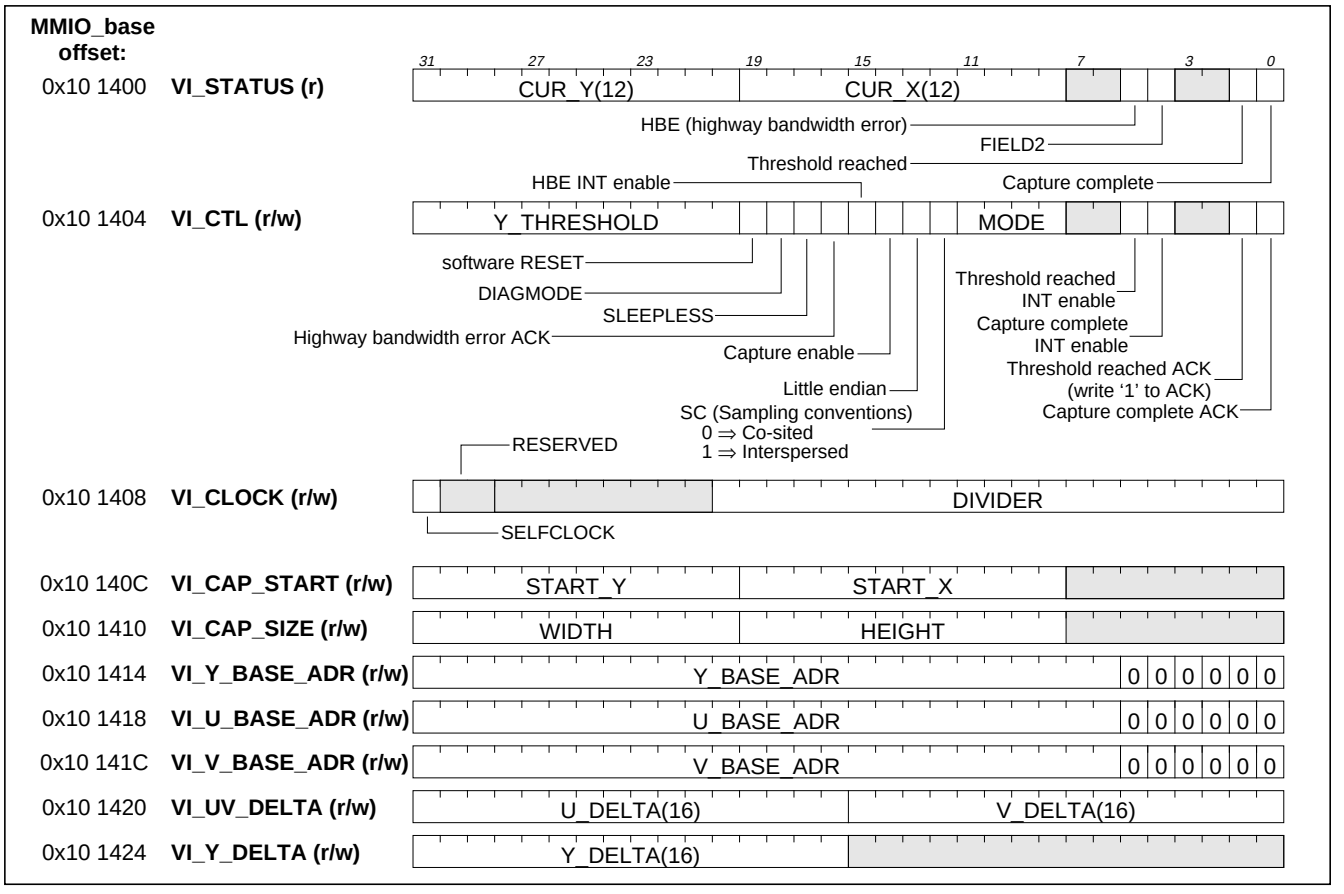


Figure 6-11. YUV capture view of VI MMIO registers.

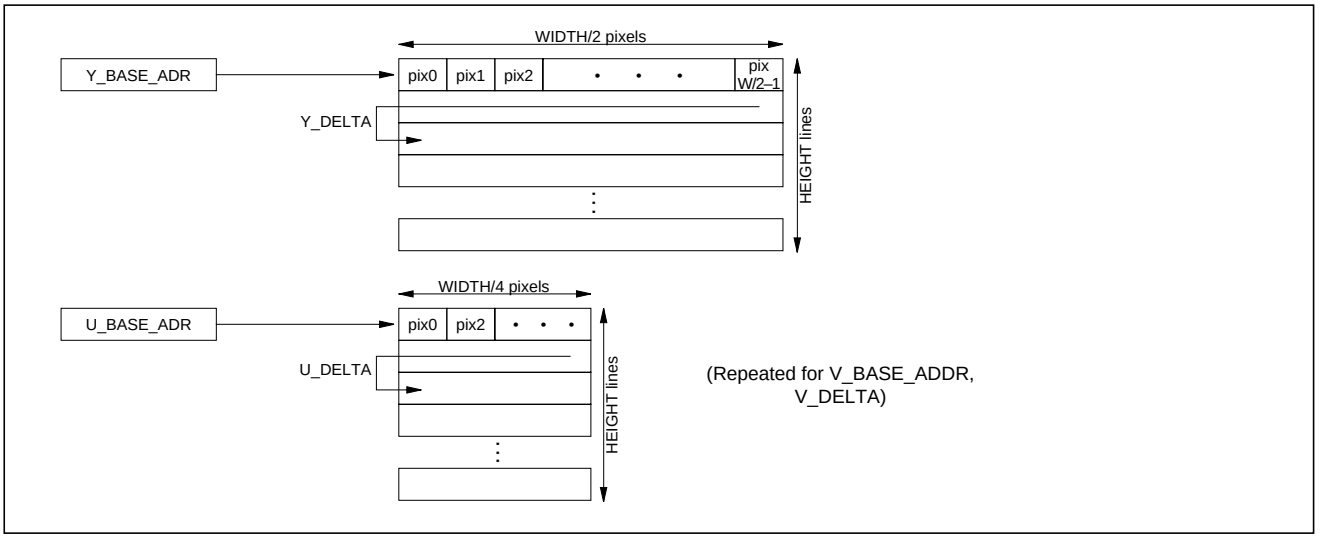


Figure 6-12. VI halfres planar memory format.

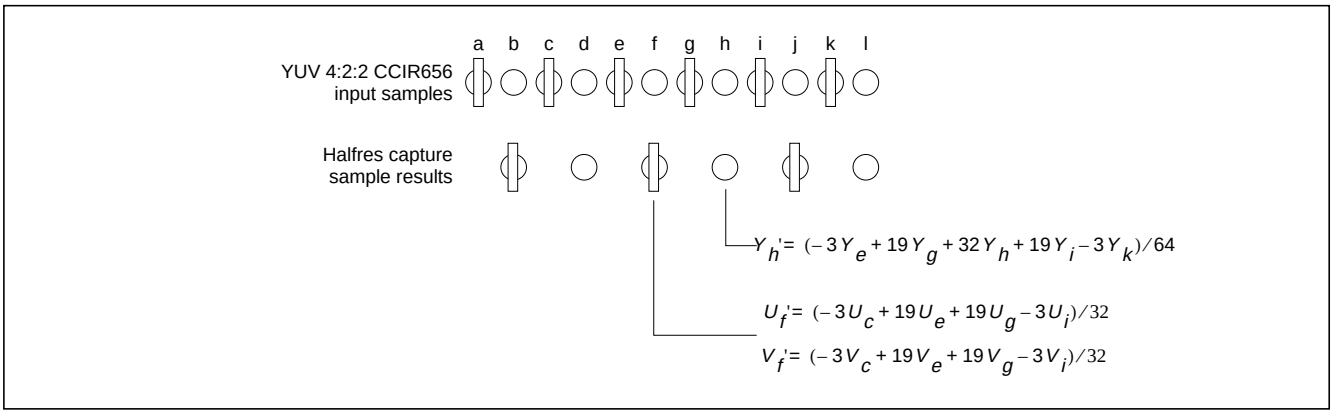


Figure 6-13. Halfres co-sited sample capture.

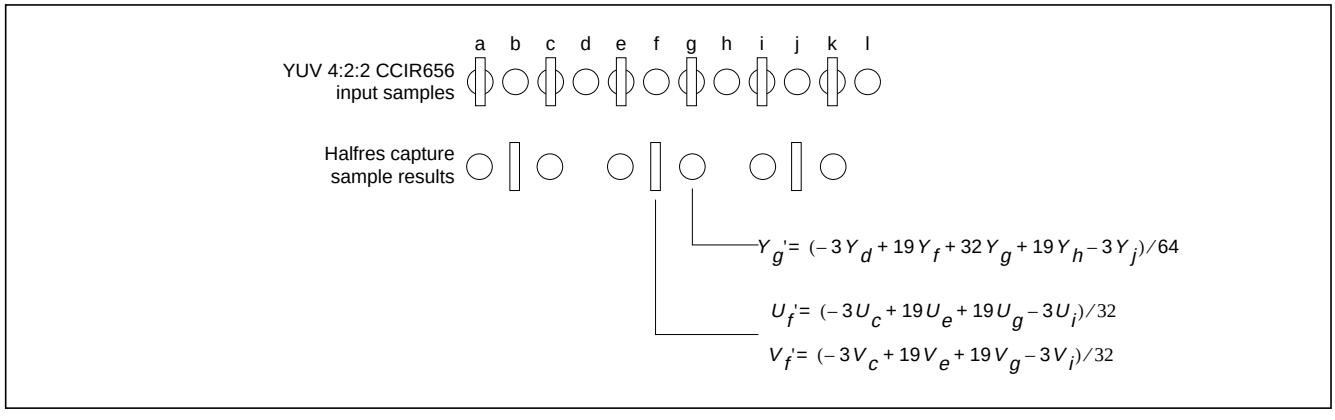


Figure 6-14. Halfres interspersed sample capture.

6.4 HALFRES CAPTURE MODE

Halfres capture mode is identical in operation to fullres capture mode except that horizontal resolution is reduced by a factor of two on both luminance and chrominance data.

Referring to Figure 6-9 and Figure 6-11, if VI is programmed to capture HEIGHT lines of WIDTH pixels in halfres mode, the resulting captured planar data is as shown in Figure 6-12. Note that WIDTH/2 luminance and WIDTH/4 chrominance samples are captured. In this mode, START_X and WIDTH must be a multiple of four.

Horizontal-resolution reduction is performed as shown in Figure 6-13 or Figure 6-14. The spatial sampling conventions of the pixels in memory depends on the SC (sampling convention) bit in the VI_CTL register. Assuming that the camera sampling positions obey the conventions shown in Figure 6-5, two possible spatial formats are supported in memory:

- If SC=0, co-sited luminance and chrominance samples result as shown in Figure 6-13. This corresponds to the standard YUV 4:2:2 sampling conventions.
- If SC=1, interspersed chrominance samples result, as shown in Figure 6-14. This form is (after vertical subsampling of the chroma components) identical to the MPEG-1 sampling conventions. If vertical subsampling is desired, it can either be performed in software on the DSPCPU or in hardware by the ICP.

The filtering process applies mirroring at the edge of the active video area, as per Figure 6-7.

For both filters, computed video data is clamped to 01h if result of the filter is less than 01h and clamped to FFh if greater than FFh.

6.5 RAW CAPTURE MODES

All raw capture modes (raw8, raw10s and raw10u) behave similarly. VI_DATA information is captured at the rate of the sender's clock, without any interpretation or start/stop of capture on the basis of the data values. Any clock cycle in which VI_DVALID is asserted leads to the capture of one data sample. Samples are 8 or 10 bits long (raw8 versus raw10 modes). For the 8-bit capture mode, four samples are packed to a word. For the 10-bit capture modes, two 16-bit samples are packed to a word. The extension from 10 to 16 bits uses sign extension (raw10s) or zero extension (raw10u).

For 8-bit and 16-bit capture, successive captured values are written to increasing memory addresses. For 16-bit

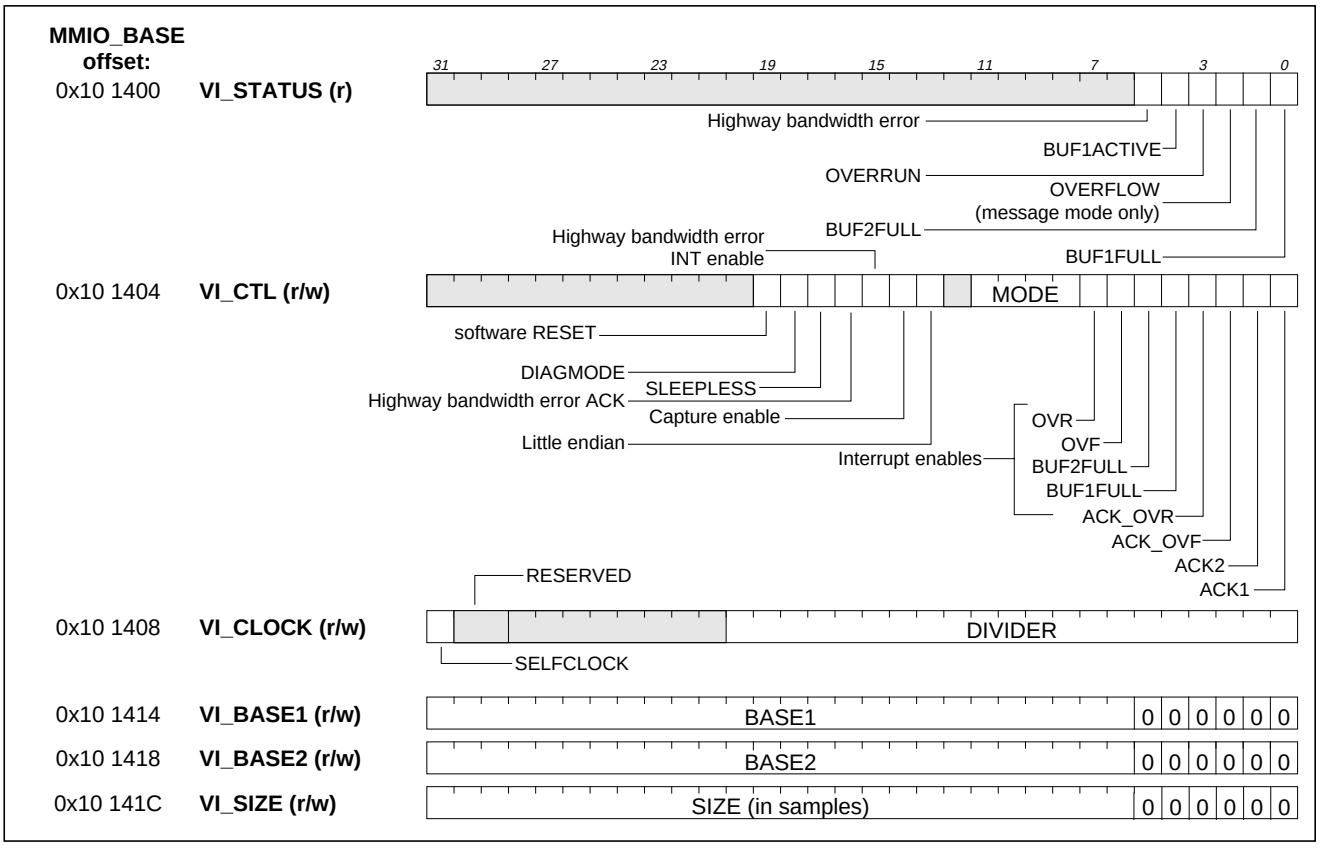


Figure 6-15. Raw and message passing modes view of VI MMIO registers.

capture, the byte order with which the 16-bit data is written to memory is governed by the LITTLE ENDIAN bit. The VI LITTLE ENDIAN bit should be set the same as the DSPCPU endianness (PCSW.BSX). This ensures that the DSPCPU sees correct 16-bit data.

Figure 6-15 illustrates the ‘raw-mode’ view of the VI MMIO registers. Figure 6-16 shows the major VI states associated with raw-mode capture. The initial state is reached on software or hardware reset as described in Section 6.1.4, “Hardware and Software Reset”. Upon reset, all status and control bits are set to ‘0’. In particular, CAPTURE_ENABLE is set to ‘0’ and no capture takes place.

Once the software has programmed BASE1 and BASE2 (with the start addresses of two SDRAM buffer areas[1]) and SIZE (in number of samples), it is safe to enable capture by setting CAPTURE_ENABLE. Note that SIZE is in samples and must be a multiple of 64, hence setting a minimum buffer size of 64 bytes for raw8 mode and 128 bytes for raw10 modes. At this point, buffer1 is the active capture buffer. Data is captured in buffer1 until capture is disabled or until SIZE samples have been captured. After every sample, a running address pointer is incremented by the sample size (one or two bytes). If SIZE samples have been captured, capture continues (without missing a sample) in buffer2. At the same time, BUF1FULL is asserted. This causes an interrupt on the DSPCPU, if enabled by BUF1FULL INTERRUPT ENABLE.

Buffer2 is now the active capture buffer and behaves as described above. In normal operation, the DSPCPU will respond to the BUF1FULL event by assigning a new BASE1 and (optionally) SIZE and performing an ACK1. If the DSPCPU fails to assign a new buffer1 and performs an ACK1 before buffer2 also fills up, the OVERRUN condition is raised and capture stops. Capture continues upon receipt of an ACK1, ACK2, or both, regardless of the OVERRUN state. The buffer in which capture resumes is as indicated in Figure 6-16. The OVERRUN condition is ‘sticky’ and can only be cleared by software, by writing a ‘1’ to the ACK_OVR bit in the VI_CTL register.

If insufficient bandwidth is allocated from the internal data highway, the VI internal buffers may overflow. This leads to assertion of the HIGHWAY BANDWIDTH ERROR condition. One or more data samples are lost. Capture resumes at the correct memory address as soon as the internal buffer is written to memory. The HBE error condition is sticky. It remains asserted until it is cleared by writing a ‘1’ to HIGHWAY BANDWIDTH ERROR ACK. Refer to Section 6.7, “Highway Latency and HBE.”

Note that VI hardware uses copies of the BASE and SIZE registers once capture has started. Modifications of BASE or SIZE, therefore, have no effect until the start of the next use of the corresponding buffer.

Note also that the VI_BASE1 and VI_BASE2 addresses must be 64-byte aligned (the six LSBs are always ‘0’).

1. SDRAM buffers must start on a 64-byte boundary.

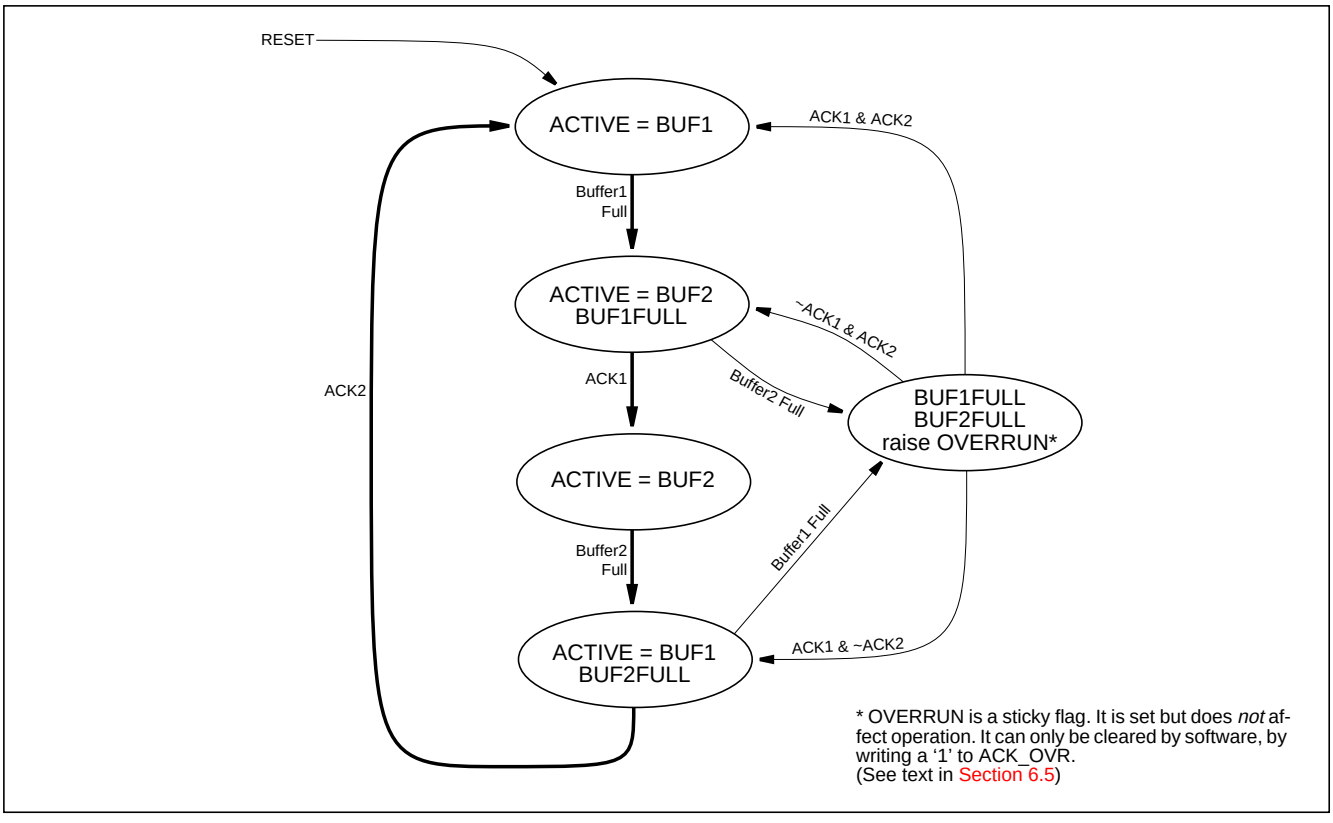


Figure 6-16. VI raw mode major states.

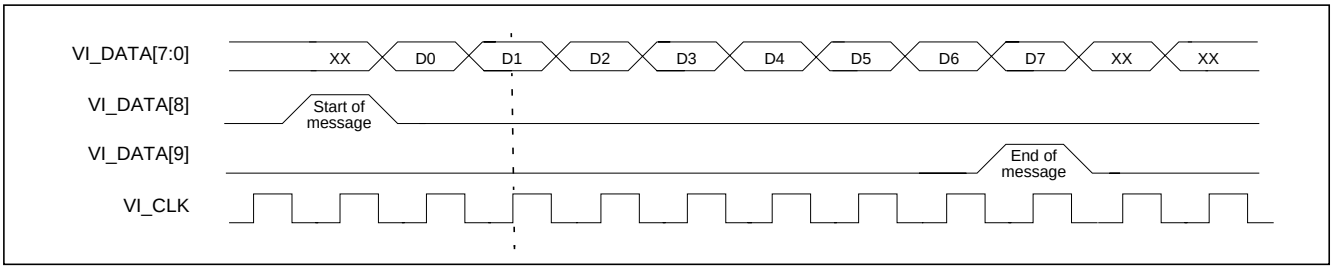


Figure 6-17. VI message passing signal example.

6.6 MESSAGE-PASSING MODE

In this mode, VI receives 8-bit message data over the VI_DATA[7:0] pins. The message data is written in packed form (four 8-bit message bytes per 32-bit word) to SDRAM. Message data capture starts on receipt of a START event on VI_DATA[8]. Message data is received until EndOfMessage (EOM) is received on VI_DATA[9] or the receive buffer is full. Note that the VI_SIZE MMIO register determines the buffer size, and hence maximum message length. It should not be changed without a VI (soft) reset.

Figure 6-17 illustrates an example of an 8-byte message transfer. The first byte (D0) is sampled on the rising edge of the VI_CLK clock after a valid START was sampled on the preceding rising clock edge. The last byte (D7) is sampled on the rising clock edge where EOM is sampled asserted.

The message passing mode view of the VI MMIO registers is shown in Figure 6-15. The major states are shown in Figure 6-18. The operation is almost identical to the operation in raw-capture mode, except that transitions to another active buffer occur upon receipt of EOM rather than on buffer full. OVERRUN is raised if the second buffer receives a complete message before a new buffer is assigned by the DSPCPU.

OVERFLOW is raised if a buffer is full and no EOM has been received. If enabled, it causes a DSPCPU interrupt. Since digital interconnection between devices is reliable, overflow is indicative of a protocol error between the two TM1300s involved in the exchange (failure to agree on message size). Detection of overflow leads to total halt of capture of this message. Capture resumes in the next buffer upon receipt of the next START event on VI_DATA[8]. The OVERFLOW flag is sticky and can only be cleared by writing a '1' to ACK_OVF.

Highway bandwidth error behavior in message passing mode is identical to that of raw mode.

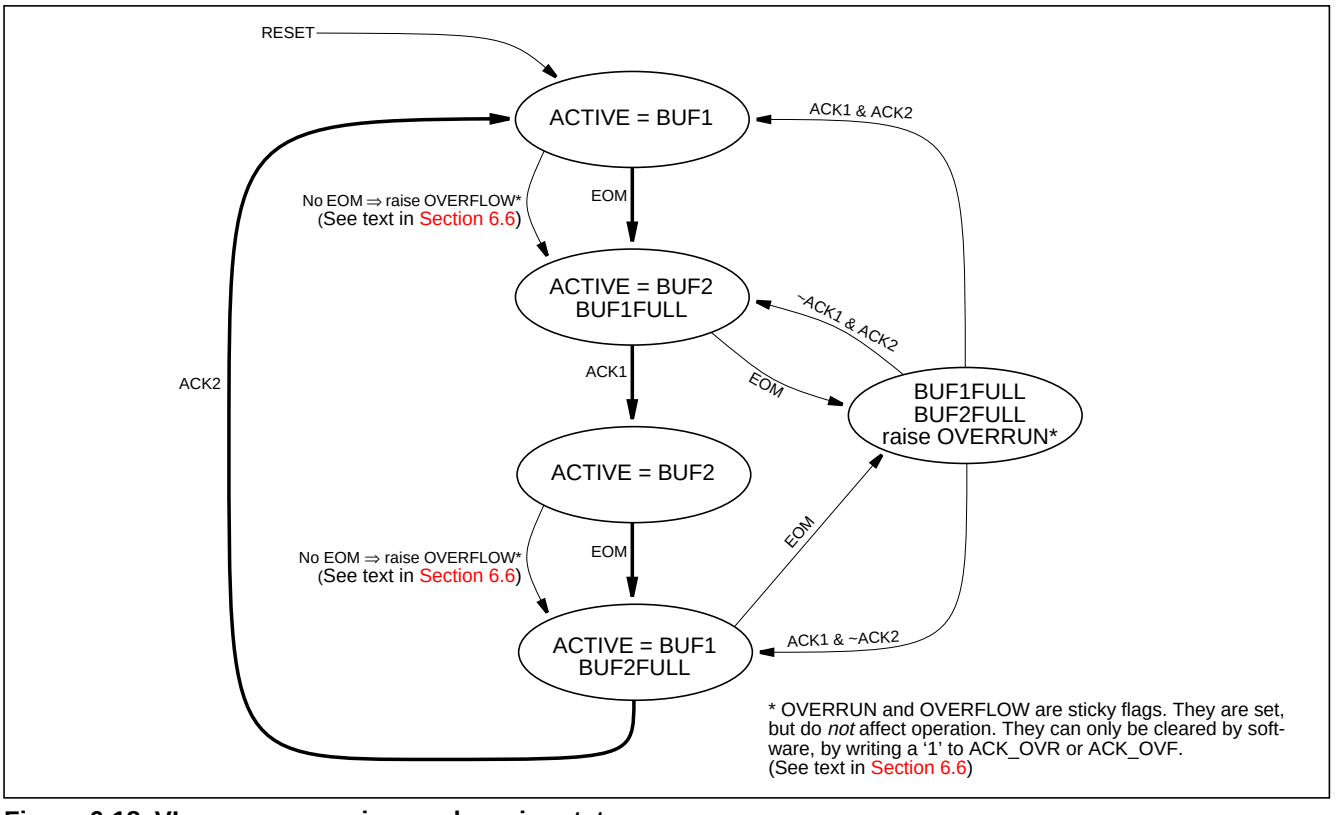


Figure 6-18. VI message passing mode major states.

6.7 HIGHWAY LATENCY AND HBE

Refer to Chapter 20, "Arbiter," for a description of the arbiter terminology used here. The VI unit uses internal buffering before writing data to SDRAM. There are two internal buffers, each 16 entries of 32 bits.

In fullres mode, each internal buffer is used for 128 Y samples, 64 U samples, and 64 V samples. Once the first internal buffer is filled, 4 highway transactions must occur before the second buffer fills completely. Hence, the requirement for not losing samples is:

- 4 requests must be served within 256 VI clock cycles.

For the typical CCIR601-resolution NTSC or PAL 27-MHz VI clock rate, the latency requirement is 4 requests in 9481 ns (25600/27). This can be used as one request every 2370 ns or, with a TM1300 SDRAM clock speed of 100 MHz, every 237 SDRAM clock cycles. The one request latency is used to define the priority raising value (see Section 20.6.3 on page 20-8).

In halfres mode, the Y, U, and V decimation by 2 takes place before writing to the internal buffers. So, the requirement for not loosing samples is:

- 4 requests served within 512 VI clock cycles.

For halfres subsampling, NTSC or PAL 27-MHz VI clock rate and TM1300 SDRAM clock speed of 100 MHz, latency is 4 requests in 51200/27 = 18962 ns (1896 highway clock cycles) or one request every 4740 ns (474 SDRAM clock cycles).

For raw8 capture and message passing modes, each internal buffer stores 64 samples at the incoming VI clock rate. The latency requirement is one request served every 64 VI clock cycles.

For the raw10 capture modes, each internal buffer stores 32 samples. Hence, the requirement for not losing samples is one request served every 32 VI clock cycles.

For a 38-MHz data rate on the incoming 10-bit samples and a TM1300 SDRAM clock speed of 100 MHz, highway latency should be set to guarantee less than 3200/38 = 842 ns (84 SDRAM clock cycles) per clock cycle. This cannot be met if any other peripherals are enabled.

Table 6-4 summarizes the maximum allowed highway latency (in SDRAM clock cycles) needed to guarantee that no samples are lost. The general formula uses 'F' to represent the VI clock frequency (in MHz).

Table 6-4. VI highway latency requirements (27-MHz data rate, 100-MHz TM1300 highway clock)

Mode	Max latency setting (27 MHz, 100 MHz)	Formula
fullres capture	237	6,400/F
halfres capture	474	12,800/F
raw8	237	6,400/F
raw10s	118	3,200/F
raw10u	118	3,200/F
message passing	237	6,400/F

In fullres mode, bandwidth requirements (in bytes) per video line with active image for VI is:

- B_{fullr} = ceil(WIDTH*2/256) * 4 * 64

ceil(X) function is the least integral value greater than or equal to X.

In halfres mode, the bandwidth is:

- B_{halfr} = ceil(WIDTH*2/512) * 4 * 64

Raw8 mode and message passing mode bandwidth depends only on VI clock speed. For raw10 mode each 10-bit value counts as 2 bytes for bandwidth computations.

Enhanced Video Out Chapter 7

by Marc Duranton, Dave Wyland, Gert Slavenburg

7.1 ENHANCED VIDEO OUT SUMMARY

The TM1300 Enhanced Video Out (EVO) improves on the design of the TM1000 Video Out (VO) unit while maintaining binary-compatibility. TM1300 EVO is fully backward compatible with TM1100, and has been extended to support byte data rates up to 81-MHz and improve the Genlock mode. The summary of new EVO features versus TM1000 includes:

- Internal clock generator (DDS) has reduced jitter
- Full alpha blending supports 129-levels
- Chroma keying
- Frame synchronization can be internally or externally generated (Genlock mode)
- External frame sync. follows the field number generated in the EAV/SAV code
- Programmable YUV output clipping
- Data-valid signal generated in data-streaming mode
- In message passing mode, message length can range from one word (4 bytes) up to 16 MB.

7.2 ABOUT THIS DOCUMENT

This chapter describes the TM1300 EVO unit which extends and improves the design of the TM1000 VO unit, and consolidates the changes introduced in the TM1100. Please refer to the TM1000 databook for a description of the VO unit's functionality.

7.3 BACKWARD COMPATIBILITY

The EVO is functionally compatible with the TM1000 VO unit. All TM1000 VO features are supported exactly in the same fashion by the TM1300 EVO. Software written for the TM1000 VO can control the TM1300 EVO without modification (with the exception of the Genlock mode which now requires EVO_CTL.GENLOCK to be set to 1 in addition to VO_CTL.SYNC_MASTER = 0).

All new features and improvements are selectively enabled by setting bits in the new EVO_CTL MMIO register, described in Section 7.16.4. A method to determine the existence of new EVO registers is given in Section 7.16.1.

The new TM1300 EVO features are disabled on hardware reset in order to remain hardware-compatible with the TM1000 VO. So it is assumed throughout this chapter that all new functions controlled by EVO_CTL are enabled by software. Any new software should use the new EVO modes. Please refer to the TM1000 databook for a description of the VO unit's funtionality.

7.4 FUNCTION SUMMARY

The TM1300 EVO generates and transmits continuous digital video images. It can connect to an off-chip video subsystem such as a digital video encoder chip (e.g., the Philips SAA7125 DENC digital encoder), a digital video recorder, or the video input of another TM1300 through a CCIR 656-compatible byte-parallel video interface. See Figure 7-1, Figure 7-1, and Figure 7-2.

The EVO can either supply video pixel clock and synchronization signals to the external interface or synchronize to signals received from the external interface (Genlock mode).

PAL, NTSC, 16:9 and other video formats including double pixel-rate, non-interlaced video formats are supported through programmable registers which control pixel clock frequency and video field or frame format.

The EVO can combine a background video image from SDRAM with an optional foreground graphics overlay image from SDRAM using 129-level, per-pixel alpha blending. The composite result is sent out as continuous video. Video image data is taken from a planar memory format, with separate Y, U and V planes in memory in YUV 4:2:2 or 4:2:0 format. The optional graphics overlay is taken from a pixel-packed YUV 4:2:2+α data structure in memory.

The EVO can also be used to stream continuous data (data-streaming mode) or send unidirectional messages (message-passing mode) from one TM1300 to another.

In data-streaming mode, the EVO generates a continuous stream of arbitrary byte data using internal or external clocking. Dual buffers allow continuous data streaming in this mode by allowing the DSPCPU to set up a buffer while another is being emptied by the EVO. Data-valid signals are generated on VO_IO1 and VO_IO2 to synchronize data streaming to other TM1300 data receivers.

In message-passing mode, unidirectional messages can be sent to the Video In (VI) port(s) of one or more TM1300s. Start and end-of-message signals are provided to synchronize message passing to other TM1300 message receivers.

7.4.1 Detailed Feature Descriptions

The EVO provides the following key functions.

- Continuous digital video output of PAL or NTSC format data according to CCIR 601.
- Transmission of YUV 4:2:2 co-sited pixel data across a standard 8-bit parallel CCIR 656[1] interface. Embedded SAV and EAV synchronization codes and separate sync control signals compatible with Philips DENC encoders are available.
- Supports the nominal PAL/NTSC data rate of 27 MB/sec (13.5 Mpix/sec), or any byte data rate up to an 81-MHz EVO clock.
- Custom video formats can be programmed with frames or fields of up to 4095 lines of up to 4095 pixels, subject only to the data rate limitation above.
- Support for video images in planar YUV 4:2:2 co-sited, planar YUV 4:2:2 interspersed, or planar YUV 4:2:0 memory formats.
- Optional 129-level alpha blending. Graphics overlay image is in pixel-packed YUV 4:2:2+α format, and is alpha blended on top of the video image. Each pixel has a 1-bit alpha, which selects one of two global 8-bit alpha values which provide 129 layers of transparency. With overlay enabled, the output byte data rate is limited to 45% of the SDRAM clock, or up to an 81-MHz EVO clock, whichever is smaller.
- Optional horizontal 2X upscaling of the video image for display. The overlay is always in display format.
- In data-streaming mode, the EVO acts as a high bandwidth continuous-output data channel. The byte data rate is limited to an 81-MHz EVO clock.
- In message-passing mode, the EVO can send messages from 1 word (4 bytes) up to 16 MB. The byte data rate is limited to an 81-MHz EVO clock.
- For diagnostic purposes, EVO output data can be internally looped back to the VI port. This is controlled by the VI DIAGMODE bit.

7.4.2 Summary of Operation

The EVO normally supplies continuous video data to its outputs. The EVO is programmed and started by the TM1300 DSPCPU. The EVO issues an interrupt to the DSPCPU at the end of each transmitted field, and/or at a programmable vertical position in the field. The DSPCPU updates the EVO video image data pointers with pointers to the next field during the vertical blanking interval so as to maintain continuous video output. During video output, the EVO supplies embedded CCIR 656 SAV (Start Active Video) and EAV (End Active Video) sync codes and optionally supplies horizontal and frame sync signals. The EVO can either supply pixel clock and horizontal and frame timing signals or it can lock to external timing signals such as those supplied by a Philips SAA7125 DENC digital encoder or similar sync source.

1. Refer to CCIR recommendation 656: Interfaces for digital component video signals in 525 line and 625 line television systems. Recommendation 656 is included in the Philips Desktop Video Data Handbook.

7.5 INTERFACE

Table 7-1 lists the interface pins of the EVO unit. Figure 7-1, Figure 7-1, and Figure 7-2 illustrate typical connections for commonly-used external devices that interface to the EVO.

The most common way to generate analog video is shown in Figure 7-1. In this setup, an SAA7125 Digital Encoder (DENC) can be programmed to derive sync either from the VO_DATA stream EAV/SAV codes, or from its RCV1/2 pins.

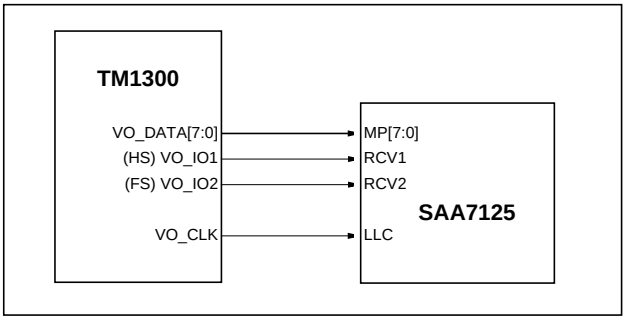


Figure 7-1. EVO connected to a digital video encoder (DENC).

Figure 7-2 illustrates how a byte-parallel ECL-level standard CCIR 656 interface can be created. In certain professional applications, serial D1 video is also used. In that case, the EVO can be connected to a Gennum GS9022 Digital Video Serializer or similar part (not shown).

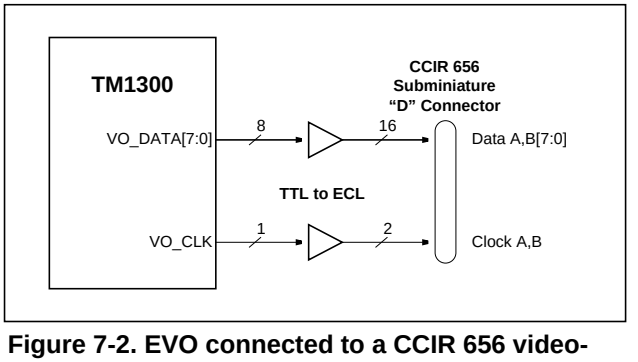


Figure 7-2. EVO connected to a CCIR 656 video-output connector.

Figure 7-3 shows the EVO unit of one TM1300 connected to the VI unit of a second TM1300.

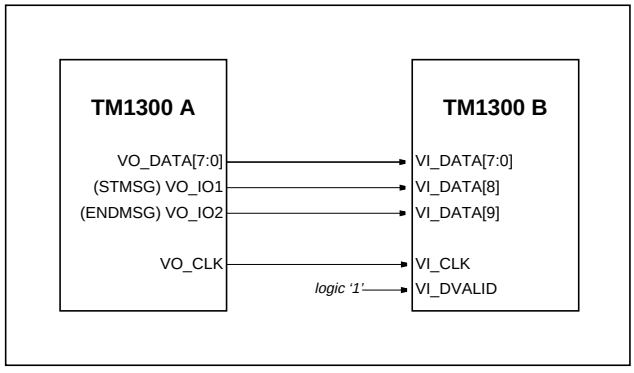


Figure 7-3. EVO unit connected to the VI unit of a second TM1300.

Table 7-1. EVO unit interface pins

Signal Name	Type	Description
VO_DATA[7:0]	OUT	CCIR 656-style YUV 4:2:2 digital output data, or general-purpose high speed data output channel. Output changes on positive edge of VO_CLK.
VO_IO1	I/O-5	Horizontal Sync (HS) output or Start Message (STMSG) output. See Figure 7-18.
VO_IO2	I/O-5	Frame Sync (FS) input, FS output or ENDMSG output. • If set as FS input, it can be set to respond to positive or negative edge transitions. • If the EVO operates in Genlock mode and the selected transition occurs, the EVO sends two fields of video data. • In message-passing mode, this pin acts as the ENDMSG output. See Figure 7-18.
VO_CLK	I/O-5	The EVO unit emits VO_DATA on a positive edge of VO_CLK. VO_CLK can be configured as an input (the hardware reset default) or output. • If configured as an input, VO_CLK is received from external display-clock master circuitry. • If configured as output, the TM1300 emits a low-jitter clock frequency programmable between approx. 4 and 81 MHz.

7.6 BLOCK DIAGRAM

Figure 7-4 shows a block diagram of the EVO unit. It consists of a clock generator, a video frame timing generator and an image or data generator. The image generator produces either a CCIR 656 digital video data stream with optional YUV overlay or a continuous-data or message-data stream. It also performs optional format conversion and optional 2:1 horizontal scaling.

The frame timing generator provides programmable image timing including horizontal and vertical blanking,

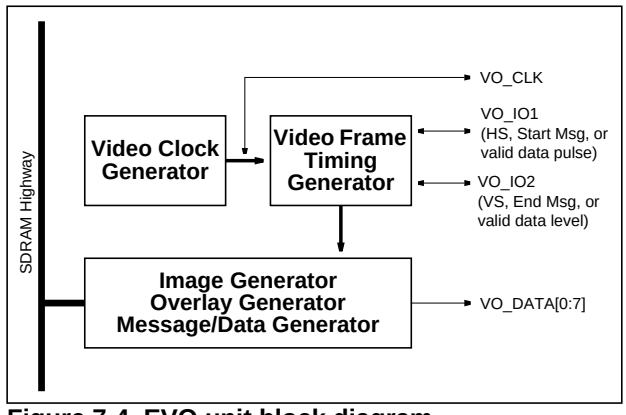


Figure 7-4. EVO unit block diagram.

SAV and EAV code insertion, overlay start and end timing, and horizontal and frame timing pulses. It also supplies data-valid timing signals in data-streaming mode and start-of-message and end-of-message timing signals in message-passing mode. The sync timing pulses can be generated by the frame timing unit, or the frame timing unit can be driven by externally-supplied sync timing pulses, when VO_CTL.SYNC_MASTER = 0 and EVO_CTL.GENLOCK = 1.

The video clock generator produces a programmable video clock. The video clock generator can supply the video clock for the frame timing generator and external devices, or it can be driven by an external clock signal.

7.7 CLOCK SYSTEM

Positive edges of VO_CLK drive all EVO output events. A block diagram of the EVO clock system is shown in Figure 7-5. The EVO clock is either supplied externally or internally generated by the EVO, as controlled by the VO_CTL.CLKOUT bit. When CLKOUT = 0, the EVO clock is supplied by an external source through the VO_CLK pin as an input. This is the default mode, entered at hardware reset. When CLKOUT = 1, an internal clock generator supplies the EVO clock and drives the VO_CLK pin as an output.

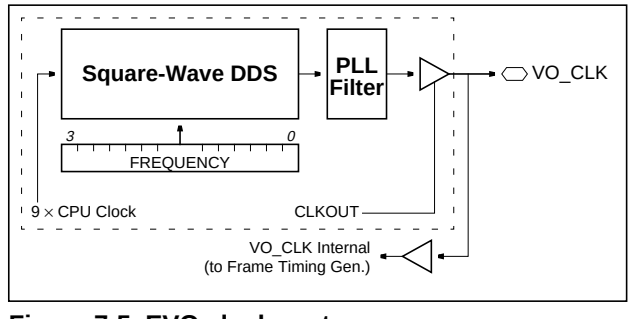


Figure 7-5. EVO clock system.

The internal clock generator system is a square wave Direct Digital Synthesizer (DDS) which can be programmed to emit frequencies from 1 Hz to 50 MHz. The output of the DDS is sent to a phase-locked loop filter (PLL) which removes clock jitter from the DDS output signal. The PLL can also be used to divide or double the DDS frequency. The PLL VCO operates from 8-MHz to

90 MHz. The PLL is enabled and programmed as described in Section 7.18.

DDS clock rate is set by the VO_CLOCK.FREQUENCY field according to the equation shown in Figure 7-6. The VO_CLK frequency can be a divider or multiplier of f_{DDS}, as determined by the PLL subsystem settings.

$$FREQUENCY = 2^{31} + \frac{f_{DDS} \cdot 2^{32}}{9 \cdot f_{DSPCPU}}$$

Figure 7-6. DDS low-jitter oscillator frequency.

Low-jitter clock mode is automatically entered whenever FREQUENCY[31] = 1. If FREQUENCY[31] = 0, the DDS operates at 1/3 the rate (for compatibility with TM1000 code), and FREQUENCY must be set as shown in Figure 7-7.

$$FREQUENCY = \frac{f_{DDS} \cdot 2^{32}}{3 \cdot f_{DSPCPU}}$$

Figure 7-7. DDS slow speed oscillator frequency

7.8 IMAGE TIMING

The EVO emits a serial byte-data stream used by CCIR 656 devices to generate a displayed image. Figure 7-8 shows an NTSC-compatible, 525-line interlaced image. The field and line numbers are shown for reference.

Interlaced images are generated by the display hardware by controlling the vertical retrace timing. For reference, Figure 7-9 shows a timing diagram of NTSC-compatible interlaced frame timing illustrating the analog vertical retrace signal. The vertical retrace signal for the second field begins in the middle of the horizontal line that ends the first field. This causes the first line of the second field to begin halfway across the display screen and the lines of the second field to be scanned between the lines of the first field, resulting in an interlaced display.

The analog timing required to generate the interlaced signal is supplied by the display device. The CCIR 656 digital video signals generated by the EVO use frame synchronization timing and do not generate any vertical retrace timing.

7.8.1 CCIR 656 Pixel Timing

The EVO generates pixels according to CCIR 656 timing in YUV 4:2:2 co-sited format and outputs these pixels as shown in Figure 7-10. Pixels are generated in groups of two, with four bytes per two pixels. Each pair of pixels has two luminance bytes (Y0, Y1) and one pair of chrominance bytes (U0, V0) arranged in the sequence shown. The chrominance samples U0 and V0 are sampled spatially co-sited with luminance sample Y0. For PAL or NTSC video, pixels are generated at a nominal rate of 13.5 Mpix/sec (27 MB/sec). Pixels are clocked out on the positive edge of VO_CLK.

7.8.2 CCIR 656 Line Timing

The CCIR 656 line timing is shown in Figure 7-11. Each line begins with an EAV code, a blanking interval and an SAV code, followed by the line of active video. The EAV code indicates end of active video for the previous line, and the SAV code indicates start of active video for the current line.

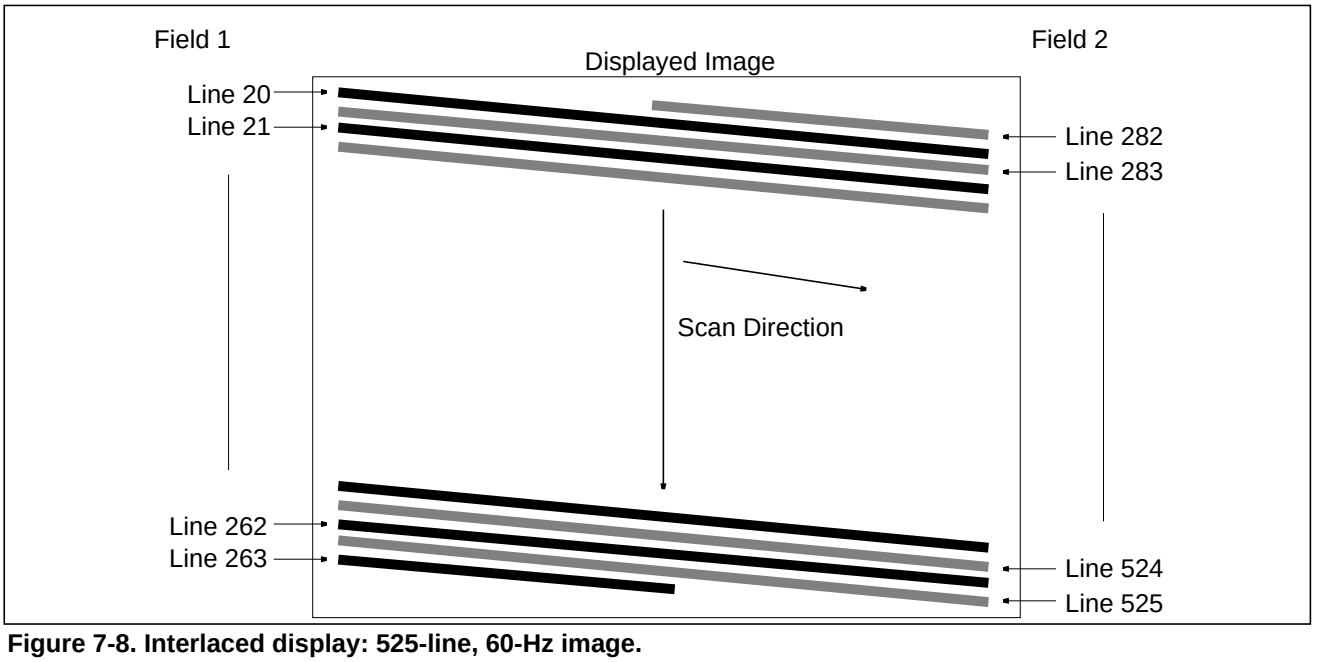


Figure 7-8. Interlaced display: 525-line, 60-Hz image.

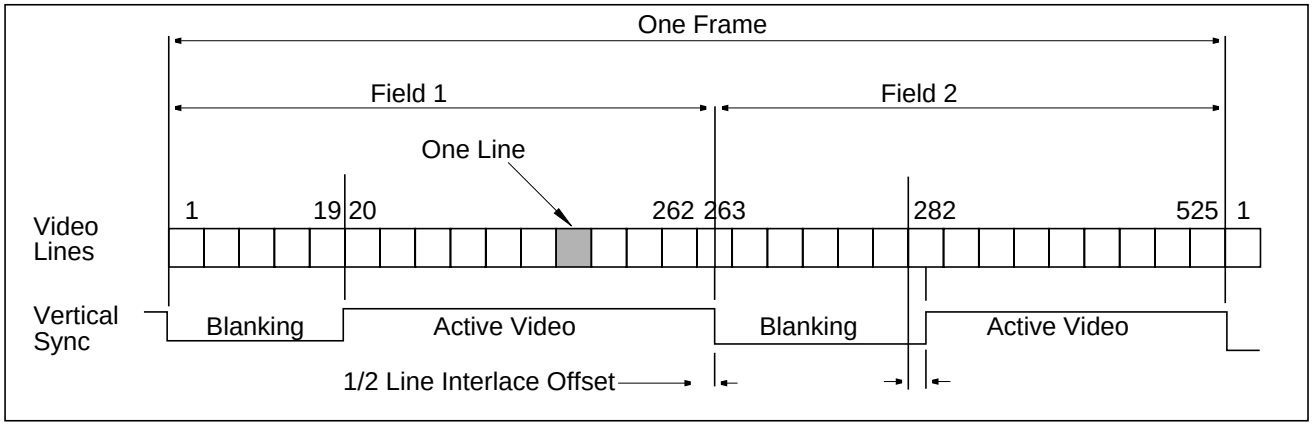


Figure 7-9. Interlaced timing—NTSC analog sync. signals.

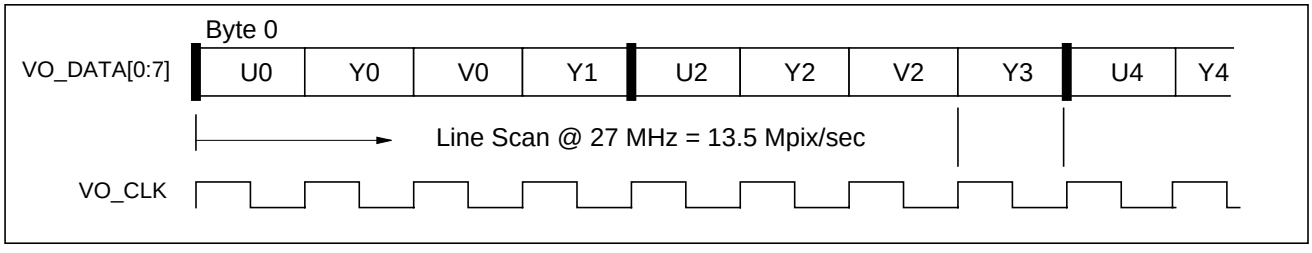


Figure 7-10. CCIR 656 pixel timing.

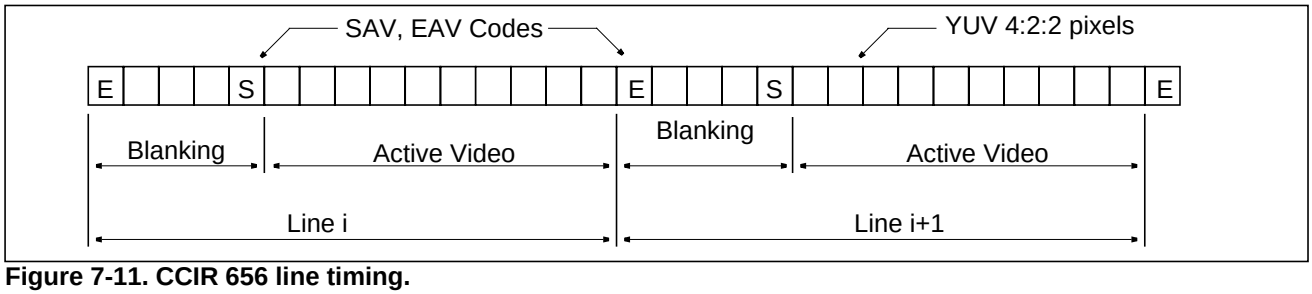


Figure 7-11. CCIR 656 line timing.

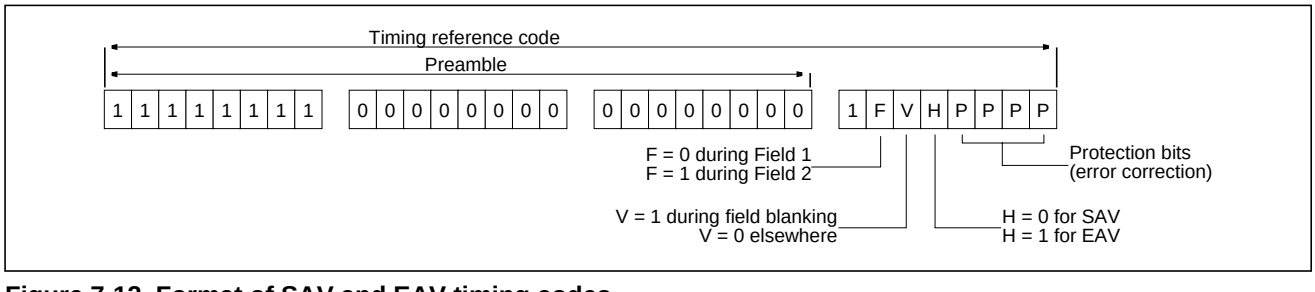


Figure 7-12. Format of SAV and EAV timing codes.

7.8.3 SAV and EAV Codes

The End Active Video (EAV) and Start Active Video (SAV) codes are issued at the start of each video line. EAV and SAV codes have a fixed format: a 3-byte preamble of 0xFF, 0x00, 0x00 followed by the SAV or EAV code byte. The EAV and SAV code byte format is shown in Figure 7-12 for reference. The EAV and SAV codes define the start and end of the horizontal blanking interval, and they also indicate the current field number and the vertical blanking interval.

The SAV and EAV codes have a 4-bit protection field to ensure valid codes. The EVO generates these protection bits as part of the SAV and EAV codes as defined by CCIR 656. There are 8 possible valid SAV and EAV codes shown with their correct protection bits in Table 7-2. The EVO generates SAV and EAV sync codes and inserts them into the video out data stream according to the CCIR 656 specification under all conditions, whether it is generating or receiving horizontal and frame timing information.

7.8.4 Video Clipping

SAV and EAV codes are identified by a 3-byte preamble of 0xFF, 0x00 and 0x00. This combination must be

Table 7-2. SAV and EAV codes

Code	Binary Value	Field	Vertical Blanking
SAV	1000 0000	1	
EAV	1001 1101	1	
SAV	1010 1011	1	X
EAV	1011 0110	1	X
SAV	1100 0111	2	
EAV	1101 1010	2	
SAV	1110 1100	2	X
EAV	1111 0001	2	X

avoided in the video data output by the EVO to prevent accidental generation of an invalid sync code. The EVO provides programmable maximum and minimum value clipping on the video data to prevent this possibility. If clipping is enabled, the EVO automatically clips the resulting image data as described in Section 7.15.3.

7.8.5 CCIR 656 Frame Timing

The interlaced frame timing defined by CCIR 656 is shown in Table 7-3. Lines are numbered from 1 to 525 for 525-line, 60-Hz systems and from 1 to 625 for 625-line, 50-Hz systems. The Field and Vertical Blanking columns indicate whether the field and vertical blanking bits, respectively, are set in the SAV and EAV codes for the indicated lines. The 525 and 625 formats have similar timing but differ in their line numbering.

Table 7-3. CCIR 656 frame timing

Line Number		F bit	V bit	Comments
525/60	625/50			
1–3	624–625	1	1	Vertical blanking for Field 1, SAV/EAV code still indicates Field 2
4–19	1–22	0	1	Vertical blanking for Field 1, change SAV/EAV code to Field 1
20–263	23–310	0	0	Active video, Field 1
264–265	311–312	0	1	Vertical blanking for Field 2, SAV/EAV code still indicates Field 1
266–282	313–335	1	1	Vertical blanking for Field 2, change SAV/EAV code to Field 2
283–525	336–623	1	0	Active video, Field 2

7.9 ENHANCED VIDEO OUT TIMING GENERATION

The EVO generates timing for frames, active video areas within frames, images within the active video area, and overlays within the image area. The relationship between these four is shown in Figure 7-13. The frame includes the timing for both interlaced fields. Progressive scan, or non-interlaced video, is accomplished by setting the timing parameters such that two identical successive fields are generated.

7.9.1 Active Video Area

Shown in Figure 7-13, the active video area begins after the horizontal and vertical blanking intervals and represents the pixels visible on the screen. The image area is the actual displayed image within the active video area. It can be slightly smaller than the active video area to avoid edge effects at the top, bottom and sides of the image. The overlay area is within the image area.

The EVO uses counters to generate and control image timing. The Frame Line Counter and Frame Pixel Counter control the overall timing for the frame and define the total number of pixels per line, lines per frame, and interlace timing, including horizontal and vertical blanking intervals.

Note that the Frame Line Counter has a starting value of one, not zero, and it counts from 1 to 525 or 625, consistent with CCIR 656 line numbering. The Image Line

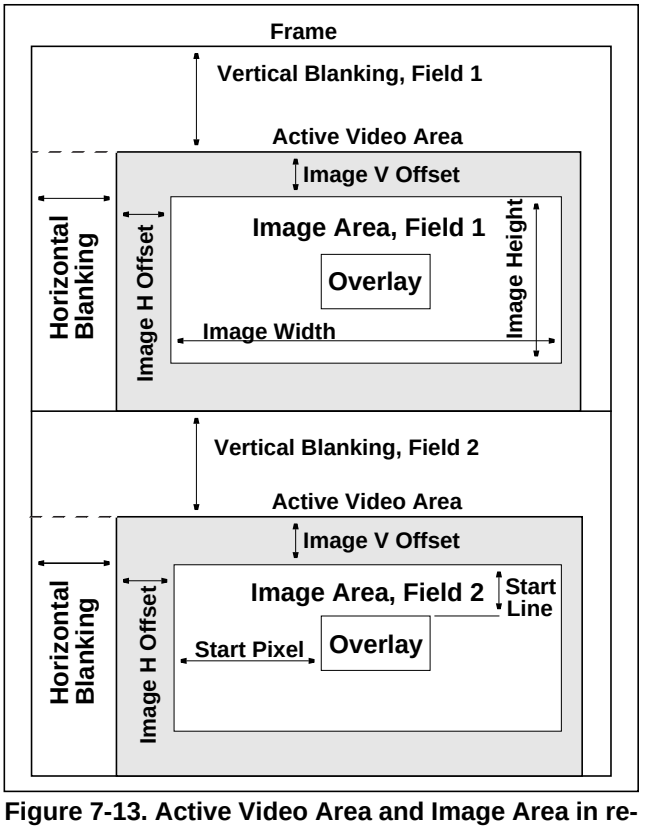


Figure 7-13. Active Video Area and Image Area in relation to vertical and horizontal blanking intervals.

Counter and Image Pixel Counter define the visible image within the field.

The geometry of the active video area is defined by the contents of several MMIO registers shown in Figure 7-29. The VO_FRAME.FIELD_2_START field defines the start line of Field 2. Field 2 is active when the Field Line Counter contents equal or exceed this value. The active video area is defined by the F1_VIDEO_LINE and F2_VIDEO_LINE fields of the VO_FIELD register for each field of the frame, and by the VIDEO_PIXEL_START field of the VO_LINE register for each line of the frame. The active video area begins when the contents of the Frame Line Counter and Frame Pixel Counter equals or exceeds these values.

7.9.2 SAV and EAV Overlap Period

The CCIR 656-compliant 525/60 and 625/50 timing specifications define an overlap period where the field number in the SAV and EAV codes from Field 1 persists into the vertical blanking interval for Field 2, and the codes for Field 2 persist into the vertical blanking interval for Field 1. The F1_OLAP and F2_OLAP fields of the VO_FIELD register define these overlap intervals.

F1_OLAP and F2_OLAP are small two's complement values in the range -8... +7. A positive value indicates that the overlap extends into the current field, while a negative value indicates that it extends backward into the previous field. See Figure 7-30 for the effect of negative and positive values.

During the overlap interval, the vertical blanking for the next field has begun; however, the field number flag in the SAV and EAV codes still shows the field number for the previous field. The field number is updated to the correct field value at the end of the overlap interval.

F1_OLAP defines the overlap from Field 1 to Field 2. This overlap occurs during the beginning of vertical blanking for Field 2. The SAV and EAV codes continue to show Field 1 during this overlap interval, and they change to Field 2 at the end of the interval.

F2_OLAP defines the overlap from Field 2 to Field 1. This overlap occurs during the beginning of vertical blanking for Field 1. The SAV and EAV codes continue to show Field 2 during this overlap interval, and they change to Field 1 at the end of the interval.

7.9.3 Control of Frame and Image Counters

The frame and image counters have different start and stop points. The frame counters begin in the vertical blanking interval of the first field and the horizontal blanking interval of the first line. They stop counting when they reach the height and width values of the frame. When the EVO generates frame timing, the frame counters are reset to their start values when they reach their stop values. When the EVO receives frame timing signals, the frame counters continue counting until reset by the external signals.

The image area is defined by VO_YTHR register fields IMAGE_VOFF and IMAGE_HOFF. These values are added to the F1_VIDEO_LINE or F2_VIDEO_LINE and VIDEO_PIXEL_START values to define the starting line and pixel, respectively, of the image area. The image area is active when the contents of the Frame Line Counter and Frame Pixel Counter equal or exceed these values.

The Image Line Counter and Image Pixel Counter start counting at the first active pixel in the image area and the first active line in the image area, respectively. The image counters start at zero and stop counting when they reach their image height and width values. The image counters are reset by frame counter values indicating the start of the image pixel in a line and the start of the image line in a field.

The image counters define the active image area of the frame, the area of interest for image processing. This allows the overlay start address to be defined relative to the active image area, for example. When the EVO is not sending out active pixels from the image area, it sends out blanking codes. The blanking codes are 0x80, 0x10, 0x80, and 0x10 for each 2-pixel group in YUV 4:2:2 image data format, as defined by CCIR 656 and shown in Figure 7-10.

7.9.4 Horizontal and Frame Timing Signals

The EVO can supply horizontal and frame timing signals or receive a frame timing signal from an external source. When VO_CTL.SYNC_MASTER = 1, the EVO generates horizontal and frame timing for the external video device. When SYNC_MASTER = 0, the EVO operates in Genlock mode and an external device, such as a DENC, must provide frame sync. This section describes EVO operation when it is sync master. See Section 7.10 for a description of Genlock mode.

If SYNC_MASTER = 1, the VO_IO1 signal generates a horizontal timing signal, and the VO_IO2 signal generates a frame timing signal. When EVO_ENABLE = 1 and FIELD_SYNC = 1, the VO_IO2 signal indicates the field number (low = Field 1, high = Field 2), according to the SAV/EAV field indication (bit[6]) as shown in Figure 7-14. The VO_IO2 signal toggles just before the first byte of the preamble that protects the EAV code and after the SAV code. Non-interlaced output can be simulated by programming the EVO to generate fields equivalent to the desired frames. In this case, VO_IO2 indicates odd or even frames.

The horizontal timing signal VO_IO1, shown in Figure 7-15, corresponds to the horizontal-blanking interval. It is active low from the EAV code at the start of the line to the SAV code at the start of active video for the line.

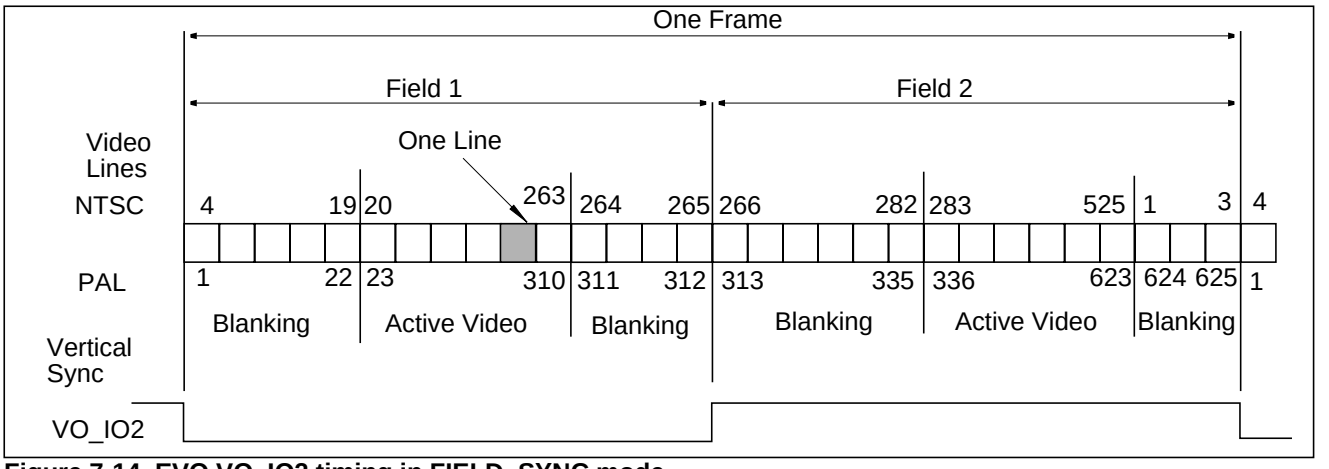


Figure 7-14. EVO VO_IO2 timing in FIELD_SYNC mode.

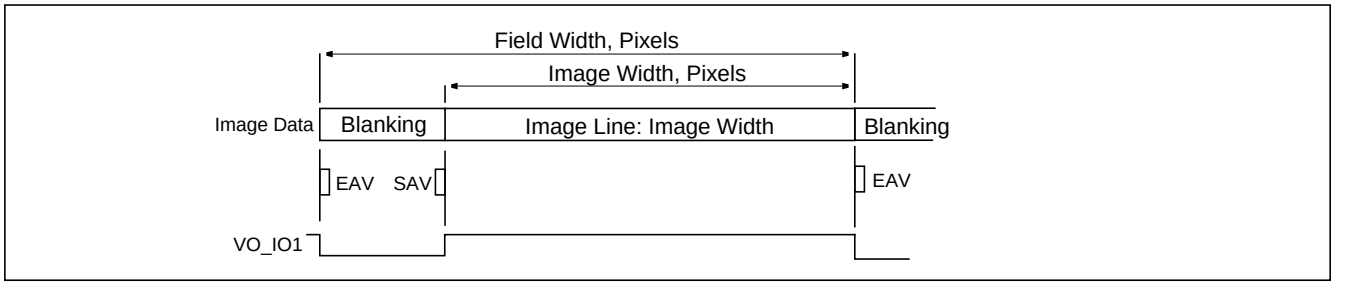


Figure 7-15. EVO VO_IO1 timing in FIELD_SYNC mode.

7.10 GENLOCK MODE

In Genlock mode, the EVO is not synchronization master but receives frame timing signals on VO_IO2. The EVO operates in Genlock mode when SYNC_MASTER = 0, EVO_CTL.EVO_ENABLE = 1 and EVO_CTL.GENLOCK = 1.

The active edge can be programmed using the VO_CTL.VO_IO2_POS bit. The initial transition of the frame timing signal on VO_IO2 causes the Frame Line Counter to be set to the value in VO_FRAME.FRAME_PRESET. After reaching FRAME_LENGTH, the Frame Line Counter starts counting again from 1.

EVO_SLVDLY.SLAVE_DLY is typically used to compensate for any delay in the frame timing source or internal pipeline synchronization anywhere in a line. Internally, the active edge of VO_IO2 is delayed by SLAVE_DLY VO_CLK clock cycles. Typically, it will allow FRAME_PRESET to be loaded at the beginning of a new line.

With correct values of SLAVE_DLY and FRAME_PRESET loaded, the TM1300 can generate frames totally synchronized with the active edge of VO_IO2. All the internal MMIO registers (except of course VO_CTL) should be programmed with the same values as for SYNC_MASTER mode. See Figure 7-16.

In Genlock mode, the EVO is free-running according to the values programmed in its internal registers before the initial VO_IO2 active edge. Just after receiving the active edge that will synchronize the EVO, output values may be erroneous for several VO_CLK cycles, but it is guaranteed that the next frame will be correct.

After the first synchronizing edge, if the next one happens according to the values programmed in the EVO MMIO registers, no change will appear in the output timing of the EVO. If the active edge of VO_OI2 does not match the programmed value, a new synchronization phase is performed.

Typically, this is programmed as follows: SLAVE_DLY is loaded with the number of clock cycles for one video line minus the number of delay cycles used by the EVO to synchronize itself. FRAME_PRESET is programmed with the value 2. With this programming, the active edge of VO_IO2 will happen just before the first byte (preamble) of the first line.

The first active edge of VO_IO2 is delayed internally by SLAVE_DLY VO_CLK cycles so that it appears internally just before the start of the second line minus the internal EVO pipeline delay. After this internal pipeline delay, the line counter is loaded by FRAME_PRESET, (‘2’), and the EVO starts sending data for line 2.

For the next frame, if the internal EVO programming matches the VO_IO2 timing, the EVO will appear to start the first byte of the first line just after the VO_IO2 active signal.

7.11 DATA TRANSFER TIMING

In data-streaming and message-passing modes, the EVO supplies a stream of 8-bit data. No data selection or

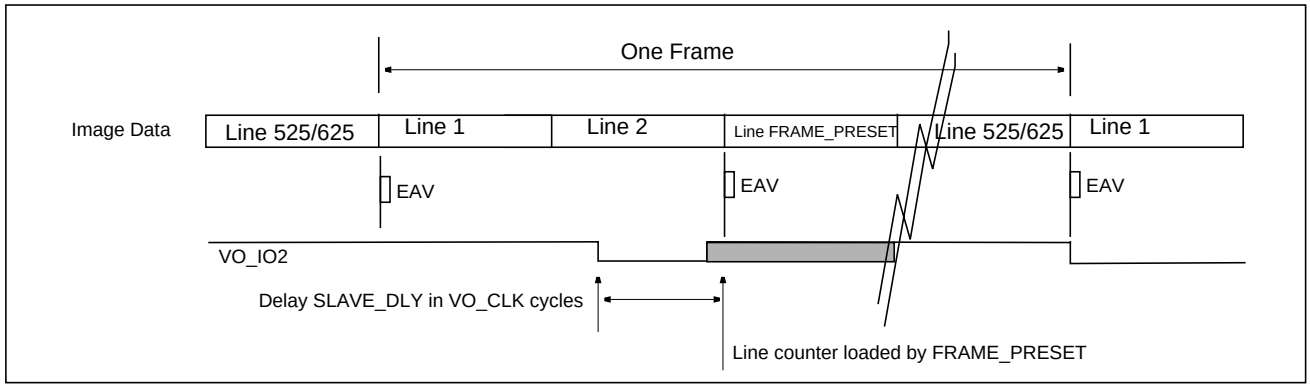


Figure 7-16. Genlock mode.

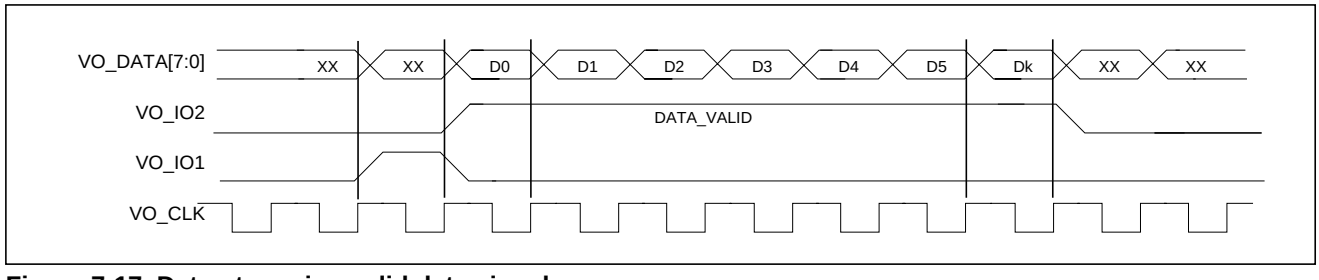


Figure 7-17. Data-streaming valid data signals.

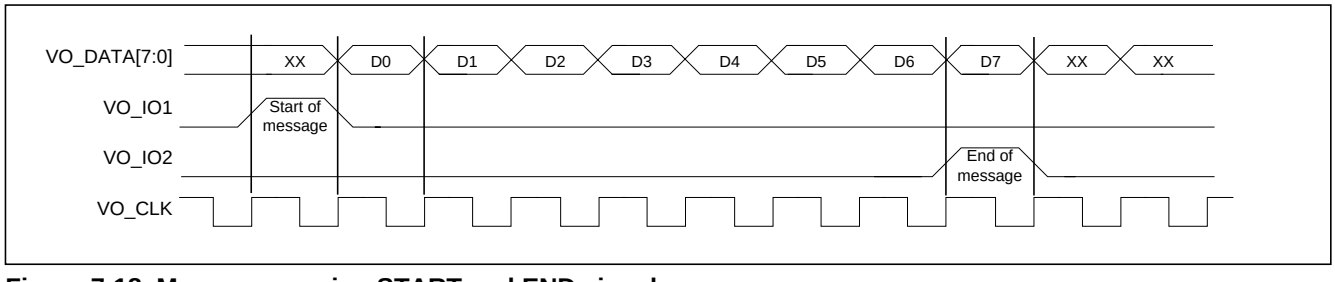


Figure 7-18. Message-passing START and END signals.

data interpretation is done, and data is transferred at the rate of one byte per VO_CLK. Data is clocked out on the positive edge of VO_CLK.

When data-streaming mode is enabled and EVO_ENABLE = 1 and SYNC_STREAMING = 1, the VO_IO2 signal indicates a data-valid condition. This signal is asserted when the EVO starts outputting valid data (that is, data-streaming mode is enabled and video out is running), and is de-asserted when data-streaming mode is disabled. As shown in Figure 7-17, the data-valid signal on VO_IO2 is asserted just before the first valid byte is present on VO_DATA[7:0], and is de-asserted just after the last valid byte was sent, or if an HBE error is signaled. All transitions of VO_IO2 occur on the rising edge of VO_CLK. The VO_IO1 signal generates a pulse one VO_CLK cycle before the first valid data is sent. The transitions of VO_IO1 occur on the rising edge of VO_CLK and last for one VO_CLK cycle.

In message-passing mode, the EVO issues signals on VO_IO1 and VO_IO2 to indicate the start and end of messages.

When message passing is started by setting VO_CTL.VO_ENABLE, the EVO sends a Start condition on VO_IO1. When the EVO has transferred the contents of the buffer, it sends an End condition on VO_IO2, sets BFR1_EMPTY, and interrupts the DSPCPU. The EVO stops, and no further operation takes place until the DSPCPU sets VO_ENABLE again to start another message, or until the DSCPU initiates other EVO operation. The timing for these signals is shown in Figure 7-18.

7.12 IMAGE DATA MEMORY FORMATS

7.12.1 Video Image Formats

The EVO accepts memory-resident video image data in three formats: YUV 4:2:2 co-sited, YUV 4:2:2 interspersed, and YUV 4:2:0. These formats are shown in Figure 7-19 through Figure 7-21.

7.12.2 Planar Storage of Video Image Data in Memory

Video image data is stored in memory with one table for each of the Y, U and V components. This is called planar format. This is shown in Figure 7-22 for YUV 4:2:2 image data. The EVO merges bytes from each of the three ta-

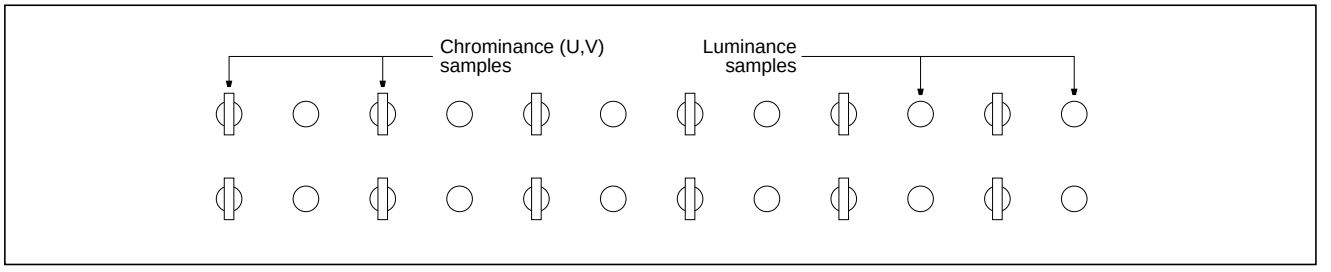


Figure 7-19. YUV 4:2:2 co-sited format.

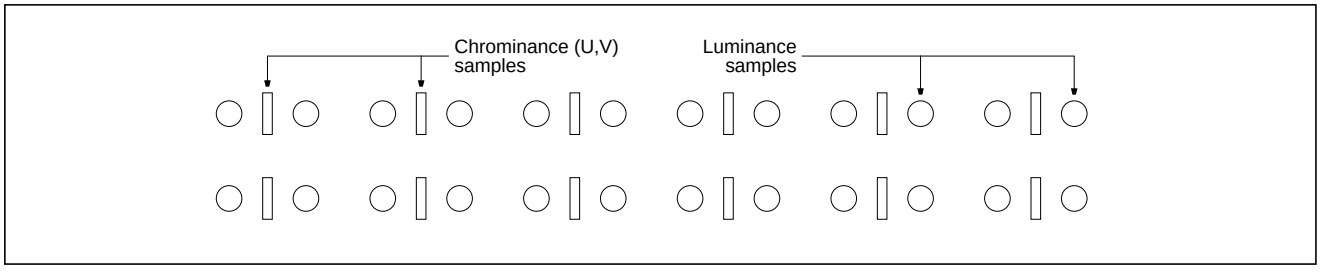


Figure 7-20. YUV 4:2:2 interspersed format.

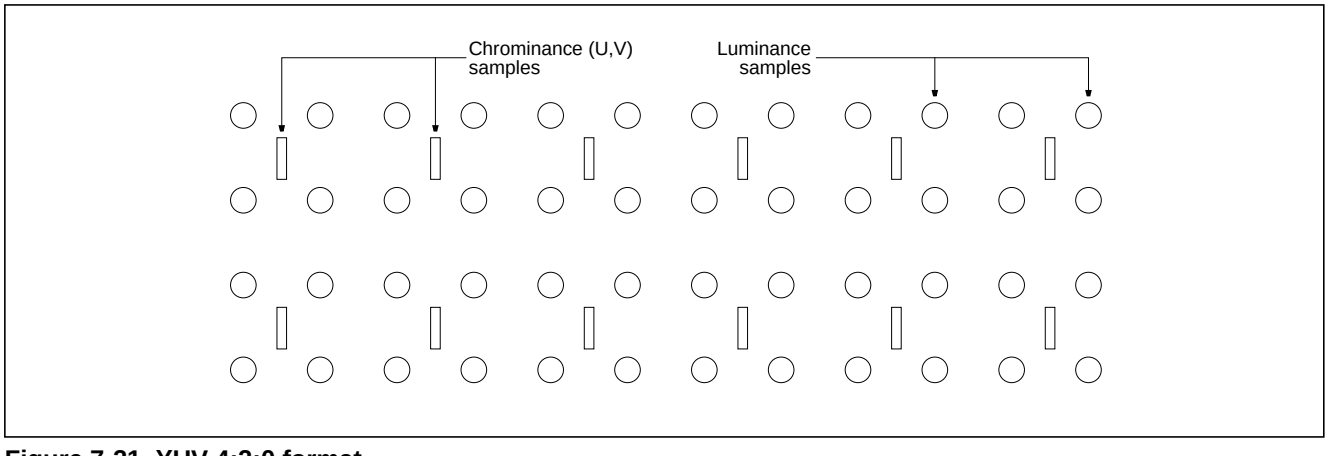


Figure 7-21. YUV 4:2:0 format.

bles to generate the CCIR 656-compatible output data. The U and V tables have the same number of lines but half the number of pixels per line as the Y table. The transfer is the same for YUV 4:2:0 format except the U and V tables will be 1/4 the size of the Y table. The U and V tables have the half the number of lines and half the number of pixels per line as the Y table.

7.12.3 Graphics Overlay Image Format

Graphics overlay image data is stored in a pixel-packed format in SDRAM. Graphics images are stored in YUV 4:2:2+alpha format. Figure 7-23 shows this format. The YUV overlay area is always within the image output resolution. The EVO does not upscale the graphics overlay image. If the EVO is upscaling the video image by 2×, the graphics overlay must be provided in upscaled format. Pixel data is a 16-bit data and follows endian-ness conventions based on 16-bit data. Refer to Appendix C, “Endian-ness” for details.

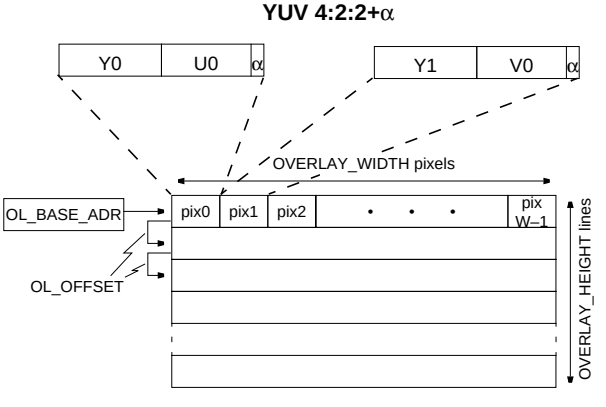


Figure 7-23. YUV 4:2:2+alpha overlay format.

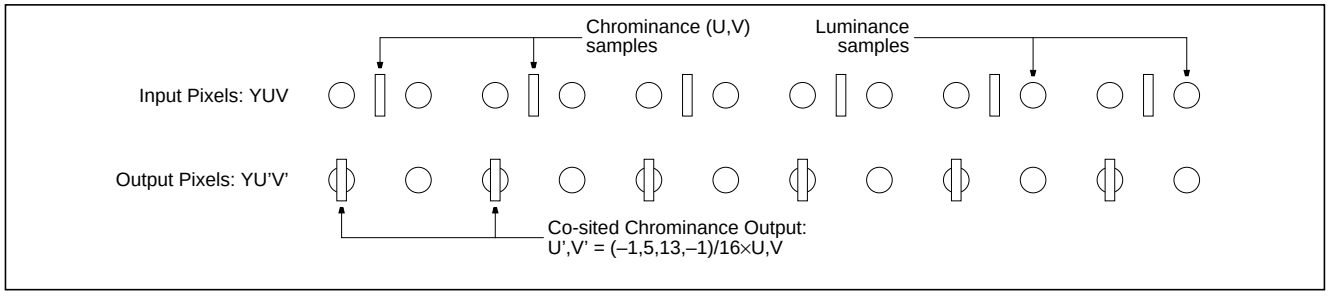


Figure 7-24. YUV interspersed to co-sited conversion.

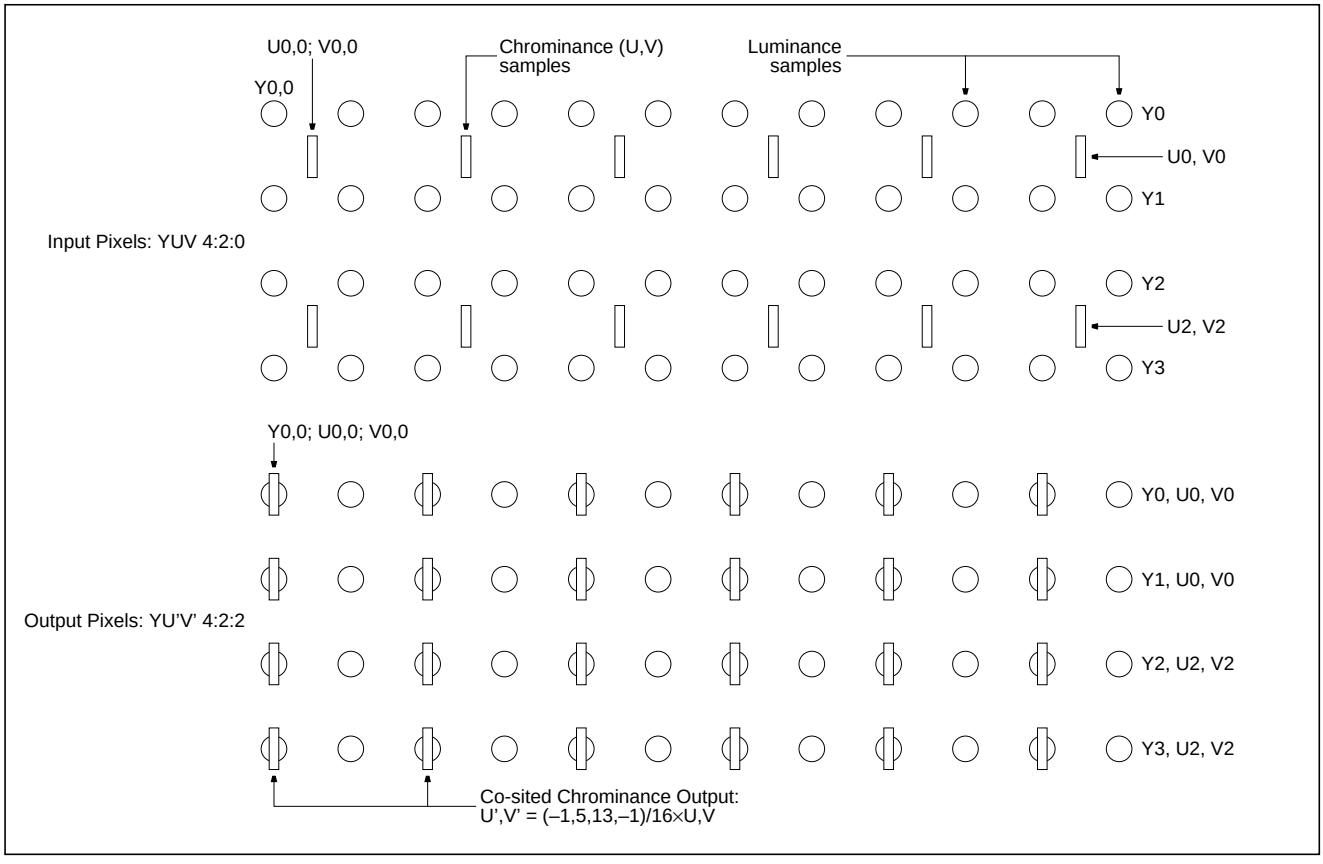


Figure 7-25. YUV 4:2:0 to YUV 4:2:2 co-sited conversion.

7.13 VIDEO IMAGE CONVERSION ALGORITHMS

The memory video image data formats are converted to the output YUV 4:2:2 co-sited format and optionally up-scaled 2× horizontally. The conversion algorithms are detailed below.

7.13.1 YUV 4:2:2 Interspersed to YUV 4:2:2 Co-sited Conversion

The EVO accepts data from SDRAM in either YUV 4:2:2 co-sited, YUV 4:2:2 interspersed, or YUV 4:2:0 interspersed formats. If the input data is in YUV 4:2:2 or YUV 4:2:0 interspersed format, interspersed-to-co-sited conversion is performed to generate co-sited output. The EVO uses a 4-tap, (–1, 5, 13, –1)/16 filter to perform this conversion on the U and V chroma data. Figure 7-24 shows an example of interspersed to co-sited conversion.

7.13.2 YUV 4:2:0 to YUV 4:2:2 Co-sited Conversion

YUV 4:2:0 to YUV 4:2:2 conversion is a variation of YUV 4:2:2 interspersed-to-co-sited conversion. The YUV 4:2:0 format has the U and V pixels positioned between lines as well as between pixels within each line. It also has half the number of U and V pixels compared to YUV 4:2:2 formats. The EVO converts YUV4:2:0 to YUV 4:2:2 co-sited by using the U and V chrominance pixel values for both surrounding lines and converting the resulting U and V pixels from interspersed to co-sited format. This is shown in Figure 7-25. For true vertical re-sampling of U and V, the TM1300 ICP unit can be invoked on U and V to convert from YUV 4:2:0 to YUV 4:2:2 interspersed.

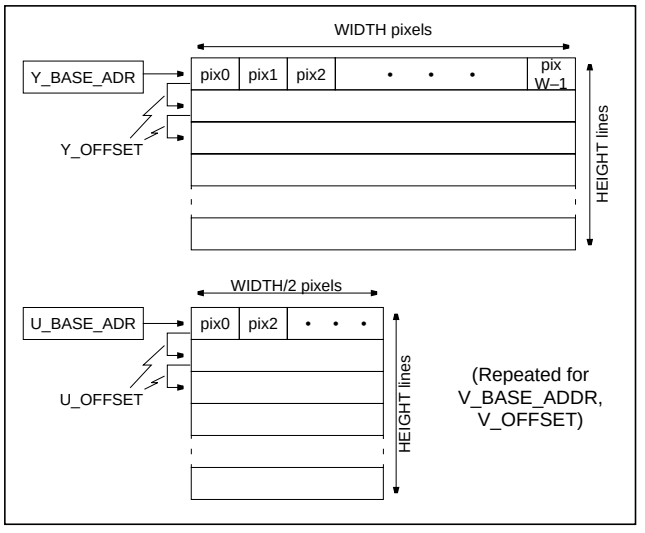


Figure 7-22. Image storage in planar memory format for YUV 4:2:2.

7.13.3 YUV-2x Upscaling

In the YUV-2× modes, the EVO performs 2× horizontal upscaling of the YUV data from SDRAM. No vertical upscaling is performed. The width of the result image (IMAGE_WIDTH) should be an even number. Upscaling is performed by 4-tap filtering. For all 3 memory formats, Y luminance data is upscaled using a (–3,19,19,–3)/32 filter to generate the missing output pixels. Output pixels at the same location as the input pixels use the corresponding input pixel values, as shown in Figure 7-26.

The U and V chrominance values are generated in the same way as the Y luminance signal for 2× upscaling, assuming that both the input and output use YUV 4:2:2 co-sited chrominance coding. The U and V output pixels at the same location as the U and V input pixels use the corresponding input pixel values. The U and V output pixels between the U and V input pixels are generated using the (–3,19,19,–3)/32 filter, as shown in Figure 7-26.

If the input chroma is interspersed, a (–1,13,5,–1)/16 filter is used to generate the U and V output pixels that are displaced by half a Y pixel from the U and V input pixels, and a (–1,5,13,–1)/16 filter is used to generate the additional upscaled U and V output pixels that are displaced by 1.5 pixels from the U and V input pixels. This is shown in Figure 7-27.

7.13.4 Pixel Mirroring for Four-tap Filters

The EVO uses a 4-tap filter for upscaling and for converting from interspersed to co-sited format. One extra pixel is needed at the beginning and two at the end of each line processed by this filter. These pixels are supplied automatically by mirroring the first and last pixels of each line. For example:

- Output pixel 1 uses input pixel 1 to generate its value. (same location, no filtering).
- Output pixel 2 uses pixels 1,1, 2 and 3 to generate its value.
- Output pixel 3 uses pixel 2 to generate its value.
- Output pixel 4 pixel uses pixels 1, 2, 3 and 4, etc.
- ...
- Output pixel 2N–2 uses pixels N–2, N–1, N, and N–1 to generate its value.
- Output pixel 2N–1 uses pixel N to generate its value.
- Output pixel 2N uses pixels N–1, N, N, and N–1 to generate its value.

Figure 7-28 shows an example of six pixels upscaled to 12 pixels.

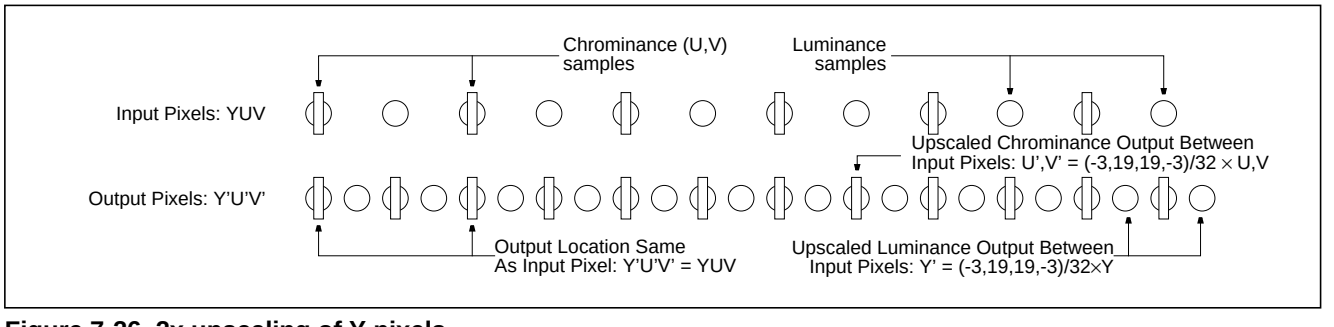


Figure 7-26. 2x upscaling of Y pixels.

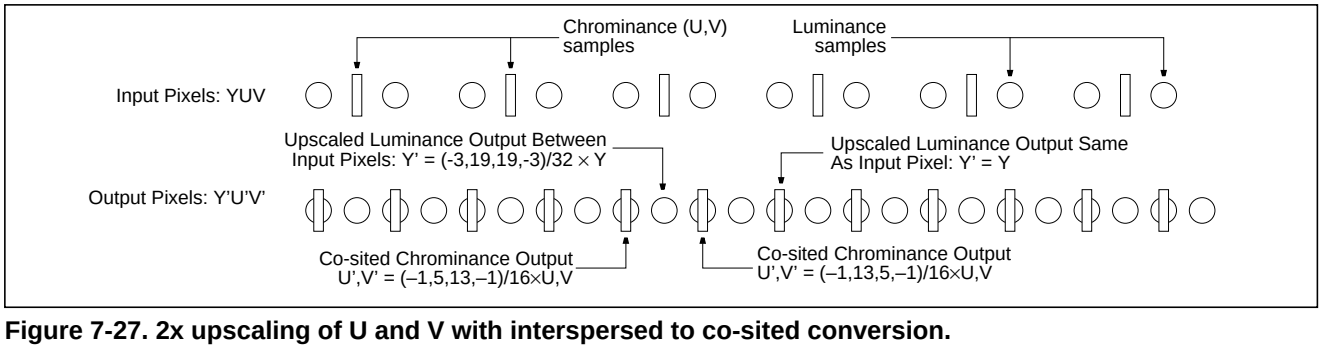


Figure 7-27. 2x upscaling of U and V with interspersed to co-sited conversion.

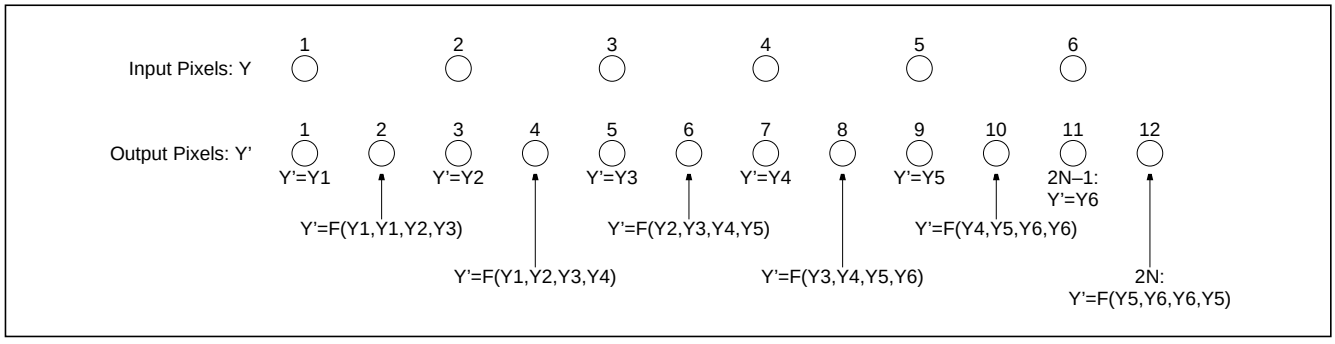


Figure 7-28. Mirroring pixels in 2x upscaling.

7.14 EVO OPERATING MODES

EVO operating modes belong to two groups as follows:

- Video-refresh modes
- Data-transfer modes

Data-transfer modes are further broken down into data-streaming mode and message-passing mode.

The operating mode is set by the VO_CTL.MODE field and the VO_CTL.OL_EN (overlay enable) control bit. The VO_CTL.MODE field determines video-refresh, message-passing or data-streaming mode. It further defines the video image format and whether or not 2× horizontal upscaling takes place. The OL_EN bit determines whether a video-refresh mode has a graphics overlay present. The modes are shown in Table 7-4.

Table 7-4. EVO Operating Modes

Mode	Function	Explanation
Video-refresh modes		
0	YUV 4:2:2C-1×	YUV 4:2:2 co-sited, no scaling
1	YUV 4:2:2I-1×	YUV 4:2:2 interspersed, no scaling
2	YUV 4:2:0-1×	YUV 4:2:0, no scaling
3	Reserved	
4	YUV 4:2:2C-2×	YUV 4:2:2 co-sited, horizontal 2× upscaling
5	YUV 4:2:2I-2×	YUV 4:2:2 interspersed, horizontal 2× upscaling
6	YUV 4:2:0-2×	YUV 4:2:0, horizontal 2× upscaling
7	Reserved	
Data-transfer modes		
8	data streaming	continuous transmission of raw 8-bit data with valid data pulse and level timing signals
9	message passing	transmission of raw 8-bit data with STMSG and ENDMSG timing signals
0xA — 0xF	Reserved	

7.15 VIDEO PROCESSING

If enabled, the TM1300 implements new functions for chroma keying, alpha blending and programmable clipping, as described in this section.

7.15.1 Alpha Blending

If enabled by setting EVO_ENABLE = 1 and FULL_BLENDING = 1, the EVO provides full 129-layer alpha blending of a background video image with a foreground graphics overlay image. If either bit is 0, the EVO implements the cruder 25% step alpha blending resolution of the TM1000. Alpha blending can operate in conjunction with chroma keying, as described in Section 7.15.2.

Alpha blending combines a graphics overlay image with the video image according to an alpha value provided with each overlay pixel. The graphics overlay is taken from a pixel-packed YUV 4:2:2+α data structure in memory. In the YUV 4:2:2+α format, each pixel has a single α-bit supplied as the LSB of the U and V pixels. The U byte LSB corresponds to the alpha for pixel Y0, the V byte LSB for pixel Y1, respectively. When the α-bit is '0', the ALPHA_ZERO register supplies the actual 8-bit α value. When the α-bit is '1', the ALPHA_ONE register supplies the 8-bit α value. In the YUV 4:2:2 format, only one set of U and V values is supplied for the two Y pixels, Y0 and Y1. In this case, the alpha bit in U0 determines the alpha value for U, Y0 and V. The alpha blend bit in V0 only sets the alpha value for Y1 and does not affect the U or V values.

The EVO uses the 8-bit content of the selected alpha blending register (ALPHA_ZERO or ALPHA_ONE) to determine the amount by which the overlay plane is merged with the image plane as follows. The least-significant 7 bits of the selected blending register encode 128 blending levels from 0 to 0x7F. The MSB is used to turn on blending (MSB = '0') or to select the overlay plane as the only output (MSB = '1'), so all values between 0x80 and 0xFF select 100% overlay. Therefore, the total number of blending levels is 129: 128 variable blending values from 0 to 0x7F plus one 'blending' value from 0x80 to 0xFF for 100% overlay. An alpha value of 0 selects 100% image plane and 0% overlay. Similarly, a value of 0x40 selects 50% image and 50% overlay blending.

The equations for the blending are illustrated below.

```
if alpha[7] = 1 then
        output[7:0] = overlay[7:0]
else
        output[7:0] = (alpha[6:0] · overlay[7:0] + (¬alpha[6:0] + 1) · image[7:0]) >> 7
    (or)
        output[7:0] = (alpha[6:0] · (overlay[7:0] – image[7:0]) >> 7) + image[7:0]
```

7.15.2 Chroma Keying

If the EVO_ENABLE and KEY_ENABLE bits are set to '1' in EVO_CTL the TM1300 activates chroma keying. The graphics overlay is taken from a pixel-packed YUV 4:2:2+α data structure in memory. The EVO_KEY register provides the value which signifies full transparency for the overlay. The overlay values (Y, U and V) are compared to the values stored in bit-fields of the EVO_KEY register. EVO_KEY has three 8-bit fields: KEY_Y, KEY_U and KEY_V, which store the values to be compared to the Y, U, and V components, respectively, of the overlay for chroma keying. Bits that correspond to bits set in MASK_Y and MASK_UV are ignored for the comparison. When there is an exact match between the pixel value and the value in EVO_KEY (disregarding any bits masked by MASK_Y and MASK_UV), then the overlay value is not present in the output stream, resulting in full transparency.

The mask bits in EVO_MASK provide for varying degrees of precision in the chroma-key matching process. The EVO_MASK.MASK_Y field can mask from 0 to 4 LSBs of the overlay Y component during the chroma key process. For example, setting MASK_Y = 1 eliminates the influence of the LSB of KEY_Y in the keying process. This can be used to widen the range of key matching to account for irregularities in the chroma-key video signal. Likewise, EVO_MASK.MASK_UV is used to mask from zero to four LSBs of the overlay U and V components during the chroma key process. For example, setting MASK_UV = 1 eliminates the influence of the LSB of KEY_U and KEY_V in the keying process.

7.15.3 Programmable Clipping

If EVO_CTL.CLIPPING_ENABLE = 1 the EVO performs fully-compliant programmable clipping. Clipping is performed as the last step of the video pipeline, after chroma keying and alpha blending. It is applied only on the image areas (Field 1 and Field 2) defined by IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_VOFF and IMAGE_HOFF inside the Active Video Area. Blanking values are not clipped.

The EVO_CLIP MMIO register stores four 8-bit fields used to clip output components. The Y output component is clipped between the values stored in LOWER_CLIPY and HIGHER_CLIPY. A value less than or equal to LOWER_CLIPY is forced to LOWER_CLIPY and a value greater than or equal to HIGHER_CLIPY is forced to HIGHER_CLIPY.

unit by clearing the HBE bit then reading VO_STATUS[4]. If the bit remains '1', the unit is an EVO.

The same behavior is implemented for U and V with the values stored in the LOWER_CLIPUV and HIGHER_CLIPUV fields.

This mode allows fully-compliant 16 to 235 Y clipping and 16 to 240 Cb and Cr clipping to be programmed. These are the default values of the EVO_CLIP register after reset.

If CLIPPING_ENABLE = 0, the EVO clips Y, U and V between the default values 16 and 240, as it is implemented in the TM1000. When LOWER_CLIP{Y,UV} registers are set to '0' and HIGHER_CLIP{Y,UV} registers are set to '255', no clipping is performed.

7.16 MMIO REGISTERS

The MMIO registers are in two groups:

- VO registers — control basic VO functions (those shared with the TM1000 VO unit)
- EVO registers — control new EVO unit functions (those new to TM1100/TM1300)

VO MMIO registers are shown in Figure 7-29. VO MMIO register names are prefixed with "VO_". Generally, their functionality is unchanged except where noted in the text (see for instance, Section 7.16.1). The register fields are described in Table 7-5, Table 7-6 and Table 7-7. They are discussed in sections 7.16.1 through 7.16.3.2.

EVO MMIO registers are shown in Figure 7-31. EVO MMIO register names are prefixed with "EVO_". The EVO_CTL register selectively enables new TM1300 functions. Other EVO-related registers support new TM1300 functions. The register fields are described in Table 7-8 and Table 7-9. They are discussed in sections 7.16.4 and 7.16.5.

To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as '0's.

7.16.1 VO Status Register (VO_STATUS)

The VO_STATUS register is a read-only register that shows the current status of the EVO. Its fields are shown in Figure 7-29 and Table 7-5.

VO_STATUS[4] is now hard-wired to '1'. This allows software to determine if the unit is an EVO unit (containing extra MMIO registers) or a TM1000 VO unit, as follows. In the TM1000, this bit is a copy of the HBE flag (VO_STATUS[5]). In the EVO unit, it is hard-wired to '1'. Software can use this bit to determine the type of (E)VO

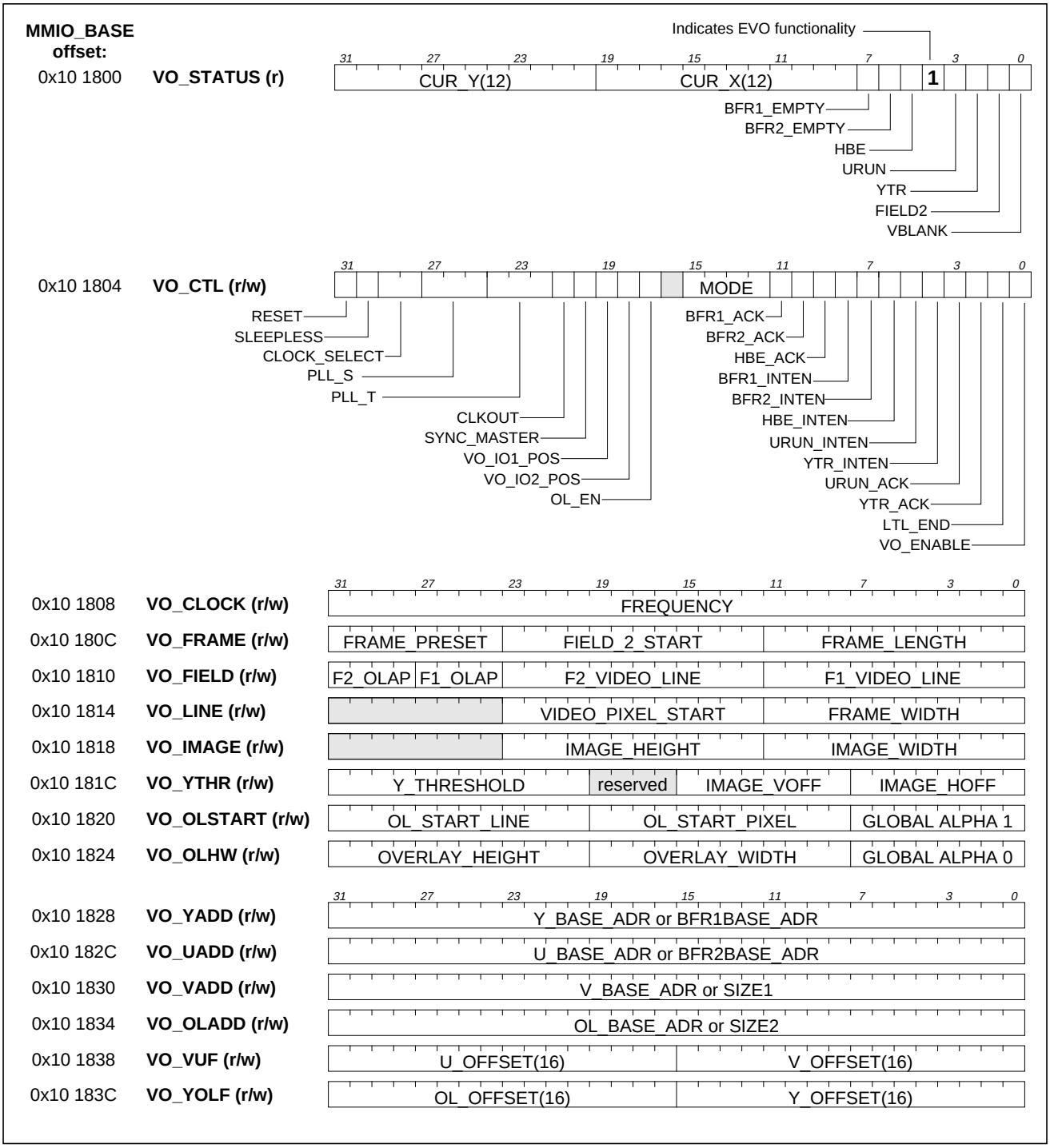


Figure 7-29. EVO MMIO registers.

7.16.2 VO Control Register (VO_CTL)

The VO_CTL register sets the operating mode, enables interrupts, clears interrupt flags, and initiates EVO operations. Its fields are unchanged from the TM1000, as shown in Figure 7-29 and Table 7-6, however the precise functionality implemented by a field may be changed if TM1300 functionality is enabled by software. Its hardware reset value is 0x32400000 which sets CLOCK_SELECT = 3, PLL_S = 1 and PLL_T = 1, and all other bits to '0'. To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as '0's.

7.16.3 VO-Related Registers

The VO-related registers and their fields are shown in Table 7-7. Their fields are unchanged from the TM1000, however their function may vary depending upon the

Table 7-5. VO_STATUS — status register fields

Field	Description
CUR_Y	Current Y. Image line index of the current line in the current field being output by the EVO. CUR_Y reflects the current state of the Image Line Counter. CUR_X and CUR_Y form a single 24-bit output data byte counter (CUR_X is the counter LSBs) when the EVO is in data-streaming or message-passing mode. This counter reflects the status of the SIZE counter for the currently active buffer. The two LSBs of this counter are not valid for reading during transfers; only the upper 22 bits (the word count) are valid.
CUR_X	Current X. Image pixel index of the most-recently-output pixel. CUR_X reflects the current state of the Image Pixel Counter.
BFR1_EMPTY BFR2_EMPTY	Buffers 1 and 2 Empty. These bits are valid in video-refresh, data-streaming and message-passing modes. • In video-refresh modes, only Buffer 1 is used. BFR1_EMPTY indicates that the last byte of a field has been transferred. It is actually raised at the completion of the transmission of the Overlap area of the field, as shown in Figure 7-30. At this point, software should assign a new field of imagery to {Y,U,V}_BASE_ADR and perform a BFR1_ACK. If BFR1_EMPTY is not cleared by BFR1_ACK before the active video area of the next field starts to be emitted, the EVO sets the URUN bit. • In data-streaming mode, BFR1_EMPTY and BFR2_EMPTY indicate that the last byte in their corresponding buffer has been transferred. When BFR1_EMPTY or BFR2_EMPTY is set, transfer stops from the corresponding buffer. • In message passing mode, BFR1_EMPTY signals completion of message transmission. These bits cause an interrupt if their interrupt-enable bits are set. One interrupt per buffer is signaled.
HBE	Highway Bandwidth Error. HBE is set when the highway fails to respond in time to a highway read request and data was not ready in time to be set on EVO data lines. HBE can be set in both image- and data-transfer modes. HBE indicates insufficient bandwidth was requested from the highway arbiter.
1	EVO unit indicator. This bit allows software to determine if the unit is an EVO (containing extra MMIO registers) or a TM1000 VO unit. In the TM1000, this bit is a copy of the HBE flag. In the EVO unit, it is hard-wired to ‘1’. Software can easily determine the type of video output unit by clearing the HBE bit then reading this bit.
YTR	Y threshold. In video-refresh modes, YTR indicates that the Image Line Counter value is equal to the Y_THRESHOLD value in VO_YTHR. The Y_THRESHOLD value can be set to provide an interrupt on any line in the valid image area.
URUN	Underrun. In video-refresh and data-streaming mode, this bit indicates that the CPU did not perform an acknowledge to indicate updated address pointers for the next field or buffer in time for continuous image or data transfer. URUN causes an interrupt if the corresponding interrupt-enable condition is set. • In video-refresh modes, URUN indicates that the SAV code marking beginning of active video has been generated without BFR1_ACK being set by the CPU. (Setting BFR1_ACK to ‘1’ clears BFR1_EMPTY). In this case, video refresh continues with previous address pointers. • In data-streaming mode, URUN indicates the last byte in the active buffer was transferred, and no BFR1_ACK or BFR2_ACK occurred to enable the next buffer. In this case, transfer continues with previous address pointers.
FIELD2	Field 2 or Buffer 2 active. • In data-streaming mode, FIELD2 = 0 when Buffer 1 is active; FIELD2 = 1 when Buffer 2 is active. • In video-refresh modes, FIELD2 indicates that the EVO is actively sending out a video image for Field 2, as defined by Figure 7-30.
VBLANK	Vertical blanking. Indicates that the EVO is in a vertical-blanking interval. VBLANK is asserted only in video-refresh modes.

new TM1300 features that are selectively enabled by EVO_CTL (see Section 7.16.4).

7.16.3.1 Frame and field timing control

The frame timing for 525/60 and 625/50 timing cases is shown pictorially in Figure 7-30. CCIR 656 line definitions are used.

7.16.3.2 Recommended values for timing registers

The recommended values for the various fields of the timing registers are shown in Table 7-10 for 525/60 and 625/50 timing cases. The FREQUENCY field value shown is for 27 MHz assuming a DSPCPU clock of 143 MHz.

7.16.4 EVO Control Register (EVO_CTL)

New TM1300 EVO features are enabled by setting the appropriate fields of the EVO_CTL register shown in Figure 7-26. The register fields are described in

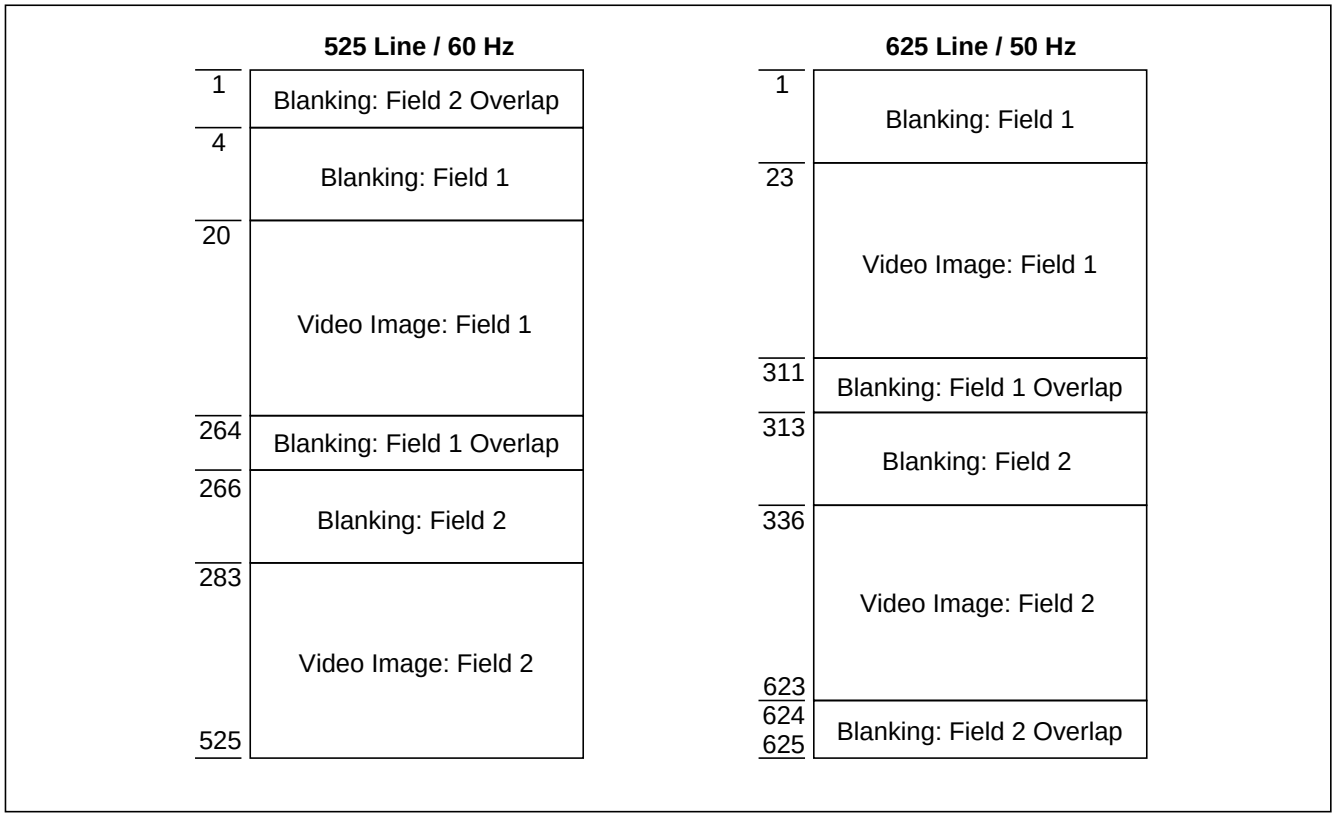


Figure 7-30. EVO frame timing.

Table 7-6. VO_CTL register fields

Field	Description
RESET	Software reset of the EVO. The recommended software reset procedure is as follows. • Write the desired VO_CTL state with the RESET bit set to '1'. • Write the desired VO_CTL state word, this time with the RESET bit cleared to '0'. Both writes should have VO_ENABLE set to 0. • Finally, enable the newly selected mode by setting VO_ENABLE. This step should be done last, as a separate transaction. After a software reset, 5 VO_CLK clock cycles are required to stabilize the internal circuitry (before enabling EVO). Note: A hardware reset clears the CLKOUT and SYNC_MASTER bits and puts VO_CLK, VO_IO1, and VO_IO2 in the input state. This results in a VO_CTL value of 0x32400000. In contrast, a software reset does not change device registers. So a software reset results in a state as specified by the VO_CTL word value written during the above-described procedure.
SLEEPLESS	Disable power management. If SLEEPLESS = 1, power-down of the EVO is prevented during global TM1300 power-down.
CLOCK_SELECT	Clock select. 00 — Select PLL VCO output as the VO_CLK source. 01 — Select PLL feedback loop divider output as VO_CLK source. 10 — Select PLL input divider output as VO_CLK source. 11 — Select DDS output directly as VO_CLK source, bypassing the PLL altogether. (Hardware reset default.)
PLL_S	PLL input divider division ratio. A value of *k* selects division by *k*+1. The hardware reset default = 1, causing division by 2.
PLL_T	PLL feedback loop divider division ratio. A value of *k* selects division by *k*+1. The hardware reset default = 1, causing division by 2.
CLKOUT	Clock output. • When CLKOUT = 1, the EVO clock generator is enabled, and VO_CLK is an output. • When CLKOUT = 0, VO_CLK is an input, and EVO clock is provided by the external device. (Hardware reset default.)

Table 7-6. VO_CTL register fields

Field	Description
SYNC_MASTER	Sync master. • When set, VO_IO1 and VO_IO2 are outputs. In video-refresh modes, the EVO generates horizontal and frame timing signals on VO_IO1 and VO_IO2 respectively. In message-passing mode and data-streaming mode, this bit should always be set so that VO_IO1 and VO_IO2 generate START and END message signals respectively. • When zero, VO_IO2 is an input. (Hardware reset default.) In video-refresh modes, VO_IO2 serves as the frame time reference. The active edge is selected by VO_IO2_POS.
VO_IO1_POS VO_IO2_POS	Polarity of VO_IOx_POS. VO_IO1_POS currently has no function. VO_IO2_POS determines the input polarity of VO_IO2. • When '0', the corresponding input triggers on the negative (high-to-low) transition of the input signal. • When '1', the input triggers on the positive (low-to-high) transition.
OL_EN	Overlay Enable. Enables the YUV overlay function in video-refresh modes.
MODE	Major operating mode. Defines the video output major operating mode, as listed in Table 7-4 on page 7-13.
BFR1_ACK BFR2_ACK	Buffer 1 and Buffer 2 acknowledge. When active in data-transfer modes, writing a '1' to BFR1_ACK clears BFR1_EMPTY and enables Buffer 1 for transfer until BFR1_EMPTY is set. Writing a '0' to BFR1_ACK has no effect. BRF2_ACK operates similarly for Buffer 2. Writing a '1' to VO_ENABLE in data-streaming mode is the same as writing a '1' to both BFR1_ACK and BFR2_ACK, and enables both buffers 1 and 2 for transfer. Writing a '1' to VO_ENABLE in message-passing mode is the same as writing a '1' to BFR1_ACK, and enables Buffer 1 for transfer. BFR2_ACK is not used in message-passing mode, since only Buffer 1 is used.
HBE_ACK URUN_ACK	Acknowledge HBE or URUN. Writing a '1' to these bits clears the HBE or URUN flags and resets their corresponding interrupt conditions.
YTR_ACK	Acknowledge Y threshold. Writing a '1' to this bit clears the YTR flag and resets its interrupt condition. YTR signals the CPU to set new pointers for the next field. If YTR_ACK is not received by the time the active image area for the next field starts, the URUN flag is set. Data transfer continues with the old pointer values.
BFR1_INTEN BFR2_INTEN HBE_INTEN URUN_INTEN YTR_INTEN	Enable interrupt conditions. Enable corresponding interrupts to be generated when the BFR1_EMPTY, BFR2_EMPTY, HBE, URUN (underrun/end of transfer), and YTR (end of field/buffer) flags are set, respectively. Note: BFR2_INTEN, URUN_INTEN, YTR_INTEN must be 0 in message passing mode.
LTL_END	Little-endian. Specifies that data in SDRAM is stored in little-endian format. This only affects the overlay packed-image format interpretation in video-refresh modes. Refer to Appendix C, "Endian-ness," for details on byte ordering.
VO_ENABLE	Enable the EVO to send image data or message data to its output. Note: This bit should not be simultaneously asserted with the RESET bit. The correct sequence to reset and enable the EVO is as follows. • Set all VO_CTL control fields as desired, writing VO_CTL with RESET = 1, VO_ENABLE = 0. • Retain all desired values of control fields, but rewrite VO_CTL with RESET = 0, VO_ENABLE = 0. • Finally, still retaining all desired control fields, rewrite VO_CTL with RESET = 0, VO_ENABLE = 1. Setting VO_ENABLE in video-refresh modes starts the EVO sending image data beginning with the first pixel in the image. Setting VO_ENABLE in data-streaming and message-passing modes starts the EVO sending data beginning with the first byte in Buffer 1. In video-refresh and data-streaming modes, VO_ENABLE remains set until cleared by the CPU. In message-passing mode, VO_ENABLE is cleared when BFR1_EMPTY is set, indicating the end of message transfer. Note: De-asserting VO_ENABLE in video-refresh modes causes SDRAM reads to stop, but sync framing and BFR1_EMPTY generation and interrupts remain fully operational. The transmitted active image data is undefined in this case. To fully halt video output, a software reset is required.

Table 7-7. VO register fields

Register	Field	Description
VO_CLOCK	FREQUENCY	VO_CLK frequency. See DDS equation in Figure 7-6, and PLL description in Section 7.18.
VO_FRAME	FRAME_LENGTH	Total number of lines per frame; the ending value of the Frame Line Counter; typically 525 or 625. Note: the Frame Line Counter counts from 1 to 525 or 625, consistent with CCIR 656 line numbering.
	FIELD_2_START	Start line number in the Frame Line Counter; where the second field of the frame begins. If non-interlaced pictures are desired, then the same value is programmed for Field 1 and Field 2. Field 1 becomes Frame 1 and Field 2 becomes Frame 2.
	FRAME_PRESET	Value loaded into the Frame Line Counter when frame timing edge is received on VO_IO2. Note: currently this must be set to 1.
VO_FIELD	F1_VIDEO_LINE	Line number in the Frame Line Counter of the first active video line of Field 1 of the frame.
	F2_VIDEO_LINE	Line number in the Frame Line Counter of the first active video line of Field 2 of the frame. If non-interlaced pictures are desired, this is programmed to the same value as F1_VIDEO_LINE
	F1_OLAP	Overlap of the SAV and EAV codes from Field 1 to Field 2. Overlap is defined as the delay in lines from start of blanking for Field 2 until SAV and EAV codes for Field 2 are emitted. Typical values are +2 for 525/60 and +2 for 625/50.
	F2_OLAP	Overlap in lines of the SAV and EAV code from Field 2 to Field 1. Overlap is defined as the delay in lines from start of blanking for Field 1 until the SAV and EAV codes for Field 1 are emitted. Typical values are +3 for 525/60 and –2 for 625/50. The negative value means Field 1 blanking actually starts two lines before end of Field 2 of previous frame. This overlap is described in Table 7-3 on page 7-6, and illustrated in Figure 7-30.
VO_LINE	FRAME_WIDTH	Total line length in pixels including blanking. Also the ending value for the Frame Pixel Counter. Lines always begin with a horizontal blanking interval, and the image starts after the blanking interval and runs to the end of the line.
	VIDEO_PIXEL_START	Pixel number in Frame Pixel Counter of starting pixel of active video area within the line. Note: Must be even.
VO_IMAGE	IMAGE_HEIGHT	Video Image height in lines.
	IMAGE_WIDTH	Video Image line (scaled) output width in pixels. Must be even for upscaling by 2×.
VO_YTHR	Y_THRESHOLD	Threshold image line number in the Image Line Counter for the YTR interrupt. Can be reprogrammed on a frame-by-frame basis.
	IMAGE_VOFF	Image vertical offset in lines from the top of the active video window.
	IMAGE_HOFF	Image horizontal offset in pixels from the start of the active video window.
VO_OLSTART	OL_START_LINE	Starting image line of YUV overlay within the image. Zero indicates that the overlay starts at the same line as the image.
	OL_START_PIXEL	Starting image pixel of the YUV overlay within the image. ‘0’ indicates that the overlay starts at same pixel as the image. Note: Must be even.
	ALPHA_ONE	Alpha blend value used for YUV 4:2:2+alpha format overlays when the alpha bit = 1.
VO_OLHW	OVERLAY_HEIGHT	Height of the YUV overlay image in lines. Note: The height of the overlay should be chosen such that it does not extend beyond the image area.
	OVERLAY_WIDTH	Width of the YUV overlay image in pixels. Note: Must be even.
	ALPHA_ZERO	Alpha blend value used for YUV 4:2:2+alpha format overlays when the alpha bit = 0.
VO_YADD	Y_BASE_ADR BFR1BASE_ADR	Y-component buffer address or Buffer 1 address. • In video-refresh modes: Y-component starting byte address. • In data-streaming and message-passing modes: Buffer 1 starting byte address. Note: must be 64-byte aligned in data-streaming mode and 4-byte aligned in message passing mode.
VO_UADD	U_BASE_ADR BFR2BASE_ADR	U-component buffer address or Buffer 2 address. • In video-refresh modes: U-component starting byte address • In data-streaming mode: Buffer 2 starting byte address; must be 64-byte aligned • Not used in message-passing mode
VO_VADD	V_BASE_ADR SIZE1	V-component buffer address or Buffer 1 length. • In video-refresh modes: V-component starting byte address • In data-streaming and message-passing modes: Buffer 1 length in bytes. Note: must be a multiple of 64 in data-streaming mode. SIZE1 is limited to 24 bits.

Table 7-7. VO register fields

Register	Field	Description
VO_OLADD	OL_BASE_ADDR SIZE2	Overlay-image buffer address or Buffer 2 length. • In video-refresh modes: overlay-image starting byte address. OL_BASE can be reprogrammed on a frame-by-frame basis. • In data-streaming mode: Buffer 2 length in bytes. Note: Must be multiple of 64 in data-streaming mode; Not used in message-passing mode.
VO_VUF	U_OFFSET	Offset in bytes from start of one line to start of next line (16-bits unsigned).
	V_OFFSET	Offset in bytes from start of one line to start of next line (16-bits unsigned).
VO_YOLF	Y_OFFSET	Offset in bytes from start of one line to start of next line (16-bits unsigned).
	OL_OFFSET	Offset in bytes from start of one line to start of next line (16-bits unsigned).

Table 7-8. If features are enabled, new TM1300 the functionality replaces TM1000 functions.

The hardware reset value of EVO_CTL register is 0x10000000, which means that EVO functions are disabled on reset and must be enabled by software. The MS four bits indicate the EVO revision number.

To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as ‘0’s.

7.16.5 EVO-Related Registers

As shown in Figure 7-31, four additional registers are introduced in the TM1300, as follows.

- EVO_MASK and EVO_KEY — used in chroma key (see Section 7.15.2).
- EVO_CLIP — provides programmable clipping (see Section 7.15.3).
-
- EVO_SLVDLY — used in Genlock mode (see Section 7.10).

These registers are shown in Figure 7-31, and their register fields are shown in Table 7-9.

To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as ‘0’s.

7.17 ENHANCED VIDEO OUT OPERATION

As described in Section 7.14, the EVO operates in either video-refresh or data-transfer modes. The DSPCPU starts the EVO by setting the appropriate VO MMIO registers and the appropriate EVO MMIO registers.

VO_CTL.MODE must be set to the appropriate transfer mode, appropriate addresses, address offsets, and image timing registers and the associated control bits in the control register must be set. Lastly, software sets

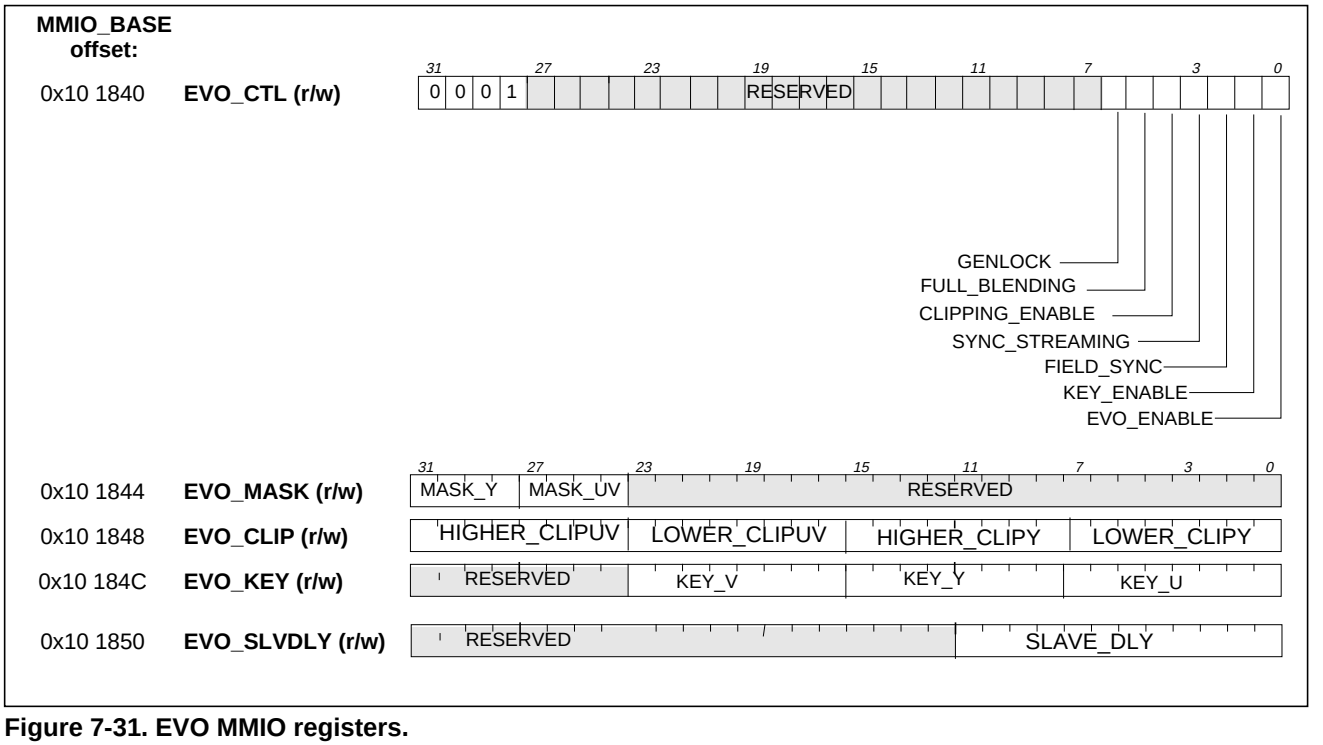


Figure 7-31. EVO MMIO registers.

Table 7-8. EVO_CTL Register Fields

Register	Field	Description
EVO_CTL	EVO_ENABLE	When set to 1, new EVO features are enabled. When set to 0 (the hardware reset value), the EVO behaves exactly like a TM1000 VO unit. Default: 0.
	FULL_BLENDING	Activates full 8-bit alpha blending when set to 1. When set to 0, only the original five TM1000 blending levels are implemented (0%, 25%, 50%, 75%, 100%). Default: 0.
	CLIPPING_ENABLE	When set to 1, the values stored in EVO_CLIP are used for the clipping of output data. Otherwise, TM1000 default values (240 and 16 for Y, U and V) are used. Default: 0.
	SYNC_STREAMING	When set to 1 in data-streaming mode, VO_IO2 generates a DATA_VALID signal. See Section 7.17.2, “Data-transfer Modes”. Default: 0.
	FIELD_SYNC	When set, VO_IO2 will generate frame synchronization signal that follows the field number in SAV/EAV codes (Field1 gives a low VO_IO2, Field2 gives a high VO_IO2). Default: 0.
	GENLOCK	Activates Genlock mode when set to 1 and VO_CTL.SYNC_MASTER = 0. Default: 0.
	KEY_ENABLE	When set, this bit activates chroma key. The overlay values (Y, U and V) are compared to the values stored in the EVO_KEY register. Bits that correspond to bits set in MASK_Y and MASK_UV are ignored for the comparison. When there is an exact match between the pixel value and the value in EVO_KEY register (less the bits selected by MASK_Y and MASK_UV), then the overlay value is not present in the output stream, resulting in full transparency. The key is 24 bits (Y, U and V are 8 bits each). Default: 0.

Table 7-9. EVO-Related MMIO Registers Fields

Register	Field	Description
EVO_MASK	MASK_Y	This 4-bit value is used to mask the four lower bits of the overlay Y component during the chroma key process. Example: Setting MASK_Y to ‘1’ will eliminate the influence of the LSB of KEY_Y in the keying process.
	MASK_UV	This 4-bit value is used to mask the four lower bits of the overlay U and V components during the chroma key process. Example: Setting MASK_UV to ‘1’ will eliminate the influence of the LSB of KEY_U and KEY_V in the keying process.
EVO_CLIP	LOWER_CLIPY	A Y value lower or equal to LOWER_CLIPY is forced to LOWER_CLIPY. Default: 16.
	HIGHER_CLIPY	A Y value higher or equal to HIGHER_CLIPY is forced to HIGHER_CLIPY. Default: 235.
	LOWER_CLIPUV	An U or Y value less than or equal to LOWER_CLIPUV is forced to LOWER_CLIPUV. Default: 16.
	HIGHER_CLIPUV	An U or and an V value higher than or equal to HIGHER_CLIPUV is forced to HIGHER_CLIPUV. Default: 240.
EVO_KEY	KEY_Y	Value compared to the Y component of the overlay for chroma keying.
	KEY_U	Value compared to the U component of the overlay for chroma keying.
	KEY_V	Value compared to the V component of the overlay for chroma keying.
EVO_SLVDLY		Number of VO_CLK cycles of internal delay for VO_IO2 in Genlock mode.

VO_CTL.VO_ENABLE to begin EVO operation. The EVO transfers the image, data, or message as commanded. In video-refresh and data-streaming modes, the EVO runs continuously. In message-passing mode, the EVO runs only until the message has been transferred.

The EVO unit is reset by a TM1300 hardware reset, or by a software reset, as described in Table 7-6 for the RESET bit.

Table 7-10. Timing register recommended values

Register	Field	525/60 Value	625/50 Value
VO_CLOCK	FREQUENCY	0x855E, E191	0x855E, E191

Table 7-10. Timing register recommended values

Register	Field	525/60 Value	625/50 Value
VO_FRAME	FRAME_LENGTH	525	625
	FIELD_2_START	264	311
	FRAME_PRESET	1	1
VO_FIELD	F1_VIDEO_LINE	20	23
	F2_VIDEO_LINE	283	336
	F1_OLAP	2	2
	F2_OLAP	3	–2 (0xE)
VO_LINE	FRAME_WIDTH	858	864
	VIDEO_PIXEL_START	138	144
VO_IMAGE	IMAGE_HEIGHT	240	288
	IMAGE_WIDTH	720	720 (704 visible)

The VO_CLK signal is normally set as an output to drive the data transfer for all modes at a programmable rate. The VO_CLK signal can be an input or output, as controlled by the VO_CTL.CLKOUT bit. When CLKOUT = 1, VO_CLK is an output, and its frequency is set by the VO_CLOCK register value. When CLKOUT = 0, VO_CLK is an input and the EVO generates data at the clock rate of the sender.

In video-refresh modes, the EVO receives or generates horizontal and frame synchronization signals on the VO_IO1 and VO_IO2 lines, as described in Section 7.9.4

7.17.1 Video Refresh Modes

In video-refresh mode, the EVO transfers an image from SDRAM to the EVO port. The VO_CTL.MODE field defines the video image memory data format and determines whether the EVO is to perform horizontal upscaling (see Table 7-4). The EVO accepts memory image data in YUV 4:2:2 co-sited, YUV 4:2:2 interspersed and YUV 4:2:0 formats, and generates a CCIR 656-compatible, YUV 4:2:2 co-sited image output stream. Scaling is identified by the *YUV-1×* and *YUV-2×* modes. In *YUV-1×* modes, luminance and chrominance pass unmodified. In *YUV-2×* modes, luminance and chrominance are horizontally upscaled by a factor of two.

During video refresh, the VO_STATUS.YTR bit is set when the Image Line Counter reaches the Y_THRESHOLD value. When an image field has been transferred, the VO_STATUS.BFR1_EMPTY bit is set. The DSPCPU is interrupted when either the YTR or BFR1_EMPTY flag is set and its corresponding interrupt is enabled. To maintain continuous transfer of image fields, the DSPCPU supplies new pointers for the next field following each BFR1_EMPTY interrupt. If the DSPCPU does not supply new pointers before the next field, the URUN bit is set, and the EVO uses the same pointer values until they are updated.

Graphics Overlay

The graphics overlay is enabled by the VO_CTL.OL_EN bit. The graphics overlay is typically a software-generated graphic overlaid onto the output video image stream. The graphics overlay is either generated in YUV by the DSPCPU or converted by the DSPCPU from an RGB to a YUV overlay image. Because RGB-to-YUV conversion can potentially lose information, this conversion is done by the DSPCPU, because it has the most information about how best to perform this conversion in the most effective manner.

The overlay height should be chosen such that the overlay does not vertically extend beyond the image area. A height greater than this causes undefined results and may result in vertical overlay wraparound.

Note: The emitted byte data rate is limited to 45% of the SDRAM clock when overlays are enabled.

The YUV overlay logic assembles the U0, Y0, V0, Y1 bytes for a pair of YUV 4:2:2 pixels for both the main image and the overlay image. The alpha bit for pixel 0 (the LSB of the U0 byte of the overlay image) selects ALPHA_ZERO or ALPHA_ONE as the alpha source, and the alpha blend logic combines U0, Y0, and V0 from the main and overlay images to generate the U0, Y0 and V0 output values. The alpha bit for pixel 1 (the LSB of the V0 byte of the overlay image) selects ALPHA_ZERO or ALPHA_ONE as the alpha source for blending the Y1 pixels to generate the Y1 output value. The alpha blended U0, Y0, V0 and Y1 bytes are sent to the EVO output port in the YUV 422 sequence. The overlay U and V values used assume an LSB of zero.

Video Image Addressing

The output image is read from SDRAM at a location defined by Y_BASE_ADR, Y_OFFSET, U_BASE_ADR, U_OFFSET, V_BASE_ADR, and V_OFFSET. The default memory packing is big-endian although little-endian packing is also supported by setting the VO_CTL.LTL_END bit.

Horizontally-adjacent samples are stored at successive byte addresses, resulting in a packed form (four 8-bit samples are packed into one 32-bit word). Upon horizontal retrace, the starting byte address for the next line is computed by adding the corresponding offset value to the previous line's starting byte address. Note that {OL,Y,U,V}_OFFSET values are 16-bit unsigned quantities. This process continues until the total image—height in lines and width in pixels per line—has been read from memory for luminance (Y). For chrominance, the same number of lines are read, but half the number of pixels per line are read in YUV 4:2:2 and YUV 4:2:0 formats[1]. The YUV 4:2:0 format has half the number of U and V lines in memory that the YUV 4:2:2 formats have, but each line of U and V data is read and used twice. See Figure 7-19 through Figure 7-22.

7.17.2 Data-transfer Modes

In data-streaming and message-passing modes, the EVO supplies a stream of 8-bit data to the VO_DATA[7:0] lines at rates up to 81 MHz.

Note: In the TM1300, the data-rate is limited to an 81-MHz EVO clock.

Data is read from SDRAM in packed form (four 8-bit bytes per 32-bit word). No data selection or data interpretation is done, and data is transferred at one byte per VO_CLK from successive byte addresses.

Data-Streaming Mode. In data-streaming mode, data is stored in SDRAM in two buffers.

When the EVO has transferred out the contents of one buffer, it interrupts the DSPCPU and begins transferring out the contents of the second buffer. The DSPCPU supplies pointers to both buffers. The EVO can provide a continuous stream of data to the EVO output if the DSPCPU updates the pointer to the next buffer before the EVO starts transferring data from the next table.

1. Note that consecutive pixel components of each line are stored in consecutive memory addresses but consecutive lines need not be in consecutive memory addresses

Note: In this mode, SYNC_MASTER must be set to ensure correct operation of VO_IO1 and VO_IO2 as outputs.

When each buffer has been transferred, the corresponding buffer-empty bit is set in the status register, and the DSPCPU is interrupted if the buffer-empty interrupt is enabled. To maintain continuous transfer of data, the DSPCPU supplies new pointers for the next data buffer following each buffer-empty interrupt. If the DSPCPU does not supply new pointers before the next field, the URUN bit is set, and the EVO uses the same pointer values until they are updated.

When data-streaming mode is enabled and EVO_ENABLE = 1 and SYNC_STREAMING = 1, the VO_IO2 signal indicates a data-valid condition. This signal is asserted when the EVO starts outputing valid data (that is, data-streaming mode is enabled and video output is running) and is de-asserted when data-streaming mode is disabled. The VO_IO1 signal generates a pulse one VO_CLK cycle before the first valid data is sent. See Section 7.11 for timing signal details.

Message-Passing Mode. In message-passing mode data is stored in SDRAM in one buffer.

Note: In this mode, SYNC_MASTER must be set to ensure correct operation of VO_IO1 and VO_IO2 as outputs.

When message passing is started by setting VO_CTL.VO_ENABLE, the EVO sends a Start condition on VO_IO1. When the EVO has transferred the contents of the buffer, it sends an End condition on VO_IO2 as shown in Figure 7-18, sets BFR1_EMPTY, and interrupts the DSPCPU. The EVO stops, and no further operation takes place until the DSPCPU sets VO_ENABLE again to start another message, or until the DSCPU initiates other EVO operation. See Section 7.11 for timing signal details.

7.17.3 Interrupts and Error Conditions

The EVO has five interrupt conditions defined by bits in the VO_STATUS register: BFR1_EMPTY, BFR2_EMPTY, HBE, URUN, and YTR. Each of these conditions has a corresponding interrupt enable flag and interrupt acknowledge bit in the VO_CTL register.

The EVO asserts a SOURCE 10 interrupt request to the TM1300 vectored interrupt controller as long as one or more enabled events is asserted.

Note: The interrupt controller should always be programmed such that the EVO interrupt operates in level-triggered mode. This ensures that no EVO events can be lost to the interrupt handler. Refer to Section 3.5.3, “INT and NMI (Maskable and Non-Maskable Interrupts),” for a description of setting level-triggered mode, as well as for recommendations on writing interrupt handlers.

The BFR1_EMPTY, BFR2_EMPTY and YTR status flags indicate to the DSPCPU that a buffer has been emptied or that the Y threshold has been reached.

The buffer-underrun (URUN) status flag indicates that the DSPCPU did not acknowledge a BFR1_EMPTY or BFR2_EMPTY interrupt before the EVO required the next buffer. In this case, the EVO uses the old address pointer value and continues image or data transfer. When the DSPCPU updates the pointer, the new pointer value will be used at the start of the next frame or buffer transfer. Therefore, the URUN flag can be interpreted as indicating to the DSPCPU that the EVO is using its old pointer values because it did not receive the new ones in time.

Note: The actual buffer pointer write operation to the MMIO registers is not seen by the hardware—only writing a ’1’ to the appropriate BFR1_ACK or BFR2_ACK bits signals buffer availability.

The Hardware Bandwidth Error (HBE) flag indicates that the EVO did not get data from SDRAM via the TM1300’s internal data highway in time to continue data transfer or video refresh. Data or video refresh will continue using whatever data is in the EVO internal data buffers. The address counter for the failing buffer(s) will continue to count, and the EVO will continue to request data from the SDRAM over the highway.

The EVO is a read-only device, transferring data from SDRAM to the EVO output port. Unlike Video In, the EVO does not modify SDRAM data. URUN and HBE are the only EVO error conditions that can arise. In the case of URUN or HBE, a scrambled image may be temporarily displayed or incorrect data may be temporarily sent. The EVO can cause no other system hardware error conditions.

Even changing operating modes can not cause system hardware error conditions to arise. For example, changing the MODE bits, the OL_EN and format bits, or the LTL_END bit while the EVO is running may cause wrong data to be displayed or transferred. However, the EVO does not detect this or stop for it.

In normal operation, the user should not change the mode or transfer-control bits while the EVO is enabled. The EVO should be disabled before changing bits such as the MODE bits, the OL_EN bit, or the LTL_END bit. However if these bits are changed while the EVO is running, they will take effect at the beginning of the next field or buffer.

7.17.4 Latency and Bandwidth Requirements

In order to avoid Hardware Bandwidth Error (HBE) conditions, the internal highway bus arbiter (see Chapter 20, “Arbiter”) must be programmed according to the latency requirements of the EVO unit described in this section. In the following discussion, it is assumed that data for video lines (in Y, U, V and overlay planar memory format) is stored in memory aligned on 64-byte boundaries. In other words, it means that the {OL,Y,U,V}_OFFSET fields are multiples of 64 bytes. Otherwise internal EVO arbitration for OL, Y, U and V requests will be different than described here, and the following latencies would not be guaranteed. The EVO uses internal 64-byte buffers.

1. Latency requirements for the EVO in image mode 4:2:2 or 4:2:0 co-sited or interspersed without upscaling and with overlay disabled is expressed as follows.

During 128 EVO clock cycles, the EVO block must have 2 requests acknowledged, that is, ([2Ys, 1U and 1V] / 2). For example, if the EVO clock is 27 MHz, then the EVO must get two requests (128 bytes) from SDRAM in 128 / 027 = 4740 ns.

The byte bandwidth B_{1x} per video line within the active image for this case is:

$$B_{1x} = \left(ceil(\frac{W}{64}) + ceil(\frac{W}{128}) \times 2 + 4\right) \times 64$$

where ceil(*X*) is a function returning the least integral value greater than or equal to *X*, and *W* is the IMAGE_WIDTH field value.

2. In the same modes but with overlay enabled, the latency is as follows:

- During the first 64 EVO clock cycles at least one request must be acknowledged for the OL data.
- During 128 EVO clock cycles, the EVO unit must have 4 requests acknowledged ([4 OLs, 2 Ys, 1 V and 1 U] / 2).

  For example, if the EVO clock runs at 54 MHz then the EVO must get the first request from SDRAM in 64/.054 = 1185 ns and must average a bandwidth latency of 4 requests in 128 / .054 = 2370 ns.

  Byte bandwidth $B_{1x,OL}$ per video line within the active image is then as follows:

$$B_{1xOL} = B_{1x} + \left(ceil(\frac{W}{32}) + 4\right) \times 64$$

3. When the EVO is set to image mode with 2× upscaling, the latency requirements are multiplied by a factor of 2. For example, if 1× mode called for one request per 64 EVO clock cycles, the latency becomes one request per 128 EVO clock cycles. Bandwidth is roughly divided by 2:

$$B_{2x} = \left(ceil(\frac{W}{128}) + ceil(\frac{W}{256}) \times 2 + 4\right) \times 64$$

$$B_{2xOL} = B_{2x} + \left(ceil(\frac{W}{64}) + 4\right) \times 64$$

4. Latency for data-streaming mode or message-passing mode is as follows:

   During 64 EVO clock cycles, the EVO unit must get one request from SDRAM. For example, if the EVO clock runs at 38 MHz, then the latency is 64 / .038 = 1684 ns and bandwidth is 38 MB/s.

7.17.5 Power Down and Sleepless

The EVO block enters in power down state whenever TM1300 is put in global power down mode, except if the SLEEPLESS bit in VO_CTL is set. In the latter case, the block continues DMA operation and will wake up the DSPCPU whenever an interrupt is generated.

The EVO block can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. Refer to Chapter 21, “Power Management.”

It is recommended that EVO be stopped (by negating VO_CTL.ENABLE) before block level power down is started, or that SLEEPLESS mode is used when global power down is activated.

7.18 DDS AND PLL FILTER DETAILS

The PLL filter reduces the phase jitter of the DDS synthesizer output. It can also be used to multiply the DDS output frequency by 2×. The DDS and PLL filter together provide a high-quality, accurately-programmable output video clock. The PLL filter block is shown in Figure 7-32.

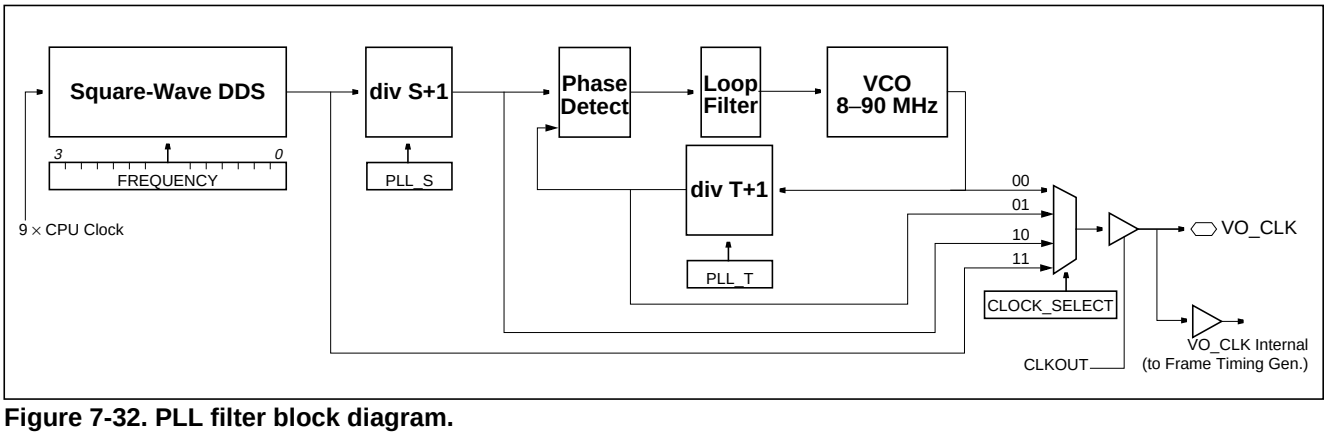


Figure 7-32. PLL filter block diagram.

At hardware reset, the output multiplexer is set to 0x3, and the PLL system is disabled. To start the PLL system, the following steps must be performed:

1. Assign a DDS frequency. This starts the DDS. Allow for at least 31 DSPCPU cycles for the DDS frequency setting to take effect.

2. Choose a value for PLL_S and PLL_T. For 8-40 MHz operation, a value of 1 (which selects division by 2) is recommended.
3. Choose a value for CLOCK_SELECT. For 8-81 MHz operation, CLOCK_SELECT = 00 is recommended.
4. Assign values to the VO_CTL register containing the above choices. The first assignment with CLOCK_SELECT not equal to 0x3 enables the PLL system. Allow for a maximum of 50 microseconds to achieve lock.

Once the PLL is locked, small changes to the DDS frequency are allowed, and the VO_CLK output will smoothly track the frequency change.

Note: Most consumer electronics equipment imposes *very high precision requirements* on the value of the color burst frequency. A video encoder will derive the color burst frequency from VO_CLK. When changing the VO_CLK frequency in software to phase-lock the EVO to a master reference, special care is required to keep the color burst signal frequency within a tolerance of about 50 ppm. When using a Philips DENC (Digital Encoder), the color burst frequency is derived from the master DENC frequency by a programmable synthesizer on the DENC chip. In this case, VO_CLK changes larger than 50 ppm are allowed by changing the DENC synthesizer over its I^2C interface to compensate for the VO_CLK change.

Table 7-11 illustrates recommended settings.

Table 7-11. DDS and PLL example settings

Desired Frequency	DDS frequency	PLL_S	PLL_T	CLOCK_SELECT	Usage
4 – 10 MHz	8 – 20 MHz	1 (divide by 2)	1 (divide by 2)	01 (T divider)	Custom low speed video
8 – 45 MHz	8 – 45 MHz	1 (divide by 2)	1 (divide by 2)	00 (VCO)	Standard or 16:9 digital video
40 – 81 MHz	20 – 40.5 MHz	1 (divide by 2)	3 (divide by 4)	00 (VCO)	High pixel rate custom video

Audio In Chapter 8

by Gert Slavenburg

8.1 AUDIO IN OVERVIEW

The TM1300 Audio In (AI) unit connects to an off-chip stereo A/D converter subsystem through a flexible bit-serial connection. The AI unit provides all signals needed to interface to high quality, low cost oversampling A/D converters, including a generator for a precisely programmable oversampling A/D system clock. Together, the AI unit and external A/D provide the following capabilities:

- One or two channels of audio input.
- 8- or 16-bit samples per channel.
- Programmable sampling rate.
- Internal or external sampling clock source.
- Supports autonomous writes of sampled audio data to memory using double buffering (DMA).
- Supports 8-bit mono and stereo as well as 16-bit mono and stereo PC standard memory data formats.
- Supports little- and big-endian memory formats.

8.2 EXTERNAL INTERFACE

Four TM1300 pins are associated with the AI unit. The AI_OSCLK output is an accurately programmable clock output intended to serve as the master system clock for the external A/D subsystem. The other three pins (AI_SCK, AI_WS and AI_SD) constitute a flexible serial input interface. Using the AI unit's MMIO registers, these pins can be configured to operate in a variety of serial interface framing modes, including but not limited to:

- Standard stereo I^2S (MSB first, 1-bit delay from AI_WS, left & right data in a frame).[1]
- LSB first with 1–16 bit data per channel.
- Complex serial frames of up to 512 bits/frame, with 'valid sample' qualifier bit.

The AI unit can be used with many serial A/D converter devices, including the Philips SAA7366 (stereo A/D), Crystal Semiconductor CS5331, CS5336 (stereo A/D's), CS4218 (codec), Analog Devices AD1847 (codec).

1. A definition of the Philips I^2S serial interface protocol, among others, can be found in the Philips IC01 databook.

Table 8-1. AI unit external signals

Signal	Type	Description
AI_OSCLK	OUT	Over-sampling clock. This output can be programmed to emit any frequency up to 40-MHz with a sub Hertz resolution. It is intended for use as the $256f_s$ or $384f_s$ over sampling clock by external A/D subsystem.
AI_SCK	I/O-5	• When the AI unit is programmed as serial-interface timing slave (power-up default), AI_SCK is an input. AI_SCK receives the serial bitclock from the external A/D subsystem. This clock is treated as fully asynchronous to TM1300 main clock. • When the AI unit is programmed as the serial-interface timing master, AI_SCK is an output. AI_SCK drives the serial clock for the external A/D subsystem. The frequency is a programmable integral divide of the AI_OSCLK frequency. AI_SCK is limited to 22 MHz. The sample rate of valid samples embedded within the serial stream is also limited by the bandwidth.latency available in the system (Section 8-7).
AI_SD	IN-5	Serial data from external A/D subsystem. Data on this pin is sampled on positive or negative edges of AI_SCK as determined by the CLOCK_EDGE bit in the AI_SERIAL register.
AI_WS	I/O-5	• When the AI unit is programmed as the serial-interface timing slave (power-up default), AI_WS acts as an input. AI_WS is sampled on the same edge as selected for AI_SD. • When the AI unit is programmed as the serial-interface timing master, AI_WS acts as an output. It is asserted on the opposite edge of the AI_SD sampling edge. AI_WS is the word-select or frame-synchronization signal from/to the external A/D subsystem.

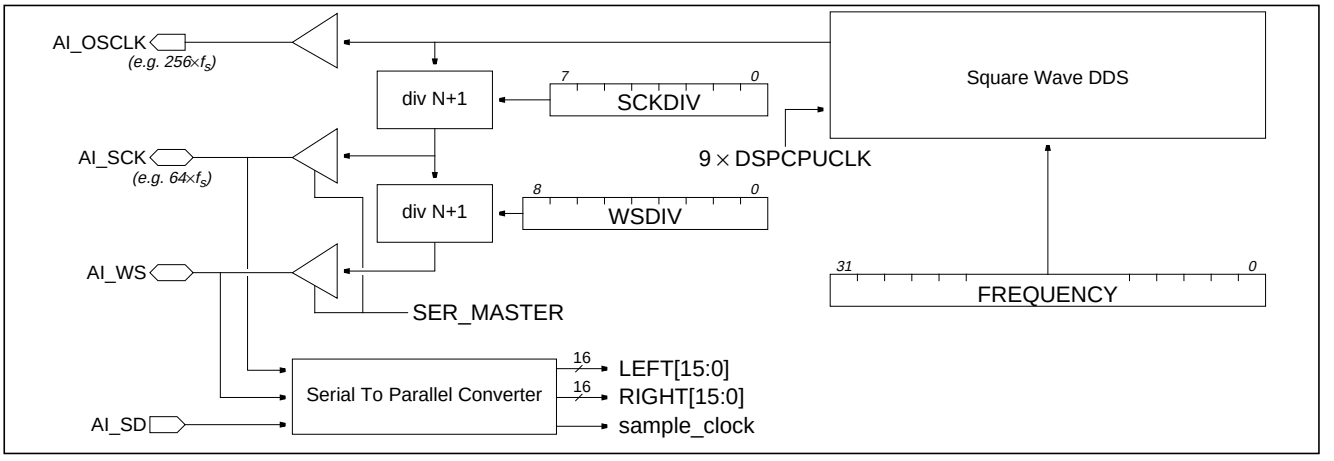


Figure 8-1. AI clock system and I/O interface.

8.3 CLOCK SYSTEM

Figure 8-1 illustrates the different clock capabilities of the AI unit. At the heart of the clock system is a square wave DDS (Direct Digital Synthesizer). The DDS can be programmed to emit frequencies from approx. 1 Hz to 40 MHz with a resolution of better than 0.3 Hz.

The output of the DDS is always sent on the AI_OSCLK output pin. This output is intended to be used as the $256f_S$ or $384f_S$ system clock source instead of a fixed frequency crystal for oversampling A/D converters, such as the Philips SAA7366T, or Analog Devices AD1847.

The TM1300 AI DDS frequency is set by writing to the FREQUENCY MMIO register. The programmer can change the FREQUENCY setting dynamically, so as to adjust the input sampling rate to track an application dependent master reference.

Depending on bit 31 (MSB), the DDS runs in one of two modes:

- bit 31 = 1 (TM1300 improved mode)
- bit 31 = 0 (TM1000 compatibility mode)

8.3.1 TM1300 Improved Mode

In improved mode, a high quality, low-jitter AI_OSCLK is generated. The setting of the FREQUENCY register to accomplish a given AI_OSCLK frequency is given by:

$$FREQUENCY = 2^{31} + \frac{f_{OSCLK} \cdot 2^{32}}{9 \cdot f_{DSPCPU}}$$

This mode, and the above formula, should be used for all new software development on TM1300. It is not available on TM1000.

8.3.2 TM1000 Compatibility Mode

TM1000 compatibility mode is provided so that TM1000 software runs without changes. It should NOT be used for new TM1300 software development. TM1000 mode is automatically entered whenever FREQUENCY[31] = 0. In TM1000 mode, AI_OSCLK frequency is set as follows:

$$FREQUENCY = \frac{f_{OSCLK} \cdot 2^{32}}{3 \cdot f_{DSPCPU}}$$

8.4 CLOCK SYSTEM OPERATION

AI_SCK and AI_WS can be configured as input or output, as determined by the SER_MASTER control field. As output, AI_SCK is a divider of the DDS output frequency. Whether input or output, the AI_SCK pin signal is used as the bit clock for serial-parallel conversion.

$$f_{AISCK} = \frac{f_{AIOSCLK}}{SCKDIV + 1} \qquad SCKDIV \in [0,255]$$

If set as output, AI_WS can similarly be programmed using WSDIV to control the serial frame length from 1 to 512 bits.

Table 8-2. Sample rate settings ($f_{DSPCPUCLK}$=133 MHz, improved TM1300 mode)

f_s	OSCLK	SCK	FREQUENCY	SCKDIV
44.1 kHz	$256f_s$	$64f_s$	2187991971	3
48.0 kHz	$256f_s$	$64f_s$	2191574340	3
44.1 kHz	$384f_s$	$64f_s$	2208246133	5
48.0 kHz	$384f_s$	$64f_s$	2213619686	5

The preferred application of the clock system options is to use AI_OSCLK as A/D master clock, and let the A/D converter be timing master over the serial interface (SER_MASTER=0).

In case an external codec (e.g. the AD1847 or CS4218) is used for common audio I/O, it may not be possible to independently control the A/D and D/A system clocks. In that case it is recommended that the Audio Out (AO) unit clock system DDS is used to provide a single master A/D and D/A clock. The AO unit, or the D/A converter, can be used as serial interface timing master, and the AI unit is set to be slave to the serial frame determined by AO

(AI SER_MASTER=0, AI_SCK and AI_WS externally wired to the corresponding AO pins). In such systems, independent software control over A/D and D/A sampling rate is not possible, but component count is minimized.

Table 8-3.AI MMIO clock & interface control bits

Field Name	Description
SER_MASTER	0 ⇒ (RESET default), the A/D converter is the timing master over the serial interface. AI_SCK and AI_WS are set to be inputs. 1 ⇒ TM1300 is timing master over the AI serial interface. The AI_SCK and AI_WS pins are set to be outputs.
FREQUENCY	Sets the clock frequency emitted by the AI_OSCLK output. RESET default 0.
SCKDIV	Sets the divider used to derive AI_SCK from AI_OSCLK. Set to 0..255, for division by 1..256. RESET default 0.
WSDIV	Sets the divider used to derive AI_WS from AI_SCK. Set to 0..511 for a serial frame length of 1..512. RESET default 0.

8.5 SERIAL DATA FRAMING

The AI unit can accept data in a wide variety of serial data framing conventions. Figure 8-2 illustrates the notion of a serial frame. If POLARITY=1 and CLOCK_EDGE=0, a frame is defined with respect to the positive transition of the AI_WS signal, as observed by a positive clock transition on AI_SCK. Each data bit sampled on positive AI_SCK transitions has a specific bit position: the data bit sampled on the clock edge after the clock edge on which the AI_WS transition is seen has bit position 0. Each subsequent clock edge defines a new bit position. As defined in Table 8-4, other combinations of POLARITY and CLOCK_EDGE can be used to define a variety of serial frame bitposition definitions.

The capturing of samples is governed by FRAMEMODE. If FRAMEMODE=00, every serial frame results in one sample from the serial-parallel converter. A sample is defined as a left/right pair in stereo modes or a single left channel value in mono modes. If FRAMEMODE=1y, the serial frame data bit in bit position VALIDPOS is examined. If it has value ‘y’, a sample is taken from the data stream (the valid bit is allowed to precede or follow the left or right channel data provided it is in the same serial frame as the data).

The left and right sample data can be in a LSB-first or MSB-first form, at an arbitrary bit position, and with an arbitrary length.

Table 8-4. AI MMIO serial framing control fields

Field Name	Description
POLARITY	0 ⇒ serial frame starts on AI_WS negedge (RESET default) 1 ⇒ serial frame starts on AI_WS posedge
FRAMEMODE	00 ⇒ accept a sample every serial frame (RESET default) 01 ⇒ unused, reserved 10 ⇒ accept sample if valid bit = 0 11 ⇒ accept sample if valid bit = 1
VALIDPOS	• Defines the bit position within a serial frame where the valid bit is found. • Default 0.
LEFTPOS	• Defines the bit position within a serial frame where the first data bit of the left channel is found. • Default 0.
RIGHTPOS	• Defines the bit position within a serial frame where the first data bit of the right channel is found. • Default 0.
DATAMODE	0 ⇒ MSB first (RESET default) 1 ⇒ LSB first
SSPOS	• Start/Stop bit position. Default 0. • If DATAMODE=MSB first, SSPOS determines the bit index (0..15) in the parallel word of the last data bit. Bits 15 (MSB) up to/including SSPOS are taken in order from the serial frame data. All other bits are set to ‘0’. • If DATAMODE=LSB first, SSPOS determines the bit index (0..15) in the parallel word of the first data bit. Bits SSPOS up to/including 15 are taken in order from the serial frame data. All other bits are set to ‘0’.
CLOCK_EDGE	• if ‘0’(RESET default) the AI_SD and AI_WS pins are sampled on positive edges of the AI_SCK pin. If SER_MASTER =1, AI_WS is asserted on negative edges of AI_SCK. • if 1, AI_SD and AI_WS are sampled on negative edges of AI_SCK. As output, AI_WS is asserted on positive edges of AI_SCK.

In MSB-first mode, the serial-to-parallel converter assigns the value of the bit at LEFTPOS to LEFT[15]. Subsequent bits are assigned, in order, to decreasing bit positions in the LEFT data word, up to and including LEFT[SSPOS]. Bits LEFT[SSPOS–1:0] are cleared. Hence, in MSB-first mode, an arbitrary number of bits are captured. They are left-adjusted in the 16-bit parallel output of the converter.

In LSB-first mode, the serial to parallel converter assigns the value of the bit at LEFTPOS to LEFT[SSPOS]. Sub-

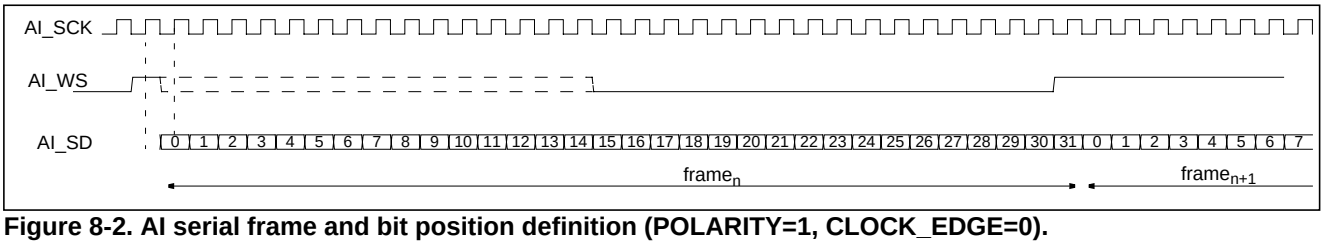


Figure 8-2. AI serial frame and bit position definition (POLARITY=1, CLOCK_EDGE=0).

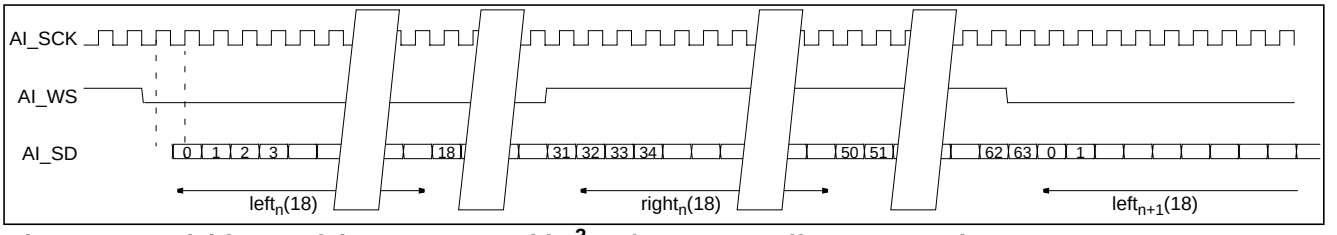


Figure 8-3. Serial frame of the SAA7366 18 bit I^{2}S A/D converter (format 2 SWS).

sequent bits are assigned, in order, to increasing bit positions in the LEFT data word, up to and including LEFT[15]. Bits LEFT[SSPOS–1:0] are cleared. Hence, in LSB-first mode, an arbitrary number of bits are captured. They are returned left-adjusted in the 16-bit parallel output of the converter.

Table 8-5. Example setup for SAA7366

Field	Value	Explanation
SER_MASTER	0	SAA7366 is serial master
FREQUENCY	161628209	256f$_S$ 44.1 kHz
SCKDIV	3	AI_SCK set to AI_OSCLK/4 (not needed since SER_MASTER=0)
WSDIV	63	Serial frame length of 64 bits (not needed since SER_MASTER=0)
POLARITY	0	Frame starts with neg. AI_WS
FRAMEMODE	00	Take a sample each ser. frame
VALIDPOS	n/a	Don't care
LEFTPOS	0	Bit position 0 is MSB of left channel and will go to LEFT[15]
RIGHTPOS	32	Bit position 32 is MSB of right channel and will go to RIGHT[15]
DATAMODE	0	MSB first
SSPOS	0	Stop with LEFT/RIGHT[0]
CLOCK_EDGE	0	Sample WS and SD on positive SCK edges for I^{2}S

Refer to Figure 8-3 and Table 8-5 to see an example of how the AI unit MMIO registers are set to collect 16-bit samples using the Philips SAA7366 I^{2}S 18-bit A/D converter. This setup assumes the SAA7366 acts as the serial master.

For example, if it were desirable to use only the 12 MSBs of the A/D converter in Figure 8-3, use the settings of Table 8-5 with SSPOS set to '4'. This results in LEFT[15:4] being set with data bits 0..11, and LEFT[3:0] being set to '0'. RIGHT[15:4] is set with data bits 32..43 and RIGHT[3:0] is set to '0'.

8.6 MEMORY DATA FORMATS

The AI unit autonomously writes samples to memory in mono and stereo 8- and 16-bits per sample formats, as shown in Figure 8-4. Successive samples are always stored at increasing memory address locations. The setting of the LITTLE_ENDIAN bit in the AI_CTL register determines how increasing memory addresses map to byte positions within words. Refer to Appendix C, "Endian-ness," for details on byte ordering conventions.

The AI hardware implements a double buffering scheme to ensure that no samples are lost, even if the DSPCPU is highly loaded and slow to respond to interrupts. The DSPCPU software assigns buffers by writing a base address and size to the MMIO control fields described in Table 8-6. Refer to Section 8.7 for details on hardware/software synchronization.

In 8-bit capture modes, the eight MSBs of the serial parallel converter output data are written to memory. In 16-bit capture modes, all bits of the parallel data are written to memory. If SIGN_CONVERT is set to '1', the MSB of the data is inverted, which is equivalent to translating from two's complement to offset binary representation. This allows the use of an external two's complement 16-bit A/D converter to generate 8-bit unsigned samples, which is often used in PC audio.

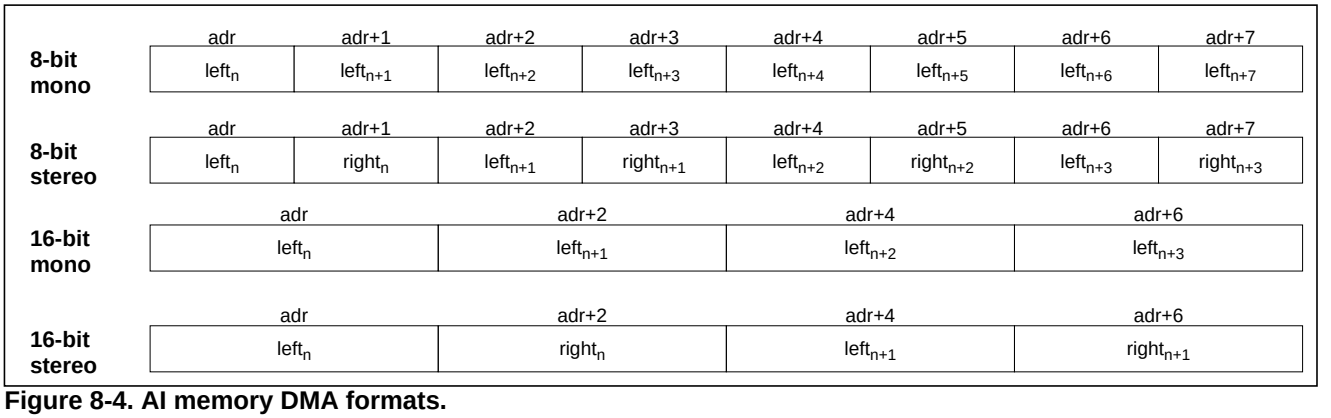


Figure 8-4. AI memory DMA formats.

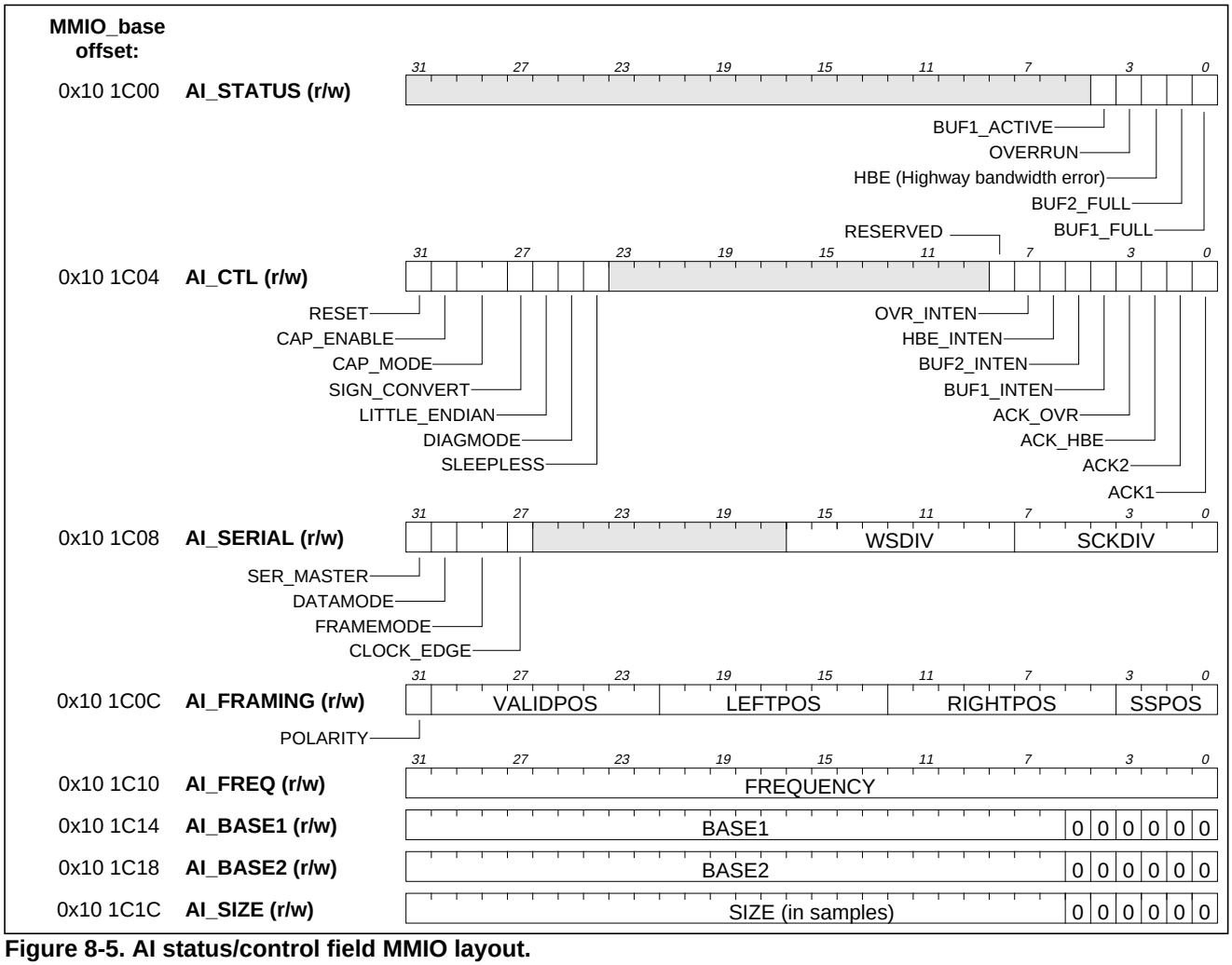


Figure 8-5. AI status/control field MMIO layout.

Table 8-6. AI MMIO DMA control fields

Field Name	Description
LITTLE_ENDIAN	0 ⇒ capture in big endian memory format (RESET default) 1 ⇒ capture little endian
BASE1	Base address of buffer1; a 64-byte aligned address in local SDRAM. RESET default 0.
BASE2	Base address of buffer2; a 64-byte aligned address in local SDRAM. RESET default 0.
SIZE	• Number of samples to be placed in buffer before switching to other buffer • Stereo modes: a pair of 8- or 16-bit data is 1 sample • Mono modes: a single value is 1 sample • RESET default 0.
CAP_MODE	00 ⇒ mono (left ADC only), 8 bits/sample. (RESET default). 01 ⇒ stereo, 2 times 8 bits/sample 10 ⇒ mono (left ADC only), 16 bits/sample 11 ⇒ stereo, 2 times 16 bits/sample
SIGN_CONVERT	0 ⇒ leave MSB unchanged (RESET default) 1 ⇒ invert MSB

Note that the AI hardware does *not* generate A-law or µ-law 8-bit data formats. If such formats are desired, the DSPCPU can be used to convert from 16-bit linear data to A-law or µ-law data.

8.7 AUDIO IN OPERATION

Figure 8-5, Table 8-9 and Table 8-8 describe the function of the control and status fields of the AI unit. To ensure compatibility with future devices, undefined bits in MMIO registers should be ignored when read, and written as ’0’s.

The AI unit is reset by a TM1300 hardware reset, or by writing 0x80000000 to the AI_CTL register. Upon RESET, capture is disabled (CAP_ENABLE = 0), and buffer1 is the active buffer (BUF1_ACTIVE=1). A minimum of 5 valid AI_SCK clock cycles is required to allow internal AI circuitry to stabilize before enabling capture. This can be accomplished by programming AI_FREQ and AI_SERIAL and then delaying for the appropriate time interval.

Programing of the AI_SERIAL MMIO register needs to follow the following sequence order:

- set AI_FREQ to ensure that a valid clock is generated (Only when AI is the master of the audio clock system)
- MMIO(AI_CTL) = 1 << 31; /* Software Reset */
- MMIO(AI_SERIAL) = 1 << 31; /* sets serial-master mode, starts AI_SCK */
- MMIO(AI_SERIAL) = (1 << 31) | (SCKDIV value); /* then set DIVIDER values */

The DSPCPU initiates capture by providing two equal size empty buffers and putting their base address and size in the $BASE_n$ and SIZE registers. Once two valid (local memory) buffers are assigned, capture can be enabled by writing a '1' to CAP_ENABLE. The AI unit hardware now proceeds to fill buffer 1 with input samples. Once buffer 1 fills up, BUF1_FULL is asserted, and capture continues without interruption in buffer 2. If BUF1_INTEN is enabled, a SOURCE 11 interrupt request is generated.

Note that the buffers must be 64-byte aligned, and a multiple of 64 samples in size (the six LSBs of AI_BASE1, AI_BASE2 and AI_SIZE are always '0').

The DSPCPU is required to assign a new, empty buffer to BASE1 and perform an ACK1, before buffer 2 fills up. Capture continues in buffer 2, until it fills up. At that time, BUF2_FULL is asserted, and capture continues in the new buffer 1, etc.

Upon receipt of an ACK, the AI hardware removes the related interrupt request line assertion at the next DSPCPU clock edge. Refer to Section 3.5.3, "INT and NMI (Maskable and Non-Maskable Interrupts)," for the rules regarding ACK and interrupt re-enabling. The AI interrupt should always be operated in level-sensitive mode, since AI can signal multiple conditions that each need independent ACKs over the single internal SOURCE 11 request line.

In normal operation, the DSPCPU and AI hardware continuously exchange buffers without ever loosing a sample. If the DSPCPU fails to provide a new buffer in time, the OVERRUN error flag is raised. This flag is *not affected* by ACK1 or ACK2; it can only be cleared by an explicit ACK_OVR.

8.8 POWER DOWN AND SLEEPLESS

The AI unit enters power down state whenever TM1300 is put in global power down mode, except if the SLEEPLESS bit in AI_CTL is set. In the latter case, the unit continues DMA operation and will wake up the DSPCPU whenever an interrupt is generated.

The AI unit can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. Refer to Chapter 21, "Power Management."

It is recommended that AI be stopped (by negating AI_CTL.CAP_ENABLE) before block level power down is started, or that SLEEPLESS mode is used when global power down is activated.

8.9 HIGHWAY LATENCY AND HBE

The AI unit uses internal buffering before writing data to SDRAM. The internal buffer consists of one stereo sample input holding register and 64 bytes of internal buffer memory. Under normal operation, the 64-byte buffer is written to SDRAM while the input register receives another sample. This normal operation is guaranteed to be maintained as long as the highway arbiter is set to guarantee a latency for the AI unit that matches the sampling interval. Given a sample rate f_S, and an associated sample interval T (in nsec), the arbiter should be set to have a latency of at most T-20 nsec. Refer to Chapter 20, "Arbiter," for information on arbiter programming. If the requested latency is not adequate, the HBE (Highway Bandwidth Error) condition may result. This error flag gets set when the input register is full, the 64-byte buffer has not yet been written to memory, and a new sample arrives.

Table 8-7 shows the required arbiter latency settings for a number of common operating modes. The rightmost column illustrates the nature of the resulting 64-byte highway requests. Is not necessary to compute arbiter settings, but they may be used to compute bus availability in a given interval.

Table 8-7. AI highway arbiter latency requirement examples

CapMode	f_S (kHz)	T (nS)	max arbiter latency (nsec)	access pattern
stereo 16 bits/sample	44.1	22,676	22,656	1 request every 362,812 nsec
stereo 16 bits/sample	48.0	20,833	20,813	1 request every 333,333 nsec
stereo 16 bits/sample	96.0	10,417	10,397	1 request every 166,667 nsec

Table 8-8. AI MMIO status fields (read only)

Field Name	Description
BUF1_ACTIVE	• If '1', buffer 1 will be used for the next incoming sample. If '0', buffer 2 will receive the next sample. • 1 after RESET.
BUF1_FULL	• If '1', buffer 1 is full. If BUF1_INTEN is also '1', an interrupt request (source 11) is pending. BUF1_FULL is cleared by writing a '1' to ACK1, at which point the AI hardware will assume that BASE1 and SIZE describe a new empty buffer. • 0 after RESET.

Table 8-8. AI MMIO status fields (read only)

Field Name	Description
BUF2_FULL	• If ‘1’, buffer 2 is full. If BUF2_INTEN is also ‘1’, an interrupt request (source 11) is pending. BUF2_FULL is cleared by writing a ‘1’ to ACK2, at which point the AI hardware will assume that BASE2 and SIZE describe a new empty buffer. • 0 after RESET.
HBE	• Highway Bandwidth Error. Condition raised when the 64-byte internal AI buffer is not yet written to SDRAM when a new input sample arrives. Indicates insufficient allocation of TM1300 highway bandwidth for the audio sampling rate/mode. Refer to Chapter 20, “Arbiter.” • 0 after RESET.
OVERRUN	• OVERRUN error occurred, i.e. the CPU did not provide an empty buffer in time, and 1 or more samples were lost. If OVR_INTEN is also 1, an interrupt request (source 11) is pending. The OVERRUN flag can ONLY be cleared by writing a ‘1’ to ACK_OVR. • 0 after RESET.

Table 8-9. AI MMIO control fields

Field Name	Description
RESET	The AI logic is reset by writing a 0x80000000 to AI_CTL. This bit always reads as a ‘0’. See Section 8.7, “Audio In Operation” for details on software reset.
DIAGMODE	0 ⇒ normal operation (RESET default) 1 ⇒ diagnostic mode (see Section 8.11, “Diagnostic Mode”)
SLEEPLESS	0 ⇒ participate in global power down (RESET default) 1 ⇒ refrain from participating in power down
CAP_ENABLE	Capture Enable flag. If 1, AI unit captures samples and acts as DMA master to write samples to local SDRAM. If ’0’ (RESET default), AI unit is inactive.
BUF1_INTEN	Buffer 1 full Interrupt Enable. Default 0. 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if buffer 1 full
BUF2_INTEN	Buffer 2 full interrupt enable. Default 0 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if buffer 2 full
HBE_INTEN	HBE Interrupt Enable. Default 0. 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if a highway bandwidth error occurs.

Table 8-9. AI MMIO control fields

Field Name	Description
OVR_INTEN	Overrun Interrupt Enable. Default 0 0 ⇒ no interrupt 1 ⇒ interrupt (SOURCE 11) if an overrun error occurs
ACK1	Write a ’1’ to clear the BUF1_FULL flag and remove any pending BUF1_FULL interrupt request. This bit always reads as 0.
ACK2	Write a ’1’ to clear the BUF2_FULL flag and remove any pending BUF2_FULL interrupt request. This bit always reads as 0.
ACK_HBE	Write a ’1’ to clear the HBE flag and remove any pending HBE interrupt request. This bit always reads as 0.
ACK_OVR	Write a ’1’ to clear the OVERRUN flag and remove any pending OVERRUN interrupt request. This bit always reads as 0.

8.10 ERROR BEHAVIOR

If either an OVERRUN or HBE error occurs, input sampling is temporarily halted, and samples will be lost. In case of OVERRUN, sampling resumes as soon as the DSPCPU makes one or more new buffers available through an ACK1 or ACK2 operation. In the case of HBE, sampling will resume as soon as the internal buffer is written to SDRAM.

HBE and OVERRUN are ‘sticky’ error flags. They will remain set until an explicit ACK_HBE or ACK_OVR.

8.11 DIAGNOSTIC MODE

Diagnostic mode is entered by setting the DIAGMODE bit in the AI_CTL register. In diagnostic mode, the AI_SCK, AI_WS and AI_SD inputs of the serial-parallel converter are taken from the output pins of the TM1300 AO unit. This mode can be used during the diagnostic phase of system boot to verify correct operation of most of the AI unit and AO unit logic circuitry.

Note that the inputs are truly taken from the TM1300 AO external pins, i.e. if an external (board level) source is driving AO_SCK or AO_WS, diagnostic mode is not capable of testing Audio Out.

Special care must be taken to enable diagnostic mode. The recommended way of entering diagnostic mode is:

- setup the AO unit such that an AO_SCK is generated
- set DIAGMODE bit followed by a 5 (AI_SCK) cycle delay
- perform a software reset of the AI unit and immediately set the DIAGMODE bit back to ‘1’.

Audio Out Chapter 9

by Gert Slavenburg, Santanu Dutta

9.1 AUDIO OUT OVERVIEW

The TM1300 Audio Out (AO) unit is new and contains many features not available in the TM1100. It has up to 8 channels, and drives up to 4 external stereo D/A converters through a flexible bit-serial connection.

It provides all signals to interface to high quality, low cost oversampling D/A converters, including a precisely programmable oversampling D/A system clock. The AO unit and external D/A's together provide the following capabilities:

- Up to 8 channels of audio output.
- 16-bit or 32-bit samples per channel.
- Programmable sampling rate.
- Internal or external sampling clock source.
- Autonomously reads processed audio data from memory using double buffering (DMA).
- Supports 16-bit mono and stereo PC standard memory data formats.
- Supports little- and big-endian memory formats.
- Provides control capability for highly integrated PC codecs such as the AD1847, CS4218 or UAD1340.

9.2 NEW AND CHANGED FEATURES

- Individual serial data outputs to each D/A
- 32-bit samples
- No 8-bit sample support

No support for connecting several D/As to one serial data output.

Table 9-1. AO unit external signals

Signal	Type	Description
AO_OSCLK	OUT	Over sampling clock. Can be programmed to emit any frequency up to 40 MHz, with sub-Hz resolution. Intended for use as the 256 or 384f_s oversampling clock by the external D/A conversion subsystem.

Table 9-1. AO unit external signals

Signal	Type	Description
AO_SCK	IO	• When AO is programmed to act as a serial interface timing slave (RESET default), AO_SCK acts as input. It receives the serial clock from the external audio D/A subsystem. The clock is treated as fully asynchronous to the TM1300 main clock. • When AO is programmed to act as serial interface timing master, AO_SCK acts as output. It drives the serial clock for the external audio D/A subsystem. Clock frequency is a programmable integral divide of the AO_OSCLK frequency. AO_SCK is limited to 22 MHz. The sample rate of valid samples embedded within the serial stream is limited by the AO_SCK maximum frequency and the available highway bandwidth.
AO_WS	IO	• When AO is programmed as the serial-interface timing slave (RESET default), AO_WS acts as an input. AO_WS is sampled on the opposite AO_SCK edge at which AO_SDx are asserted. • When AO is programmed as serial-interface timing master, AO_WS acts as an output. AO_WS is asserted on the same AO_SCK edge as AO_SDx. AO_WS is the word-select or frame-sync signal from/to the external D/A subsystem. Each audio channel receives 1 sample for every WS period. AO_WS can be set to change on AO_OSCLK positive or negative edges by the CLOCK_EDGE bit.
AO_SD1	OUT	Serial data to stereo external audio D/A subsystem. AO_SD1 can be set to change on AO_OSCLK positive or negative edges by the CLOCK_EDGE bit.
AO_SD2	OUT	Serial data to stereo external audio D/A subsystem. AO_SD2 can be set to change on AO_OSCLK positive or negative edges by the CLOCK_EDGE bit.
AO_SD3	OUT	Serial data to stereo external audio D/A subsystem. AO_SD3 can be set to change on AO_OSCLK positive or negative edges by the CLOCK_EDGE bit.
AO_SD4	OUT	Serial data to stereo external audio D/A subsystem. AO_SD4 can be set to change on AO_OSCLK positive or negative edges by the CLOCK_EDGE bit.

9.3 EXTERNAL INTERFACE

Seven TM1300 pins are associated with the AO unit. The AO_OSCLK output is an accurately programmable clock output intended to be used as the master system clock for the external D/A subsystem. The other pins (AO_SCK, AO_WS and AO_SDx) constitute a flexible serial output interface. Using the AO MMIO registers, these pins can be configured to operate in a variety of serial interface framing modes, including but not limited to:

- Standard stereo I^2S (MSB first, 1-bit delay from AO_WS, left & right data in a frame).
- LSB first, with 1–16-bit data per channel.
- Complex serial frames of up to 512 bits/frame.

9.4 SUMMARY OF OPERATION

The AO unit consists of three major subsystems, a programmable sample clock generator, a DMA engine and a data serializer.

The DMA engine reads 16 or 32-bit samples from memory using a double buffered DMA approach. The DSPCPU initially assigns two full sample buffers containing an integral number of samples for all active channels. The DMA engine retrieves samples from the first buffer until exhausted and continues from the second buffer, while requesting a new first sample buffer from the DSPCPU, etc.

The samples are given to the data serializer, which sends them out in a MSB first or LSB first serial frame format that can also contain 1 or 2 codec control words of up to 16 bits. The frame structure is highly programmable by a series of MMIO fields.

9.5 INTERNAL CLOCK SOURCE

Figure 9-1 illustrates the different clock capabilities of the AO unit. At the heart of the clock system is a square wave DDS (Direct Digital Synthesizer). The DDS can be programmed to emit frequencies from approx. 1 Hz to 80 MHz with a sub Hertz resolution.

The output of the DDS is always sent to the AO_OSCLK output pin. This output is intended to be used as the $256f_s$ or $384f_s$ system clock source for oversampling D/A converters, such as the Philips SAA7322, or codecs such as the AD1847, CS4218, or UAD1340.

The TM1300 DDS frequency is set by writing to the FREQUENCY MMIO register. The programmer is free to change the FREQUENCY setting dynamically, in order to adjust the outgoing audio sample rate. In ATSC transport stream decoding, this is the method by which the system software locks audio output sample rate to the original program provider sample rate.

Depending on bit 31 (MSB), the DDS runs in one of the two following modes:

- bit 31 = 1 (standard mode)
- bit 31 = 0 (TM1000 compatibility mode)

9.5.1 TM1300 Standard Mode

This mode was first available in the TM1100. In this mode, a high quality, low-jitter AO_OSCLK is generated. The setting of the FREQUENCY register to accomplish a given AO_OSCLK frequency is given by the formula:

$$FREQUENCY = 2^{31} + \frac{f_{OSCLK} \cdot 2^{32}}{9 \cdot f_{DSPCPU}}$$

This mode, and the above formula, should be used for all new software development on TM1300.

Table 9-2. Clock system setting (f_{DSPCPU}=133 MHz)

f_s	OSCLK	SCK	FREQUENCY	SCKDIV
44.1 kHz	256fs	64fs	2187991971	3
48.0 kHz	256fs	64fs	2191574340	3
44.1 kHz	384fs	64fs	2208246133	5
48.0 kHz	384fs	64fs	2213619686	5

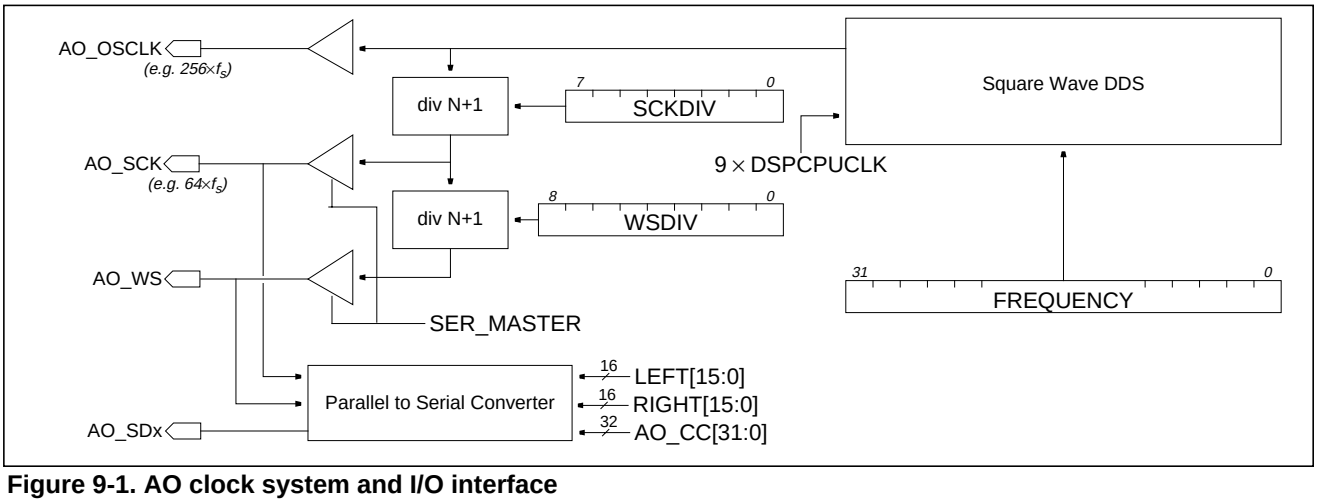


Figure 9-1. AO clock system and I/O interface

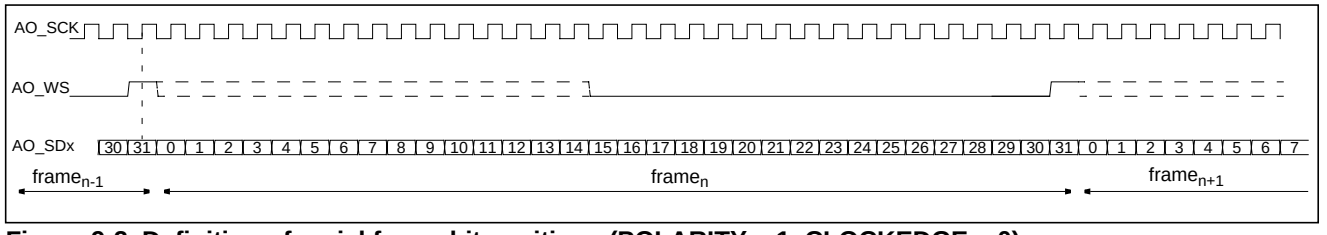


Figure 9-2. Definition of serial frame bit positions (POLARITY = 1, CLOCKEDGE = 0)

9.5.2 TM1000 Clock Compatibility Mode

TM1000 clock compatibility mode is provided so that TM1000 audio software runs without changes. It should NOT be used for new software development, due to a 3x higher jitter. TM1000 mode is automatically entered whenever FREQUENCY[31] = 0. In TM1000 mode, AO_OSCLK frequency is set as follows:

$$FREQUENCY = \frac{f_{OSCLK} \cdot 2^{32}}{3 \cdot f_{DSPCPU}}$$

9.6 CLOCK SYSTEM OPERATION

The output of the DDS is always sent to the AO_OSCLK output pin. This output is typically used as the $256f_s$ or $384f_s$ system clock source for oversampling D/A converters, such as the Philips SAA7322, or codecs such as the AD1847, CS4218 or UD1340.

AO_WS and AO_SCK are sent to each external D/A converter in the master mode.

AO_WS, the word strobe, determines the sample rate: each active channel receives one sample for each AO_WS period.

AO_SCK is the data bit clock. The number of AO_SCK clocks in an AO_WS period is the number of data bits in a serial frame required by the attached D/A converter.

$$f_{AOSCK} = \frac{f_{AOOSCLK}}{SCKDIV + 1} \qquad SCKDIV \in [0,255]$$

AO_WS is a divider of the bit clock and is set using WSDIV to control the serial frame length. The number of bits per frame is equal to WSDIV+1. There are some minimum length requirements for a serial frame, refer to Section 9.7.1.

AO_SCK and AO_WS can be configured as input or output, as determined by the SER_MASTER control field. If set as output, AO_SCK can be set to a divider of the DDS output frequency.

Whether set as input or output, the AO_SCK pin signal is always used as the bit clock for parallel-serial conversion. The AO_WS pin always acts as the trigger to start the generation of a serial frame. AO_WS can similarly be programmed using WSDIV to control the serial frame length. The number of bits per frame is equal to WSDIV+1.

The preferred use of the clock system options is to use AO_OSCLK as D/A master clock, and let the D/A converter be a timing slave of the serial interface (SER_MASTER=1). This is important in view of compatibility with future Trimedia devices, which may only support the AO unit as serial interface master.

Table 9-3. AO MMIO Clock & Interface Control

Field Name	Description
SER_MASTER	0 ⇒ (RESET default), the D/A subsystem is the timing master over the AO serial interface. AO_SCK and AO_WS act as inputs. 1 ⇒ TM1300 is the timing master over the serial interface. AO_SCK and AO_WS act as outputs. This mode is required for 4,6 or 8 channel operation. The SER_MASTER bit should only be changed while the AO unit is disabled, i.e. TRANS_ENABLE = 0.
FREQUENCY	Sets the clock frequency emitted by the AO_OSCLK output. RESET default 0.
SCKDIV	Sets the divider used to derive AO_SCK from AO_OSCLK. Set to 0..255, for division by 1..256. RESET default 0.
WSDIV	Sets the divider used to derive AO_WS from AO_SCK. Set to 0..511 for a serial frame length of 1..512. RESET default 0.

Some D/A converters however, like the AD1847, provide better SNR properties if they are configured as serial master, with the AO unit as slave (SER_MASTER=0). As illustrated by Figure 9-1, the internal parallel to serial converter that constructs the serial frame is oblivious to which component is timing master.

9.7 SERIAL DATA FRAMING

The AO unit can generate data in a wide variety of serial data framing conventions. Figure 9-2 illustrates the notion of a serial frame. If POLARITY=1, a frame starts with a positive edge of the AO_WS signal. If POLARITY=0, a serial frame starts with a negative edge on AO_WS. If CLOCK_EDGE=0, the parallel to serial converter samples AO_WS on a positive clock edge transition, and outputs the first bit (bit 0) of a serial frame on the next falling edge of AO_SCK.

If CLOCK_EDGE=1, the parallel to serial converter samples AO_WS on the negative edge of AO_SCK, while audio data is output on the positive edge, i.e. the AO_SCK polarity would be reversed with respect to Figure 9-2.

Table 9-4. AO Serial Framing Control Fields

Field Name	Description
POLARITY	0 ⇒ serial frame starts with an AO_WS negedge (RESET default) 1 ⇒ serial frame starts with an AO_WS posedge This bit should NOT be changed during operation of the AO unit, i.e. only update this bit when TRANS_ENABLE = 0.
LEFTPOS(9)	Defines the bit position within a serial frame where the first data bit of the left channel is placed. Reset default '0'.
RIGHTPOS(9)	Defines the bit position within a serial frame where the first data bit of the right channel is placed. Reset default '0'.
DATAMODE	0 ⇒ MSB first (RESET default) 1 ⇒ LSB first
SSPOS	Start/Stop bit position. Reset default 0. Note that SSPOS is a 5-bit field, with SSPOS bit 4 not-adjacent. This is for backwards compatibility in 16 bits/sample modes with TM1000/1100. • If DATAMODE=MSB first, transmission starts with the MSB of the sample, i.e. bit 15 for 16 bits/sample modes or bit 31 for 32 bits/sample modes. SSPOS determines the bit index (0..31) in the parallel input word of the last transmitted data bit. • If DATAMODE=LSB first, SSPOS determines the bit index (0..31) in the parallel word of the first transmitted data bit. Bits SSPOS up to/including the MSB are transmitted, i.e. up to bit 15 in 16 bits/sample mode and bit 31 in 32 bits/sample mode. See Table 9-5 for more information.
CLOCK_EDGE	0 ⇒ the parallel to serial converter samples AO_WS on positive edges of AO_SCK and outputs data on the negative edge of AO_SCK (RESET default). 1 ⇒ the parallel to serial converter samples AO_WS on negative edges of AO_SCK and outputs data on positive edges of AO_SCK.
WS_PULSE	0 ⇒ emit 50% AO_WS (RESET default). 1 ⇒ emit single AO_SCK cycle AO_WS
NR_CHAN	00 ⇒ Only AO_SD1 is active 01 ⇒ AO_SD1 and 2 are active 10 ⇒ AO_SD1, 2 and 3 are active 11 ⇒ AO_SD1..SD4 are active Each SD output either receives 1 or 2 channels depending on TRANS_MODE mono resp. stereo. Non-active channels receive 0 value samples. In mono modes, each channel of a SD output receives identical left & right samples. See also Table 9-9.

Every serial frame transmits a single left and right channel sample, and optional codec control data to each D/A converter. The left and right sample data can be in an LSB first or MSB first form, at an arbitrary serial frame bit position, and with an arbitrary length.

In MSB-first mode (DATAMODE = 0), the parallel to serial converter sends the value of LEFT[MSB] in bit position LEFTPOS in the serial frame. Subsequently, bits from decreasing bit positions in the LEFT data word, up to and including LEFT[SSPOS], are transmitted in order.

In LSB-first mode (DATAMODE = 1), the parallel-to-serial converter sends the value of LEFT[SSPOS] in bit position LEFTPOS in the serial frame. Subsequent bits from the LEFT data word, up to and including LEFT[MSB], are transmitted in order. Table 9-5. shows the transmitted bits in different modes.

Table 9-5. Bits transmitted for each memory data item S

operating mode	first bit	last bit	valid SSPOS values
16 bits/sample, MSB-first	S[15]	S[SSPOS]	0..15
16 bits/sample, LSB-first	S[SSPOS]	S[15]	0..15
32 bits/sample, MSB-first	S[31]	S[SSPOS]	0..31
32 bits/sample, LSB-first	S[SSPOS]	S[31]	0..31

Frame bits that do not belong to either LEFT[MSB:SSPOS] or RIGHT[MSB:SSPOS] or a codec control field (Section 9.8, "Codec Control") are shifted out as zero. This zero extension ensures that TM1300 can be used in combination with D/A converters of higher precision than the actual number of transmitted bits in the current operating mode, e.g. 18-bit D/As operating with 16-bit memory data.

9.7.1 Serial Frame Limitations

Due to the implementation, there is a minimum serial frame length required that is operating mode dependent. This is shown in Table 9-6.

Table 9-6. Minimum serial frame length in bits

operating mode	minimum serial frame length
16 bits/sample, mono	13 bits
32 bits/sample, mono	13 bits
16 bits/sample, stereo	13 bits
32 bits/sample, stereo	36 bits

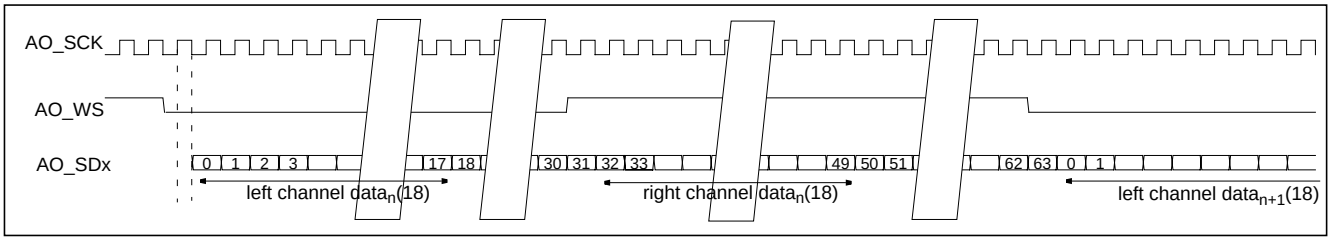


Figure 9-3. Serial frame (64 bits) of a 18-bit precision I^{2}S D/A converter.

9.7.2 I^{2}S Serial Framing Example

Refer to Figure 9-3 and Table 9-7 to see how the AO unit MMIO registers should be set to transmit 16 or 32 bits of stereo data via an I^{2}S serial standard to an 18-bit D/A converter with a 64-bit serial frame.

Table 9-7. Example setup for 64-bit I^{2}S framing

Field	Value	Explanation
POLARITY	0	Frame starts with negedge AO_WS.
LEFTPOS	0	LEFT[msb] will go to serial frame position 0.
RIGHTPOS	32	RIGHT[msb] will go to serial frame position 32.
DATAMODE	0	MSB first.
SSPOS	0	Stop with LEFT/RIGHT[0], send 0's after. (for 32 bits/sample mode, this field could be set to 14 to ensure zeroes in all unused bit positions)
CLOCK_EDGE	0	AO_SDx change on negedge AO_SCK
WSDIV	63	Serial frame length = 64.
WS_PULSE	0	emit 50% duty cycle AO_WS.

9.8 CODEC CONTROL

In addition to the left and right data fields that are generated based on autonomous DMA action, a serial frame generated by the AO unit can be set to contain 1 or 2 control fields up to 16 bits in length. Each control field can be independently enabled/disabled by the CC1_EN, CC2_EN bits in AO_CTL. The content shifted into the frame is taken from the CC1 and CC2 field in the AO_CC register. The CC1_POS and CC2_POS fields in the AO_CFC register determine the first bit position in the frame where the control field is emitted. The field is emitted observing the setting of DATAMODE, i.e. LSB or MSB first.

The CC_BUSY bit in AO_STATUS indicates if the AO unit is ready to receive another CC1, CC2 value pair. Writing a new value pair to AO_CC writes the value into a buffer register, and raises the CC_BUSY status. As soon as both CC1 and CC2 values have been copied to a shadow register in preparation for transmission, CC_BUSY is negated, indicating that the AO logic is ready to accept a new codec control pair. The old CC1/CC2 data keeps being transmitted - i.e. software is not required to provide new CC1 and CC2 data.

Table 9-8. AO MMIO codec control/status fields

Field Name	Description
CC1 (16)	The 16-bit value of CC1 is shifted into each emitted serial frame starting at bit position CC1_POS, as long as CC1_EN is asserted.
CC1_POS	Defines the bit position within a serial frame where the first data bit of CC1 is placed. RESET Default 0.
CC1_EN	0 ⇒ CC1 emission disabled (RESET default) 1 ⇒ CC1 emission enabled.
CC2(16)	The 16-bit value of CC2 is shifted into each emitted serial frame starting at bit position CC2_POS, as long as CC2_EN is asserted.
CC2_POS	Defines the bit position within a serial frame where the first data bit of CC2 is placed. Default 0.
CC2_EN	0 ⇒ CC2 emission disabled (RESET default) 1 ⇒ CC2 emission enabled.
CC_BUSY	0 ⇒ AO is ready to receive a CC1, CC2 pair (RESET default). 1 ⇒ AO is not ready to receive a CC1, CC2 pair. Try again in a few SCK clock intervals.

Software always needs to ensure that the CC_BUSY status is negated before writing a new CC1, CC2 pair. By polling CC_BUSY, the DSPCPU can emit a sequence of individual audio frames with distinct control field values reliably. This can, for example, be used during codec initialization. No provision is made for interrupt driven operation of such a sequence of control values; it is assumed that after initialization, the value of control fields determine slow, asynchronous changing parameters such as volume.

It is legal to program the control field positions within the frame such that CC1 and CC2 overlap each other and/or left/right data fields. If two fields are defined to start at the same bit position, the priority is left (highest), right, CC1 then CC2. The field with the highest priority will be emitted starting at the conflicting bit position. If a field *f2* is defined to start at a bit position *i* that falls within a field *f1* starting at a lower bit position, *f2* will be emitted starting from *i* and the rest of *f1* will be lost. Any bit positions not belonging to a data or control field will be emitted as '0'.

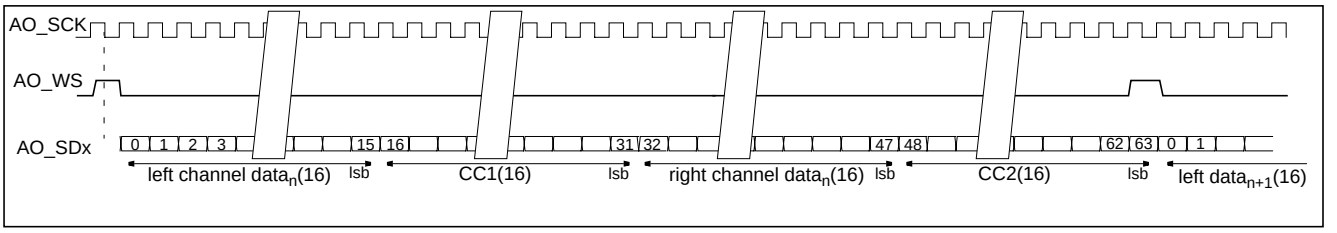


Figure 9-4. Example codec frame layout for a Crystal Semi, CS4218.

Figure 9-4 shows a 64-bit frame suitable for use with the CS4218 codec. It is obtained by setting POLARITY=1, LEFTPOS=0, RIGHTPOS=32, DATAMODE=0, SSPOS=0, CLOCK_EDGE=1, WS_PULSE=1, CC1_POS = 16, CC1_EN=1, CC2_POS=48, CC2_EN=1.

Note that frames are generated (externally or internally) even when TRANS_ENABLE is de-asserted. Writes to CC1 and CC2 should only be done after TRANS_ENABLE is asserted. The 'first' CC values will then go out on the next frame. For a summary of codec control fields see Table 9-8

9.9 MEMORY DATA FORMATS

The AO unit autonomously reads samples from memory in 16 or 32 bit-per-sample memory formats, as shown in Figure 9-5 for some example modes. Memory samples are retrieved and used as described in Table 9-9. Successive samples are always read from increasing memory address locations. The setting of the LITTLE_ENDIAN bit in the AO_CTL register determines the byte order of retrieved 16 or 32-bit samples. Refer to Appendix C, "Endian-ness," for details on byte ordering conventions.

AO hardware implements a double buffering scheme to ensure that there are always samples available to transmit, even if the DSPCPU is highly loaded and slow to respond to interrupts. The DSPCPU software assigns 2 equal size buffers by writing a base address and size to the MMIO control fields described in Figure 9-6. Refer to Section 9.10, "Audio Out Operation," for details on hardware/software synchronization.

If SIGN_CONVERT is set to one, the MSB of the memory data is inverted, which is equivalent to translating from offset binary representation to two's complement. This allows the use of an external two's complement 16-bit D/A converter to generate audio from 16-bit unsigned samples. This MSB inversion also applies to the '0' values transmitted to non-active output channels.

Note that the AO hardware does *not* support A-law or µ-law eight-bit data formats. If such formats are desired, the DSPCPU should be used to convert from A-law or µ-law data to 16-bit linear data.

Table 9-9. Operating modes and memory formats

NR_CHAN	MODE	destination of successive samples
00	mono	SD1.left
00	stereo	SD1.left, SD1.right
01	mono	SD1.left, SD2.left
01	stereo	SD1.left, SD1.right, SD2.left, SD2.right
10	mono	SD1.left, SD2.left, SD3.left
10	stereo	SD1.left, SD1.right, SD2.left, SD2.right, SD3.left, SD3.right
11	mono	SD1.left, SD2.left, SD3.left, SD4.left
11	stereo	SD1.left, SD1.right, SD2.left, SD2.right, SD3.left, SD3.right, SD4.left, SD4.right.

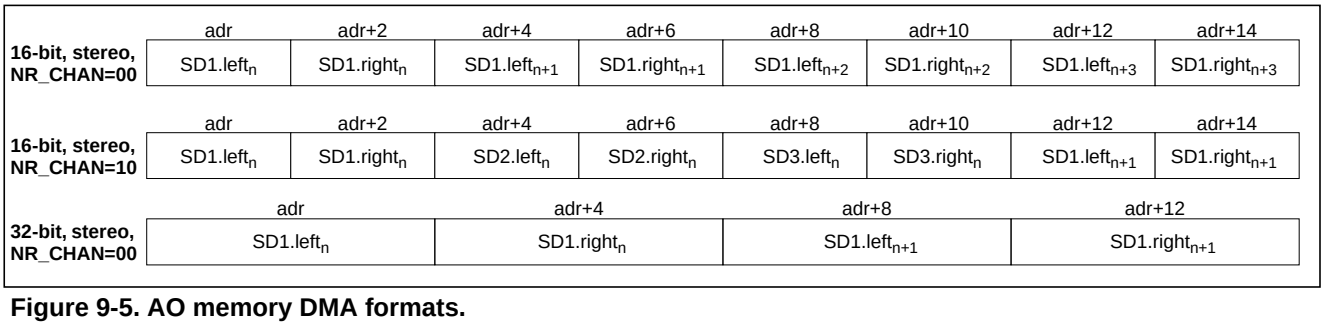


Figure 9-5. AO memory DMA formats.

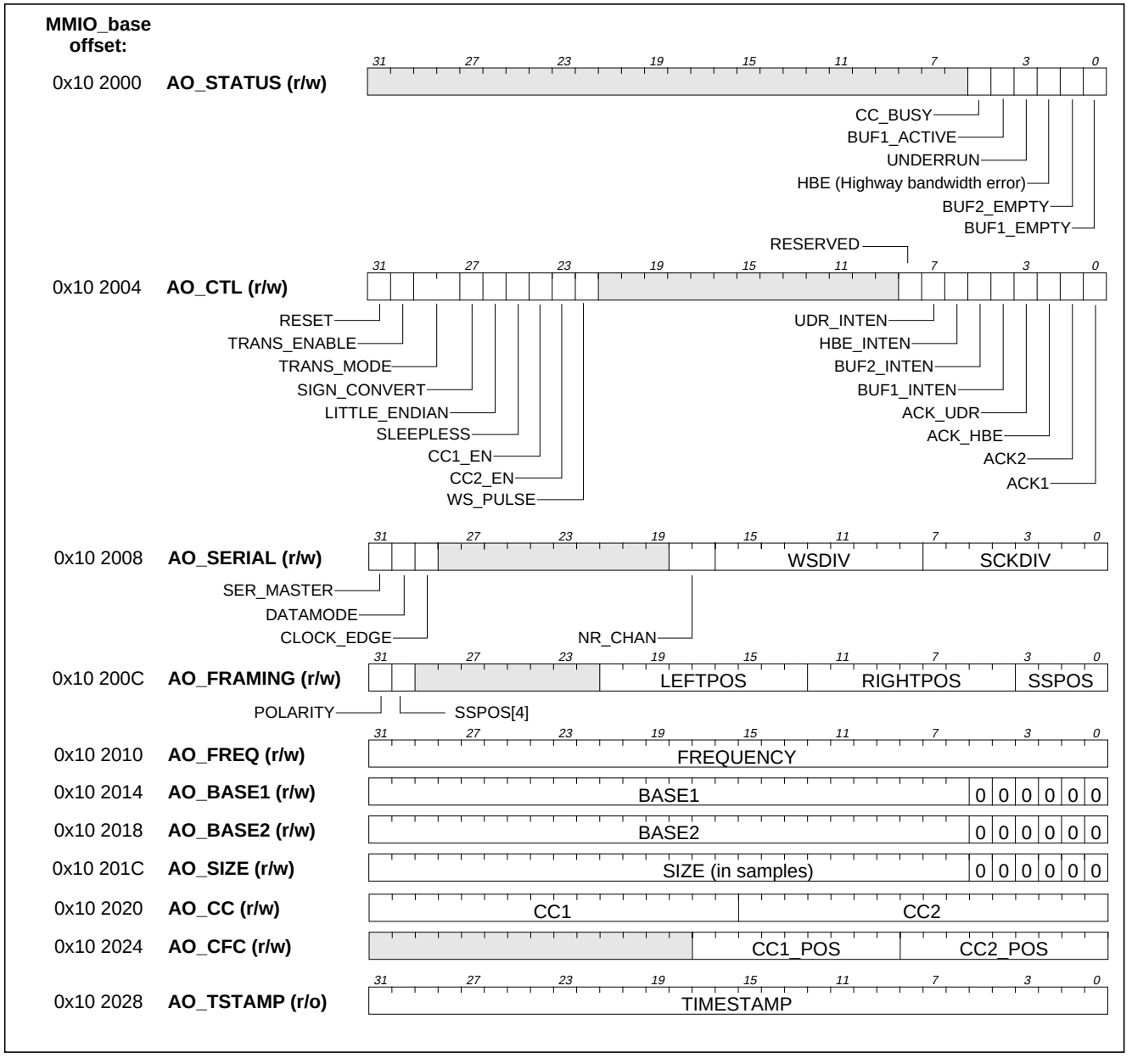


Figure 9-6. AO status/control field MMIO layout.

9.10 AUDIO OUT OPERATION

Figure 9-6, Table 9-10 and Table 9-11 describe the function of the control and status fields of the AO unit. To ensure compatibility with future devices, any undefined or reserved MMIO bits should be ignored when read, and written as zeroes

The AO unit is reset by a TM1300 hardware reset, or by writing 0x80000000 to the AO_CTL register. The AO unit is not affected by DSPCPU reset initiated through the BIU_CTL register. Either reset method sets all MMIO fields as indicated in the tables.

The timestamp counter is reset by TRI_RESET# or by DSPCPU reset initiated through BIU_CTL. It is not affected by AO_CTL reset. This ensures that the timestamp counter stays synchronous with the DSPCPU CCCOUNT register.

After an AO reset, 5 AO_SCK clock cycles are required to stabilize the internal circuitry before enabling Audio Out. This can be accomplished by programming the AO_FREQ and AO_SERIAL registers to start AO_SCK generation then waiting for the appropriate 5 AO_SCK cycle interval.

Programing of the AO_SERIAL MMIO register needs to follow the following sequence order:

- set AO_FREQ to ensure that a valid clock is generated (Only when AO is the master of the audio clock system)
- MMIO(AO_CTL) = 1 << 31; /* Software Reset */

- MMIO(AO_SERIAL) = 1 << 31; /* sets serial-master mode, starts AO_SCK */
- MMIO(AO_SERIAL) = (1 << 31) | (SCKDIV value); /* then set DIVIDER values */

Upon reset, transmission is disabled (TRANS_ENABLE = 0), and buffer 1 is the active buffer (BUF1_ACTIVE=1).

The DSPCPU initiates transmission by providing two full equal size buffers and putting their base address and size in the $BASE_n$ and SIZE registers. Once two valid buffers are assigned, transmission can be enabled by writing a '1' to TRANS_ENABLE. The AO hardware now proceeds to empty buffer 1 by transmission of output samples. Once buffer 1 empties, BUF1_EMPTY is asserted, and transmission continues without interruption from buffer 2. If BUF1_INTEN is enabled, a SOURCE 12 interrupt request is generated.

Note that buffers must be 64-byte aligned (the six LSBs of AO_BASE1, AO_BASE2 are zero). Buffer sizes must be a multiple of 64 samples (the 6 LSB's of AO_SIZE are zero).

Table 9-10. AO MMIO DMA control fields

Field Name	Description
LITTLE_ENDIAN	0 ⇒ big endian memory format (RESET default) 1 ⇒ little endian
BASE1	Base Address of buffer1. Must be a 64-byte aligned address in local SDRAM. RESET default 0.
BASE2	Base Address of buffer2. Must be a 64-byte aligned address in local SDRAM. RESET default 0.
SIZE	DMA buffer size, in samples. This number of mono samples or stereo sample pairs is read from a DMA buffer before switching to the other buffer. Buffer size in bytes is as follows: 16 bps, mono : 2 * SIZE 32 bps, mono : 4 * SIZE 16 bps, stereo : 4 * SIZE 32 bps, stereo : 8 * SIZE RESET default 0.
TRANS_MODE	00 ⇒ mono, 32 bits/sample. (RESET default). Left data and Right data sent to each active output are the same. 01 ⇒ stereo, 32 bits/sample 10 ⇒ mono, 16 bits/sample. Left data and Right data are the same. 11 ⇒ stereo, 16 bits/sample Refer to Table 9-9 for an explanation of how TRANS_MODE and NR_CHAN map to output behavior.
SIGN_CONVERT	0 ⇒ leave MSB unchanged (RESET default) 1 ⇒ invert MSB (not applied to codec control fields)

The DSPCPU is required to assign a new, full buffer to BASE1 and perform an ACK1 before buffer 2 empties. Transmission continues from buffer 2 until it is empty. At that time, BUF2_EMPTY is asserted and transmission continues from the new buffer 1, etc. An ACK performs two functions: it tells the AO unit that the corresponding BASE register now points to a buffer filled with samples, and it clears BUF_EMPTY. Upon receipt of an ACK, the AO hardware removes the BUF_EMPTY related interrupt request line assertion at the next DSPCPU clock edge. Refer to the interrupt controller documentation for details on interrupt handler programming. The AO interrupt (SOURCE 12) should always be operated in level sensitive mode

Table 9-11. AO DMA status fields (read only)

Field Name	Description
BUF1_ACTIVE	• If 1, buffer 1 will be used for the next sample to be transmitted. • If 0, buffer 2 will contain the next sample (1 after RESET).
BUF1_EMPTY	• If 1, buffer 1 is empty. • If BUF1_INTEN is also 1, an interrupt request (source 12) is asserted. • BUF1_EMPTY is cleared by writing a '1' to ACK1, at which point the AO hardware will assume that BASE1 and SIZE describe a new full buffer. • 0 after RESET.
BUF2_EMPTY	• If 1, buffer 2 is empty. • If BUF2_INTEN is also 1, an interrupt request (source 12) is asserted. • BUF2_EMPTY is cleared by writing a '1' to ACK2, at which point the AO hardware will assume that BASE2 and SIZE describe a new full buffer. • 0 after RESET.
HBE	• Highway Bandwidth Error. • 0 after RESET. • Indicates that no data was transmitted due to inability to read the local AO buffer from SDRAM in time. This indicates an insufficient allocation of TM1300 Highway bandwidth for the audio sampling rate/ mode.
UNDERRUN	• An UNDERRUN error has occurred, i.e. the CPU failed to provide a full buffer in time, and no samples were transmitted, although requested by the D/A converter. • If UDR_INTEN is also 1, an interrupt request (source 12) is pending. The UNDERRUN flag can ONLY be cleared by writing a '1' to ACK_UDR. • 0 after RESET.

9.11 INTERRUPTS

The AO unit has a private interrupt request line to the DSPCPU vectored interrupt controller. It uses SRC# 12 (same as TM1000/TM1100 AO).

An interrupt is asserted as long as one or more of the UNDERRUN, HBE, BUF1_EMPTY or BUF2_EMPTY condition flags and the corresponding INTEN bit are asserted. Interrupts are sticky, i.e. an interrupt remains asserted until the software explicitly clears the condition flag by an ACK_x action.

Table 9-12. AO MMIO Control Fields

Field Name	Description
RESET	Resets the audio-out logic. See Section 9.10, “Audio Out Operation” for a description of the recommended procedure.
TRANS_ENABLE	Transmission Enable flag. 0 ⇒ (RESET default) AO inactive. 1 ⇒ AO transmits samples and acts as DMA master to read samples from local SDRAM. Do NOT change the POLARITY bit while transmission is enabled.
SLEEPLESS	0 ⇒ (power up default) AO goes into power-down mode if TM1300 goes to global powerdown mode. 1 ⇒ AO continues operation when TM1300 goes to global powerdown mode. Samples are read from memory as needed, and AO interrupts, when enabled, will wake up the DSPCPU.
BUF1_INTEN	Buffer 1 Empty Interrupt Enable. 0 ⇒ (default) no interrupt 1 ⇒ interrupt (SOURCE 12) if buffer 1 empty
BUF2_INTEN	Buffer 2 Empty Interrupt Enable. 0 ⇒ (default) no interrupt 1 ⇒ interrupt (SOURCE 12) if buffer 2 empty
HBE_INTEN	HBE Interrupt Enable. 0 ⇒ (default) no interrupt 1 ⇒ interrupt (SOURCE 12) if a highway bandwidth error occurs.
UDR_INTEN	UNDERRUN Interrupt Enable. 0 ⇒ (default) no interrupt 1 ⇒ interrupt (SOURCE 12) if an UNDERRUN error occurs
ACK1	• Write a 1 to clear the BUF1_EMPTY flag and remove any pending BUF1_EMPTY interrupt request. • ACK1 always reads 0.
ACK2	• Write a 1 to clear the BUF2_EMPTYflag and remove any pending BUF2_EMPTY interrupt request. • ACK2 always reads 0.
ACK_HBE	• Write a 1 to clear the HBE flag and • remove any pending HBE interrupt request. • ACK_HBE always reads as 0.
ACK_UDR	• Write a 1 to clear the UNDERRUN flag and remove any pending UNDERRUN interrupt request. • ACK_UDR always reads 0.

9.12 TIMESTAMP

The AO_TSTAMP MMIO register provides a 32-bit timestamp value that contains the CCCOUNT time value at which the last sample of the last DMA buffer transmitted was sent across the SD output pin. This value is available for software inspection (read-only) in the interrupt handler for BUFx_EMPTY.

The implementation involves an internal DSPCPU clock cycle counter that is reset to have the same value as the DSPCPU CCCOUNT register. It is guaranteed to be in sync with the 32 LSB of CCCOUNT provided that PCSW.CS=1.

9.13 POWERDOWN AND SLEEPLESS

The AO unit enters powerdown state whenever TM1300 is put in global powerdown mode, except if the SLEEPLESS bit in AO_CTL is set. In the latter case, the block continues DMA operation and will wake up the DSPCPU whenever an interrupt is generated. The internal timestamp counter never powers down to ensure that it remains synchronous with CCCOUNT.

The AO unit can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. Refer to Chapter 21, “Power Management.”

If the block enters powerdown state, AO_SCK, AO_SDx, and AO_WS hold their value stable. AO_OSCLK continues to provide a D/A converter clock. The signals resume their original transitions at the point where they were interrupted once the system wakes up. The external D/A converter subsystem is most likely confused by this behavior, hence it is recommended AO unit to be stopped (by negating TRANS_ENABLE) before block level powerdown is started, or that SLEEPLESS mode is used when global powerdown is activated.

9.14 HIGHWAY LATENCY AND HBE

The AO unit uses an internal 64-byte buffer as well as an output holding register that contains a single mono sample or single stereo sample pair. Under normal operation, the internal buffer is refreshed from SDRAM fast enough to avoid any missing samples, while data is being emitted from the holding register. If the highway arbiter is set up with an insufficient latency guarantee, the situation can arise that the 64-byte buffer is not refilled and the holding register is exhausted by the time a new output sample is due. In that case the HBE error is raised. The last sample for each channel will be repeated until the buffer is refreshed. The HBE condition is sticky, and can only be cleared by an explicit ACK_HBE. This condition indicates an incorrect setting of the highway bandwidth arbiter.

Given a sample rate f_s, and an associated sample interval T (in ns), the arbiter should be set to have a latency of at most T-20 ns for all modes. The latency for 4,6 and 8 channel modes can be computed as if the system is operating in stereo mode with a 2x, 3x respectively 4x sample rate.

Table 9-13 shows the required arbiter latency settings for a number of common operating modes. The right most column in illustrates the nature of the resulting 64-byte highway requests. Is not necessary to compute arbiter settings, but they may be used to compute bus availability in a given interval.

Refer to Chapter 20, “Arbiter,” for information on arbiter programming.

Table 9-13. AO highway arbiter latency requirement examples

TransMode	f_s (kHz)	T (ns)	max. arbiter latency (ns)	access pattern
stereo 16 bits/sample	44.1	22,676	22,656	1 request every 362,812 ns
stereo 16 bits/sample	48.0	20,833	20,813	1 request every 333,333 ns
stereo 16 bits/sample	96.0	10,417	10,397	1 request every 166,667 ns
6 channel 16 bits/sample	48.0	20,833	6,924	1 request every 111,111 ns
stereo 32 bits/sample	48.0	20,833	20,813	1 request every 166,667 ns
6 channel 32 bits/sample	48.0	20,833	6,924	1 request every 55,556 ns

9.15 ERROR BEHAVIOR

In normal operation, the DSPCPU and AO hardware continuously exchange buffers without ever failing to transmit a sample. If the DSPCPU fails to provide a new buffer in time, the UNDERRUN error flag is raised, and the last valid sample or sample pair is repeated until a new buffer of data is assigned by an ACK1 or ACK2. The UNDERRUN flag is *not affected* by ACK1 or ACK2; it can only be cleared by an explicit ACK_UDR.

If an HBE error occurs, the last valid sample or sample pair is repeated until the AO hardware retrieves a new sample buffer across the highway.

SPDIF Out Chapter 10

by Gert Slavenburg, Santanu Dutta

10.1 SPDIF OUT OVERVIEW

The TM1300 SPDIF Output unit (SPDO) allows generation of a 1-bit high-speed serial data stream. The primary application is to make SPDIF (Sony/Philips Digital Interface) data available for use by external audio equipment.

The SPDO unit has the following features:

- fully compliant with IEC958, for both consumer and professional applications
- supports 2-channel linear PCM audio, with 16 or 24 bits per sample
- supports one or more Dolby Digital(r) 6-channel data streams embedded per Project 1937
- supports one or more MPEG-1 or MPEG-2 audio streams embedded per Project 1937
- allows arbitrary, programmable, sample rates from 1 Hz to 300 kHz
- can output data with a sample rate independent of and asynchronous to the sample rate of the Audio Out (AO) unit
- hardware performs autonomous DMA of memory resident IEC958 sub-frames
- hardware performs parity generation and bi-phase mark encoding
- allows software to have full control over all data content, including user and channel data

Alternate use of the SPDO unit to generate a general-purpose high-speed data stream is possible. Potential applications include use as a high-speed UART or high speed serial data channel. In this case features are:

- up to 40 Mbit/sec data rate
- full software control over each bit cell transmitted
- LSB first or MSB first data format

10.2 EXTERNAL INTERFACE

The external interface consists of only one pin, SPDO, which is described in Table 10-1.

Table 10-1. SPDO external signals

Signal	Type	Description
SPDO	I/O	SPDIF output. Self clocking interface carrying either 2-channel PCM data with samples up to 24 bits, or encoded Dolby AC-3(r) or MPEG audio data for decoding by an external audio component.

An external circuit (see Figure 10-1) is required to provide an electrically isolated output and convert the 3.3 V output pin to a drive level of 0.5 V peak-peak into a 75-ohm load, as required for consumer applications of IEC-958.

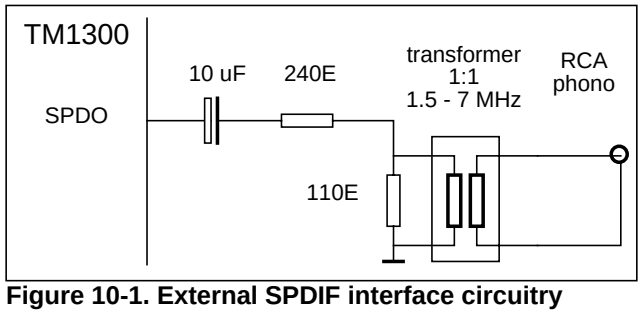


Figure 10-1. External SPDIF interface circuitry

10.3 SUMMARY OF OPERATION

In both SPDIF and transparent DMA modes, SPDO sends alternating memory data buffers out across the output pin. Software initially gives SPDO two memory data buffers and enables the SPDO unit. When the first buffer is sent, SPDO requests a new buffer from software while switching over to use the other buffer, etc. Transmission continues uninterrupted until the unit is disabled.

10.3.1 SPDIF Mode

SPDIF driver software assembles SPDIF data in each memory data buffer. Each memory data buffer consists of groups of 32-bit words in memory. Each word describes the data to be transmitted for a single IEC-958 sub-frame, including what type of preamble is to be included. Each sub-frame is transmitted in 64-clock cycle intervals of the SPDO clock, a programmable clock generated by the SPDO Direct Digital Synthesizer (DDS).

10.3.2 Transparent DMA Mode

In transparent DMA mode, software prepares each data bit exactly as it is to be transmitted, in a series of 32-bit words in each memory data buffer. Each 32-bit word is transmitted LSB first or MSB first in 32-clock cycle intervals of the SPDO clock, a programmable clock generated by the SPDO Direct Digital Synthesizer.

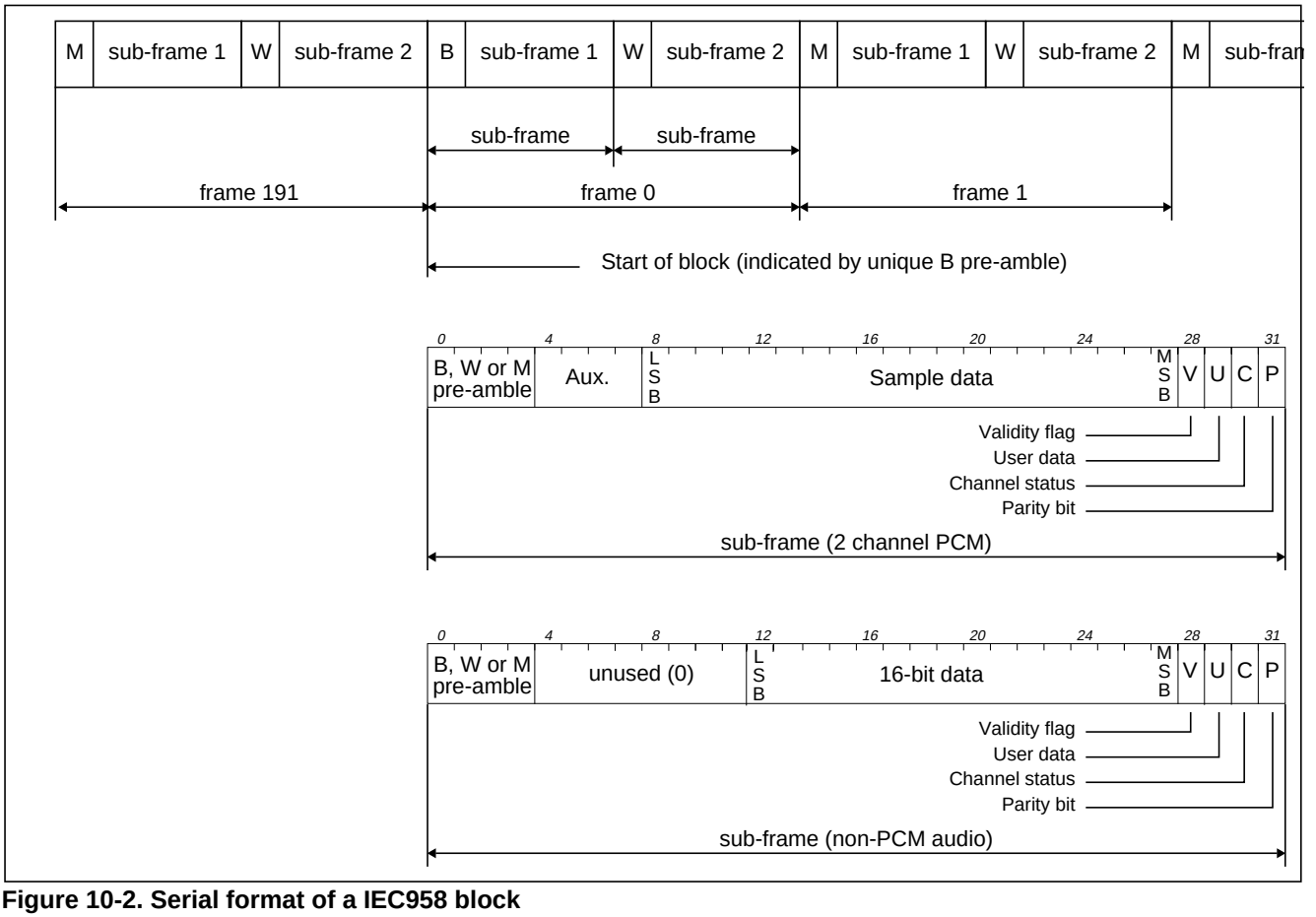


Figure 10-2. Serial format of a IEC958 block

10.4 IEC-958 SERIAL FORMAT

Figure 10-2 shows the serial format layout of a IEC-958 block. A block starts with a special 'B' pre-amble, and consists of 192 frames. The sample-rate of all embedded audio data is equal to the frame rate. Each frame consists of 2 sub-frames. Sub-frame 1 always starts with a 'M' pre-amble, except for sub-frame 1 in frame 0, which starts with a 'B'. Sub-frame 2 always starts with a 'W' pre-amble.

When IEC-958 data carries 2-channel PCM data, one audio sample is transmitted in each sub-frame, 'left' in sub-frame 1 and 'right' in sub-frame 2. Each sample can be 16 or 24 bits in length, where the MSB is always aligned with bit slot 28 of the sub-frame. In case of more than 20 bits/sample, the Aux field is used for the 4 LSBs.

When IEC-958 data carries non-PCM audio, such as 1 or more streams of Dolby AC-3 encoded data and/or MPEG audio, each sub-frame carries 16-bit data. The data of successive frames adds up to a payload data-stream which carries its own burst-data.This is described in [2].

Programmers should refer to the IEC-958 documents [1] and Project 1937 document [2] for a precise description of the required values in each field for different types of consumer equipment. A complete discussion of this issue is outside the scope of this document.

The SPDO block hardware only concerns itself with generating B, W and M preambles as well as generating the P (parity) bit. All other bits in the sub-frame are completely determined by software and copied verbatim from memory to output, subject only to bit-cell coding.

The programmer must construct valid IEC-958 blocks by constructing the right sequence of 32-bit words as described in Section 10.7, "IEC-958 Memory Data Format."

10.5 IEC-958 BIT CELL AND PRE-AMBLE

Each data bit in IEC-958 is transmitted using bi-phase mark encoding. In bi-phase mark encoding, each data bit is transmitted as a cell consisting of two consecutive binary states. The first state of a cell is always inverted from the second state of the previous cell. The second state of a cell is identical to the first state if the data bit value is a "0", and inverted if the data bit value is a "1".

Pre-ambles are coded as bi-phase mark violations, where the first state of a cell is not the inverse of the last state of the previous cell.

The duration of each state in a cell is called a UI (Unit Interval), so that each cell is 2 UI's long. In SPDO, the length of a UI is 1 SPDO clock cycle as determined by the settings of the DDS (see Section 10.8, "Sample Rate Programming").

Figure 10-3 illustrates the transmission format of 8-bit data value "10011000", as well as the transmission format of the 3 pre-ambles. Note that each pre-amble al-

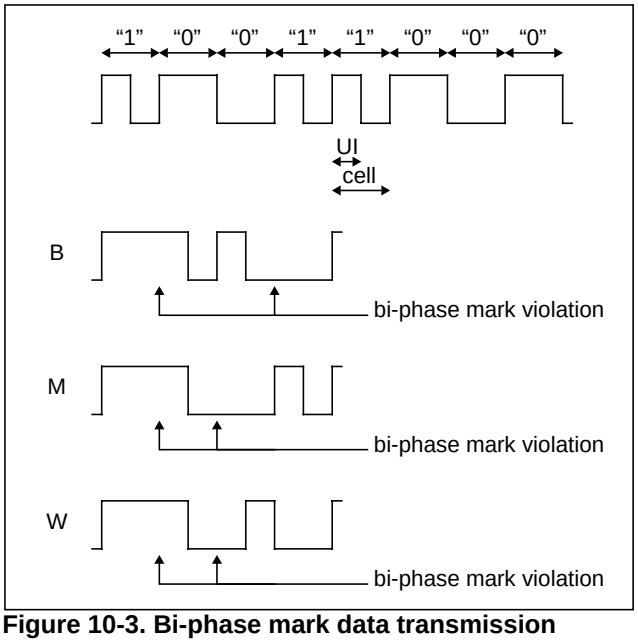


Figure 10-3. Bi-phase mark data transmission

ways starts with a rising edge. This is made possible thanks to the presence of the parity bit, which always guarantees an even number of '1' bits in each sub-frame.

10.6 IEC-958 PARITY

The parity bit, or P bit in Figure 10-2, is computed by the SPDO hardware. The P bit value should be set such that bit cells 4 to 31 inclusive contain an even number of '1's (and hence even number of '0's). The P bit is bi-phase mark encoded using the same method as for all other bits.

10.7 IEC-958 MEMORY DATA FORMAT

The DSPCPU software must prepare a memory data structure that instructs the SPDO hardware to generate correct IEC-958 blocks. This data structure consists of 32-bit words with the following content:

Table 10-2. SPDIF sub-frame descriptor word

bits	definition
31 (MSB)	this bit must be a '0' for future compatibility
30..4	Data value for bits 4..30 of the subframe, exactly as they are to be transmitted. Hardware will perform the bi-phase mark encoding and parity generation.
3..0 (LSB)	0000 - generate a B preamble 0001 - generate a M preamble 0010 - generate a W preamble 0011 .. 1111 reserved for future

The data structure for a block consists of 384 of these 32-bit descriptor words, one for each subframe of the block, with the correct B, M, W values. All data content, including the U, C and V flag are fully under control of the software that builds each block.

A DMA buffer handed to the hardware is required to be a multiple of 64 bytes in length. It can contain 1 or more complete blocks, or a block may straddle DMA buffer boundaries. The 64-byte length will result in DMA buffers that contain a multiple of 16 sub-frames.

Note that the descriptor structure is a 32-bit word memory data structure, and is hence subject to processor endian-ness. To allow software to be efficient in both little-endian and big-endian operation, the SPDO block SPDO_CTL register has an endian-ness bit 'LITTLE_ENDIAN'. The SPDO block performs byte swapping when loading the SPDIF descriptors as follows.

- If LITTLE_ENDIAN = 1, 32-bit words at address 'a' will be assembled from bytes (a+3,a+2,a+1,a), with the byte at 'a+3' containing the MSB's and the byte at 'a' the LSB's.
- If LITTLE_ENDIAN = 0, 32-bit words at address 'a' will be assembled from bytes (a,a+1,a+2,a+3), with the byte at 'a' containing the MSB's and the byte at 'a+3' the LSB's.

10.8 SAMPLE RATE PROGRAMMING

In he SPDO unit, the frame rate always equals f_s, the sample rate of embedded audio. This relation holds for PCM as well as for Dolby AC-3 and MPEG encoded audio. Each frame consists of 128 Unit Intervals (UI's). The length of a UI is determined by the frequency setting of the DDS (Direct Digital Synthesizer) in the SPDO block.

$$f_s = \frac{(f_{DDS})}{128} \qquad \text{Eq. 1}$$

The DDS can be programmed to emit frequencies from approx. 1 Hz to 80 MHz in steps of approx. 0.3 Hz, with a jitter of approx. 750 psec (at DSPCPU frequency of 143 MHz, see equations below).

Programming is accomplished through the FREQUENCY MMIO register: the relation between FREQUENCY register value, DSPCPU clock value and synthesized frequency is:

$$FREQUENCY = 2^{31} + \frac{f_{DDS} \cdot 2^{32}}{9 \cdot f_{DSPCPU}} \qquad \text{Eq. 2}$$

Putting equation 1 and 2 above together yields the formula for setting FREQUENCY to accomplish a given sample rate:

$$FREQUENCY = 2^{31} + \frac{f_s \cdot 2^{39}}{9 \cdot f_{DSPCPU}}$$

The DDS synthesizer maximum jitter can be computed as follows:

$$jitter = \frac{1}{9 \cdot f_{DSPCPU}}$$

Table 10-3 shows settings for common sample rate and DSPCPU clock combinations:

Table 10-3. SPDIF sample rate setting

f_s (kHz)	f_{DSPCPU} (MHz)	FREQUENCY (hexadecimal)	UI (nSec)	jitter (nSec)
32.000	143	0x80D0,9316	244.14	0.777
32.000	166	0x80B3,ACF8	244.14	0.669
32.000	180	0x80A5,B36E	244.14	0.617
44.100	143	0x811F,711B	177.15	0.777
44.100	166	0x80F7,9D93	177.15	0.669
44.100	180	0x80E4,5B47	177.15	0.617
48.000	143	0x8138,DCA1	162.76	0.777
48.000	166	0x810D,8375	162.76	0.669
48.000	180	0x80F8,8D25	162.76	0.617

The programmer is free to change FREQUENCY, and hence the system sample rate to perform long-term tracking of any absolute timing source and/or control software buffer fullness. Changes to the FREQUENCY register pull-in or delay the next clock edge and have no instantaneous effect on clock level, i.e. the rate of phase progression is changed, not the phase.

10.9 TRANSPARENT MODE

When SPDO is set to operate in transparent mode, it takes all 32 bits of the memory data and shifts them out verbatim, without bi-phase mark encoding, parity generation, or preamble.

Two transparent modes are provided, as determined by TRANS_MODE in SPDO_CTL: LSB first and MSB first.

One bit of memory data is transmitted for each DDS clock, such that the FREQUENCY register value for a desired bitrate is given by the following equation:

$$FREQUENCY = 2^{31} + \frac{2^{32} \cdot bitrate}{9 \cdot f_{DSPCPU}} \qquad \text{Eq. 2}$$

The 32-bit memory word is constructed according to the same rules for LITTLE_ENDIAN as in Section 10.7, “IEC-958 Memory Data Format.”

10.10 DMA OPERATION

Before enabling the SPDO block, software must assign two buffers with data to SPDO_BASE1, SPDO_BASE2, and SPDO_SIZE (buffer size in bytes). Each memory buffer size must be a multiple of 64 bytes regardless of the operating mode.

The SPDO block is enabled by writing a ‘1’ to SPDO_CTL.TRANS_ENABLE. Once enabled, the first DMA buffer is sent out at the programmed sample rate. Once the first buffer is empty, BUF1_ACTIVE is negated, a timestamp is generated (see Section 10.13, “Timestamps”) and the BUF1_EMPTY flag in SPDO_STATUS is asserted. If BUF1_INTEN in SPDO_CTL is also asserted, an interrupt to the DSPCPU is generated. The SPDO block continues emitting the data in DMA buffer 2. In normal operation, the DSPCPU assigns a new buffer 1 full of data to SPDO and signals this by writing a ‘1’ to ACK_BUF1. The SPDO block immediately negates the BUF1_EMPTY condition and the related interrupt request. Once buffer 2 is empty, similar signaling occurs and the hardware switches back to using buffer 1.

10.11 DMA ERROR CONDITIONS

Two types of error can occur during DMA operation.

If the software fails to provide a new buffer of data in time, and both DMA buffers empty out, the SPDO hardware raises the UNDERRUN flag in SPDO_STATUS. Transmission switches over to the use of the next buffer, but the data transmitted is incorrect. If UDR_INTEN is asserted, an interrupt will be generated. The UNDERRUN flag is sticky, i.e. it will remain asserted until the software clears it by writing a ‘1’ to ACK_UDR.

A lower level error can also occur when the limited size internal buffer empties out before it can be refilled across the highway. This situation can arise only if insufficient bandwidth has been requested from the highway. In this case, the HBE error flag is raised. Refer to Section 10.17, “HBE and Highway Latency” for a description of how to set the arbiter latency correctly.

10.12 INTERRUPTS

The SPDO block uses interrupt SRC NUM 25, with interrupt vector MMIO offset 0x1008E4.

It is highly recommended that the interrupt be operated in level-sensitive mode only.

The SPDO block generates an interrupt if one of the following status bit flags, and its corresponding INTEN_xxx flag are set: BUF1_EMPTY, BUF2_EMPTY, HBE, UNDERRUN.

All these status flags are sticky, i.e. they are asserted by hardware when a certain condition occurs, and remain set until the interrupt handler explicitly clears them by writing a ‘1’ to the corresponding ACK bit in SPDO_CTL. The SPDO hardware takes the flag away in the clock cycle after the ACK is received. This allows immediate return from interrupt once performing an ACK.

10.13 TIMESTAMPS

Any outgoing DMA buffer is assigned a 32-bit ‘time of departure’ timestamp. The counter used to generate timestamps uses the DSPCPU clock and the same reset time as the DSPCPU CCCOUNT register, resulting in a value that corresponds to the 32 LSB’s of CCCOUNT - provided that PCSW.CS=1, i.e. the real CCCOUNT counter increments on every clock cycle.

The timestamp can be read in the DMA interrupt handler as MMIO register SPDO_TSTAMP. Its contents corresponds to the (synchronized) clock edge at which the last bit in the DMA buffer was sent across the output signal pin.

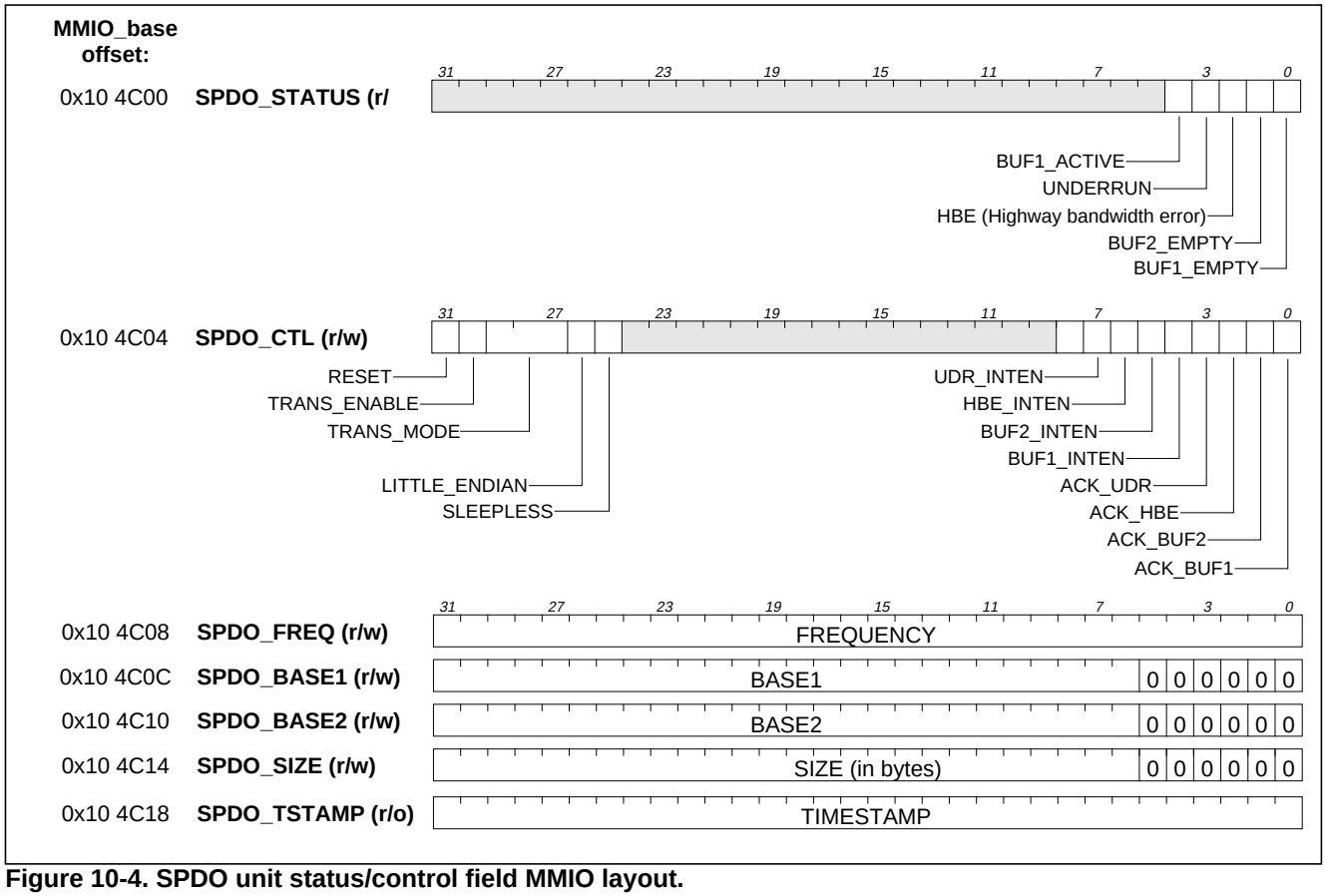


Figure 10-4. SPDO unit status/control field MMIO layout.

10.14 MMIO REGISTER DESCRIPTION

Table 10-4. SPDO_STATUS MMIO register

field	type	description
BUF1_EMPTY	r/o	Sticky flag - set if DMA buffer 1 emptied by the SPDO hardware. Can only be cleared by software write to ACK_BUF1.
BUF2_EMPTY	r/o	Sticky flag - set if DMA buffer 2 emptied by the SPDO hardware. Can only be cleared by software write to ACK_BUF2.
HBE	r/o	Highway Bandwidth Error. Sticky flag - set if internal SPDO buffers emptied before new data brought from memory. Refer to Section 10.17, “HBE and Highway Latency.” Can be cleared only by a software write to ACK_HBE.
UNDERRUN	r/o	Sticky flag - set if both DMA buffers were emptied before a new full buffer was assigned by the DSPCPU. The hardware has performed a normal buffer switch over and is emitting old data. Can only be cleared by software write to ACK_UDR.
BUF1_ACTIVE	r/o	Flag - set if the hardware is currently emitting DMA buffer 1 data; negated when emitting DMA buffer 2 data.

Table 10-5. SPDO_CTL MMIO register

field	type	description
ACK_BUF1	w/o	Always reads as ‘0’. Write a ‘1’ here to clear BUF1_EMPTY. This informs SPDO that DMA buffer 1 is now full. Writing a ‘0’ has no effect.
ACK_BUF2	w/o	Always reads as ‘0’. Write a ‘1’ here to clear BUF2_EMPTY. This informs SPDO that DMA buffer 2 is now full. Writing a ‘0’ has no effect.
ACH_HBE	w/o	Always reads as ‘0’. Writing a ‘1’ here clears HBE.
ACK_UDR	w/o	Always reads as ‘0’. Writing a ‘1’ here clears UNDERRUN.
BUF1_INTEN	r/w	If BUF1_EMPTY asserted and this bit asserted, the SRC 25 interrupt line is asserted.
BUF2_INTEN	r/w	If BUF2_EMPTY asserted and this bit asserted, the SRC 25 interrupt line is asserted.
HBE_INTEN	r/w	If HBE asserted and this bit asserted, the SRC 25 interrupt line is asserted.
UDR_INTEN	r/w	If UNDERRUN asserted and this bit asserted, the SRC 25 interrupt line is asserted.

Table 10-5. SPDO_CTL MMIO register

field	type	description
SLEEPLESS	r/w	If ‘1’, the SPDO block does not power down when TM1300 goes into global power-down mode. If ‘0’, the block does power down.
LITTLE_ENDIAN	r/w	If asserted, the 32-bit data SPDIF descriptor word or transparent mode data word is assembled using little endian byte ordering, otherwise big-endian.
TRANS_MODE	r/w	• 000 - IEC-958 mode. Hardware performs bi-phase mark encoding, preamble generation, and parity generation, and transmits one IEC-958 subframe for each data descriptor word. • 010 transparent mode, LSB first. The 32-bit data descriptor words are transmitted as is, LSB first. • 011 transparent mode, MSB first. The 32-bit data descriptor words are transmitted as is, MSB first. • Any other code reserved for future extensions. The transmission mode should only be changed while transmission is disabled.
TRANS_ENABLE	r/w	Writing a ‘1’ to this bit enables transmission per the selected mode. Writing a ‘0’ here stops any ongoing transmission after completing any actions related to the current data descriptor word.
RESET	w/o	Writing a ‘1’ to this bit resets the SPDO unit and should be used with extreme caution. Ongoing transmission will be interrupted, receivers may be left in a strange state.

To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as ’0’s.

The SPDO_FREQ register determines the frequency of operation of the DDS, and hence the sample rate of outgoing audio. Refer to Section 10.8, “Sample Rate Programming.” and Section 10.9, “Transparent Mode.”

SPDO_BASE1 contains the memory address of DMA buffer 1. SPDO_BASE2 contains the memory address of DMA buffer 2. SPDO_SIZE determines the size, in bytes, of both DMA buffers. Assignment to SPDO_BASE1, SPDO_BASE2 and SPDO_SIZE have no effect on the state of the SPDO_STATUS flags; the ACK_BUF1 and ACK_BUF2 bits signal the assignment of valid data to the DMA buffers. Any change to the BASE register should only be done to an inactive buffer and should precede the ACK to that buffer.

SPDO_TSTAMP is a read-only register containing the cycle count at which the last bit from the last emptied buffer was transmitted across the output pin. Refer to Section 10.13, “Timestamps.”

10.15 RESET

The SPDO block is reset by global TM1300 reset pin TRI_RESET# or by writing a ‘1’ to the RESET bit in SPDO_CTL. The SPDO block is not affected by DSPCPU reset initiated though the PCI block BIU_CTL register. Either reset method sets the SPDO block in the following state:

- SPDO_BASE1, SPDO_BASE2, SPDO_SIZE = 0
- SPDO_STATUS: all defined fields set to ’0’, except BUF1_ACTIVE = 1
- SPDO_CTL all defined fields set to value 0

The SPDO block timestamp counter is reset by TRI_RESET# or by DSPCPU reset initiated through BIU_CTL, so as to ensure that it stays synchronous to the CCCOUNT DSPCPU register.

10.16 POWER DOWN AND SLEEPLESS

The SPDO block enters powerdown state whenever TM1300 is put in global powerdown mode, except if the SLEEPLESS bit in SPDO_CTL is set. In the latter case, the block continues DMA operation and will wake up the DSPCPU whenever an interrupt is generated.

SPDO can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. For a description of powerdown, see Chapter 21, “Power Management.”

The SPDO block should not be active when applying global powerdown (TRANS_ENABLE = 0), or if active, SLEEPLESS should be asserted. SPDO should not be active if powered down separately.

If the block enters power-down state while transmission is enabled, its operation continues from the interrupted clock cycle, but the output signal generated by the block has undergone a pause that is unacceptable to external equipment.

10.17 HBE AND HIGHWAY LATENCY

The SPDO unit uses one internal 64-byte buffer and two 32-bit holding registers. Under normal operation, the internal buffer is refilled from SDRAM fast enough to avoid missing any data, while data is being sent from the two 32-bit registers. If the highway arbiter is set up with an insufficient latency guarantee, the situation can arise in which the 64-byte buffer is not refilled in time. In that case the HBE error is raised, and some data has been irrevocably lost. The HBE condition is sticky, and can only be cleared by an explicit ACK_HBE.

The highway arbiter needs to be programmed such that the SPDO unit’s latency requirement can always be met. Refer to Chapter 20, “Arbiter” for details. The required latency can be computed as indicated below.

Given an output data rate f_S in samples/sec, 2x 32 bits are required each sample interval. The arbiter should be set to have a latency so that the buffer is refilled before a sample interval expires. See Table 10-6 for example practical settings.

Table 10-6. SPDO block highway latency requirements

f_s (kHz)	Max. latency (nSec)
32.000	31250
44.100	22675
48.000	20833

10.18 LITERATURE REFERENCES

[1] IEC-958 Digital Audio Interface, Part 1: General; Part 2: Professional applications; Part 3: Consumer applications.

[2] ‘Interface for non-PCM encoded Audio bitstreams applying IEC958’, Philips Consumer Electronics, June 6 1997. IEC 100c/WG11(project 1937)

PCI Interface Chapter 11

by Gert Slavenburg, Ken-Sue Tan, Babu Kandimalla

11.1 NEW IN TM1300

TM1300 DMA read transactions use the more efficient ‘memory read multiple’ PCI transactions, unless explicitly disabled. Section 11.7.5.

TM1300 contains an on-board PCI_CLK generator for low-cost configurations. It can be enabled/disabled at boot time. See Section 13.2.

TM1300 has a sideband control signal that allows glueless connection of simple slave peripherals directly to the PCI bus wires. This can be used to connect Flash, ROM, SRAM, UARTs, etc. with 8-bit data and demultiplexed addresses. Refer to Chapter 22, “PCI-XIO External I/O Bus.”

11.2 PCI OVERVIEW

TM1300 includes a PCI interface for easy integration into personal computer applications—where the PCI-bus is the standard for high-speed peripherals. In embedded applications, with TM1300 serving as the main CPU, the PCI bus can interface to peripheral devices that implement functions not provided by the on-chip peripherals. See Figure 11-1.

The main function of the PCI interface is to connect the TM1300 on-chip highway and PCI buses. A bus cycle on the internal highway that targets an address mapped into PCI space will cause the PCI interface to create a PCI bus cycle. Similarly, a bus cycle on PCI that targets an address mapped into TM1300 memory space will cause the PCI interface to create a highway bus cycle targeted at SDRAM. For some operations, the PCI interface is explicitly programmed by the DSPCPU.

From TM1300, only the DSPCPU and the image coprocessor (ICP) unit can cause the PCI interface to create PCI bus cycles; the other on-chip peripherals cannot see external hardware through the PCI interface. From PCI, SDRAM and most of the registers in MMIO space can be accessed by external PCI initiators.

The PCI interface implements DMA (also called block or burst) and non-DMA transfers. DMA transfers are interruptible on 64-byte boundaries. The PCI interface can service outbound (TM1300 → PCI) and inbound (PCI → TM1300) data flows simultaneously.

Table 11-1 lists some of the features of the PCI interface.

Table 11-1. PCI interface characteristics

Characteristic	Comments
PCI Compliance	PCI Local Bus Specification Rev. 2.1
PCI Speed	Up to 33 MHz
Data bus width	32-bit only
Address space	32 bits (4 GB)
Voltage levels	Drive & receive at either 3.3 V or 5V
Burst mode	Yes, w/ double buffering so maximum transfer rate (132 MB/sec) is sustainable
Posted write	Yes, can be disabled
PCI ‘special cycle’	Not recognized
PCI ‘memory write & invalidate’	Supported for TM1300 as initiator
PCI ‘interrupt acknowledge’	Not generated
PCI ‘dual-address cycle’	Not generated

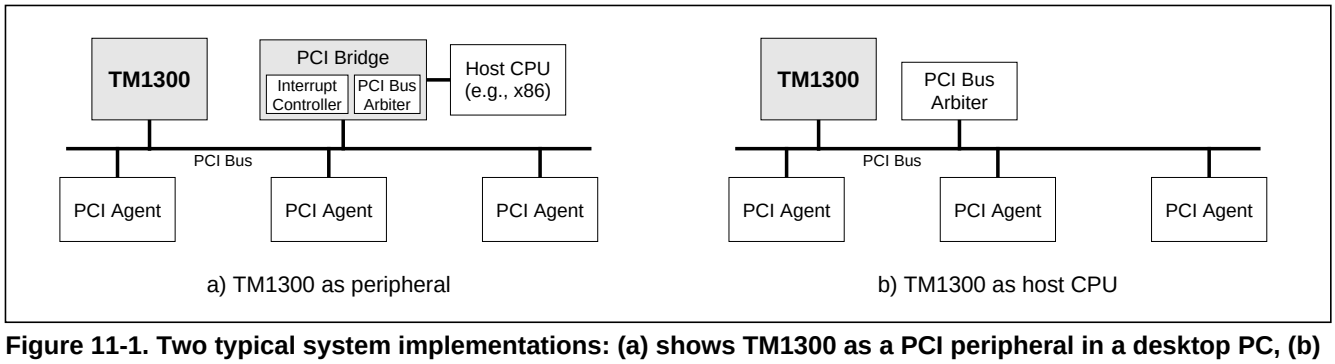


Figure 11-1. Two typical system implementations: (a) shows TM1300 as a PCI peripheral in a desktop PC, (b) shows an embedded system with TM1300 as the host CPU.

11.3 PCI INTERFACE AS AN INITIATOR

The following classes of operations invoked by TM1300 cause the PCI interface to act as a PCI initiator:

- Transparent, single-word (or smaller) transactions caused by DSPCPU loads and stores to the PCI address aperture
- Explicitly programmed single-word I/O or configuration read or write transactions
- Explicitly programmed multi-word DMA transactions.
- ICP DMA

11.3.1 DSPCPU Single-Word Loads/Stores

From the point of view of programs executed by TM1300's DSPCPU, there are three apertures into TM1300's 4-GB memory address space:

- SDRAM space (0.5 to 64 MB; programmable)
- MMIO space (2 MB)
- PCI space

MMIO registers control the positions of the address-space apertures (see Chapter 3, "DSPCPU Architecture"). The SDRAM aperture begins at the address specified in the MMIO register DRAM_BASE and extends upward to the address in the DRAM_LIMIT register. The 2-MB MMIO aperture begins at the address in MMIO_BASE (defaults to 0xEFE00000 after power-up). All addresses that fall outside these two apertures are assumed to be part of the PCI address aperture. References by DSPCPU loads and stores to the PCI aperture are reflected to external PCI devices by the coordinated action of the data cache and PCI interface.

When a DSPCPU load or store targets the PCI aperture (i.e., neither of the other two apertures), the DSPCPU's data cache automatically carries out a special sequence of events. The data cache writes to the PCI_ADR and (if the DSPCPU operation was a store) PCI_DATA registers in the PCI interface and asserts (load) or de-asserts (store) the internal signal pci_read_operation (a direct connection from the data cache to the PCI interface).

While the PCI interface executes the PCI bus transaction, the DSPCPU is held in the stall state by the data cache. When the PCI interface has completed the transaction, it asserts the internal signal pci_ready (a direct connection from the PCI interface to the data cache).

When pci_ready is asserted, the data cache finishes the original DSPCPU operation by reading data from the PCI_DATA register (if the DSPCPU operation was a load) and releasing the DSPCPU from the stall state.

Explicit Writes to PCI_ADR, PCI_DATA

The PCI_ADR and PCI_DATA registers are intended to be used only by the data cache. Explicit writes are not allowed and may cause undetermined results and/or data corruption.

11.3.2 I/O Operations

Explicit programming by DSPCPU software is the only way to perform transactions to PCI I/O space. DSPCPU software writes three MMIO registers in the following sequence:

1. The IO_ADR register.
2. The IO_DATA register (if PCI operation is a write).
3. The IO_CTL register (controls direction of data movement and which bytes participate).

The PCI interface starts the PCI-bus I/O transaction when software writes to IO_CTL. The interface can raise a DSPCPU interrupt at the completion of the I/O transaction (see BIU_CTL register definition in Section 11.7.5, "BIU_CTL Register") or the DSPCPU can poll the appropriate status bit (see BIU_STATUS register definition in Section 11.7.4, "BIU_STATUS Register"). Note that PCI I/O transactions should NOT be initiated if a PCI configuration transaction described below is pending. This is a strict implementation limitation.

The fully detailed description of the steps needed can be found in Section 11.7.13, "IO_CTL Register."

11.3.3 Configuration Operations

As with I/O operations, explicit programming by DSPCPU software is the only way to perform transactions to PCI configuration space. DSPCPU software writes three MMIO registers in the following sequence:

1. The CONFIG_ADR register.
2. The CONFIG_DATA register (if PCI operation is a write).
3. The CONFIG_CTL register (controls direction of data movement and which bytes participate).

The PCI interface starts the PCI-bus configuration transaction when software writes to CONFIG_CTL. As with the I/O operations, the biu_status and BIU_CTL registers monitor the status of the operation and control interrupt signaling. Note that PCI configuration space transactions should NOT be initiated if a PCI I/O transaction described above is pending. This is a strict implementation limitation.

The fully detailed description of the steps needed can be found in Section 11.7.10, "CONFIG_CTL Register."

11.3.4 DMA Operations

The PCI interface can operate as an autonomous DMA engine, executing block-transfer operations at maximum PCI bandwidth. As with I/O and configuration operations, DSPCPU software explicitly programs DMA operations.

General-purpose DMA

For DMA between SDRAM and PCI, DSPCPU software writes three MMIO registers in the following sequence:

1. The SRC_ADR and DEST_ADR registers.
2. The DMA_CTL register (controls direction of data movement and amount of data transferred).

The PCI interface begins the PCI-bus transactions when software writes to DMA_CTL. As with the I/O and configuration operations, the BIU_STATUS and BIU_CTL registers monitor the status of the operation and control interrupt signaling.

The fully detailed description of the steps needed to start a DMA transaction can be found in Section 11.7.16, "DMA_CTL Register."

Image-Coprocessor DMA

The PCI interface also executes DMA transactions for the Image Coprocessor (ICP). The ICP performs rapid post-processing of image data and writes it at PCI DMA speed to a PCI graphics card frame buffer. The ICP cannot perform PCI read transactions. BIU_CTL.IE (ICP DMA Enable) should be asserted before attempting ICP PCI operation. Programming of ICP DMA is described in Section 14.6, "Operation and Programming."

11.4 PCI INTERFACE AS A TARGET

The TM1300 PCI interface responds as a target to external initiators for a limited set of PCI transaction types:

- Configuration read/write
- Memory read/write, read line, and read multiple to the TM1300 SDRAM or MMIO apertures. See Section 11.9, "Limitations."

TM1300 ignores PCI transactions other than the above.

11.5 TRANSACTION CONCURRENCY, PRIORITIES, AND ORDERING

The PCI interface can be processing more than one operation at a given time. There are five distinct classes of operations implemented by the PCI interface:

1. DSPCPU load/store to PCI space.
2. PCI I/O read/write and PCI configuration read/write.
3. General-purpose DMA read/write.
4. ICP DMA write.
5. External-PCI-agent-initiated read/write (to TM1300 on-chip resource).

If the active general-purpose DMA transaction is a read, up to five transactions, one from each, can be active simultaneously. If the active general-purpose DMA operation is a write, then only four transactions can be active simultaneously because general-purpose DMA writes force ICP DMA writes to wait until the general-purpose DMA completes. When a general-purpose DMA write is pending, an in-progress ICP DMA operation is suspended at the next 64-byte block boundary and waits until the completion of the DMA write operation. General-purpose DMA reads are interleaved with ICP DMA writes, so both can be active concurrently.

PCI single-data-phase transactions (DSPCPU load/store, I/O read/write, and configuration read/write) are executed in the order they are issued to the PCI interface. Note the strict implementation limitation that PCI - I/O and PCI configuration transactions cannot be simultaneously active.

11.6 REGISTERS ADDRESSED IN PCI CONFIGURATION SPACE

Since it is a PCI device, TM1300 has a set of configuration registers to determine PCI behavior. PCI configuration registers allow full relocation of interrupt binding and address mapping by the system's host processor. This relocatability of PCI-space parameters eases installation, configuration, and system boot.

The PCI standard specifies a 64-byte PCI configuration header region within a reserved 256-byte block. During system initialization, host system software scans the PCI bus, looking for PCI headers, to determine what PCI devices are present in the system. The fields in the header region uniquely identify the PCI device and allow the host to control the device in a generic way. Figure 11-2 shows the layout of the configuration header region.

Figure 11-2 also shows the initial values for the configuration registers. Some registers, such as Device ID, have hardwired values, while others are programmed by software. Still others are set automatically from the external boot ROM during TM1300's power-up initialization.

11.6.1 Vendor ID Register

For TM1300, the value of the 16-bit Vendor ID field is hardwired to 0x1131 (Philips). This value identifies the manufacturer of a PCI device. Valid vendor identifiers are assigned by the PCI special interest group (PCI SIG) to ensure uniqueness. The value 0xFFFF is reserved and must be returned by the host/PCI bridge when an attempt is made to read a non-existent device's Vendor ID configuration register.

11.6.2 Device ID Register

For TM1300, the value of the 16-bit Device ID field is hardwired to 0x5402. The Device ID is assigned by the manufacturer to uniquely identify each PCI device it makes.

11.6.3 Command Register

The 16-bit command register provides basic control over a PCI device's ability to generate and/or respond to PCI bus cycles. According to the PCI specification, after reset, all bits in this register are cleared to '0' (except for a device that must be initially enabled). Clearing all bits to '0' logically disconnects the device from the PCI bus for all accesses except configuration accesses.

The command register format is shown in Figure 11-3. Table 11-2 summarizes the field values. Note that the values listed as 'normally taken' are not necessarily the reset values, i.e. the Command register is reset to all '0's, meaning the features are disconnected on reset.

Following are detailed descriptions of the command register fields.

I/O (I/O access enable). This bit controls a device's ability to respond to I/O-space accesses. A value of '0' disables PCI device response; a value of '1'enables response. This bit is hardwired to '0' because all TM1300 internal registers are memory mapped.

MA (Memory access enable). This bit controls response to memory-space accesses. A value of '0' disables TM1300 response; a value of '1' enables response. This bit is set to '0' at power-up; software can set this bit to '1' with a configuration write.

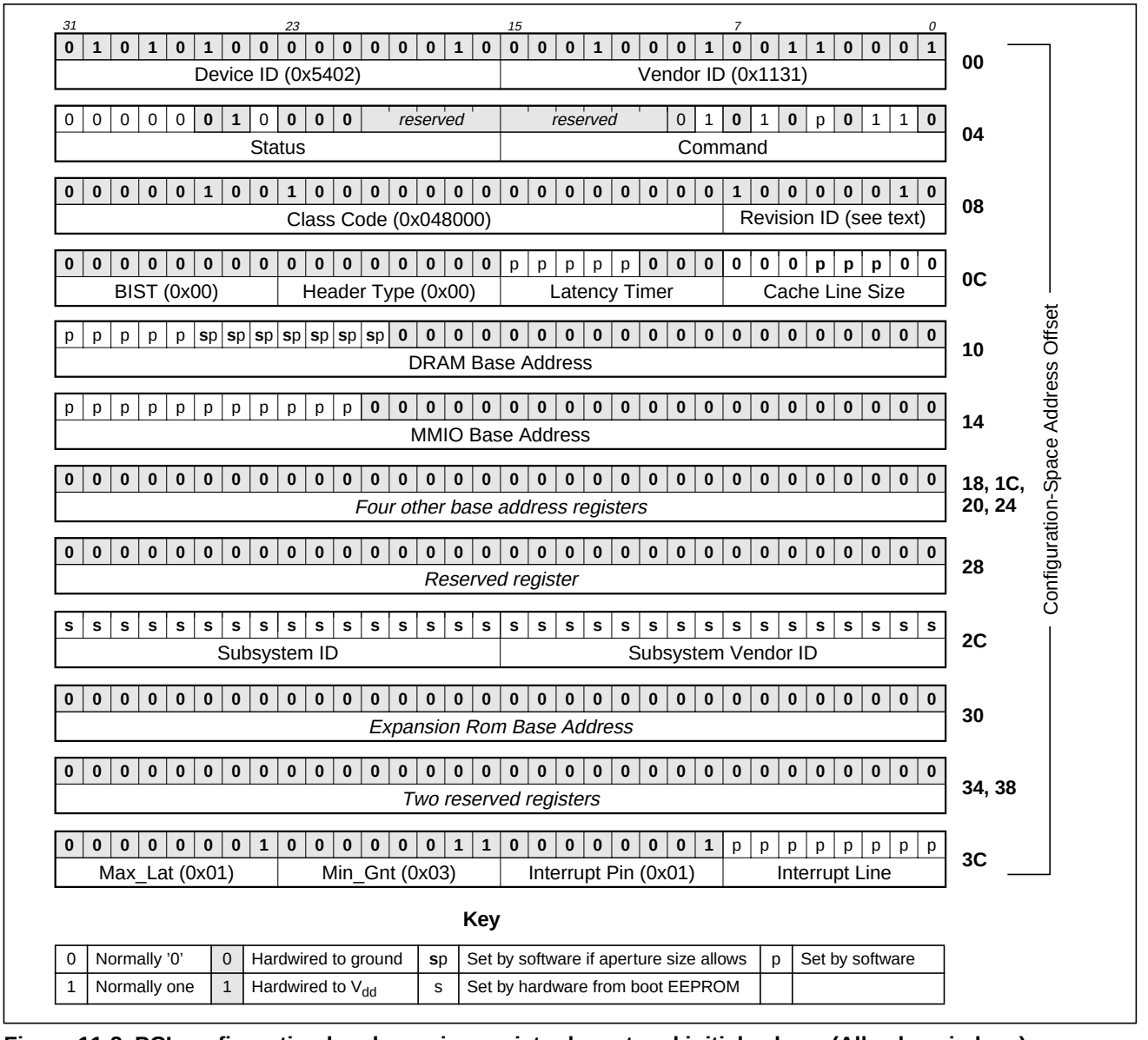


Figure 11-2. PCI configuration header region register layout and initial values. (All values in hex.)

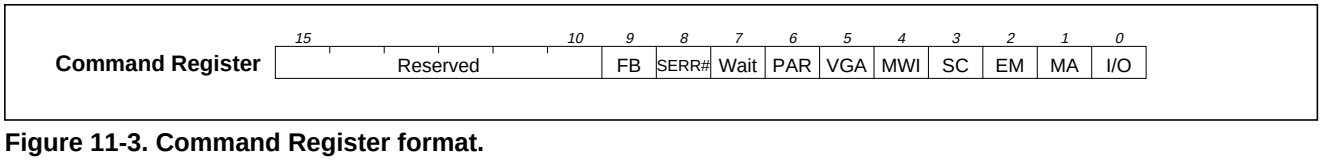


Figure 11-3. Command Register format.

Table 11-2. Field values for Command Register

Field	Value Explanation
I/O	Hardwired to 0 (ignore I/O space accesses)
MA	0 ⇒ no recognition of memory-space accesses 1 ⇒ recognizes memory-space accesses
EM	0 ⇒ cannot act as PCI initiator 1 ⇒ can act as PCI initiator
SC	Hardwired to 0 (ignore special cycle accesses)
MWI	0 ⇒ cannot generate memory write and invalidate 1 ⇒ can generate memory write and invalidate
VGA	Hardwired to 0
Par	0 ⇒ ignore parity errors 1 ⇒ acknowledge parity errors
SERR#	0 ⇒ disable driver for serr# pin 1 ⇒ enable driver for serr# pin
FB	0 ⇒ fast back-to-back only to same agent 1 ⇒ fast back-to-back to different agents
Reserved	Write ignored; reads return 0

EM (Enable mastering). This bit controls the TM1300 PCI interface's ability to act as a PCI master. A value of '0' prevents the PCI interface from initiating PCI accesses; a value of '1' allows the PCI interface to initiate PCI accesses.

Note that the EM bit is automatically set to '1' whenever the HE bit in the BIU_CTL register is set to '1' (see Section 11.7.5, "BIU_CTL Register"). Mastering must be enabled for TM1300 to serve as PCI host processor.

EM is set to '0' at power-up. Host system software can set this bit to '1' with a configuration write.

SC (Special cycle). This bit controls PCI device recognition of special-cycle operations. A value of '0' causes a PCI device to ignore all special cycles; a value of '1' allows a PCI device to monitor special cycle operations. This bit is hardwired to '0' in TM1300.

MWI (Memory write and invalidate). This bit determines a PCI device's ability to generate memory-write-and-invalidate commands. A value of '1' allows a PCI device to generate memory-write-and-invalidate commands; a value of '0' forces the PCI device to use memory-write commands instead. TM1300 implements this bit. The conditions under which TM1300 DMA transactions generate memory-write-and-invalidate are described in Section 11.7.16, "DMA_CTL Register." Details of operation can be found in Section 11.6.7, "Cache Line Size Register." Image Coprocessor DMA writes always use regular memory-write transactions.

VGA (VGA palette snoop). This bit controls how VGA-compatible PCI devices handle accesses to their palette registers. This bit is hardwired to '0'.

PAR (Parity error response). This bit controls signaling of parity errors (data or address). A value of '0' causes the PCI interface to ignore parity errors; a value of '1' causes the PCI interface to report parity errors on the perr# PCI signal. This bit is set to '0' at power-up; since the PCI interface checks parity, software can set this bit to '1' with a configuration write.

Wait (Wait-cycle control). This bit controls whether or not a PCI device does address/data stepping. PCI devices that never do stepping must hardwire this bit to 0. Since TM1300 does not implement stepping, this bit is hardwired to '0'.

SERR# (serr# enable). This bit enables the driver of the serr# pin (system error): a value of '0' disables it, a value of '1' enables it. All PCI devices that have an serr# pin must implement this bit. This bit is set to '0' after reset; it can be set to '1' with a configuration write. SERR# and PAR must both be set to '1' to allow signaling of address parity errors on the serr# signal.

FB (Fast back-to-back enable). This bit controls whether or not a PCI master can do fast back-to-back transactions to different devices. A value of '0' means fast back-to-back transactions are only allowed when the transactions are to the same agent; a value of '1' means the master is allowed to generate fast back-to-back transactions to different agents. Initialization software will set this bit if all targets are capable of fast back-to-back transactions. In TM1300, this bit is hardwired to '0'.

Reserved. Reads from reserved bits returns '0'; writes to reserved bits cause no action.

11.6.4 Status Register

The status register is used to record information about PCI bus events. The status register format is shown in Figure 11-4. Table 11-3 lists the Status register fields.

Reserved. Reads from reserved bits return '0'; writes to reserved bits cause no action.

66M (66-MHz capable). This bit is hardwired to '0' for TM1300 (PCI runs at 33-MHz maximum).

UDF (user-definable features). Since the TM1300 PCI interface does not implement PCI user-definable features, this bit is hardwired to '0'.

FBC (Fast back-to-backcapable). The TM1300 PCI interface does not support fast back-to-back capability, so this bit is hardwired to '0'.

DPD (Data parity detected). Since the TM1300 PCI interface can act as a PCI bus initiator, this bit is implemented. DPD is set in the initiator's status register when:

- The PAR (parity-error response) bit in the command register is set, and

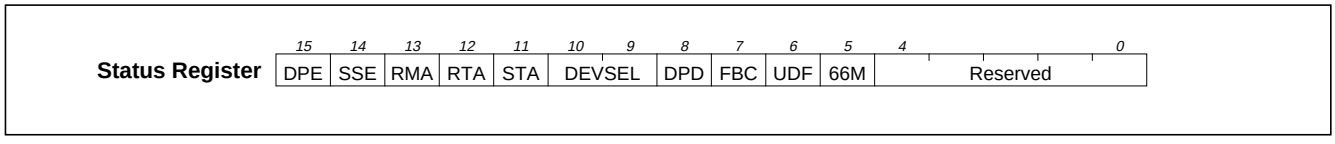


Figure 11-4. Status register format.

- The initiator asserted perr# or detected it asserted by the target (during a write cycle).

Table 11-3. Status register fields

Field	Characteristics
Reserved	Writes ignored; reads return 0
66M	PCI bus speed (hardwired to 0 ⇒ 33-MHz)
UDF	User-definable features (hardwired to 0 ⇒ none)
FBC	Fast back-to-back capable (hardwired to 0 ⇒ unsupported)
DPD	Data parity detected
DEVSEL	devsel# signal timing (hardwired to 1 ⇒ 'medium')
STA	Signaled target abort
RTA	Receive target abort
RMA	Receive master abort
SSE	Signaled system error
DPE	Detected parity error

DEVSEL (Device select timing). This read-only field defines the slowest timing that will be used for the devsel# signal when TM1300 is a target on the PCI bus. Table 11-4 shows the allowable encodings and meanings. These bits are hardwired to '01' to indicate that

Table 11-4. DEVSEL encodings

DEVSEL	Meaning
00	Fast
01	Medium
10	Slow
11	Reserved

TM1300 uses a 'medium' devsel# timing.

STA (Signaled target abort). TM1300's PCI interface sets this bit when it is a target device and aborts a transaction.

RTA (Receive target abort). TM1300's PCI interface sets this bit when it is the initiating device and the transaction is aborted by the target device. (All initiating devices must implement this bit.)

RMA (Receive master abort). TM1300's PCI interface sets this bit when it is the initiating device and aborts a transaction (except when the transaction is a special cycle). (All initiating devices must implement this bit.)

SSE (Signaled system error). TM1300's PCI interface sets this bit when it asserts the serr# signal. (TM1300 can generate serr#, so this bit is implemented; devices incapable of generating serr# need not implement SSE.)

DPE (Detected parity error). TM1300's PCI interface sets this bit when it detects a parity error, even if parity error handling is disabled. (The PAR bit in the command register enables the handling of parity errors.)

11.6.5 Revision ID Register

The value in the Revision ID register is a read only value chosen by the manufacturer to indicate product revisions. For the TM1300 product family, the two MSBs of the revision ID indicate the fab where the part was manufactured. The next two bits indicate an all-layer revision number, and the 4 LSBs indicate metal layer revisions. Each all-layer revision adds 0x10 to the revision ID and resets the 4 LSBs to '0'. Non-pin or -function compatible TriMedia devices will use the same Revision ID convention, but with a revised Device ID.

Table 11-5. Actual revision ID values

Value (hex)	Product description
0x80	TM1300 original mask - tm1f 1.0
0x81	TM1300 1st metal revision - tm1f 1.1
0x82	TM1300 2nd metal revision - tm1f 1.2

11.6.6 Class Code Register

The value in the Class Code register is read-only. System software uses the Class Code register to identify the generic function of the device, and in some cases, the Class Code can specify a register-level programming interface.

Class Code consists of three 1-byte fields as shown in Figure 11-5. The value of the upper byte, Base Class Code, broadly classifies the function of the device. The value of the middle byte, Subclass Code, identifies the function more specifically. The value of the lower byte specifies a register-level programming interface so that device-independent software can interact with the device. The meanings of the Base Class byte values are shown in Table 11-6.

The value of Base Class is hardwired to 0x04 since TM1300 is a multimedia device. Currently, there are no specific register-level programming interfaces defined for multimedia devices.

Table 11-7 lists the defined subclasses of multimedia devices. TM1300 is both a video and audio multimedia device, so its subclass value is hardwired to 0x80.

11.6.7 Cache Line Size Register

This field only matters when the MWI bit in configuration space is set. The value of the Cache Line Size register specifies the host system cache line size in units of 32-

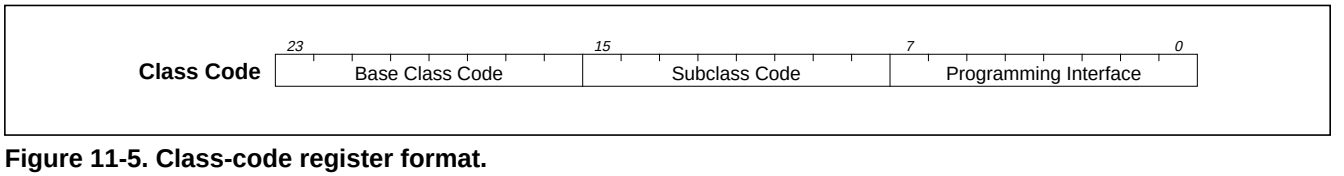


Figure 11-5. Class-code register format.

Table 11-6. Base Class Encodings

Base Class (in hex)	Meaning
00	Device was built before class code definitions were finalized
01	Mass-storage controller
02	Network controller
03	Display controller
04	Multimedia device
05	Memory controller
06	Bridge device
07	Simple communications controller
08	Base system peripheral
0A	Docking station
0B	Processor
0C	Serial bus controller
0D–FE	Reserved
FF	Device does not fit any of the above classes

Table 11-7. Subclass & programming interface fields

Subclass (in hex)	Programming Interface (in hex)	Meaning
00	00	Video device
01	00	Audio device
80	00	Other multimedia device

bit words. Initiating devices, such as the TM1300, that can generate memory-write-and-invalidate commands must implement this register. When implemented, the cache line size allows initiators participating in the PCI caching protocol to retry burst accesses at cache-line boundaries.

This register is implemented in TM1300. In the TM1300, PCI DMA performs write-and-invalidate cycles as per the table below. ICP DMA and CPU PCI writes are performed using normal memory-write cycles.

Table 11-8. Cache line size values

Cache Line Size (binary)	Effect
0000,0100	write-and-invalidates are done in 4-DWORD, i.e. 16-byte chunks
0000,1000	write-and-invalidate in 8-DWORD chunks
0001,0000	write-and-invalidate in 16-DWORD chunks
all other values	only normal ‘memory-write’ is performed

11.6.8 Latency Timer Register

The value of the Latency Timer register specifies the minimum number of PCI clock cycles the TM1300 BIU (as initiator) is allowed to own the PCI bus. This register is readable and writable in PCI configuration space.

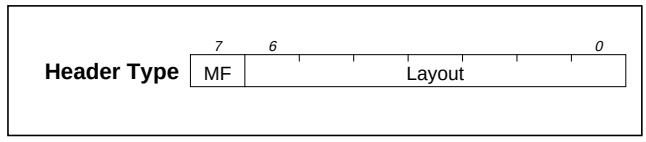


Figure 11-6. Header type register format.

This register must be writable in any PCI-initiating device that can burst more than two data phases. In the TM1300 PCI interface, the least-significant three bits are hard-wired to ’0’ and software can program any value into the most-significant five bits. This permits software to specify the time slice with a minimum granularity of eight PCI clocks. A value of ’0’ signifies maximum latency, i.e. 256 PCI clocks.

11.6.9 Header Type Register

The value of the Header Type register defines the format of words 16 through 63 in configuration space and whether or not the device contains multiple functions. Figure 11-6 shows the format of Header Type.

Bit 7 of Header Type is ’0’ for single-function devices, ’1’ for multi-function devices. TM1300 is a single-function device, so bit 7 is ’0’. Table 11-9 shows the encodings of the Layout field.

Table 11-9. Layout encodings

Layout (in hex)	Meaning
00	Non-bridge PCI device
01	PCI-to-PCI bridge device

11.6.10 Built-In Self Test Register

When implemented, the BIST register is used to control the operation of a device’s built-in self testing capability. TM1300 does not implement BIST, so this register is hardwired to return ’0’s when read.

11.6.11 Base Address Registers

The TM1300 PCI interface implements two configuration space memory Base Address registers: DRAM_BASE and MMIO_BASE. DRAM_BASE relocates TM1300’s SDRAM within the system address space; MMIO_BASE relocates the 2-MB memory-mapped I/O address aperture.

The values in the Base Address registers determine the address map as seen by both the DSPCPU and external PCI masters. These values are normally set once, and not changed dynamically once the DSPCPU operates.

Hardware RESET initializes DRAM_BASE to 0x0 and MMIO_BASE to 0xefe0,0000, after which the TM1300 boot protocol sets the final value.

In standalone systems, the autonomous boot sequence is executed. In this case, the values of DRAM_BASE and MMIO_BASE are copied from the content of the serial boot EEPROM, as described in Section 13.3.2, “Initial DSPCPU Program Load for Autonomous Bootstrap.”

In X86 or other host-assisted platforms, the PCI host assisted boot sequence is executed. In this case, the base registers are not set from the EEPROM. Instead, the host BIOS executes a scan for devices on each PCI bus. During this scan, memory apertures needed by each device are determined, and a suitable base is assigned by the host BIOS. The details of this process are described below.

Figure 11-7 shows the formats for DRAM_BASE and MMIO_BASE. Following are descriptions of the register fields.

M (Memory). The value of the M bit indicates whether the desired resource is a memory or PC I/O aperture. The M bit is hardwired to '0', indicating a memory type aperture for both the DRAM_BASE and MMIO_BASE registers.

T (Type). The value of the T field indicates the size of the base address register and constraints on its relocatability. Table 11-10 lists the encodings and meanings of the T field.

Table 11-10. Type field encodings

Type	Meaning
00	Base register is 32 bits wide; mapping can relocate anywhere in 32-bit memory space
01	Base register is 32 bits wide; mapping must relocate below 1 MB in memory space
10	Base register is 64 bits wide; mapping can relocate anywhere in 64-bit address space
11	Reserved

TM1300's PCI-interface base registers are 32 bits wide and can be relocated in the 32-bit address space; thus, the value of the T field is '00' for both DRAM_BASE and MMIO_BASE.

P (Prefetchable). The value of the P bit indicates to other devices whether or not the range is prefetchable.

The P bit in DRAM_BASE reflects the DRAM prefetchable attribute as set by the prefetchable bit in the boot prom (Refer to Table 13-5 on page 13-7 for programming).

MMIO is not prefetchable, so the P bit is hardwired to '0' for MMIO_BASE.

Being prefetchable means there are no side effects on reads, the device returns all bytes on reads regardless of the byte enables, and host bridges can merge processor writes into this range without causing errors.

Note: the setting of the P bit does not change the behavior of the cache or memory interface. It simply signals the host if the range is assumed to be prefetchable.

DRAM/MMIO base address. In X86 or other host platforms, the configuration space DRAM Base Address and MMIO Base Address fields serve two purposes. First, the host BIOS software can use them to determine the sizes of the SDRAM and MMIO apertures. Second, the BIOS can write to these fields to cause the apertures to be relocated within the PCI memory address space.

To determine the sizes of an aperture, the BIOS first writes all '1's (0xFFFFFFFF) to the address field. When the BIOS reads the field immediately after, the value returned will have '0's in all don't-care bits and '1's in all required address bits. Required address bits form a left-aligned (i.e., starting at the MSB) contiguous field of '1's, thus effectively specifying the size of the aperture.

For example, the MMIO aperture is a fixed 2-MB space. After writing all '1's to the MMIO Base Address field, a subsequent read returns the value 0xFFE00000. The M, T, and P fields are all '0' indicating the aperture is memory (not I/O), can be relocated anywhere in a 32-bit address space, and is not prefetchable. Since the aperture has 21 address bits (the position of the first '1' bit), MMIO space is a 2-MB aperture (2^{21} bytes). The host BIOS now assigns a suitable 2-MB aligned base address by writing to the MMIO_BASE register in configuration space.

The DRAM aperture can range in size from 1 MB to 64 MB (but the size must be a power of 2). Thus, the number of required address bits can range from 20 to 26. The actual amount of SDRAM present is determined by the content of the first byte of the boot EEPROM, as described in Section 13.5, "Detailed EEPROM Contents." The PCI BIU uses this size to determine which of the bits marked 'sp' in Figure 11-7 are writable and which are set to '0'. This causes the BIOS to determine the correct actual DRAM aperture size.

11.6.12 Subsystem ID, Subsystem Vendor ID Register

The subsystem and subsystem vendor ID are new in PCI Rev 2.1. These fields are optional, but their use is highly recommended as a means to have software drivers identify the board rather than the chip on the board.

This register is implemented starting with TM1300 and onwards, and replaces the 'Personality' register functionality in the TriMedia CTC chip.

The board manufacturer chooses the values of both 16 bits fields by modifying the TM1300 Boot EEPROM. The

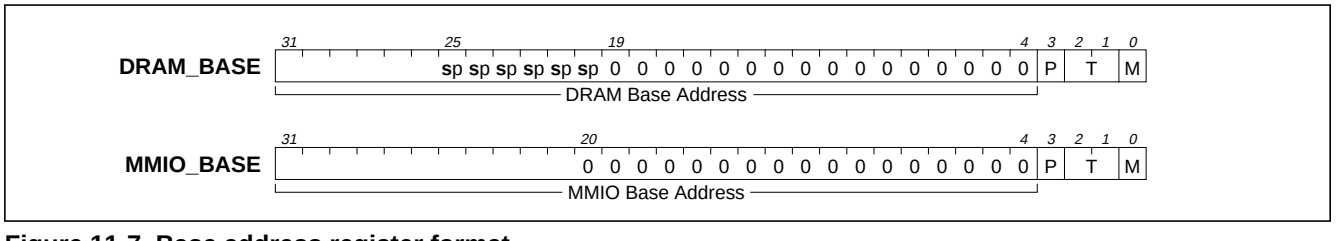


Figure 11-7. Base address register format.

location of these bits is described in Section 13.5, "Detailed EEPROM Contents." A legal Vendor ID must be obtained from the PCI SIG. The vendor is free to assign subsystem ID's.

11.6.13 Expansion ROM Base Address Register

The Expansion ROM Base Address register is similar in purpose to the SDRAM and MMIO Base Address registers. This register relocates a separate memory aperture for PCI devices that wish to implement additional ROM.

TM1300 does not implement expansion ROM; consequently, the least-significant bit of this register—which indicates whether or not TM1300 responds to expansion ROM accesses—is hardwired to '0'. All other bits also read as '0's.

11.6.14 Interrupt Line Register

The value of the Interrupt Line Register determines which input of the system interrupt controller is driven by TM1300's interrupt pin. As it configures the system and assigns resources, host system software writes this register to assign one of the system interrupt lines to TM1300.

11.6.15 Interrupt Pin Register

The value of the Interrupt Pin Register determines which interrupt pin TM1300 uses. Table 11-11 lists the possible values for this register.

Table 11-11. Interrupt pin encodings

Interrupt Pin	Meaning
1	Use interrupt pin inta#
2	Use interrupt pin intb#
3	Use interrupt pin intc#
4	Use interrupt pin intd#
all others	Reserved

Since TM1300 uses inta#, the value of this register is hardwired to '1'.

11.6.16 Max_Lat, Min_Gnt Registers

The value in the Max_Lat register specifies how often the TM1300 PCI interface needs access to the PCI bus. The value in the Min_Gnt register specifies the minimum length for a burst period on the PCI bus.

Both of these timer values are specified as multiples of 250 ns. Values of '0' indicate that a device has no specific requirements for latency and burst-length.

For TM1300, Max_Lat is hardwired to 0x01 (250 ns), and Min_Gnt is hardwired to 0x03 (750 ns).

11.7 REGISTERS IN MMIO SPACE

The TM1300 PCI interface contains 13 MMIO registers; most, except the status bits in BIU_Status, are usually written only by the DSPCPU. Table 11-12 lists the supported cycles sequenced by the PCI interface and the registers involved in each cycle. To ensure compatibility with future devices, all undefined MMIO bits should be ignored when read, and written as '0's.

The MMIO registers are all accessible to DSPCPU software, and all but the PCI_ADR and PCI_DATA registers are accessible to external PCI initiators. The facilities of TM1300's PCI interface can be useful to external initiators in certain circumstances. For example:

- The PCI DMA engine might be useful during host-assisted boot.
- Host-resident diagnostics may want to test the PCI interface during boot.
- The MMIO registers can be used to diagnose malfunctioning parts.

Note, however, that external PCI initiators can access MMIO registers in only one way: as 32-bit words on naturally aligned, 32-bit addresses. If any other type of access is attempted, the results are undefined. Also, the byte order of the external initiator and the PCI interface must be the same; otherwise, the result of an access with disagreeing byte order is undefined.

For easy reference, Table 11-13 lists the MMIO registers together with their offsets from MMIO_BASE and their accessibility by the DSPCPU and external PCI initiators.

Figure 11-8 shows the formats of the PCI interface MMIO registers. The following are detailed descriptions of the MMIO registers.

11.7.1 DRAM_BASE Register

The DRAM_BASE register in MMIO space is a shadow copy of the DRAM_BASE register in PCI Configuration space. See Section 11.6.11, "Base Address Registers," for more details. This copy provides MMIO-space access to this register. The P,T and M bitfields of this MMIO register are read-only.

11.7.2 MMIO_BASE Register

The MMIO_BASE register in MMIO space is a copy of the MMIO_BASE register in PCI Configuration space. See Section 11.6.11, "Base Address Registers," for more details. This shadow copy provides MMIO-space access to this register. The P,T and M bitfields of this MMIO register are read-only.

11.7.3 MMIO/DRAM_BASE updates

The DRAM_BASE and MMIO_BASE registers are not normally written through MMIO; their value is determined by the boot process. Though not recommended, the registers are writable in MMIO. Special care should be exercised when writing these registers:

- writing to SDRAM_BASE moves the origin of any executing DSPCPU program, which will cause it to fail
- writing to MMIO_BASE moves devices around, and moves MMIO_BASE and SDRAM_BASE around
- writing to both registers in sequence requires a delay, due to the implementation. It is recommended to space such writes far apart, or iterate until the first register written to reads back with the new value before writing the second one.

11.7.4 BIU_STATUS Register

The BIU_Status register holds bits that track the status of bus cycles initiated by the DSPCPU and bus cycles from external devices that write into SDRAM.Two bits of status are provided for each type of bus cycle: a busy bit and a done bit. The DSPCPU can read both bits; a done bit is cleared by writing a ‘1’ to it. The status register also holds two error-flag bits.

DSPCPU software must check the busy bits to avoid issuing a PCI interface bus cycle request while a request of a similar type is in progress. If a bus cycle is issued

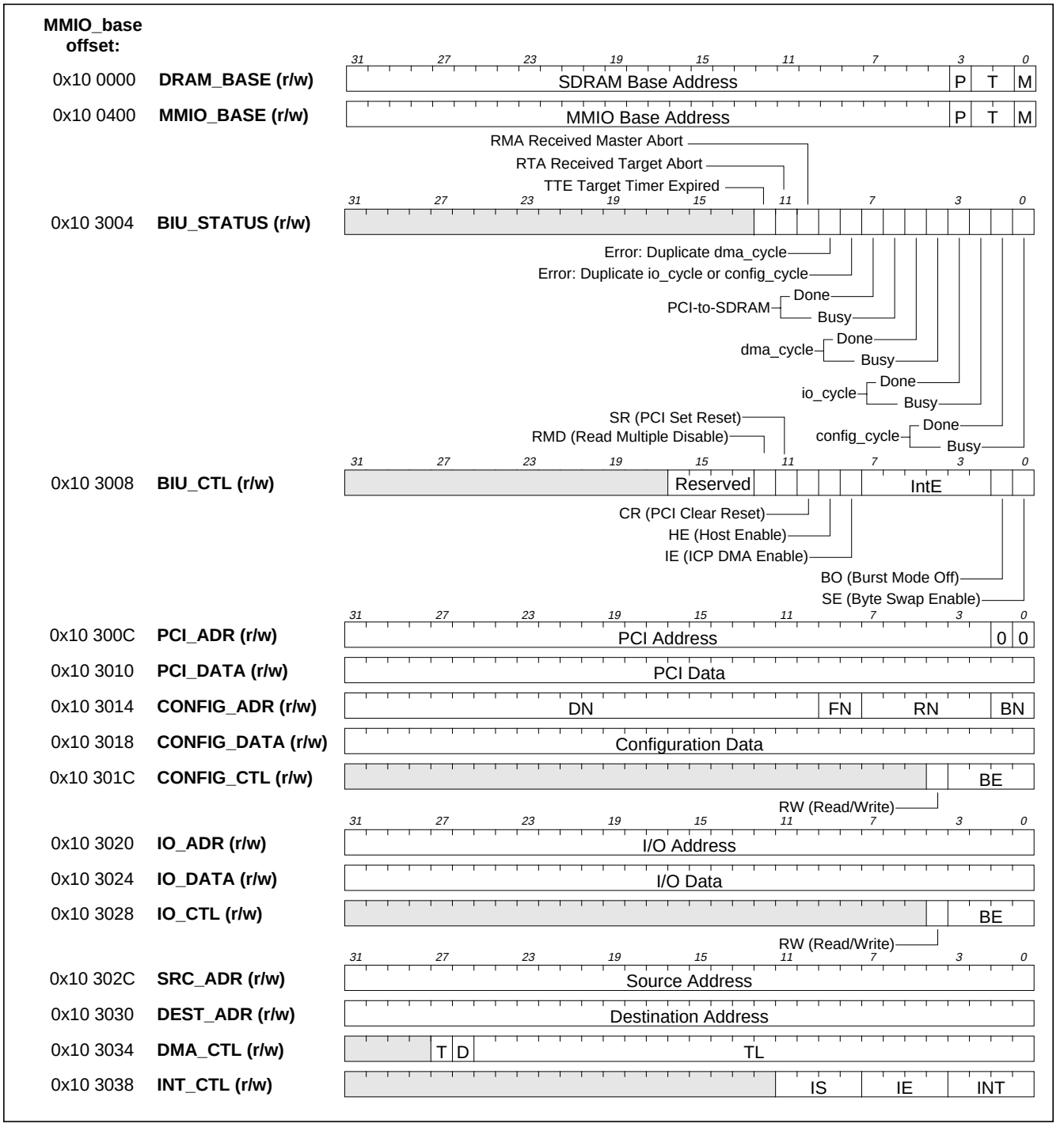


Figure 11-8. PCI interface registers accessible in MMIO address space.

while a request of similar type is in progress, the PCI interface ignores the second command and sets the appropriate error bit in the status register.

When the DSPCPU issues either an io_cycle or config_cycle request while a previous request of either type is already in progress, the PCI interface sets bit 8 in BIU_STATUS. When the DSPCPU issues a dma_cycle while a previous one is already in progress, the PCI interface sets bit 9 in BIU_STATUS. To reset either of the error bits 8 or 9 in BIU_STATUS write a '1' to it.

RTA (Received target abort). This bit is set when TM1300 initiated a transaction that was aborted by the target. To reset this bit, write a '1' to this bit position. This bit is set simultaneous with the RTA bit in the configuration space status register, but is cleared independently.

RMA (Received master abort). This bit is set when TM1300 initiated a transaction and aborts it. This usually signals a transaction to a nonexistent device. To reset this bit, write a '1' to this bit position. This bit is set simultaneous with the RMA bit in the configuration space status register, but is cleared independently.

TTE (Target timer expired). In normal operation, a read of a TM1300 data item is performed on retry basis: TM1300 tells the external master to retry, meanwhile it fetches the data item across the highway. This bit is set if an external master did not retry a read of a TM1300 data item within 32768 PCI clocks. The requested data is discarded. To reset this bit, write a '1' to this bit position. This is purely a software information bit. No software action is required when this condition occurs, but it may indicate a non-compliant or defective master on the bus.

11.7.5 BIU_CTL Register

The BIU_CTL register contains bits that control miscellaneous aspects of the PCI interface operation. Following are descriptions of the fields.

Table 11-12. PCI MMIO registers and bus cycles

Internal Cycle	Registers Involved
mmio_cycle (MMIO register R/W)	All registers accessible by external PCI devices
mem_cycle (PCI-space memory R/W)	PCI_ADR, PCI_DATA
dma_cycle (Block data transfer)	SRC_ADR, DEST_ADR, DMA_CTL
IO_cycle (I/O register R/W)	IO_ADR, IO_DATA, IO_CTL
config_cycle (Configuration register R/W)	CONFIG_ADR, CONFIG_DATA, CONFIG_CTL

SE (Swap bytes enable). This bit is initialized after reset to '0', which causes the PCI interface to operate in its default big-endian mode. Writing a '1' to SE causes accesses to MMIO registers over the PCI interface to be made in little endian mode.

Table 11-13. PCI MMIO register accessibility

Register	MMIO_BASE Offset	Accessibility	
		DSPCPU	External Initiator
DRAM_BASE	0x10 0000	R/W	R/W
MMIO_BASE	0x10 0400	R/W	R/W
BIU_STATUS	0x10 3004	R/W	R/W
BIU_CTL	0x10 3008	R/W	R/W
PCI_ADR	0x10 300C	R/W	–/–
PCI_DATA	0x10 3010	R/W	–/–
CONFIG_ADR	0x10 3014	R/W	R/W
CONFIG_DATA	0x10 3018	R/W	R/W
CONFIG_CTL	0x10 301C	R/W	R/W
IO_ADR	0x10 3020	R/W	R/W
IO_DATA	0x10 3024	R/W	R/W
IO_CTL	0x10 3028	R/W	R/W
SRC_ADR	0x10 302C	R/W	R/W
DEST_ADR	0x10 3030	R/W	R/W
DMA_CTL	0x10 3034	R/W	R/W
INT_CTL	0x10 3038	R/W	R/W

BO (Burst mode off). This bit is initialized to '0', which allows the PCI interface to support burst-mode writes as a target on the PCI bus. Setting this bit to '1' disables burst-mode writes.

With burst mode enabled, the PCI interface buffers as much data as possible into r_buffer before issuing a disconnect to the PCI initiator. With burst mode disabled, the PCI interface buffers only one data phase before issuing a disconnect to the PCI initiator.

IntE (Interrupt enables). The bits in the IntE field control the signaling of interrupts to the DSPCPU for PCI interface events. These events raise DSPCPU interrupt 16 if enabled. Interrupt 16 must be set up as a level triggered interrupt. Table 11-14 lists the function of each IntE bit. IntE is initially set to '0's (interrupts disabled).

Note that the error condition masked by bit 6 (see Section 11.7.4, "BIU_STATUS Register") occurs when either a config_cycle or an io_cycle is requested and a request of either type is already in progress. That is, the second request need not be of exactly the same type that is already in progress.

Table 11-14. IntE bit functions

BIU_CTL Bit	If set to '1', interrupt DSPCPU when...
2	config_cycle done
3	io_cycle done
4	dma_cycle done
5	pci_dram write cycle done
6	second config_cycle or io_cycle requested
7	second dma_cycle requested

IE (ICP DMA enable). This bit is must be set to ’1’ to allow the ICP to write pixel data through the PCI interface. If this bit is cleared to ’0’, the ICP is not allowed to use the PCI interface. Programming of ICP DMA is described in Section 14.6, “Operation and Programming.”

HE (Host enable). This bit is initialized to ’0’, which prevents the DSPCPU from serving as the host CPU in the PCI system. If this bit is set to one, the Enable Mastering (EM) bit in the PCI Configuration register (see Section 11.6.3, “Command Register”) is also set to ’1’ (since TM1300 must be enabled to serve as a PCI bus initiator to perform PCI configuration).

CR (PCI clear reset). This bit releases the DSPCPU from its reset state. The TM1300 device driver (executing on an external host CPU) sets this bit to ’1’ after it completes TM1300’s configuration.

SR (PCI set reset). This bit forces the DSPCPU into its reset state. Writing ’1’ to this bit resets the CPU; writing ’0’ causes no action. The TM1300 device driver (executing on an external host CPU) can set this bit to reset the DSPCPU. This form of reset resets only CPU and Lcache. The Dcache is NOT reset, nor are any peripherals.

RMD (Read Multiple Disable). In default operating mode, the RMD bit should be set to ‘0’. In that case, the BIU uses ‘memory read multiple’ PCI transactions for BIU DMA, and ‘memory read’ PCI transactions for DSPCPU reads to PCI space. If the RMD bit is set, DMA transactions are forced to also use the - less efficient - memory read transactions. Note that TM1000 only used memory read transactions.

11.7.6 PCI_ADR Register

The 30-bit PCI_ADR register is intended to be written only by the data cache. PCI_ADR participates in the special two-cycle data-cache-to-PCI protocol. See Section 11.7.7, “PCI_DATA Register,” for more information.

Only the DSPCPU can write to PCI_ADR. External PCI initiators can neither read nor write this register.

DSPCPU software should not write to this register (by writing to PCI_ADR in MMIO space). This register is intended only to support the special protocol between the data cache and PCI bus. An unexpected write to PCI_ADR via MMIO space will not be prevented by hardware and may result in data corruption on the PCI bus.

11.7.7 PCI_DATA Register

The 32-bit PCI_DATA register is intended to be used only by the data cache. PCI_DATA participates in the special two-cycle data-cache-to-PCI protocol.

The PCI_DATA and PCI_ADR registers are used together by the data cache to perform a single data phase PCI memory-space read or write. A read operation is triggered when the data cache has written the transaction address into PCI_ADR and asserted the internal signal pci_read_operation (a direct internal connection between the data cache and PCI interface). A write operation is triggered when the data cache has written both PCI_ADR and PCI_DATA with the signal pci_read_operation deasserted.

While the PCI interface is performing the PCI read or write, the DSPCPU is stalled waiting for the completion of the PCI transaction. When the PCI transaction is complete, the PCI interface asserts pci_ready (a direct internal connection between the data cache and PCI interface). To finish a read operation, the data cache reads the PCI_DATA register, forwards the data to the DSPCPU, and then unlocks the DSPCPU. To finish a write, the data cache simply unlocks the DSPCPU.

Note that, if the DSPCPU attempts to access a non-existent PCI address, an RMA condition occurs. In this case, the value in the PCI_DATA register is set to ‘0’. Hence, the DSPCPU always reads non-existent PCI locations as ‘0’.

Normal MMIO write operations to PCI_DATA have no effect. Reads return the register’s current value. External PCI initiators can neither read nor write this register.

11.7.8 CONFIG_ADR Register

The CONFIG_ADR register is written by the DSPCPU to set up for a configuration cycle. When TM1300 is acting as the host CPU, it must configure devices on the PCI bus. The DSPCPU writes CONFIG_ADR to select a configuration register within a specific PCI device. See Section 11.7.10, “CONFIG_CTL Register,” for more information on initiating configuration cycles.

Following are descriptions of the fields of CONFIG_ADR.

BN (PCI bus number). The BN field (the two least-significant bits of CONFIG_ADR) selects one of four possible PCI buses. A value of ’0’ for BN means that the targeted device is on the PCI bus directly connected to TM1300 and that any PCI-to-PCI bridges should ignore the configuration address. Any value for BN other than ’0’ means that the targeted device is on a PCI bus connected to a PCI-to-PCI bridge and that all devices directly connected to TM1300’s local PCI bus should ignore the configuration address.

RN (Register number). The RN field (bits 2..7 of CONFIG_ADR) is used to specify one of the 64 configuration words within the target device’s configuration space.

FN (Function number). The FN field (bits 8..10 of CONFIG_ADR) is used to specify one of up to eight functions of the addressed PCI device.

DN (Device number). The DN field (bits 11..31 of CONFIG_ADR) is used to select the targeted PCI device. Each bit corresponds to one of the 21 possible PCI devices on a single PCI bus, i.e., each bit corresponds to the idsel signal of one PCI device. Only one idsel signal—and, therefore, only one DN bit—can be asserted during a given configuration cycle.

11.7.9 CONFIG_DATA Register

The 32-bit CONFIG_DATA register is used by the DSPCPU to buffer data for a configuration cycle. When TM1300 is acting as the host CPU, it must configure the

PCI bus and devices. The DSPCPU writes or reads CONFIG_DATA depending on whether it is performing a write or read to a PCI device's configuration space. See Section 11.7.10, "CONFIG_CTL Register," for more information on initiating configuration cycles.

11.7.10 CONFIG_CTL Register

The DSPCPU writes to CONFIG_CTL to trigger a configuration read or write cycle on the PCI bus. A PCI configuration read or write should not be performed during an ongoing PCI I/O read or write.

The steps involved in a DSPCPU PCI configuration access are:

1. Wait until BIU_STATUS io_cycle.Busy and config_cycle.Busy are both de-asserted
2. Write to CONFIG_ADR as described above, and (in case of a write operation) write to CONFIG_DATA.
3. Write to CONFIG_CTL to start the read or write.This action sets config_cycle.Busy.
4. Wait (polling or interrupt based) until config_cycle.Done is asserted by the hardware.
5. Retrieve the requested data in CONFIG_DATA (in case of a read)
6. Clear config_cycle.Done by writing a '1' to it.

Following are descriptions of the fields of CONFIG_CTL and a discussion of how a DSPCPU write to CONFIG_CTL triggers configuration cycles.

BE (Byte enables). The BE field (the four LSBs of CONFIG_CTL) determines the state of PCIs 4-line c/be# bus during the data phase of a configuration cycle. Since the c/be# bus signals are active low, a '0' in a BE field bit means byte participates; a '1' in a BE field bit means 'byte does not participate.' Table 11-15 shows the correspondence between BE bits and bytes on the PCI bus *assuming little-endian byte order*.

RW (Read/Write). The RW field (bit 4 of CONFIG_CTL) determines whether the configuration cycle will be a read or a write. Table 11-16 shows the interpretation of RW.

Table 11-15. BE field interpretation (assumes little-endian byte ordering)

BE Bit	Interpretation
0	0 ⇒ byte 0 (LSB) participates 1 ⇒ byte 0 (LSB) does not participate
1	0 ⇒ byte 1 participates 1 ⇒ byte 1 does not participate
2	0 ⇒ byte 2 participates 1 ⇒ byte 2 does not participate
3	0 ⇒ byte 3 (MSB) participates 1 ⇒ byte 3 (MSB) does not participate

A write by the DSPCPU to the CONFIG_CTL register starts a configuration cycle on the PCI bus. The CONFIG_DATA (for a write) and CONFIG_ADR registers must be set up before writing to CONFIG_CTL.

Table 11-16. RW Interpretation

RW	Interpretation
0	Write
1	Read

During a configuration read, the PCI interface drives the PCI bus with the address from CONFIG_ADR and the BE field from CONFIG_CTL. The returned data is buffered in CONFIG_DATA. When the data is returned, the PCI interface will generate a DSPCPU interrupt if the appropriate IntE bit is set in BIU_CTL. Alternatively, DSPCPU software can poll the appropriate "done" status bin in BIU_STATUS. Finally, DSPCPU software reads the CONFIG_DATA register in MMIO space to access the data returned from the configuration cycle.

A write operation proceeds as for a read, except that PCI data is driven from CONFIG_DATA during the transaction and no data is returned in CONFIG_DATA.

11.7.11 IO_ADR Register

The 32-bit IO_ADR register is written by the DSPCPU to set up for an access to a location in PCI I/O space. The DSPCPU writes the address of the I/O register into IO_ADR. See Section 11.7.13, "IO_CTL Register," for more information on initiating I/O cycles.

11.7.12 IO_DATA Register

The 32-bit IO_DATA register is used by the DSPCPU to set up for an access to a location in PCI I/O space. The DSPCPU writes or reads IO_DATA depending on whether it is performing a write or read from IO space. See Section 11.7.13, "IO_CTL Register," for more information on initiating I/O cycles.

11.7.13 IO_CTL Register

The DSPCPU writes to IO_CTL to trigger a read or write access to PCI I/O space. The function of this register is similar to that of CONFIG_CTL, and the protocol for an I/O cycle is similar to the configuration cycle protocol. A PCI I/O read or write should not be performed during an ongoing PCI configuration read or write.

The steps involved in a DSPCPU PCI I/O access are:

1. Wait until BIU_STATUS io_cycle.Busy and config_cycle.Busy are both de-asserted
2. Write IO address to IO_ADR, and (in case of a write operation) write data to IO_DATA.
3. Write to IO_CTL to start the read or write.This action sets io_cycle.Busy.
4. Wait (polling or interrupt based) until io_cycle.Done is asserted by the hardware.
5. Retrieve the requested data in IO_DATA (in case of a read)
6. Clear io_cycle.Done by writing a '1' to it.

Following are descriptions of the fields of IO_CTL and a discussion of how a DSPCPU write to IO_CTL triggers I/O cycles.

BE (Byte enables). The BE field (the four least-significant bits of IO_CTL) determines the state of PCI's 4-line c/be# bus during the data phase of an I/O cycle. Since the c/be# bus signals are active low, a '0' in a BE field bit means 'byte participates;' a '1' in a BE field bit means 'byte does not participate.' Table 11-15 shows the correspondence between BE bits and bytes on the PCI bus *assuming little-endian byte order*.

RW (Read/Write). The RW field (bit 4 of IO_CTL) determines whether the I/O cycle will be a read or a write. Table 11-16 shows the interpretation of RW (0 ⇒ write, 1 ⇒ read).

A write by the DSPCPU to the IO_CTL register starts an I/O cycle on the PCI bus. The IO_DATA (for a write) and IO_ADR registers must be set up before writing to IO_CTL.

During an I/O read, the PCI interface drives the PCI bus with the address from IO_ADR and the BE field from IO_CTL. The returned data is buffered in IO_DATA. When the data is returned, the PCI interface will generate a DSPCPU interrupt if the appropriate IntE bit is set in BIU_CTL. Alternatively, DSPCPU software can poll the appropriate 'done' status bit in BIU_STATUS. Finally, DSPCPU software reads the IO_DATA register in MMIO space to access the data returned from the I/O cycle.

A write operation proceeds as for a read, except that PCI data is driven from IO_DATA during the transaction and no data is returned in IO_DATA.

11.7.14 SRC_ADR Register

The 32-bit SRC_ADR register is used to set the source address for a block transfer DMA operation. The address in SRC_ADR must be word (4-byte) aligned, i.e. the 2 LSBs have to be '0'. The content of this register during or after DMA is not defined, hence it cannot be used to track progress or verify completion of a DMA transaction.

11.7.15 DEST_ADR Register

The 32-bit DEST_ADR register is used to set the destination address for a block transfer DMA operation. The address is DEST_ADR must be word (4 byte) aligned, i.e. the 2 LSBs must be '0'. The content of this register during or after DMA is not defined, hence it cannot be used to track progress or verify completion of a DMA transaction.

11.7.16 DMA_CTL Register

A write by the DSPCPU to the DMA_CTL register starts a DMA block transfer on the PCI bus. The SRC_ADR and DEST_ADR registers must be set up before writing to DMA_CTL.

The steps involved in a DMA transfer are:

1. Wait until BIU_STATUS dma_cycle.Busy is de-asserted
2. Write to SRC_ADR and DEST_ADR as described above
3. Write to DMA_CTL to start the DMA transaction.This action sets dma_cycle.Busy
4. Wait (polling or interrupt based) until dma_cycle.Done is asserted by the hardware
5. Clear dma_cycle.Done by writing a '1' to it

The fields of DMA_CTL are described below.

TL (Transfer length). The TL field (bits 0..25 of DMA_CTL) specifies the number of data bytes to be transferred during the DMA operation. It must be a multiple of 4 bytes. The maximum length of a DMA operation is limited to 64 MB, the maximum amount of SDRAM supported by TM1300. The content of this field during or after a DMA transaction is not defined.

D (DMA direction). The D field (bit 26 of DMA_CTL) determines the direction of data movement during the block transfer. Table 11-17 (shows the interpretation of the D field.

Table 11-17. D interpretation

D	Data Movement Direction
0	SDRAM → PCI memory space (DMA write)
1	PCI memory space → SDRAM (DMA read)

T (DMA Transaction type). The T field (bit 27 of DMA_CTL) determines the transaction type of a write, as described below.

Table 11-18. T interpretation

T	DMA Write transaction type
0	memory write
1	memory write-and-invalidate

TM1300 generates memory write-and-invalidate PCI transactions if all conditions below are satisfied, otherwise it generates regular memory write transactions:

- The MWI bit in the Command Register is set.
- The Cache Line Size register is set to 4,8, or 16 32-bit words.
- The DMA source address is 64 byte aligned.
- The DMA destination address is cache line size aligned.
- The T bit is set

TM1300 generates 'memory read multiple' PCI transactions for DMA reads, unless the RMD (Read Multiple Disable) bit is set in BIU_CTL, in which case the less efficient 'memory read' transactions are used.

During a PCI → SDRAM block transfer, the PCI interface drives the PCI bus with the address from SRC_ADR. The returned data is buffered in r_buffer. The PCI interface then drives the address from DEST_ADR and the data from r_buffer to the SDRAM controller. SRC_ADR and DEST_ADR are incremented, the TL field in DMA_CTL

is decremented, and this sequence repeats until TL reaches ‘0’.

At the end of the PCI → SDRAM block transfer, the PCI interface will generate a DSPCPU interrupt if the appropriate IntE bit is set in BIU_CTL. Alternatively, DSPCPU software can poll the appropriate ‘done’ status bit in BIU_STATUS.

During an SDRAM → PCI block transfer, the PCI interface drives the address from SRC_ADR to the SDRAM controller. The returned data is buffered in w_buffer. The PCI interface then drives the address from DEST_ADR and the data from w_buffer to the PCI bus. SRC_ADR and DEST_ADR are incremented, the TL field in DMA_CTL is decremented, and this sequence repeats until TL reaches ‘0’.

At the end of the SDRAM → PCI block transfer, the PCI interface can generate a DSPCPU interrupt if the appropriate IntE bit is set in BIU_CTL. Alternatively, DSPCPU software can poll the appropriate ‘done’ status bit in BIU_STATUS.

11.7.17 INT_CTL Register

The INT_CTL register contains three fields for setting, enabling, and sensing the four PCI interrupt lines. Table 11-19 shows the interpretation of the fields in INT_CTL.

INT (Interrupt bits). The INT field (bits 0..3 of INT_CTL) can force a PCI interrupt to be signalled.

IE (Interrupt enable). The IE field (bits 4..7 of INT_CTL) enables TM1300 to drive PCI interrupt lines.

IS (Interrupt state). The IS field (bits 8..11 of INT_CTL) senses the state of the PCI interrupt lines.

Table 11-19. INT_CTL Bits

INT_CTL		PCI Signal	Programming
Field	Bit		
INT	0	inta#	0 ⇒ Deassert intx# 1 ⇒ Assert intx# (if enabled); i.e., pull intx# pin to a low logic level
	1	intb#	
	2	intc#	
	3	intd#	
IE	4	inta#	0 ⇒ Disable open-collector output to intx# 1 ⇒ Enable open-collector output to intx#
	5	intb#	
	6	intc#	
	7	intd#	
IS	8	inta#	Reads state of intx# pin: 0 ⇒ No interrupt asserted (intx# is high) 1 ⇒ Interrupt is asserted (intx# is low)
	9	intb#	
	10	intc#	
	11	intd#	

Figure 11-9 shows a conceptual realization of the logic used to implement the control of each intx# pin.

See also Section 3.6, “TM1300 to Host Interrupts.”

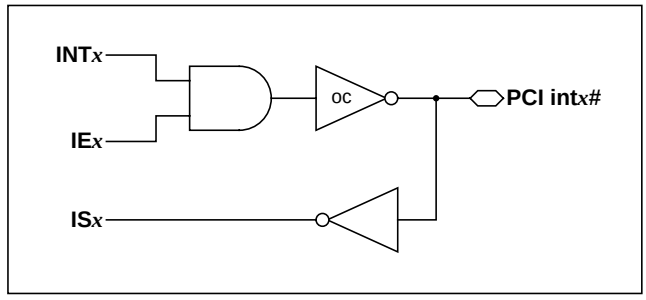


Figure 11-9. Conceptual realization of intx# pin control logic.

11.8 PCI BUS PROTOCOL OVERVIEW

TM1300’s PCI interface can generate and respond to several types of PCI bus commands. Table 11-20 lists the 12 possible commands and whether or not TM1300 can generate them.

Table 11-20. TM1300 PCI Commands as Initiator

TM1300 Generates	TM1300 Cannot Generate
Configuration read Configuration write Memory read Memory read multiple Memory write Memory write and invalidate I/O read I/O write	Interrupt acknowledge Special cycle Dual address Memory read line

Table 11-21 lists the 12 possible commands and whether or not TM1300 can respond to them.

Table 11-21. TM1300 PCI commands as target

TM1300 Responds To	TM1300 Ignores
Configuration read Configuration write Memory read Memory write Memory write and invalidate Memory read line Memory read multiple	I/O read I/O write Interrupt acknowledge Special cycle Dual address

The basic transfer mechanism on the PCI bus is a burst, which consists of an address phase followed by one or more data phases. In TM1300, the DSPCPU and ICP are the only two units that can cause TM1300 to become a PCI-bus initiator, i.e., only the DSPCPU and ICP can access external resources.

11.8.1 Single-Data-Phase Operations

When the DSPCPU reads or writes PC memory, the PCI transaction has only a single data phase. A typical single-data-phase read operation is illustrated in Figure 11-10. During the first clock period, the TM1300

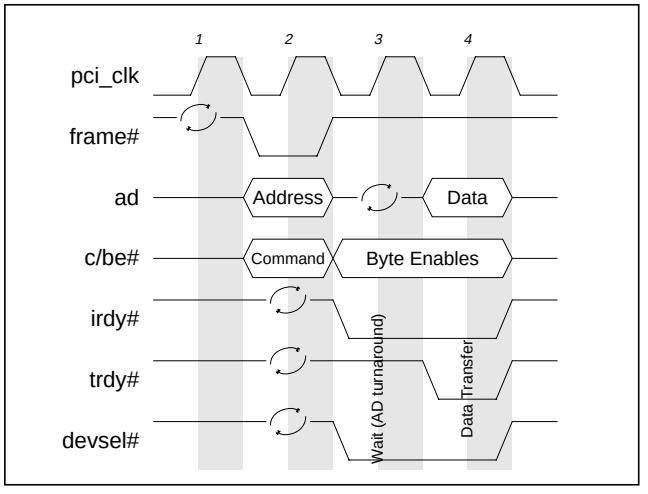


Figure 11-10. Basic single-data-phase read opera-

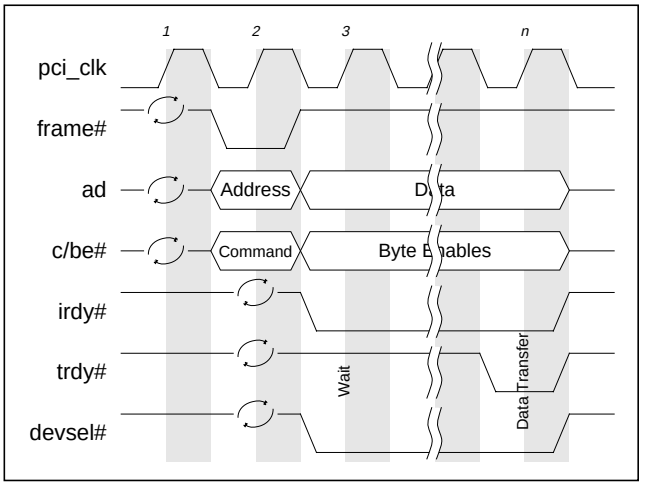


Figure 11-11. Basic single-data-phase write opera-

asserts the frame# signal to indicate that the transaction has begun and that an address and command are stable on ad and c/be#, respectively.

TM1300 then releases the ad bus, deasserts frame#, asserts irdy#, asserts byte enables on c/be#, and waits for the target to claim the transaction by asserting devsel#. The target asserts trdy# to signal the master that the ad bus contains stable data. The assertion of trdy# causes the initiator (TM1300 in this case) to sample the ad bus data and deassert irdy# to complete the single-data-phase read transaction.

Figure 11-11 shows a typical single-data-phase write operation. The operation begins like a read: TM1300 asserts the frame# signal and drives the ad bus with the target address and drives the command onto the c/be# bus.

The operation continues when TM1300 deasserts frame#, asserts irdy#, and drives the byte enables as before, but it also drives the data to be written on the ad bus. The target device asserts devsel# to claim the transaction. Eventually, the target asserts trdy# to signal that it is sampling the data on the ad bus. TM1300 continues to drive the data on the ad bus until after the target deasserts trdy#, which completes the write operation.

11.8.2 Multi-Data-Phase Operations

As with the single-data-phase operations, DMA operations begin with the assertion of frame# and valid address and command information. See Figure 11-12. The target knows a burst is requested because frame# remains asserted when irdy# becomes asserted.

In the example timing of Figure 11-12, a fast device is receiving the burst from TM1300. The target asserts devsel# and trdy# simultaneously. The trdy# signal remains asserted while TM1300 sends a new word of data on each PCI clock cycle. The burst operation shown is a 16-word burst transfer. Since only the starting address is sent by the initiator, both initiator and target must increment source and destination addresses during the burst.

The initiator signals the end of the burst of data in Figure 11-12 when it deasserts frame# in clock 17. The last word (or partial word) of data is transferred in the clock cycle after frame# is deasserted. Finally, the target acknowledges the last data phase by deasserting trdy# and devsel#.

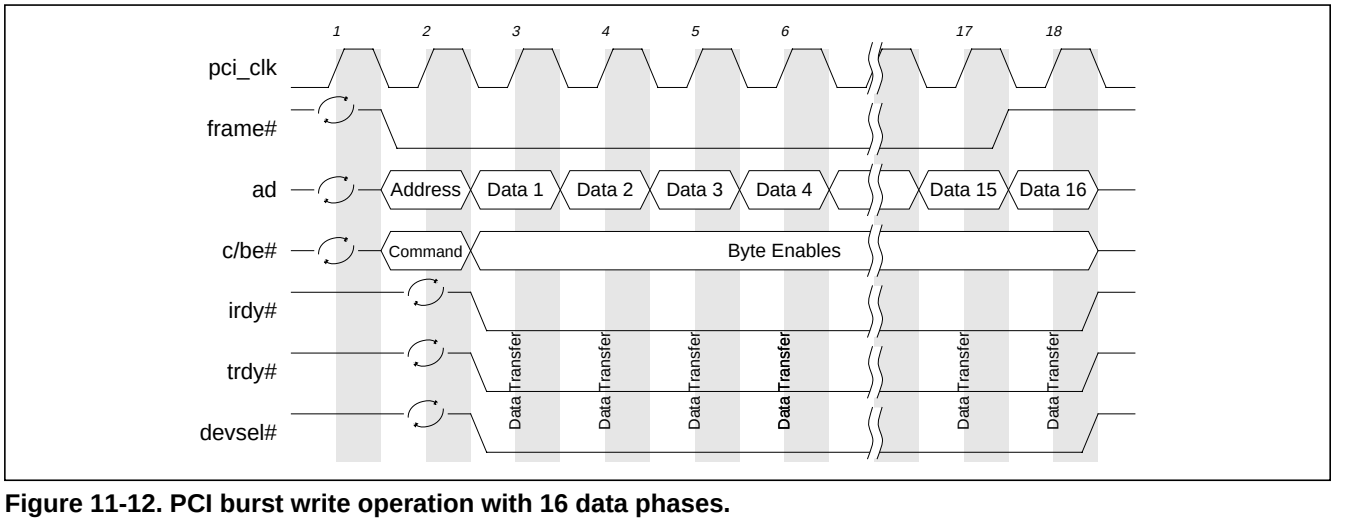


Figure 11-12. PCI burst write operation with 16 data phases.

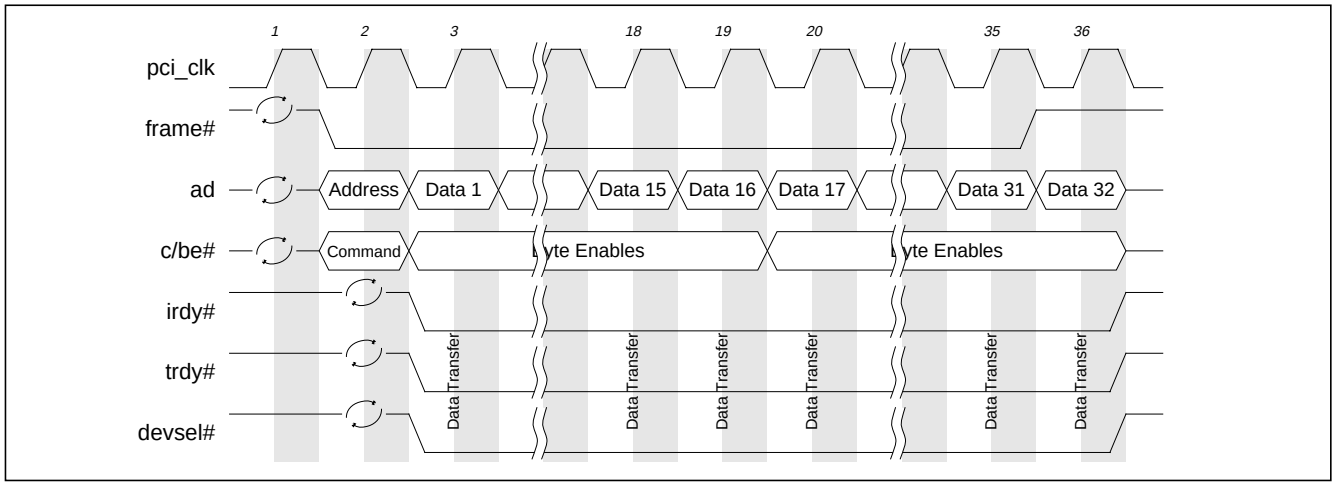


Figure 11-13. Back-to-back PCI burst write operations with 16 data phases which might be generated by the ICP when writing image data to a PCI-resident video frame buffer.

Figure 11-13 illustrates back-to-back DMA burst data transfers. The ICP is capable of exploiting the high bandwidth available with back-to-back DMA operations when it is writing image data to a frame buffer on a PCI video card.

The timing of Figure 11-13 assumes that the PCI bus is granted to TM1300 until at least the beginning of the second DMA burst operation. For as long as bus ownership is granted to TM1300 and the ICP has queued requests for data transfer, the PCI interface will perform back-to-back DMA operations. If the target eventually becomes unable to accept more data, it signals a disconnect on the TM1300 PCI interface. The PCI interface remembers where the DMA burst was interrupted and attempts to restart from that point after two bus clocks.

11.9 LIMITATIONS

11.9.1 Bus Locking

The PCI interface does not implement lock#, sbo, and sbone pins. Consequently, it is possible for both the DSPCPU and external PCI initiators to write to a critical memory section simultaneously. Software must implement policies to guarantee memory coherency.

11.9.2 No Expansion ROM

TM1300 does not implement the PCI expansion ROM capability.

11.9.3 No Cacheline Wrap Address Sequence

The PCI interface does not implement the PCI cacheline-wrap address mode for external PCI initiators that access TM1300 SDRAM.

11.9.4 No Burst for I/O or Configuration Space

Only single-data-phase transactions to configuration and I/O spaces are supported. The byte-enable signals select the byte(s) within the addressed word.

11.9.5 Word-Only MMIO Register Access

External initiators can access TM1300 MMIO registers only as full words. The byte-enable signals have no effect on the data transferred. External initiators must read and write all four bytes of MMIO registers.

SDRAM Memory System Chapter 12

by Eino Jacobs, Chris Nelson, Thorwald Rabeler, Luis Lucas

12.1 NEW IN TM1300

- Support of 64-Mbit SDRAMs organized in x16 and 128-Mbit organized in x32.
- Partial support of 64-Mbit SDRAMs organized in x8 and 128-Mbit SDRAMs organized in x16.
- External MM_MATCHOUT to MM_MATCHIN line is no longer required.

12.2 TM1300 MAIN MEMORY OVERVIEW

TM1300 connects to its local memory system with a dedicated memory bus, shown in Figure 12-1. This bus interfaces only with SDRAM or SGRAM (synchronous graphics DRAM) with its DSF pin tied low; TM1300 is the only master on this bus.

A variety of device types, speeds, and rank[1] sizes are supported allowing a wide range of TM1300 systems to be built. Table 12-1 summarizes the memory system features.

The main memory interface provides all control and data signals with sufficient drive capacity for a glueless connection to a 143-MHz memory system with up to four memory devices. Note that memory-system speed can be different from TM1300 core speed; the ratio between the memory system clock and TM1300 core clock is programmable.

With current memory technology, TM1300 supports a glueless memory interface of up to 32 MBytes with four 4×1M×16 SDRAM chips (four devices with 4 banks of one million words, each 16 bits wide) or four 4x512Kx32 or two 8×1M×16 SDRAM devices. Larger memories require a lower memory system clock frequency (though the TM1300 core clock can be higher).

1. In this document, the term ‘rank’ is used to refer to a group of memory devices that are accessed together. Historically, the term ‘bank’ has been used in this context; to avoid confusion, this document uses bank to refer to on-chip organization (SDRAM devices have two or four internal banks) and rank to refer to off-chip, system-level organization.

Table 12-1. Memory System Features

Characteristic	Comments
Data width	32 bits
Number of ranks	Four chip-select signals support up to four ranks
Memory size	From 512 KB to 64 MB
Devices supported	• Jedec SGRAM (DSF tied low) • Jedec SDRAM (×4, ×8, ×16, ×32) • PC100/133
Clock rate	Up to 143 MHz SDRAM speed (programmable ratio between TM1300 core clock and memory system clock)
Bandwidth	572 MB/sec (at 143 MHz)
Glueless interface	• Up to 4 chips at 143 MHz (e.g., 32 MB memory with 4x512Kx32 SDRAM) • Up to 8 chips at 133 MHz (e.g., 64 MB memory with 4x1Mx16 SDRAM)
Signal levels	3.3-V LVTTL

12.3 MAIN-MEMORY ADDRESS APERTURE

TM1300’s local main memory is just one of three apertures into the 4-GB address space of the DSPCPU:

- SDRAM (0.5 to 64 MB in size),
- MMIO (2 MB in size), and
- PCI (any address not in SDRAM or MMIO).

MMIO registers control the positions of the address-space apertures. The SDRAM aperture begins at the absolute address specified in the MMIO register DRAM_BASE and extends upward to the address specified in the DRAM_LIMIT register. If the SDRAM aperture overlaps the memory hole, the memory hole is ignored. The MMIO aperture begins at the address in MMIO_BASE, which defaults to 0xEFE00000 after power-up, and extends upwards 2 MB. (See Chapter 3, “DSPCPU Architecture,” for a detailed discussion.) All addresses that fall outside these two apertures are assumed to be part of the PCI address aperture.

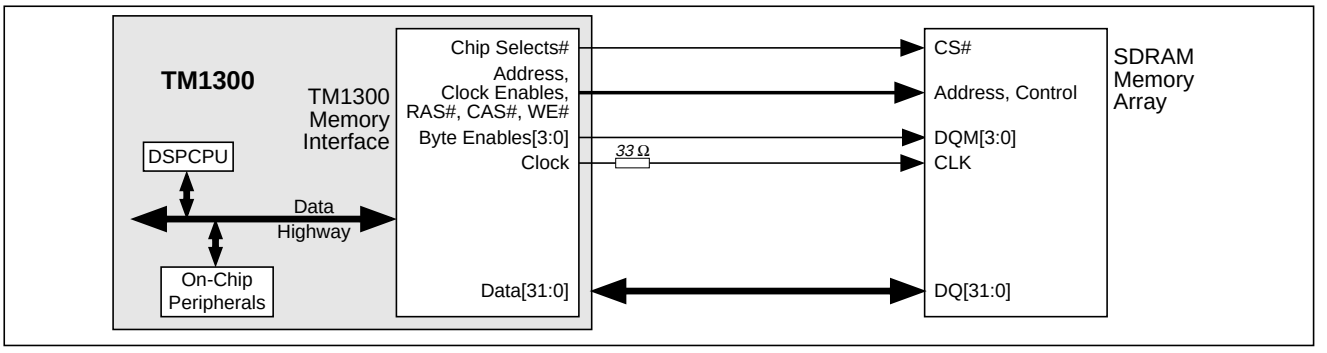


Figure 12-1. A high-performance memory interface connects the TM1300 internal highway bus to external SDRAM or SGRAM. The interface is glueless for an array of up to four devices.

12.4 MEMORY DEVICES SUPPORTED

The devices and organizations supported can be configured as listed in Table 12-2. All devices must have a LVTTL, 3.3-V interface.

Table 12-2. Supported Rank Configurations

Device Size (Mbit)	Device(s)	Rank Size
2	2 × 64K × 16 SDRAM	512 KB
4	2 × 128K × 16 SDRAM	1 MB
8	2 × 128K × 32 SGRAM	1 MB
16	2 × 256K × 32 SGRAM	2 MB
	2 × 512K × 16 SDRAM	4 MB
	2 × 1M × 8 SDRAM	8 MB
	2 × 2M × 4 SDRAM	16 MB
64	4 × 512K × 32 SDRAM	8 MB
	4 × 1M × 16 SDRAM	16 MB
	4 × 2M × 8 SDRAM[a]	32[b] MB
128	4 × 1M × 32 SDRAM	16 MB
128[a]	4 × 2M × 16 SDRAM	32[b] MB

a. Limited support for a 32-MB configuration only.
b. However MM_CONFIG.SIZE is 16 MB (i.e. 6).

Refer to Section 12.8, “Address Mapping,” in order to evaluate the support of 2-bank, 64-Mbit devices. These devices are not widely used and not described in the following sections.

12.4.1 SDRAM

TM1300 supports synchronous DRAM chips directly. SDRAM has a fast, synchronous interface that permits burst transfers at 1 word per clock cycle. The memory inside an SDRAM device is divided into two or four banks; the SDRAM implements interleaved bank access to sustain maximum bandwidth.

SDRAM devices implement a power down mechanism with self-refresh. TM1300’s power management takes advantage of this mechanism.

TM1300 supports only Jedec-compatible SDRAM with two or four internal banks of memory per device.

12.4.2 SGRAM

Also supported in TM1300 systems, SGRAM is essentially an SDRAM with additional features for raster graphics functions. The device type is standardized by Jedec and offered by multiple DRAM vendors. Tying the DSF input of an SGRAM low makes the device operates like a standard 32-bit-wide SDRAM and thus compatible with the TM1300 memory interface.

12.5 MEMORY GRANULARITY AND SIZES

TM1300 supports a variety of memory sizes thanks to:

- Many possible configurations of SDRAM devices
- Support for up to four memory ranks

The minimum memory size is 512 KB using two 2×64K×16 SDRAM devices on the 32-bit data bus. Up to four memory devices can be connected without any glue logic and without sacrificing performance. The maximum memory size with full performance is 32 MB using four 4×1M×16 SDRAM chips, four 4 x 512K x 32, or two 8×1M×16 on a 32-bit data bus.

Larger memories can be constructed using more devices. To do so, the frequency of the memory interface must be lowered to account for extra propagation delay due to the excessive loading on the interface signals (see Section 12.13, “Output Driver Capacity”).

The following rules apply to memory rank design:

- All devices in a rank must be of the same type.
- All ranks must be a power of two in size.
- All ranks must be of equal size.

Table 12-3 lists some example memory system designs.

Refer to the TM1100 Databook for smaller memory configurations. Note:

- Some of these configurations may not be economically attractive due to the price premium.
- ‘Max. MHz’ refers to the memory interface/SDRAM speed, not the TM1300 core operating frequency.

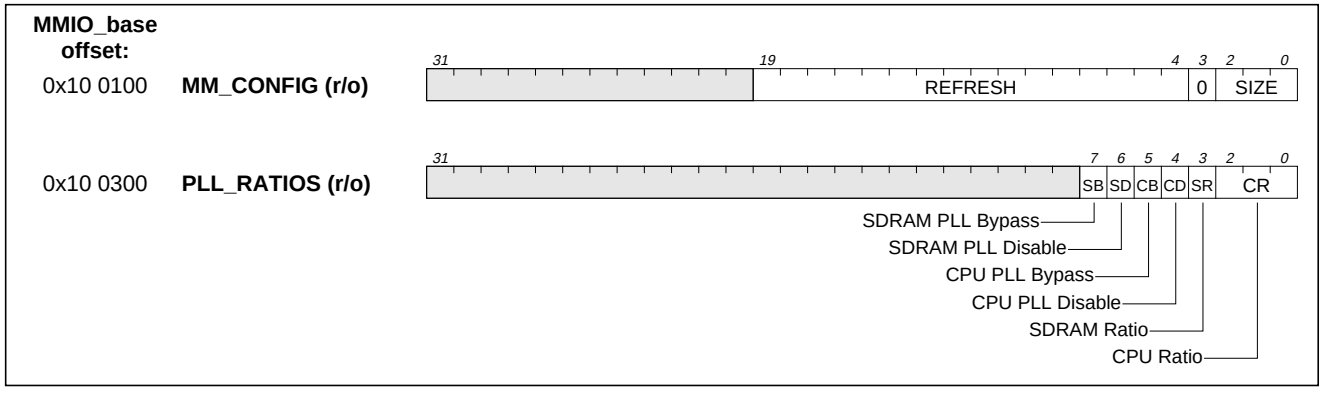


Figure 12-2. Memory interface configuration registers.

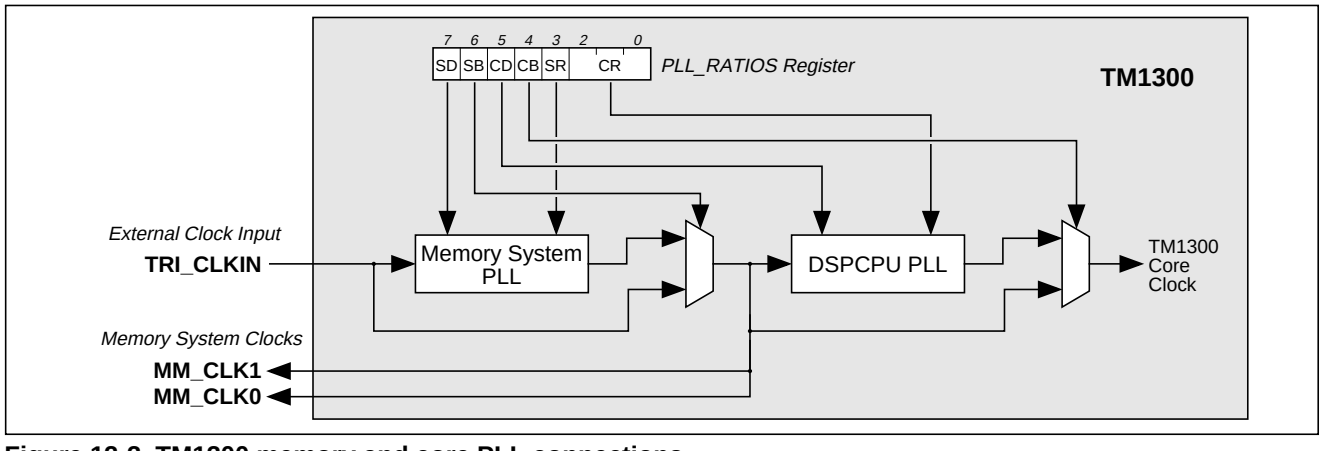


Figure 12-3. TM1300 memory and core PLL connections.

12.6 MEMORY SYSTEM PROGRAMMING

Memory system parameters are determined by the contents of two configuration registers, MM_CONFIG and PLL_RATIOS. Table 12-4 describes the function of these registers, and Figure 12-2 shows their formats. To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read.

MM_CONFIG and PLL_RATIOS are loaded from the boot EEPROM, as described in Section 13.5, "Detailed EEPROM Contents." During this boot process, the memory interface is held in reset state. After the memory interface is released from reset, the contents of these registers cannot be altered.

These registers are visible in MMIO space. They can be read, but writes have no effect.

12.6.1 MM_CONFIG Register

The MM_CONFIG register tells the memory interface how to use the local DRAM memory. The fields in this register tell the interface the rank size and the refresh rate of the memory. Table 12-6 summarizes the field functions.

REFRESH (Refresh interval). The 16-bit REFRESH field specifies the number of memory-system clock cycles between refresh operations. The default value of this register is 1000 (0x03E8). See Section 12.11, "Refresh," for more information.

Bit 3 of MM_CONFIG must be set to '0' for normal operation.

SIZE (Rank size). The 3-bit SIZE field specifies the size of each rank of DRAM. Each rank must be the size specified by SIZE. The default is a rank size of 4MB. Refer to Table 12-5 for the interpretation of this field.

12.6.2 PLL_RATIOS Register

The PLL_RATIOS register controls the operation of the separate memory-interface and CPU PLLs. Fields in this register determine if the PLLs are active and what input:output ratio each PLL should generate. Table 12-6 summarizes the field functions. Figure 12-3 shows how the PLLs are connected and how fields in the PLL_RATIOS register control them.

CR (CPU-to-memory PLL ratio). The 3-bit CR field selects one of five input-to-output clock ratios for the CPU PLL. The input clock is the memory system clock; the output clock determines the TM1300 core operating frequency. The default value is '0', which implies a 1:1 CPU:memory ratio. See Table 12-6 for other encodings.

SR (Memory-to-external PLL ratio). The 1-bit SR field selects one of two memory-to-external clock ratios for the memory interface PLL. The PLL input is TM1300's

Table 12-3. Examples of Memory Configurations

Size (MB)	Ranks	Rank Configurations	Max. MHz	Peak MB/s
8	1	four 2×1M×8 SDRAM	143	572
	2	two 2×512K×16 SDRAM two 2×512K×16 SDRAM	143	572
	1	one 4×512K×32 SDRAM	143	572
16	1	two 4×1M×16 SDRAM	143	572
	2	one 4×512K×32 SDRAM one 4×512K×32 SDRAM	143	572
24	3	one 4×512K×32 SDRAM one 4×512K×32 SDRAM one 4×512K×32 SDRAM	143	572
32	1	two 4×2M×16 SDRAM	143	572
	1	four 4×2M×8 SDRAM	143	572
	2	two 4×1M×16 SDRAM two 4×1M×16 SDRAM	143	572
	4	one 4×512K×32 SDRAM one 4×512K×32 SDRAM one 4×512K×32 SDRAM one 4×512K×32 SDRAM	143	572
48	3	two 4×1M×16 SDRAM two 4×1M×16 SDRAM two 4×1M×16 SDRAM	133	532
64	4	two 4×1M×16 SDRAM two 4×1M×16 SDRAM two 4×1M×16 SDRAM two 4×1M×16 SDRAM	133	500

Table 12-4. Memory Interface Configuration Registers

Register	Purpose
MM_CONFIG	Describes external memory configuration
PLL_RATIOS	Controls separate memory and CPU PLLs (phase-locked loops)

Table 12-5. MM_CONFIG Fields

Field	Function		
REFRESH	Refresh interval in memory clock cycles. Default value 1000 (0x03E8).		
SIZE	Memory rank size	0	Reserved
		1	512KB
		2	1MB
		3	2MB
		4	4MB
		5	8MB
		6	16MB
		7	Reserved

external input clock TRI_CLKIN; the PLL output determines the operating frequency of the memory interface and SDRAM devices. The default value is ‘0’, which implies a 2:1 memory:external ratio. A value of ‘1’ implies a 3:1 ratio.

Table 12-6. PLL_RATIOS Fields

Field	Function		
CR	CPU:memory ratio	**0**	1:1
		1	2:1
		2	3:2
		3	4:3
		4	5:4
		5–7	Reserved
SR	Memory:external ratio	**0**	2:1
		1	3:1
CD	CPU PLL Disable	0	CPU PLL on
		1	CPU PLL off
CB	CPU PLL bypass	0	CPU ← PLL
		1	**CPU ← Memory**
SD	SDRAM PLL Disable	0	SDRAM PLL on
		1	SDRAM PLL off
SB	SDRAM PLL bypass	0	Memory ← PLL
		1	**Memory ← external**

CD (CPU PLL disable). The 1-bit CD field determines whether or not the CPU PLL is turned on. The reset value is ‘1’, which disables operation of the CPU PLL and dissipates almost no power. For normal operation the value should be zero, enabling the CPU PLL.

CB (CPU PLL bypass). The 1-bit CB field determines whether the input or the output of the CPU PLL drives TM1300’s core logic. The default value is ‘1’, which causes the TM1300 core to be clocked by the input of the CPU PLL (i.e., the memory interface clock). A value of ‘0’ causes normal operation, and the core is clocked by the output of the CPU PLL.

Note that if both CB and SB are set to ‘1’ (bypass the CPU PLL and the SDRAM PLL), TM1300’s core logic is effectively clocked at the external input frequency.

Note: it is illegal to use the output of a disabled PLL. For example, it is illegal to have CD set to ‘1’ while CB is set to ‘0’.

SD (SDRAM PLL disable). The 1-bit SD field determines whether or not the SDRAM PLL is turned on. The default value is ‘1’, which disables the SDRAM PLL. In this state, it dissipates almost no power. For normal operation the value should be ‘0’, enabling the SDRAM PLL.

SB (SDRAM PLL bypass). The 1-bit SB field determines whether the input or the output of the SDRAM PLL drives the memory interface and memory devices. The default value is ‘1’, which causes the memory system to be clocked by the input of the SDRAM PLL (TM1300’s external input clock). A value of ’0’ causes normal operation, and the memory system is clocked by the output of the SDRAM PLL.

12.7 MEMORY INTERFACE PIN LIST

The memory interface consists of 61 signal pins including clocks (but excluding power and ground pins). Table 12-7 lists the interface signal pins.

Table 12-7. Memory Interface Signal Pins

Name	Function	I/O	Active...
MM_CLK[1:0]	Memory bus clock	O	High
MM_CS#[3..0]	Chip selects for the four memory ranks	O	Low
MM_RAS#	Row-address strobe	O	Low
MM_CAS#	Column address strobe	O	Low
MM_WE#	Write enable	O	Low
MM_A[13:0]	Address	O	High
MM_CKE[1:0]	Clock enable	O	High
MM_DQM[3:0]	Byte enables for dq bus	O	High
MM_DQ[31:0]	Bi-directional data bus	I/O	High

12.8 ADDRESS MAPPING

Table 12-8 shows how internal address bits from the TM1300 data highway bus are mapped to main-memory address-bus pins (MM_A[13:0]). The mapping is determined by the state of the rank-size bits in the MM_CONFIG register.

Table 12-8. Address Mapping Based on Rank Size

Rank Size	Rank Addr.	Row Address		Column Address		Bank Address	
	H.Way Bits	Pins	H.Way Bits	Pins	H.Way Bits	Pin	H.Way Bit
512 KB	20-19	8, 6–0	18, 17–11	7–0	10–6, 4–2	9	5
1 MB	21-20	8–0	19–11	7–0	10–6, 4–2	9	
2 MB	22-21	9–0	20–11	7–0	10–6, 4–2	10	
4 MB	23–22	10–0	21–11	7–0	10–6, 4–2	11	
8 MB	24-23	12, 10–0	11, 22–12	12, 8–0	11, 11–6, 4–2	11	
16 MB	25-24	13-12 10–0	12-11, 23–13	12, 9–0	11, 12–6, 4–2	11	

The column “Rank Addr./H.Way Bits” specifies which internal data-highway address bits select the preliminary SDRAM rank. The actual rank used is subject to the limitation implied by the relationship between SDRAM aperture size (described in Section 13.3.1) and the rank size. The rank is selected via the chip select bits, MM_CS#[3:0].

The column “Row Address/H.Way Bits” specifies which internal data-highway address bits map to the SDRAM row address. “Row Address/Pins” specifies which lines of TM1300’s MM_A address bus serve as the SDRAM row address.

The column ‘Column Address/H.Way Bits’ specifies which data-highway address bits map to the SDRAM column address. ‘Column Address/Pins’ specifies which lines of TM1300’s MM_A address bus serve as the SDRAM column address.

MM_A[12] is only defined for a 8- or 16-MB rank size. MM_A[12] contains H.Way bit 11 during the RAS and CAS operations. MM_A[12] can be used as a bank select (4-bank SDRAMs) or as a Row address (two bank SDRAMs).

MM_A[13] is only defined for a 16-MB rank size. MM_A[13] contains H.Way bit 12 during the RAS operation. MM_A[13] can only be used as a Row address

Highway address bits 5–0 are the offset within a 64-byte block. All ‘0’ for an aligned block transfer. Table 12-8 lists the mapping of bits 5–2 to identify in which SDRAM positions the words of a block are located. Bit 5 is always mapped to (one of) the SDRAM internal bank selects; thus, each SDRAM bank receives half (32 bytes) of the block transfer.

Highway address bits 4–2 are the word offset in a cache block. Bits 1–0 are the byte offset within a 32-bit word.

12.9 MEMORY INTERFACE AND SDRAM INITIALIZATION

Immediately after reset, the main-memory interface is initialized by placing default values in the MM_CONFIG and PLL_RATIOS registers (see Section 12.6, “Memory System Programming”). During the subsequent hardware boot process, when TM1300 reads initial values from an external ROM, these registers can be set to different values.

After TM1300 is released from the reset state, the memory interface automatically executes 10 refresh operations, then initializes the mode register in each SDRAM chip. Table 12-9 shows the settings in the SDRAM mode register(s).

Table 12-9. SDRAM Mode Register Settings

Parameter	Value
Burst length	4
Wrap type	Interleaved
CAS latency	3

12.10 ON-CHIP SDRAM INTERLEAVING

The main-memory interface (MMI) takes advantage of the on-chip interleaving of SDRAM devices. Interleaving allows the precharge, RAS, and CAS commands needed to access one internal bank to be performed while useful data transfer is occurring with the other internal bank. Thus, the overhead of preparing one bank is hidden during data movement to or from the other.

The benefit of on-chip interleaving is sustainable full-bandwidth data transfer (1 word per clock cycle). The transition from one internal bank to the other happens on 8-word boundaries; transferring 8 words gives the inactive bank time to prepare (perform precharge, RAS, and CAS) so that when the last word of the 8-word block in the active bank has been transferred, the next word from the just-precharged bank is ready on the next cycle.

The seamless transitions between the two on-chip banks can be sustained for a stream of contiguous addresses with the same direction (read or write). That is, a stream of contiguous reads or contiguous writes can sustain full bandwidth. If a write follows a read, then a small gap between transfers is needed.

Each bank access is terminated with a read or write with automatic precharge, making a separate precharge command before the next RAS unnecessary.

12.11 REFRESH

The MMI performs SDRAM refresh cycles autonomously using the CAS-before-RAS (CBR) mechanism. SDRAMs have a 4K refresh interval: either 4096 rows must be refreshed every 64 ms or 2048 rows every 32 ms.

The MMI performs refresh at timed intervals: one CBR refresh command must be issued every 15.6 µsec. A counter in the MMI keeps track of the number of SDRAM clock cycles between refresh operations. This counter starts after the CBR operation has completed; this CBR operation take 19 cycles. When the counter reaches a programmed limit, the next refresh operation is due, and the next-in-line data transfer request from the data-highway is delayed until the CBR operation is executed.

All devices in the main-memory system are refreshed simultaneously. The REFRESH field in the MM_CONFIG register determines the number of memory-system clock cycles (as distinguished from TM1300 core clock cycles) between the CBR refresh operations. Table 12-10 lists the number of memory-system clocks for typical SDRAM operation speeds.

Table 12-10. Refresh Intervals

SDRAM Operation Speed	Value For REFRESH Field (decimal)
100 MHz	1540
125 MHz	1930
133 MHz	2060
143 MHz	2210

Each CBR refresh operation takes 19 SDRAM clock cycles. Thus, at 100-MHz, refresh consumes about 1.2% of maximum available SDRAM bandwidth (19 cycles out of 1560). The bandwidth impact is slightly higher at lower frequencies.

12.12 POWER-DOWN MODE

When TM1300TM1300 is put into power-down mode to reduce power consumption, the MMI responds by putting the SDRAM devices into their power-down mode. In this mode, the SDRAM devices retain their contents through self-refresh.

12.13 OUTPUT DRIVER CAPACITY

TM1300's output driver circuits for the memory address and control signals (output signals in Table 12-7), can drive up to four memory devices when the memory interface is operating at 143 MHz. If more devices are connected, then a lower SDRAM clock frequency must be chosen.

Table 12-11 lists the clock frequency as a function of the number of memory devices connected to unbuffered memory interface signals.

Two identical outputs are provided for both the MM_CKE (clock-enable) and MM_CLK signals. Each MM_CKE and MM_CLK signal is capable of driving two SDRAM devices at 143 MHz, thus the total of four devices.

12.14 SIGNAL PROPAGATION DELAY COMPENSATION

The TM1300 MMI no longer has the two special pins, MM_MATCHOUT and MM_MATCHIN, that were used in the TM1100 and TM1000. This loop helped the interface compensate for the propagation delay through circuit-board traces to and from the external SDRAM devices. It is now integrated into the MMI. Read timing is internally derived.

To avoid excessive ringing of the clock signals, series termination with a 33-ohm resistor is advised at the clock outputs.

The delay of the memory clock with respect to the internal sending and receiving clocks is adjusted inside the memory interface to achieve reliable communication and guarantee correct setup and hold times.

Figure 12-4 shows a conceptual circuit board layout. Two SDRAM devices share a single clock output. The clock signals should have source-series termination.

12.15 CIRCUIT BOARD DESIGN

TM1300 and its memory array form a high-speed digital system. Even though only a small number of chips is involved, this digital system operates at frequencies high enough to make the analog characteristics of the connections between the chips significant. Consequently, the system designer must take care to ensure reliable operation.

12.15.1 General Guidelines

- In general, TM1300 and its memory chips should be as close together as possible to minimize parasitic

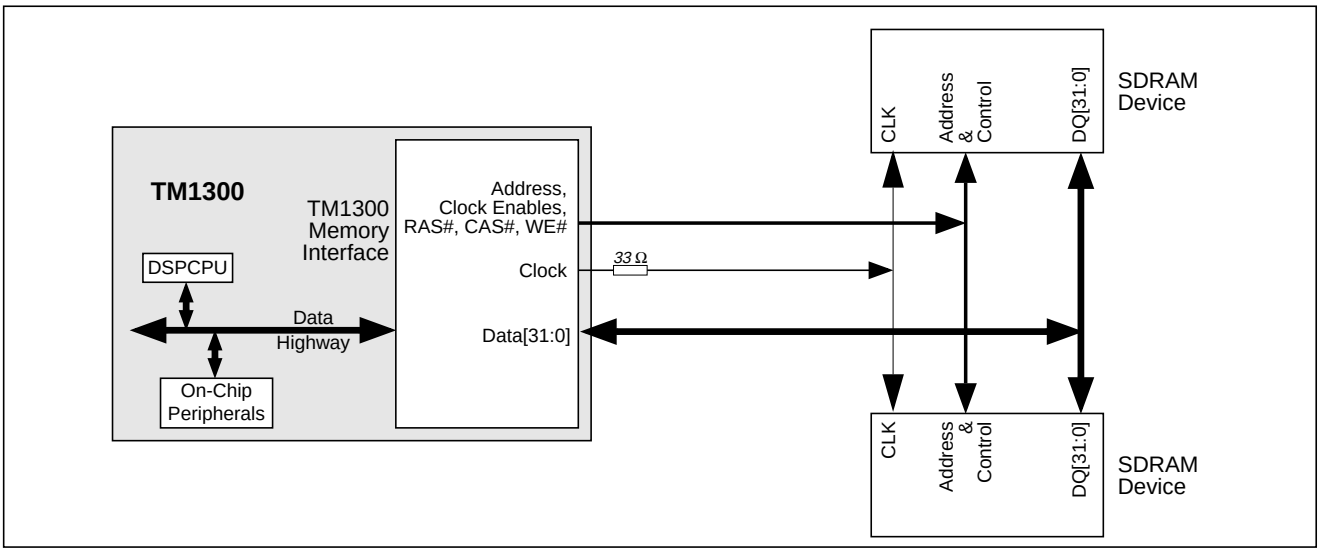


Figure 12-4. Conceptual board layout.

Table 12-11. Glueless interface limits for address/ clocks

Memory Chips	Maximum Clock Frequency
4	143 MHz
6	133 MHz
8	133 MHz

capacitance. Close proximity is especially important for a 143-MHz memory system.

- Signal traces between TM1300 and the memory chips should be matched in length as closely as possible to minimize signal skew.
- The clock-signal trace(s) should be as short as possible.
- Address and control-signal traces should also be short, but their length is less critical than the clock's.
- Data-signal traces should also be short, but their length is less critical than the clock's, especially if only one or two ranks are connected.
- Connections to several loads should follow a "T" connection scheme in order to limit the reflections.

12.15.2 Specific Guidelines

- The maximum length for a signal trace is 10 cm. For 143-MHz operation, signal trace length should not be longer than 7 cm.
- The maximum capacitive load is 30 pF per trace, including loads.
- The signal traces on the TM1300 circuit board must be designed as 50-ohm transmission lines.
- At most two SDRAM devices may be connected to each MM_CLK signal at 143 MHz.

12.15.3 Termination

No termination is required for address, data, and control signals. Address and control signals are driven only by TM1300; the output impedance of the drivers is sufficiently matched to prevent excessive ringing. TM1300 design assumes that when driving data lines, the output drivers of SDRAM chips are also sufficiently impedance matched.

Series termination of the clock outputs with a 33-ohm resistor is advised.

12.16 TIMING BUDGET

The glueless interface of the TM1300 main-memory interface makes the memory system simple and straightforward from one point of view, but to ensure reliable operation at high clock rates, system designers must follow the board design guidelines (see Section 12.15, "Circuit Board Design").

SDRAM devices must meet the critical specifications listed in Table 12-12 to ensure reliable operation of an 143-MHz (T_{cycle} = 7 ns) memory system.

Table 12-12. Required SDRAM performance for 143-MHz memory system

Timing Parameter		Value
Max. output delay	t_{AC}	6.0 ns
Min. output hold time	t_{OH}	2.0 ns
Max. input setup time	t_{IS}	2.0 ns
Max. input hold time	t_{IH}	1.0 ns

These values leave virtually no margin for the critical timing parameters in a high-speed system and assume a total worst case delay of 0.5 ns for the board traces (T_{board})

and a T_{SU} for TM1300 less or equal than 0.4 ns. In other words the following equation needs to be met:

$$T_{cycle} \geq t_{AC} + T_{board} + T_{CS} + T_{SU} \quad .$$

Where T_{CS} is the skew between MM_CLK0 and MM_CLK1, and T_{SU} the input data setup time as defined in Section 1.9.4.7 on page 1-15.

12.16.1 Main AC Parameter requirements

The TM1300 SDRAM interface was designed to support a wide range of SDRAM vendors. Table 12-13 describes some of the minimum SDRAM AC requirements for TM1300 to operate correctly. The symbols or names are not really standardized and may differ from one vendor to another one. The table is not meant to be exhaustive and shows only the main parameters. Parameters are expressed in clock cycles rather than ns.

Table 12-13. Minimum AC Parameters

Description	Symbol	Clocks
ACTIVE command period	t_{RC}	10
ACTIVE to PRECHARGE command	t_{RAS}	7
PRECHARGE command period	t_{RP}	3

Table 12-13. Minimum AC Parameters

Description	Symbol	Clocks
ACTIVE Bank A to ACTIVE bank B	t_{RRD}	3
ACTIVE to READ or WRITE command	t_{RCD}	3
WRITE recovery time	t_{WR}	2

12.17 EXAMPLE BLOCK DIAGRAMS

Figures 112-5 through 12-10 illustrate some of the possible memory configurations that can be built with TM1300. For all the following schemes MM_A[12:11] when used as bank addresses, are interchangeable (i.e. it does not matter whether one is connected to Bank 1 or Bank 0.).

12.17.1 16-Mbit Devices or Less

These devices allow small memory configurations to be built. They are described in the TM1000 and TM1100 databooks. Figure 12-5 shows a 4-MB memory system.

12.17.2 64-Mbit Devices

64-Mbit SDRAMs organized in x16 can be used to build a 16-, 32-, 48- or 64-MB memory systems. Figure 12-6

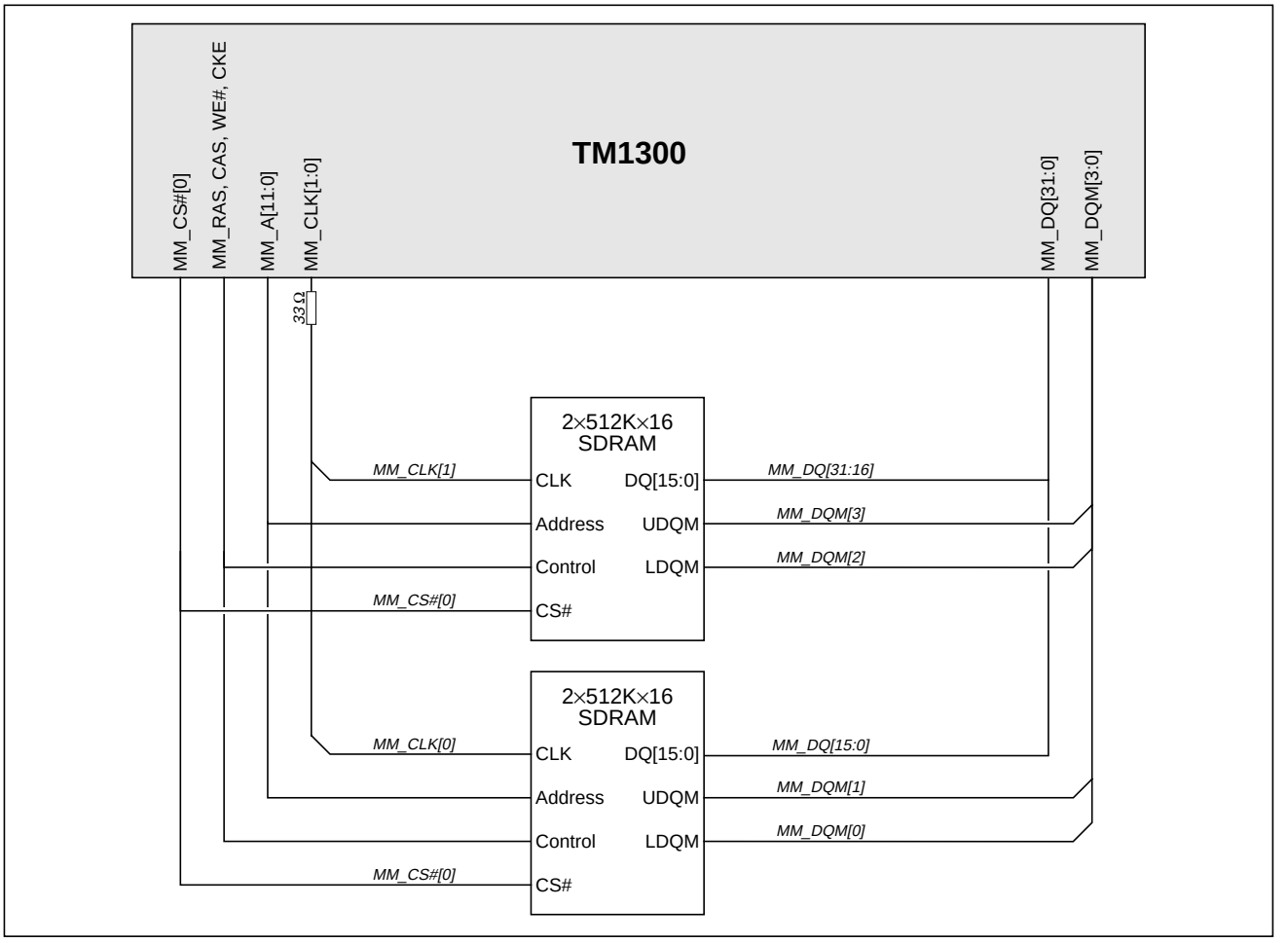


Figure 12-5. Schematic of a 4-MB memory system consisting of two 2×512K×16 SDRAM chips (one rank).

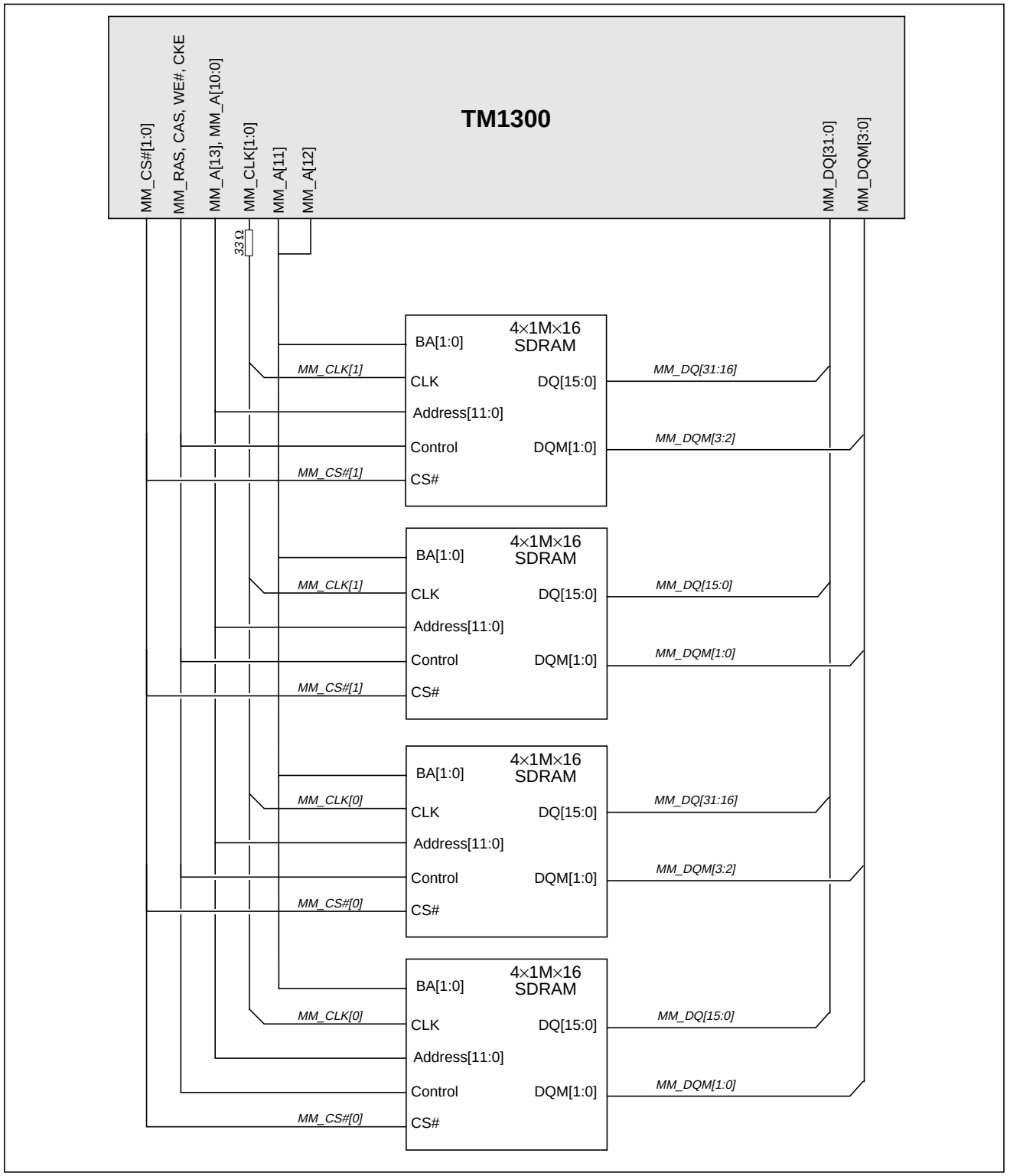


Figure 12-6. Schematic of a 32-MB memory system consisting of four 4×1M×16 SDRAM chips (two ranks)

details a 32-MB memory system. Removing the device controlled by MM_CS#[1] makes a 16-MB system. 64-Mbit SDRAMS organized in x32 can be used to build an 8-, 16-, 24-, or 32-MB memory system. Figure 12-7

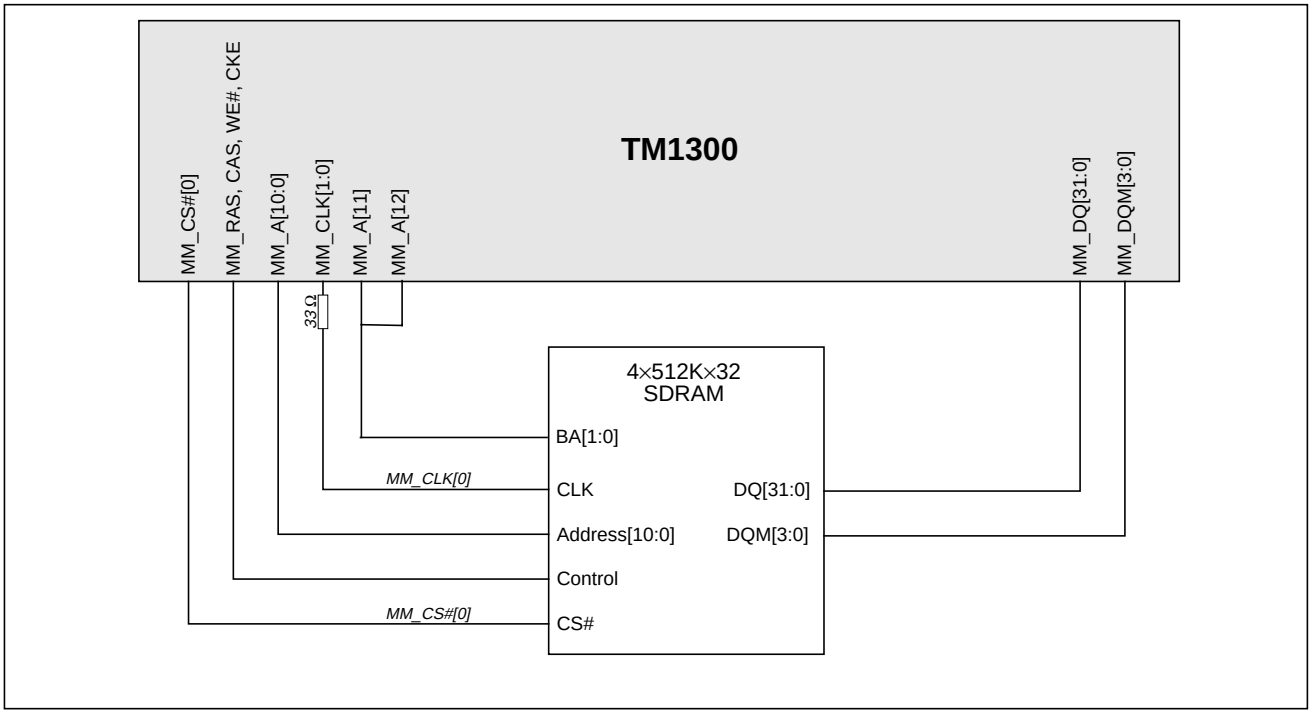


Figure 12-7. Schematic of a 8-MB memory system consisting of one 4×512K×32 SDRAM (one rank).

shows an 8-MB memory system (one device only) and Figure 12-8 a 16-MB configuration.

SDRAMs organized in x16 and x32 could be mixed in order to create, for example, a 24 (16 + 8) MB memory system.

Finally x8 devices could be used to build a 32-MB memory system as illustrated in Figure 12-9. Note that due to

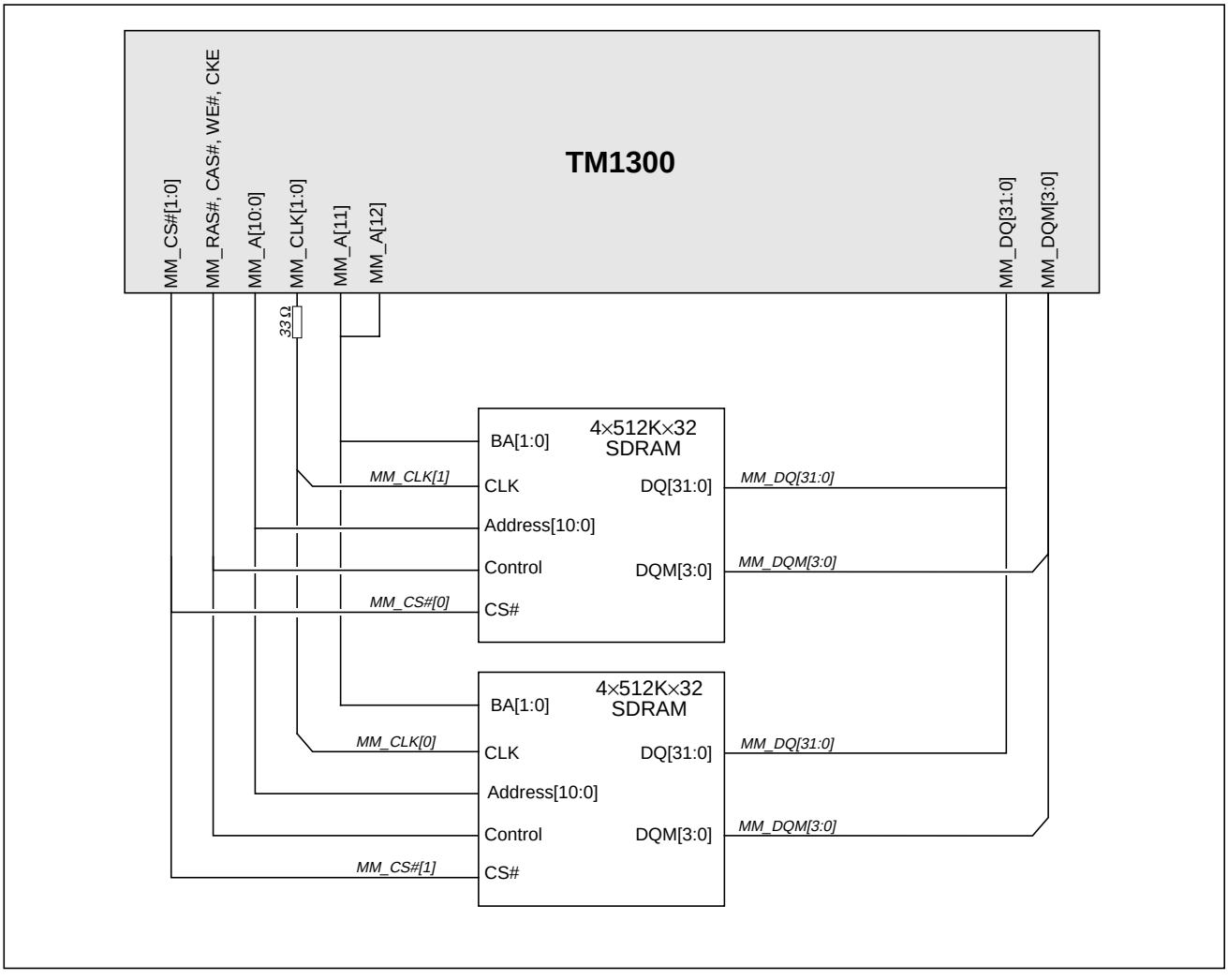


Figure 12-8. Schematic of a 16-MB memory system consisting of two ranks of 4×512K×32 SDRAM chips.

the unusual way of using the devices, it is the only supported configuration with x8 devices. **MM_CONFIG.SIZE** must be set to 6 (i.e. 16-MB rank size, Section 12.6.1).

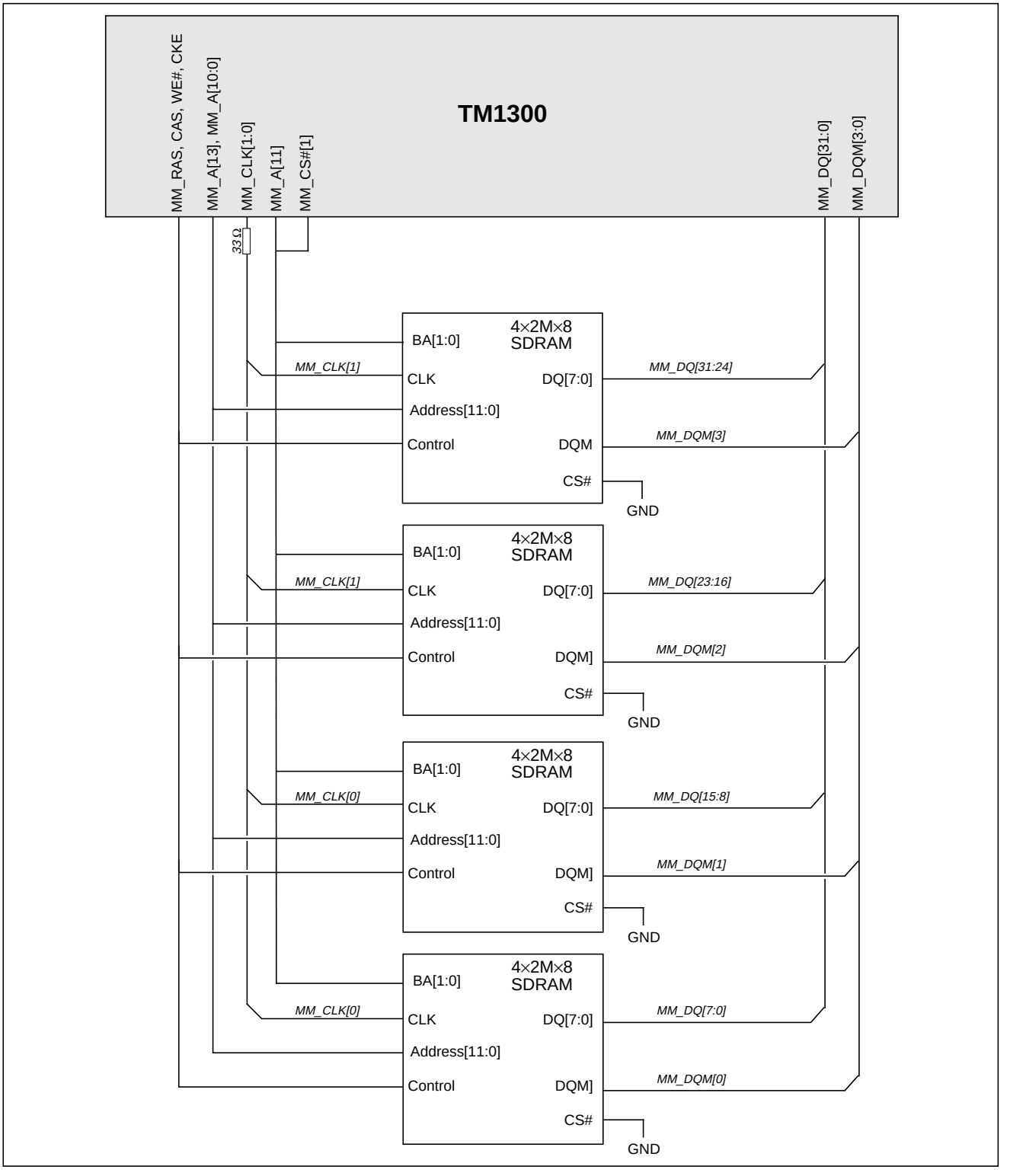


Figure 12-9. Schematic of a 32-MB memory system consisting of four 4×2M×8 SDRAM chips (one rank)

12.17.3 128-Mbit Devices

128-Mbit SDRAMs are partially supported in x16, enabling a 32-MB memory system to be built (cannot be extended using the other MM_CS# pins). Refer to

Figure 12-10 for a more detailed connection scheme. **MM_CONFIG.SIZE** must be set to 6 (i.e. 16 MB rank size, Section 12.6.1). Note that the connections described for the 128-Mbit SDRAMs organized in x16 can also be used to connect 64-Mbit SDRAM devices organized in x16 allowing the same footprint on the board for two different memory size configurations (i.e. 32 MB and 16 MB). 128-Mbit SDRAMs organized in x32 are also supported as pictured in Figure 12-11.

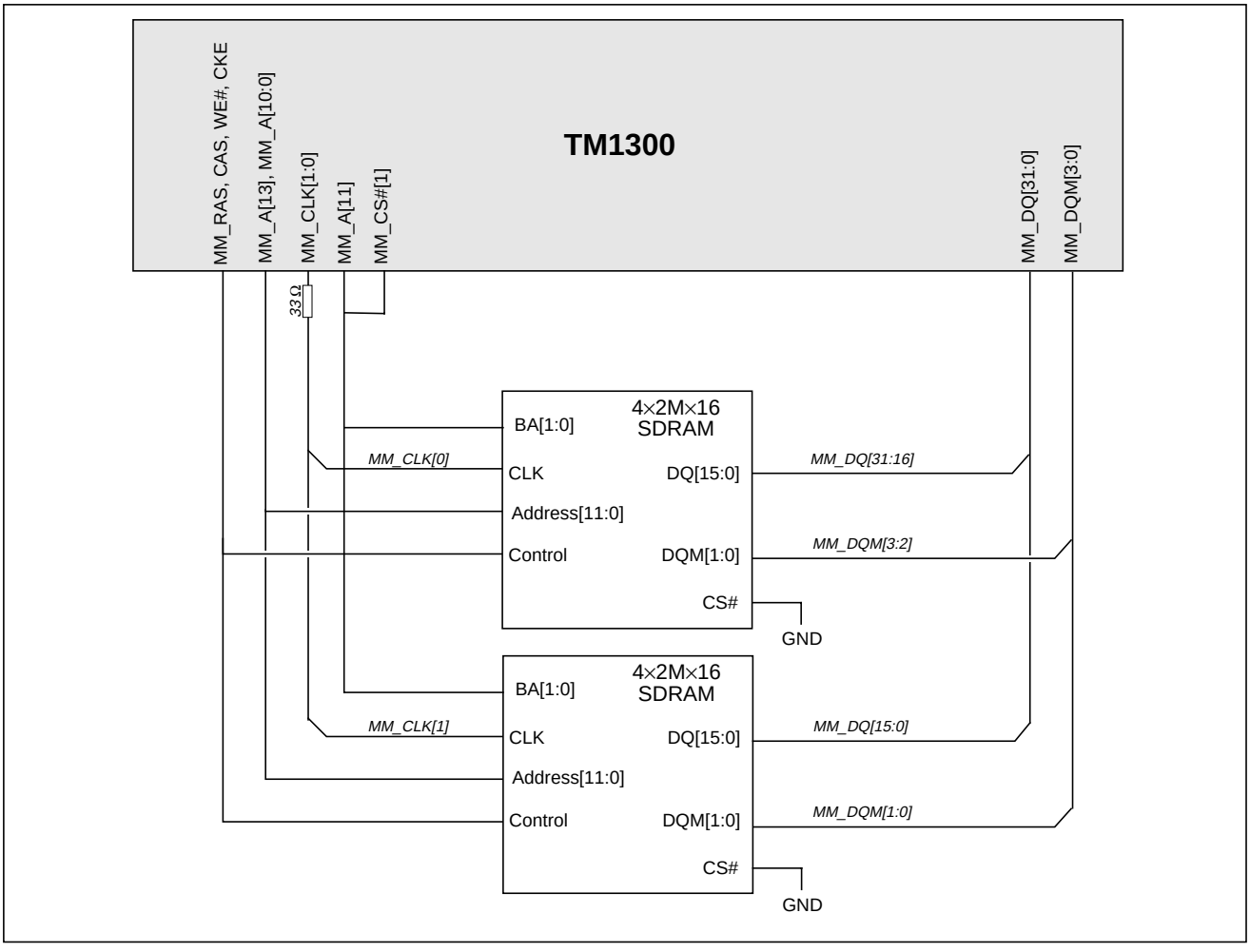


Figure 12-10. Schematic of a 32-MB memory system consisting of two 4×2M×16 SDRAM chips (one rank)

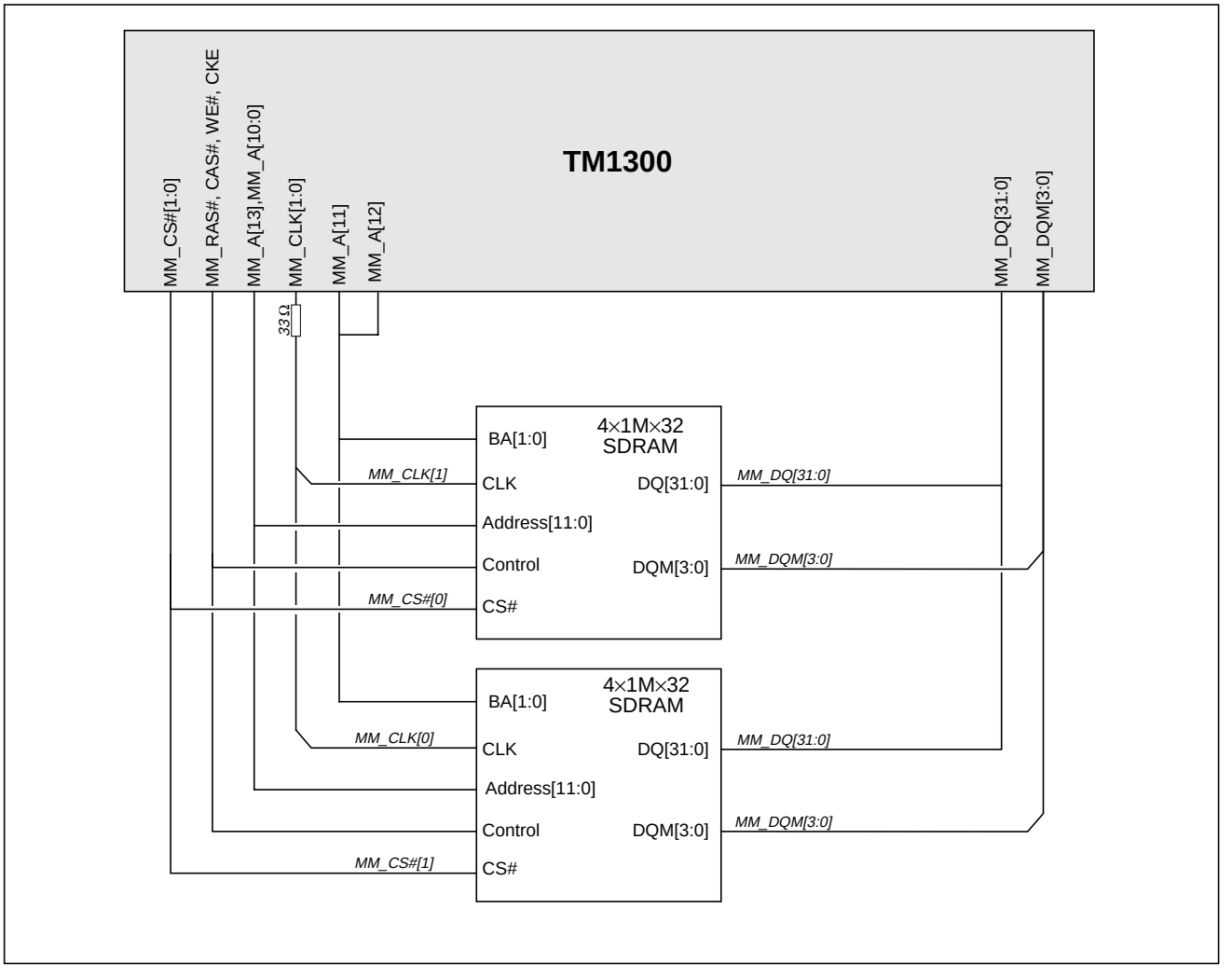


Figure 12-11. Schematic of a 32-MB memory system consisting of two ranks of 4×1M×32 SDRAM chips.

System Boot Chapter 13

by Gert Slavenburg, Bob Bradfield, and Hani Salloum

13.1 NEW IN TM1300

A new bit in the boot EEPROM allows an internal PCI_CLK clock source for low-cost standalone systems

13.2 TM1300 BOOT SEQUENCE OVERVIEW

Before a TM1300 system can begin operating, the main-memory interface (MMI) registers and on-chip clock ratio register must be configured. Since the DSPCPU cannot begin operating until after these registers and circuits are initialized, the DSPCPU cannot be relied on to initialize these resources. Consequently, TM1300 needs an independent bootstrap facility for low-level initialization.

TM1300 implements low-level system initialization by combining a small block of on-chip system boot logic with a single external serial boot EEPROM connected to the I^2C interface. See Figure 13-1. Serial EEPROMs with an I^2C interface are slow but have the advantages of being space-efficient and inexpensive. The amount of information needed for initial system boot is small, so speed is not a concern.

The TM1300 system boot block performs differently for each of two major types of TM1300 system, distinguished by host-assisted and autonomous bootstrapping. The most significant bit of the tenth byte in the external EEPROM determines the system boot procedure and must match the system configuration.

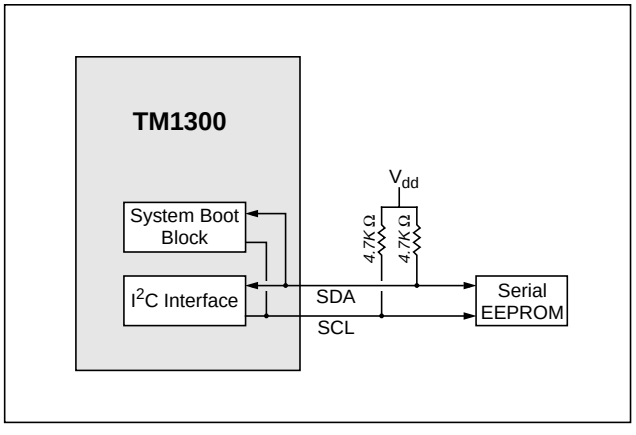


Figure 13-1. The system boot logic uses the I2C interface to access a serial EEPROM that contains main-memory and system timing information.

In host-assisted bootstrapping, a TM1300 device is integrated into a system where some other processor serves as the host. For example, a TM1300 chip might be part of a PCI card in a standard personal computer (PC). In this case, the TM1300 system boot only needs to load enough information from the serial EEPROM to configure the on-chip timing circuits and MMI; the host processor can perform all other TM1300 setup chores.

Table 13-1. System Boot Features

Characteristic	Comments
Boot Configurations Supported	• Host assisted, e.g., TM1300 is a PCI slave in a standard PC. • Autonomous, e.g., TM1300 is the host PCI processor.
ROM Device Types Supported	• Single standard I^2C serial EEPROMs from 128 bytes to 2KB in size. • EEPROMs connect via the TM1300 built-in 2-wire I^2C interface. • The use of EEPROMs with hardware Write Protect (WP) is recommended. A jumper on WP allows user control over in-system reprogramming using the I^2C interface. • The EEPROM must respond to I^2C device address 1010.
ROM device examples	• Atmel 24C01A (128 bytes, WP) • Atmel 24C08 (1KB, WP) • Atmel 24C16 (2KB, WP).
ROM size	• From 128 bytes to 2 KB (one device) for initial program load.

In the second type of system, autonomous bootstrapping takes place. In this configuration, a TM1300 device serves as the host (main) processor; consequently, the TM1300 system boot must perform more work. In addition to configuring on-chip timing and the MMI, the system boot must set the base addresses of the main memory and MMIO address apertures and load into main memory a level 1 bootstrap program for the DSPCPU.

Only the first 10 bytes of the serial EEPROM are needed when TM1300 is not the host PCI processor; thus, such systems can use a very low-cost 128-byte EEPROM device. When TM1300 serves as the system's host processor, the boot logic permits almost 2 KB of storage for the level 1 bootstrap DSPCPU program in a single eight-pin EEPROM device.

13.3 BOOT HARDWARE OPERATION

The TM1300 boot sequence begins with the assertion of the reset signal TRI_RESET#. After reset is de-asserted, only the system boot block, I^2C, and PCI interfaces are allowed to operate. In particular, the DSPCPU and the internal data highway bus will remain in the reset state until they are explicitly released during the boot procedure. In autonomous boot, the system boot block is responsible for releasing the DSPCPU and highway from reset. In host-assisted boot, the boot logic releases the highway from reset and the TM1300 software driver (which runs on the host processor) releases the DSPCPU from reset.

The system boot block operation is illustrated in a flow chart shown in Figure 13-2.

13.3.1 Boot Procedure Common to Both Autonomous and Host-Assisted Bootstrap

There should be no other I^2C master active from reset until boot EEPROM load completes. The system boot procedure begins by loading a few critical pieces of information from the serial EEPROM. This part of the procedure is common to both autonomous and host-assisted bootstrapping. See Table 13-2 for a summary and Table 13-5 for full bit-accurate EEPROM layout details.

The first byte of the EEPROM is read using a serial clock equal to BOOT_CLK/1000, which is guaranteed to be less than 100 kHz. After reading the first byte, which contains the actual BOOT_CLK rate as well as the EEPROM speed capability, the boot block proceeds to read subsequent bytes at the highest valid speed.

The number of lines in the EEPROM device should be ‘0’ in case of a 128-byte device and ‘1’ for larger devices.

The SDRAM aperture size should be set to the smallest size that is larger than or equal to the actual size of SDRAM connected to TM1300. The SDRAM aperture size information is forwarded to the PCI interface for use in host BIOS configuration, as described in Section 13.4.2, “Stage 2: Host-System PCI Configuration.”

The BOOT_CLK speed bits should be set to match the closest rounded up frequency of the external clock circuit, i.e. for an external clock of 40 MHz or 50 MHz the value should be 10. This field, together with the EEPROM maximum clock speed bit are used to decide the best possible divider ratio for generation of the I^2C clock, as shown in Table 13-3. In addition, the delay actions in Figure 13-2 are taken based on the specified BOOT_CLK value.

The EEPROM maximum clock speed bit is set to match the speed grade of the serial EEPROM device.

The test mode bit should always be set to ‘0’. It is only set to one for factory ATE testing.

The Subsystem ID and Subsystem Vendor ID data has no meaning to the TM1300 hardware; its meaning is entirely software defined. The value is loaded by the system boot block from the EEPROM and published in the PCI configuration space register at offset 0x2C to provide the 16-bit Subsystem ID and Subsystem Vendor ID values. These values are used by driver software to distinguish the board vendor and product revision information for multiple board products based on the TM1300 chip. Refer to Section 11.6.12, “Subsystem ID, Sub-

Table 13-2. Information Loaded During First Part of Bootstrapping Procedure

Information	Size	Interpretation	
Number of lines in EEPROM device	1 bit	0	128 lines
		1	256 or more lines
SDRAM aperture size	3 bits	000	1 MB
		001	1 MB
		010	2 MB
		011	4 MB
		100	8 MB
		101	16 MB
		110	32 MB
		111	64 MB
BOOT_CLK speed	2 bits	00	100 MHz
		01	75 MHz
		10	50 MHz
		11	33 MHz
I^2C clock speed	1 bit	0	100 KHz
		1	400 KHz
Test mode	1 bit	0	normal operation
		1	rapid ATE testing
Subsystem ID	16 bits	Value is copied to Subsystem ID register in PCI configuration space.	
Subsystem Vendor ID	16 bits	Value is copied to Subsystem Vendor ID register in PCI config space.	
MM_CONFIG register initialization	20 bits	Value is simply written to the MM_CONFIG register; see Section 12.6.1, “MM_CONFIG Register.”	
PLL_RATIOS register initialization	8 bits	Value is simply written to the PLL_RATIOS register; see Section 12.6.2, “PLL_RATIOS Register.”	
Autonomous/host-assisted boot	1 bit	0	host-assisted
		1	autonomous
Enable internal PCI_CLK	1 bit	0	PCI_CLK taken from outside
		1	use on-chip XIO PCI_CLK clock generator Note: MUST be set if no external PCI clock is supplied
SDRAM prefetchable	1 bit	0	not prefetchable
		1	prefetchable

system Vendor ID Register," for more information on the choice of values.

Table 13-3I²C speed as a function of EEPROM byte 0

BOOT_CLK bits	EEPROM speed bit	divider value	actual I2C speed
00 (100 MHz)	0 (100 KHz)	1008	99.2 KHz
00	1 (400 KHz)	256	390.6 KHz
01 (75 MHz)	0 (100 KHz)	752	99.7 KHz
01	1 (400 KHz)	192	390.6 KHz
10 (50 MHz)	0 (100 KHz)	512	97.6 KHz
10	1 (400 KHz)	128	390.6 KHz
11 (33 MHz)	0 (100 KHz)	336	98.2 KHz
11	1 (400 KHz)	96	343.8 KHz

The MM_CONFIG and PLL_RATIOS registers control the hardware of the MMI and TM1300 on-chip clock circuits. These registers are described in detail in Section 12.6, "Memory System Programming." The boot value should be set to reflect the exact capabilities of the actual SDRAM in the system.

The 'enable internal PCI_CLK generator' bit determines the PCI_CLK pin operating mode. If this bit is '0', PCI_CLK acts compatible with TM1000 and normal PCI operation, i.e. it is an input pin that takes PCI clock from the external world. If this bit is '1', an on-chip clock divider in the XIO logic becomes the source of PCI_CLK, and the PCI_CLK pin is configured as an output. In the latter case, the PCI_CLK frequency can be programmed to a divider of the TM1300 highway clock by setting the XIO_CTL register 'Clock Frequency' divider value. Refer to Chapter 22, "PCI-XIO External I/O Bus." Note: This bit must be set if no external PCI clock is supplied.

The 'SDRAM prefetchable' bit is copied to the PCI configuration space register DRAM_BASE and only visible as bit #3 (P bit) of DRAM_BASE in a PCI configuration read, but not visible by MMIO access. Its purpose is to tell the PCI host, that SDRAM reads will cause no side effects. The host may apply optimizations on PCI access, if this bit is set.

The 'autonomous/host-assisted boot' bit determines whether the system boot logic will continue reading more information from the EEPROM or halt its operation so the host can complete system initialization. After the information listed in Table 13-2 has been loaded into TM1300 registers, an external PCI host processor can finish the initialization of TM1300. If no external PCI host processor is present, the autonomous/host-assisted boot bit should be set to '1' to allow the system boot logic to load the information described in the next section.

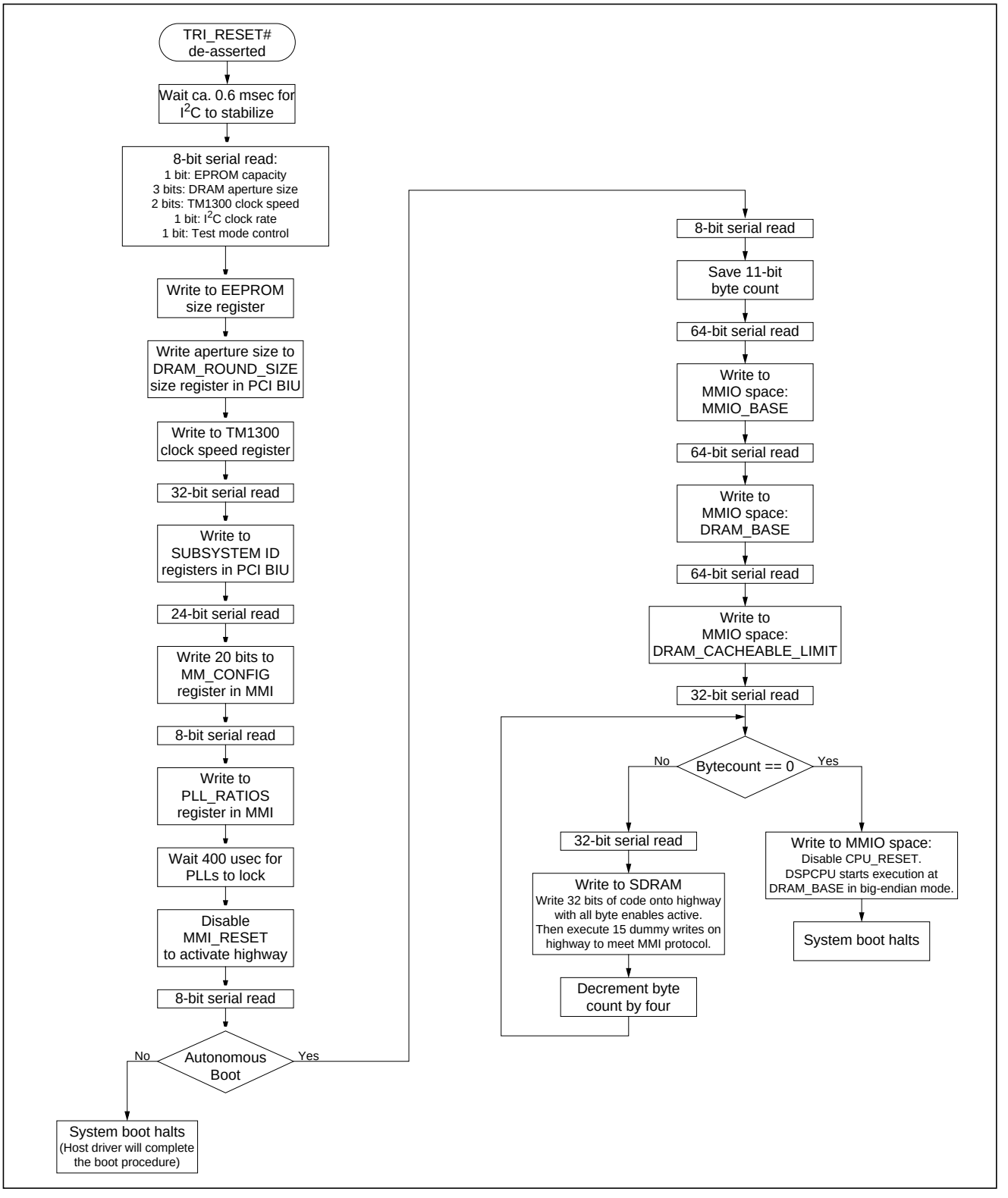


Figure 13-2. Flow chart of system boot procedure for both host-assisted and autonomous configurations.

13.3.2 Initial DSPCPU Program Load for Autonomous Bootstrap

In a system where TM1300 serves as the host CPU, the system boot block performs an autonomous boot procedure. For an autonomous boot, the system boot block reads all the information described in Section 13.3.1, "Boot Procedure Common to Both Autonomous and Host-Assisted Bootstrap," and then—because the autonomous boot bit is set—continues reading information from the EEPROM. After this part of the system boot procedure is done, the DSPCPU starts executing. See Table 13-4.

The DSPCPU bootstrap program byte count encodes the number of bytes of DSPCPU program code contained in the EEPROM(s). This 11-bit unsigned byte count can encode up to 2048 bytes, which is also the maximum amount of EEPROM storage supported. The actual amount of EEPROM available for the DSPCPU bootstrap program is limited to 2000 bytes. Other information consumes 47 bytes, and the DSPCPU code must be an integral number of 32-bit words.

Four pairs of 32-bit MMIO-register addresses and values follow the bootstrap program byte count. Each address tells the boot block where in the 32-bit DSPCPU address space to store the corresponding 32-bit value.

The first pair initializes the MMIO_BASE. The MMIO_BASE sets the base address of the 2-MB MMIO-register address aperture within the DSPCPU 32-bit address space. All MMIO registers are addressed using an offset that is relative to the value of MMIO_BASE. For this pair, the address is required to be 0xEFF00400 because that is the default MMIO_BASE enforced when TM1300 is reset. The new value for MMIO_BASE is encoded in the corresponding value.

The DRAM_BASE address/value pair determine the base address of the SDRAM address aperture within the 32-bit DSPCPU address space. The address must be equal to 0x100000 plus the new value of MMIO_BASE set previously in the boot procedure. The DRAM_BASE value must be naturally aligned given the rounded DRAM aperture size, i.e. a 6 MB DRAM aperture should start on a 8 MB address multiple.

The DRAM_LIMIT address/value pair determine the extent of the SDRAM address aperture. The address must be equal to 0x100004 plus the new value of MMIO_BASE set previously in the boot procedure. The value in DRAM_LIMIT should be 1 higher than the address of the last valid byte of SDRAM memory, and must be a 64 KB multiple.

The DRAM_CACHEABLE_LIMIT address/value pair determine the extent of the cacheable aperture of the SDRAM address space. The address must be equal to 0x100008 plus the value of MMIO_BASE set previously in the boot procedure. The cacheable aperture always begins at the address value in DRAM_BASE; the value in DRAM_CACHEABLE_LIMIT is one higher than the address of the last byte of cacheable SDRAM memory, and must be a 64 KB multiple. It is safe to initially set the value of DRAM_CACHEABLE_LIMIT equal to DRAM_LIMIT. The RTOS can, if desired, change the value later.

Table 13-4. Information Loaded During Second Part of Bootstrapping Procedure for Autonomous Boot

Information	Size	Interpretation
DSPCPU bootstrap program byte count *n*	11 bits	up to 500 32-bit words (2048 bytes less 47 header bytes)
MMIO_BASE address	32 bits	Value must be 0xEFF00400
MMIO_BASE value	32 bits	Value is simply written to 0xEFF00400 to determine new base address of 2-MB MMIO register aperture within 32-bit DSPCPU address space
DRAM_BASE address	32 bits	MMIO_BASE + 0x100000
DRAM_BASE value	32-bits	Value is simply written to DRAM_BASE to determine base address of SDRAM aperture within 32-bit DSPCPU address space
DRAM_LIMIT address	32-bits	MMIO_BASE + 0x100004
DRAM_LIMIT value	32-bits	Value is simply written to DRAM_LIMIT to determine limit address of SDRAM aperture within 32-bit DSPCPU address space
DRAM_CACHEABLE_LIMIT address	32-bits	MMIO_BASE + 0x100008
DRAM_CACHEABLE_LIMIT value	32-bits	Value is simply written to DRAM_CACHEABLE_LIMIT to determine limit address of cacheable part of SDRAM aperture within 32-bit DSPCPU address space
DRAM_BASE value	32-bits	Copy of the DRAM_BASE; must be equal to value specified above
SDRAM code word 0	32-bits	First 32-bit word of initial DSPCPU bootstrap program
SDRAM code word 1	32-bits	Second 32-bit word of initial DSPCPU bootstrap program
. . .	. . .	. . .
SDRAM code word *n*/4	32 bits	Last 32-bit word of initial DSPCPU bootstrap program

The next 32-bit value in boot EEPROM memory is a copy of the DRAM_BASE value encoded previously. The system boot hardware loads the DSPCPU bootstrap program into SDRAM starting at DRAM_BASE.

The bytes of the DSPCPU bootstrap program follow the copy of the SDRAM_BASE value. The bootstrap pro-

gram can consist of up to 500 32-bit words of DSPCPU instructions. The byte count must be a multiple of four. Note that the bytes are stored in the EEPROM in a byte swapped order per group of 4 compared to SDRAM, as detailed in Table 13-5.

After the entire DSPCPU bootstrap program is loaded into SDRAM at DRAM_BASE, the system boot logic releases the DSPCPU from the reset state. At this point, the DSPCPU begins executing the bootstrap program starting at DRAM_BASE and TM1300 is fully operational. At the same time, the boot logic releases the I^2C interface.

13.4 HOST-ASSISTED BOOT DESCRIPTION

For a host-assisted bootstrap, the complete bootstrap process consists of three distinct stages, but the system boot hardware performs only the first stage. The other two stages are the responsibility of the host system.

13.4.1 Stage 1: TM1300 System Boot Hardware

In the first stage, the TM1300 hardware must be initialized enough to allow the host system to query and manipulate TM1300 resources. The system boot hardware, using the procedure described above in Section 13.3.1, "Boot Procedure Common to Both Autonomous and Host-Assisted Bootstrap," initializes the Subsystem ID, Subsystem Vendor ID, MM_CONFIG, and PLL_RATIOS registers, waits for the PLLs to lock, enables the internal highway and MMI, but leaves the DSPCPU in the reset state. After this minimal initialization, the host system can finish the bootstrap process.

At the completion of stage 1, the TM1300 hardware is ready to respond to PCI configuration space accesses, and the boot block has released the I^2C interface.

13.4.2 Stage 2: Host-System PCI Configuration

Stage 2 is carried out either by the host-system PCI BIOS or by a combination of the BIOS and the host operating system (e.g., Windows 95). During this stage, the host system configures all PCI-bus clients.

The PCI-bus configuration consists of querying the bus clients to determine the following:

- The number of PCI base-address registers implemented by each client. For TM1300, the number of PCI base-address registers is always two (MMIO_BASE and DRAM_BASE).
- The size of each aperture associated with the base-address registers. For TM1300, the size of the MMIO aperture is always 2 MB. The size of the SDRAM aperture can range from 1 MB to 64 MB, and the size must be a power of two (seven distinct sizes).

Using this information, the host system relocates each address aperture to eliminate overlaps in the PCI address space. The host system accomplishes the relocation by considering each aperture's size and then writing an appropriate starting address to each base-address register. For TM1300, the base addresses of the MMIO and SDRAM apertures must be relocated in this way. Note that in the case of autonomous boot, this relocation is done statically by the system boot hardware when it simply copies the values of MMIO_BASE and DRAM_BASE from the serial EEPROM into these registers.

The steps of the PCI protocol for determining the size of an address aperture are as follows (see Section 11.6.11, "Base Address Registers," for a more complete discussion):

- The host writes a 32-bit word of all '1's (0xffffffff) to the base-address register.
- The host reads the base-address register immediately after the write. The value returned will have '0's in all don't-care bits and '1's in all required address bits. The required address bits form a left-aligned (i.e., starting at the most-significant bit) contiguous field of '1's.
- This left-aligned field of '1's effectively specifies the size of the address aperture by indicating the bits of the base-address register that are significant for relocation. That is, an address aperture of size 2^n can only begin on a 2^n-byte-aligned boundary.

As an example, consider the case of the MMIO aperture. The host will perform the following steps during stage 2 of the bootstrap process:

- Write 0xffffffff to MMIO_BASE.
- Read from MMIO_BASE, which returns the value 0xffe00000. The host sees that this value has an 11-bit left-aligned field of '1's, which indicates that the aperture can only be relocated on 2-MB boundaries; thus, the aperture size is 2 MB.
- Write a new value to MMIO_BASE with the top 11 bits set to relocate the MMIO aperture to a 2-MB region of PCI address space that does not conflict with other PCI address apertures.

At the completion of stage 2, the TM1300 hardware is ready to respond to host configuration space accesses, host MMIO accesses and host SDRAM aperture accesses. The DSPCPU is still in RESET state.

13.4.3 Stage 3: TM1300 Driver Executing on the Host

During the final stage of the bootstrap process, the TM1300 software driver executing on the host system will write to SDRAM a program for the DSPCPU, and initialize any MMIO registers. When the initial program load is complete, the driver releases the DSPCPU from its reset state by a write to the BIU_CTL register with the CR bit set. See Chapter 11, "PCI Interface." Now, with the DSPCPU and host both running, the TM1300 bootstrap process is complete.

13.5 DETAILED EEPROM CONTENTS

Table 13-5 shows the serial EEPROM contents needed for an autonomous boot procedure. For the host-assisted boot procedure, only the contents up to line nine are needed.

Note that the 32-bit words in the serial EEPROM are not stored on 32-bit word-aligned addresses.

Table 13-5. Serial boot EEPROM contents

Line	Data Byte							
	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
0	#lines 0: 128 lines 1: 256 or more lines	SDRAM size[2:0] 000: 1MB 001: 1MB 010: 2MB 011: 4MB 100: 8MB 101: 16MB 110: 32MB 111: 64MB			BOOT_CLK[1:0] 00: 100 MHz 01: 75 MHz 10: 50 MHz 11: 33 MHz		EEPROM clock 0: 100 KHz 1: 400 KHz	Test Mode 0: normal 1: rapid ATE
1 2 3 4	Subsystem ID, 8 msb Subsystem ID, 8 lsb Subsystem Vendor ID, 8 msb Subsystem Vendor ID, 8 lsb							
5	—	—	—	—	MM_CONFIG[19:16]			
6 7	MM_CONFIG[15:8] MM_CONFIG[7:0]							
8	PLL_RATIOS[7:0]							
	sdram PLL bypass	sdram PLL disable	cpu PLL bypass	cpu PLL disable	sdram ratio	cpu ratio[2:0]		
9	boot type 0: host assist. 1: autonomous	enable internal PCI_CLK	SDRAM prefetchable 0:no 1:yes	—	—	byte count [10:8]		
10	byte count [7:0]							
11 12 13 14	MMIO_BASE address [31:24] (must be 0xEF) MMIO_BASE address [23:16] (must be 0xF0) MMIO_BASE address [15:8] (must be 0x04) MMIO_BASE address [15:8] (must be 0x00)							
15 16 17 18	MMIO_BASE value [31:24] MMIO_BASE value [23:16] MMIO_BASE value [15:8] MMIO_BASE value [7:0]							
19 20 21 22	DRAM_BASE address [31:24] (must be byte 3 of MMIO_BASE + 0x100000) DRAM_BASE address [23:16] (must be byte 2 of MMIO_BASE + 0x100000) DRAM_BASE address [15:8] (must be byte 1 of MMIO_BASE + 0x100000) DRAM_BASE address [7:0] (must be byte 0 of MMIO_BASE + 0x100000)							
23 24 25 26	DRAM_BASE value [31:24] DRAM_BASE value [23:16] DRAM_BASE value [15:8] DRAM_BASE value [7:0]							
27 28 29 30	DRAM_LIMIT address [31:24] (must be byte 3 of MMIO_BASE + 0x100004) DRAM_LIMIT address [23:16] (must be byte 2 of MMIO_BASE + 0x100004) DRAM_LIMIT address [15:8] (must be byte 1 of MMIO_BASE + 0x100004) DRAM_LIMIT address [7:0] (must be byte 0 of MMIO_BASE + 0x100004)							
31 32 33 34	DRAM_LIMIT value [31:24] DRAM_LIMIT value [23:16] DRAM_LIMIT value [15:8] DRAM_LIMIT value [7:0]							
35 36 37 38	DRAM_CACHEABLE_LIMIT address [31:24] (must be byte 3 of MMIO_BASE + 0x100008) DRAM_CACHEABLE_LIMIT address [23:16] (must be byte 2 of MMIO_BASE + 0x100008) DRAM_CACHEABLE_LIMIT address [15:8] (must be byte 1 of MMIO_BASE + 0x100008) DRAM_CACHEABLE_LIMIT address [7:0] (must be byte 0 of MMIO_BASE + 0x100008)							

Table 13-5. Serial boot EEPROM contents

Line	Data Byte							
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
39 40 41 42	DRAM_CACHEABLE_LIMIT value [31:24] DRAM_CACHEABLE_LIMIT value [23:16] DRAM_CACHEABLE_LIMIT value [15:8] DRAM_CACHEABLE_LIMIT value [7:0]							
43 44 45 46	repeat of DRAM_BASE value [31:24] repeat of DRAM_BASE value [23:16] repeat of DRAM_BASE value [15:8] repeat of DRAM_BASE value [7:0]							
47 48 49 50	byte 0 of DSPCPU bootstrap program (stored at DRAM_BASE + 3) byte 1 of DSPCPU bootstrap program (stored at DRAM_BASE + 2) byte 2 of DSPCPU bootstrap program (stored at DRAM_BASE + 1) byte 3 of DSPCPU bootstrap program (stored at DRAM_BASE + 0)							
. . .	. . .							
j+47	byte j of DSPCPU bootstrap program (stored at DRAM_BASE + ((j div 4) + (3 – (j mod 4))))							
. . .	. . .							
(n–1) +47	last byte of DSPCPU bootstrap program (bits [7:0] of last 32-bit word, stored at DRAM_BASE + n – 4)							

13.6 EEPROM ACCESS PROTOCOLS

Figure 13-3 shows the SDA (serial data) line protocols for three types of read accesses supported by I^2C serial EEPROMs. A read from the address currently latched inside the EEPROM can be for either a single byte or for an arbitrary series of sequential bytes. The master makes the choice by setting the ACK bit after a byte has been transferred.

A random-access read is accomplished by performing a dummy write, which overwrites the latched address stored inside the EEPROM. Once the internal address latch is set to the desired value, one of the other two read protocols can be used to read one or more bytes.

The boot logic inside TM1300 uses a single random read transaction to location 0 of device address 1010000 followed by a sequential read extension to read all required EEPROM bytes in a single pass.

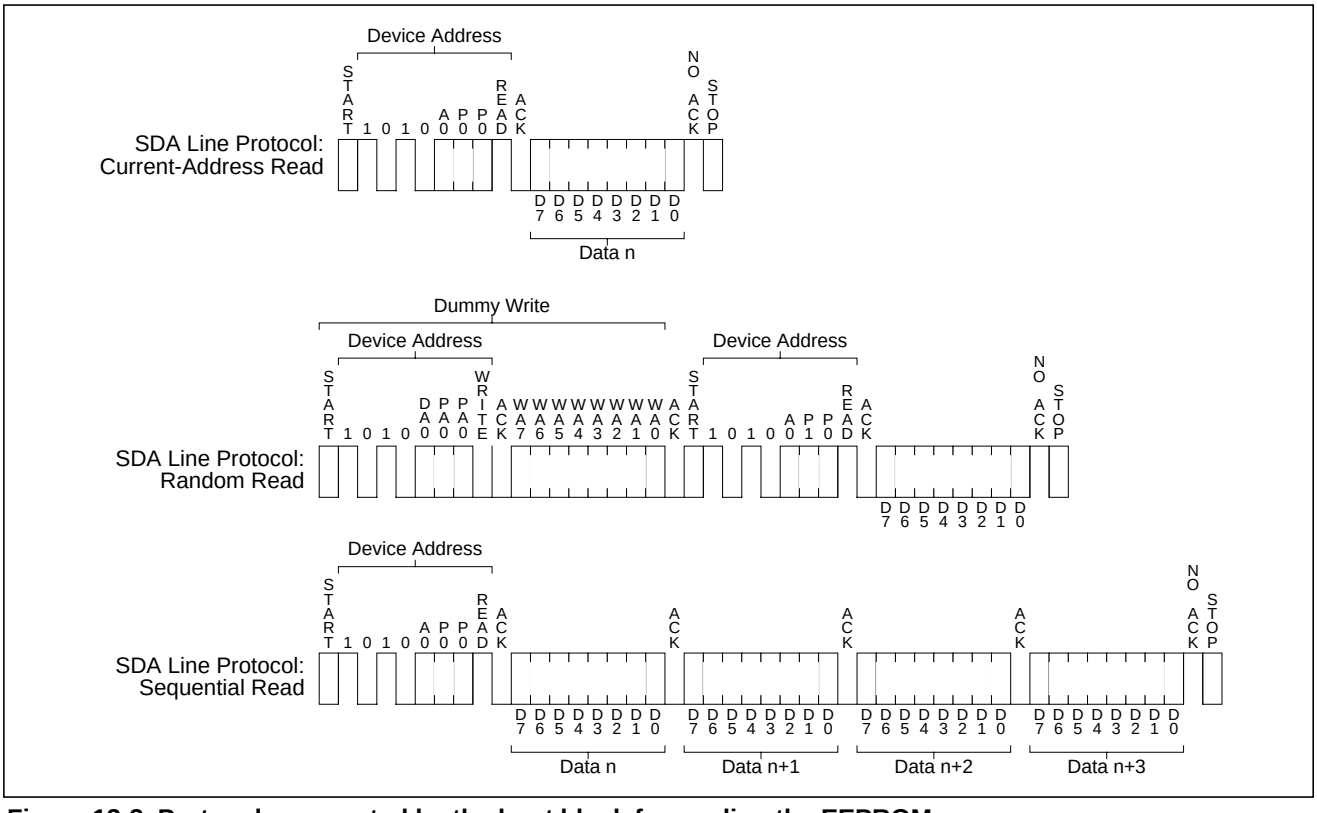


Figure 13-3. Protocols supported by the boot block for reading the EEPROM

14.1 IMAGE COPROCESSOR OVERVIEW

The Image Coprocessor (ICP) connects to the TM1300 on-chip data highway to perform SDRAM block read and write actions. It also connects to the PCI interface to allow block write transactions across PCI.

The major functions of the ICP are:

- Filter an image by reading the image from SDRAM and writing the image back to SDRAM, while applying a user-defined polyphase filter with optional horizontal up- or down-scaling.
- Filter an image by reading the image from SDRAM and writing the image back to SDRAM, while applying a user defined polyphase filter with optional vertical up- or down-scaling.
- Filter an image and convert it from planar to RGB or YUV composite by reading the image from SDRAM and writing the image out to PCI bus memory (graphics card) or SDRAM, while performing horizontal scaling and conversion to one of a several RGB or YUV formats. The programmer can add optional bitmap masking to selectively enable/disable pixel writes to PCI (to refresh only the exposed part of a video window) and an optional image overlay with alpha blending and optional chroma keying (PCI output only).
- Move an image by reading the image from SDRAM and writing it back to SDRAM.

All of the ICP functions move and transform data from memory to memory or memory to the PCI bus. Hence, the DSPCPU can use the ICP in a time-sharing fashion to simultaneously achieve:

1. Vertical and horizontal resizing/subsampling on the image stream from the Video In (VI) unit.
2. Vertical and horizontal resizing/upsampling on the image stream sent to the Video Out (VO) unit.
3. Presentation of a collection of live video windows with programmable up and down scaling and arbitrary overlap configuration on PCI graphics cards.[1]

Full 2D scaling and filtering requires two passes over the data: one for horizontal scaling and filtering and one for vertical scaling and filtering.

1. Note that function 2 and 3 don't normally occur simultaneously, and if an application attempts both simultaneously, some performance limitations are incurred.

Figure 14-1 shows a block diagram of the TM1300 with the ICP. Figure 14-2 shows a block diagram of the internal structure of the ICP. The ICP contains a 5-tap filter, YUV to RGB converter, an overlay and alpha blending unit, and an output formatter. These blocks communicate with each other through FIFOs that also buffer the block data to and from the TM1300 Data Highway. The ICP uses a microprogram-controlled sequencer to control its internal timing. The program for this sequencer is in a table in SDRAM. The ICP reads the appropriate portion from the SDRAM each time the ICP is commanded to perform a function. Microprogram control simplifies and minimizes the ICP hardware and increases the flexibility of the ICP to perform additional tasks without adding hardware.

14.2 REQUIREMENTS

14.2.1 Functions

The major functions of the ICP include:

1. Read an image from SDRAM and write the image back to SDRAM, while applying a user defined polyphase filter with optional up or down scaling in horizontal direction.
2. Read an image from SDRAM and write the image back to SDRAM, while applying a user defined polyphase filter with optional up or down scaling in vertical direction.
3. Read an image from SDRAM and write the image out to PCI bus memory (graphics card) or SDRAM, while performing horizontal scaling and conversion to one of a several RGB and YUV formats. The PCI output mode includes optional bitmap masking to selectively enable/disable pixel writes to PCI (to refresh only the exposed part of a video window) and optional RGB overlay with alpha blending and optional chroma keying.

14.2.2 Bandwidth

ICP bandwidth can be estimated from the worst-case image processing bandwidth. If the worst case image is 1024 x 768 at 30 Hz in YUV 4:2:2 format, the pixel rate is 1024 x 768 x 30 = 23.59 Mpix/sec. For YUV 4:2:2 image coding at 2 bytes per pixel, this is 23.59 x 2 = 47.19 MB/sec. The minimum bandwidth for the ICP function is therefore 47.18 MB/sec., or approximately 50 MB/sec.

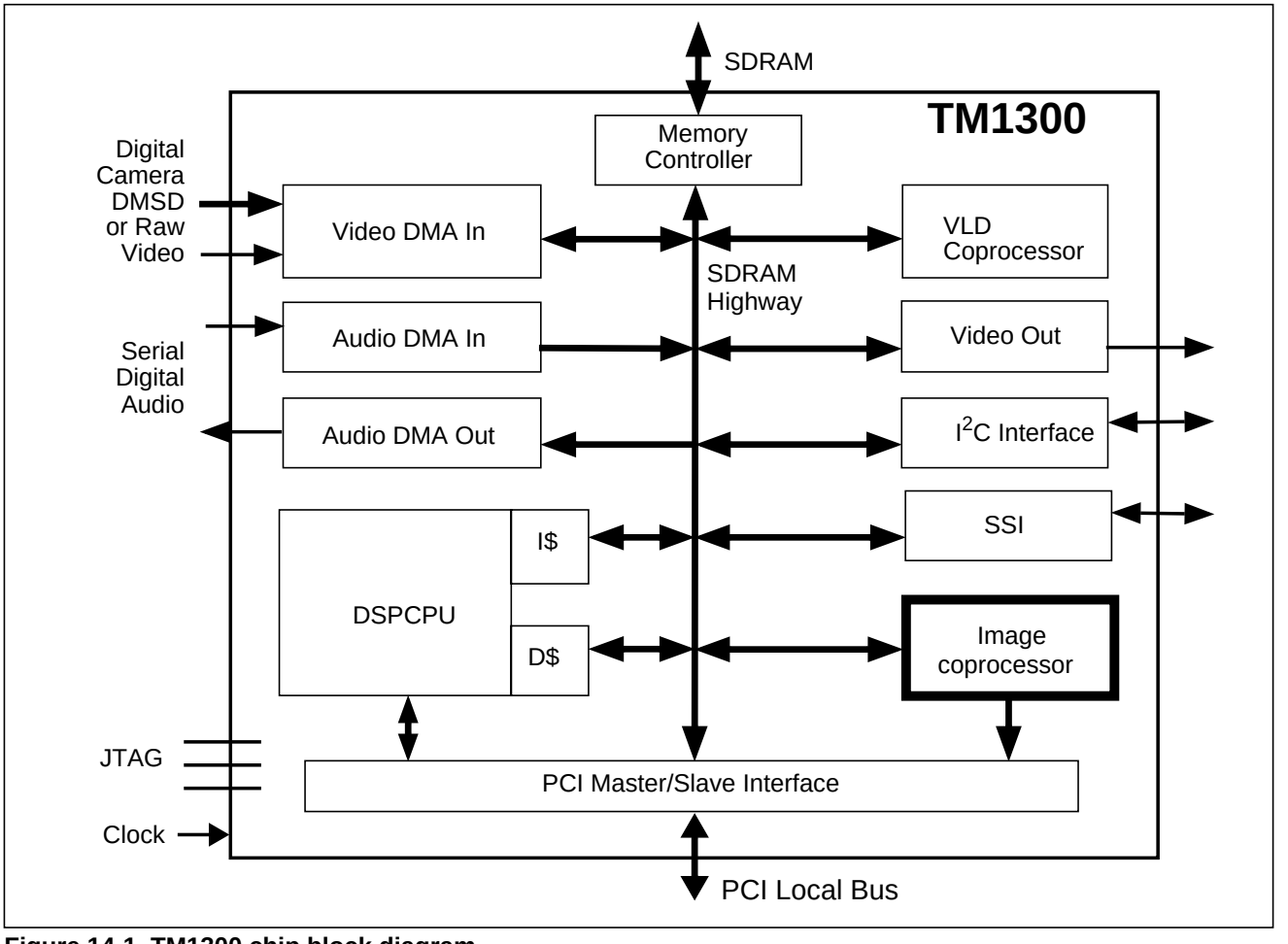


Figure 14-1. TM1300 chip block diagram

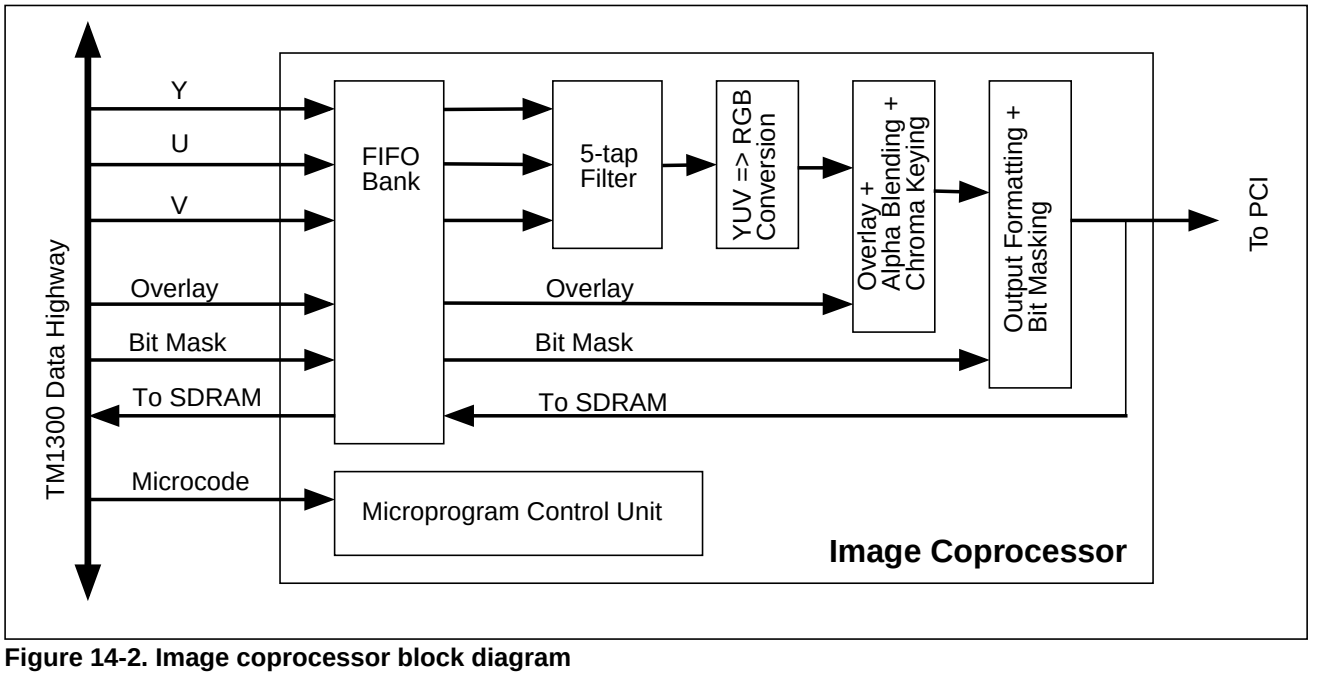


Figure 14-2. Image coprocessor block diagram

Scaling and filtering of the two dimensional image requires two passes of the image data through the filter, one for vertical and one for horizontal. Scaling an image and sending it to the PCI bus requires three transfers of the image over the SDRAM bus: one transfer to read the image for vertical filtering, one transfer to write the filtered data back, and one transfer to read the image for horizontal filtering and output to the PCI bus. This means an average of SDRAM bus bandwidth of 3 x 50 = 150 MB/sec for the 1024 x 768 image case described above, assuming a scaling factor of 1.0. A larger or smaller scaling factor means that either the input or output image will be smaller than 1024 x 768. The bandwidths required are determined by the larger of the two images, input or output. This is because all input pixels must be scanned to generate all the output pixels.

14.2.3 Image Size and Scaling

Image sizes in the TM1300 have a nominal range of 16 x 16 to 1024 x 768. Sizes smaller than 16 x 16 are possible, but are too small to be recognizable images. Images larger than 1024 x 768 (up to 64 K x 64 K) are possible but they cannot be processed in real time and require larger SDRAM sizes. Scaling factors have a nominal range of 1/4 (down scaling by 4) to 4 (upscaling by 4). Larger up and down scaling factors are possible, up to 1000 and beyond; however, very large upscaling factors result in a large magnification of a few pixels, and very large down scaling factors give only a few pixels as a result.

14.3 INTERFACE

The ICP unit has no TM1300 external pins. It interfaces internally to the Data Highway and the PCI Interface.

14.4 DATA FORMATS

The ICP unit accepts input and overlay image data to generate output image data. The ICP accommodates a variety of formats for the input, overlay and output data. These image data formats define the relationship between the Y, U, and V or R, G, and B components of the image as they are stored in memory. The ICP accepts input image data in planar format, where the Y, U and V components are in separate tables in SDRAM. The various input image data formats differ in the position of the U and V components relative to the Y component and the amount of U and V data relative to the Y data.

In all modes except the YUV to RGB conversion modes, each ICP operation processes one Y, U, or V image component. Three separate commands are required to process all three components of an image. Since each component is scaled and filtered separately, the software defines the image format and format conversion by how it scales each component.

For pixel format conversion for PCI or SDRAM output mode, each output pixel is a combination of RGB or YUV components as defined by the output format. The YUV input data and the RGB or YUV overlay data are combined by the ICP hardware pixel by pixel to form the RGB or YUV output pixels. Because all three YUV components are simultaneously woven together to create each output pixel, the ICP hardware must know the image data format in SDRAM, defined as how the components of the image data are to be found and combined.

In the YUV to RGB conversion mode, the ICP accepts the following input data formats: YUV 4:2:2 co-sited, YUV 4:2:2 interspersed, and YUV 4:2:0. In this mode, the ICP will also accept image overlay data when PCI output is specified. The ICP accepts image overlay data in several combined formats: RGB 24+α, RGB 15+α, and YUV 4:2:2+α. In this mode, the ICP generates output data in several RGB and YUV formats. These formats are compatible with a wide variety of PCI frame buffers.

14.4.1 Image Input Formats

The ICP image input formats define the relative positions of the Y component and the U and V components of the input image pixel data. There are three input formats to the ICP: 4:2:2 co-sited, 4:2:2 interspersed, and 4:2:0 interspersed. The 4:2:2 formats have 2 U and 2 V pixels for every 4 Y pixels, so the ratio of Y to U or V is 2:1. The 4:2:0 format has 1 U and 1 V pixel for every 4 Y pixels, so the ratio of Y to U or V is 4:1. The input formats are given below. The input formats have a significant impact on the 2 dimensional scaling operation.

14.4.1.1 YUV 4:2:2 Co-Sited

In the YUV 4:2:2 co-sited format, the U and V pixels coincide with the Y pixel on every other pixel, as shown in Figure 14-3.

14.4.1.2 YUV 4:2:2 Interspersed

In the YUV 4:2:2 interspersed format, the U and V pixels lie between the Y pixels on every other pixel of the horizontal line, as shown in Figure 14-4.

14.4.1.3 YUV 4:2:0 XY Interspersed

In the YUV 4:2:0 interspersed format, the U and V pixels lie between the Y pixels on every other pixel of the horizontal line, as shown in Figure 14-5.

14.4.1.4 YUV 4:1:1 Co-Sited

In the YUV 4:1:1 co-sited format, the U and V pixels coincide with the Y pixel on every fourth pixel, as shown in Figure 14-6.

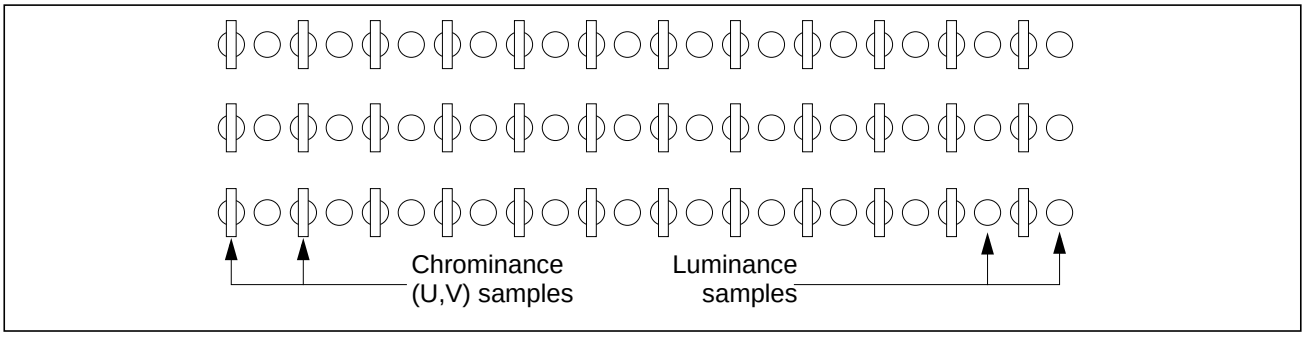


Figure 14-3. 4:2:2 Co-sited input format

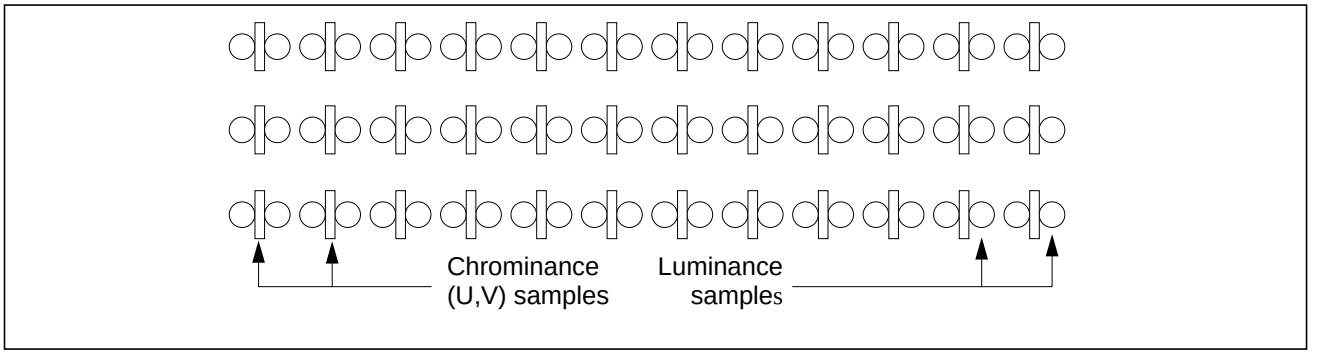


Figure 14-4. 4:2:2 Interspersed input format

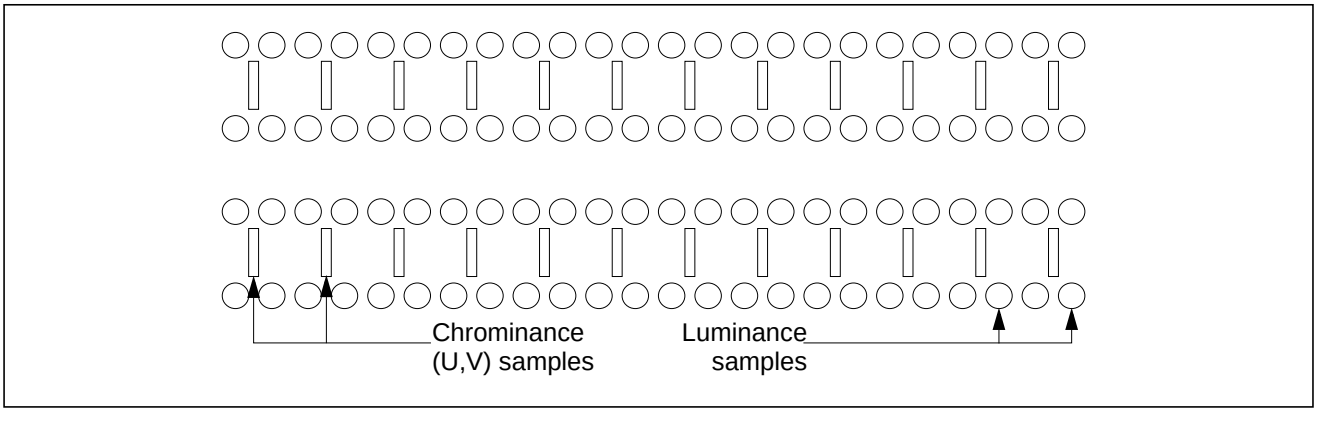


Figure 14-5. 4:2:0 XY Interspersed input format

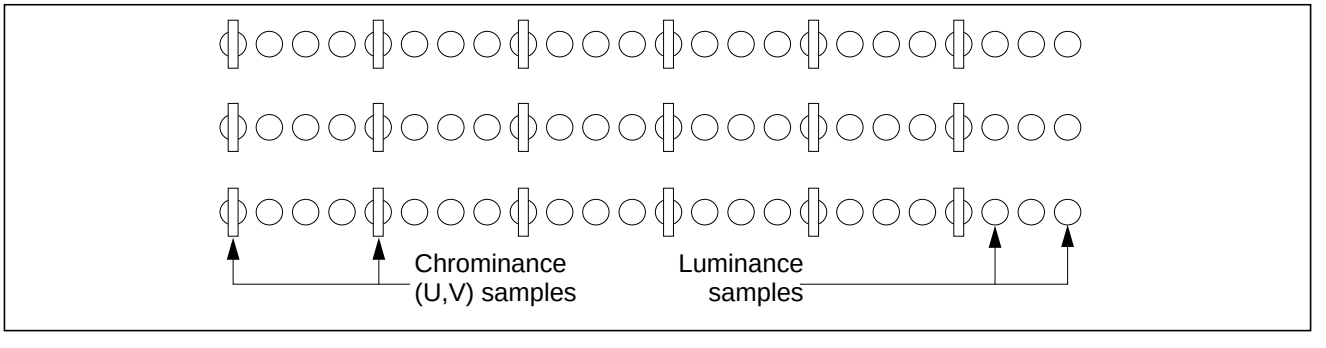


Figure 14-6. 525-60 YUV 4:1:1 Co-Sited input format

Table 14-1. Image Overlay Formats

Format	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
RGB 24+α	a7 - a0	r7 - r0	g7 - g0	b7 - b0
YUV-4:2:2+α	Y1	(v7-v1) + α	Y0	(u7-u1) + α
	Pixel 1		Pixel 0	
RGB 15+α	α r4 r3 r2 r1 r0 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0	α r4 r3 r2 r1 r0 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0

14.4.2 Image Overlay Formats

The ICP accepts image overlay data in three formats, RGB 24+α, RGB 15+α, and YUV-4:2:2+α as shown in Table 14-1. The overlay image format must be the same type as the output image format generated by the ICP for the main image. For example, if the output image is one of the RGB formats, the overlay must be one of the two RGB overlay formats, RGB-24-α and RGB-15+α. If the output image format is YUV, the overlay format must be in YUV-4:2:2+α format. The formats must be of the same type because the ICP does no conversion on the overlay data.

In RGB 24+α, pixels are packed 1 pixel/word, a full byte of alpha information (stored in the most significant byte) is included with each pixel. In RGB 15+α, one bit of alpha is included for each pixel. The pixels in the overlay image are packed as 2 pixels per 32-bit word, and the alpha bit is the most significant bit of each half word. In the same manner, the YUV-4:2:2+α format packs two pixels into one 32-bit word, and has one bit of alpha for each pixel. The least significant bit of the U and V components supplies the alpha bit for the Y0 and Y1 pixels, respectively. The alpha bit in these formats selects between two alpha values stored in the ICP, alpha 1 and alpha 0. The alpha 1 and alpha 0 values are loaded from the parameter block when the ICP is started.

14.4.3 Alpha Blending Codes

Image overlay uses alpha blending, which combines the overlay image with the main image according to the alpha value. The alpha value is supplied by the alpha byte in RGB 24+α format and by the alpha registers, Alpha 0 and Alpha 1 in the other formats. The alpha code format is shown in Table 14-2.

Table 14-2. Alpha Blending Codes

Alpha Code	Alpha Value	Image	Overlay
00h	0	100%	0%
20h	32	75%	25%
40h	64	50%	50%
60h	96	25%	75%
80h - FFh	128-255	0%	100%

14.4.4 Output Formats

The output formats are the RGB image formats sent to the PCI interface or SDRAM. These formats are shown in Table 14-3. Note: B1 = Byte 1 of blue = $[b7...b0]_1$.

Table 14-3. Output Data Formats

Format	Word	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
		Pixel 3	Pixel 2	Pixel 1	Pixel 0
RGB 8A: 233	1	r1 r0 g2 g1 g0 b2 b1 b0	r1 r0 g2 g1 g0 b2 b1 b0	r1 r0 g2 g1 g0 b2 b1 b0	r1 r0 g2 g1 g0 b2 b1 b0
RGB 8R: 332	1	r2 r1 r0 g2 g1 g0 b1 b0	r2 r1 r0 g2 g1 g0 b1 b0	r2 r1 r0 g2 g1 g0 b1 b0	r2 r1 r0 g2 g1 g0 b1 b0
		Pixel 1		Pixel 0	
RGB 15+α	1	α r4 r3 r2 r1 r0 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0	α r4 r3 r2 r1 r0 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0
RGB-16	1	r4 r3 r2 r1 r0 g5 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0	r4 r3 r2 r1 r0 g5 g4 g3	g2 g1 g0 b4 b3 b2 b1 b0
		1 Pixel/Word			
RGB 24+α	1	a7 - a0	r7 - r0	g7 - g0	b7 - b0
		Packed 4 Pixels/3 Words			
RGB 24-packed	1	B1	R0	G0	B0
	2	G2	B2	R1	G1
	3	R3	G3	B3	R2
		Packed 2 Pixels/Word			
YUV- 4:2:2	1	Y1	V0	Y0	U0

14.5 ALGORITHMS

14.5.1 Introduction

The ICP provides filtering, resizing (scaling) and YUV to RGB conversion of the source image. Filtering provides image enhancement. Scaling generates a new image that is larger or smaller than the current image. YUV to RGB conversion is used to generate an RGB version of the image for output to an RGB format frame buffer through the PCI interface or to SDRAM.

The filtering, scaling, and YUV to RGB conversion algorithms are discussed separately. The ICP uses these algorithms in two ways.

1. It provides one pass horizontal scaling with horizontal 5-tap filtering of Y, U, or V.
2. It provides one pass vertical scaling with vertical 5-tap filtering of Y, U, or V.

14.5.2 Filtering

The ICP provides high quality, 5-tap polyphase filtering, both horizontal and vertical, of Y, U, or V data. Each filter type is performed as a separate one dimensional filter pass. Two dimensional filtering of the image requires two passes of the one dimensional filters.

Multi-tap FIR filtering

In multi-tap FIR filtering of an image, the new filter output (pixel) value is a weighted sum of adjacent pixels. The weighting coefficients determine the type of filtering used. A 5-tap filter generates the new pixel value as a weighted sum of the current value and the two pixels on either side (2 left and 2 right for horizontal filtering, 2 above and 2 below for vertical).

A multi-tap FIR filter can be used to generate values for new pixels that are displaced from the original (‘center’) pixel in the same way as linear interpolation. For example, assume the new pixel location is shifted slightly to the right of the center pixel of the input image. A horizontal filter can be used to estimate the new pixel value by weighting the right pixel filter coefficients more heavily than the left, proportional to the relative position offset of the new pixel. (In this sense, interpolation is a 2-tap filter.) This is shown in Figure 14-7. The ICP horizontal and vertical filter operations use this method to combine scaling with filtering.

Mirroring pixels at the start and end of a line or window

A line may start and/or end at the edge of the input image. In this case, the two start and/or end pixels needed for the first and last pixels of the line, respectively, are missing. The ICP uses pixel mirroring to solve this problem. In pixel mirroring, the two available pixels are used to substitute the two missing pixels. The first pixel, uses copies of the two pixels to the right as though they were the two pixels to the left. Specifically, P+2 substitutes for P-2, and P+1 substitutes for P-1. The last pixel uses copies of the two pixels to the left as though they were the two pixels to the right. Since the left and right pixels are now the same, this is called pixel mirroring.

There are five states of pixel mirroring: first output pixel, second output pixel, middle pixels, next to last output pixel and last output pixel. The first output pixel uses pixels numbered (2,1,0,1,2). The second pixel uses (1,0,1,2,3). The middle pixels use (P-2, P-1, P, P+1, P+2). The next to last pixel uses (N-3, N-2, N-1,N, N-1), where N is the number of the last input pixel. The last pixel uses (N-2, N-1, N, N-1, N-2).

In some cases of upscaling, one more input pixel may be needed at the end of the line. In these cases, the pixel value(s) are not generated by the mirror logic. Instead, the ICP uses a copy of the last output pixel as the best estimate of the required output pixel.

14.5.3 Scaling

Scaling overview

Resizing, or scaling, the image means generating a new image that is larger or smaller than the original. The new image will have a larger or smaller number of pixels in the horizontal and/or vertical directions than the original image. A larger image is scaling up (more new pixels); a smaller image is scaling down (fewer newer pixels). A simple case is a 2:1 increase or decrease in size. A 2:1 decrease could be done by throwing away every other pixel (although this simple method results in poor image quality). A 2:1 increase is more interesting. The new pixels can be generated in between the old ones by:

1. Duplicating the original pixels
2. Linear interpolation, where the new in-between pixels are the weighted average of the adjacent input pixels

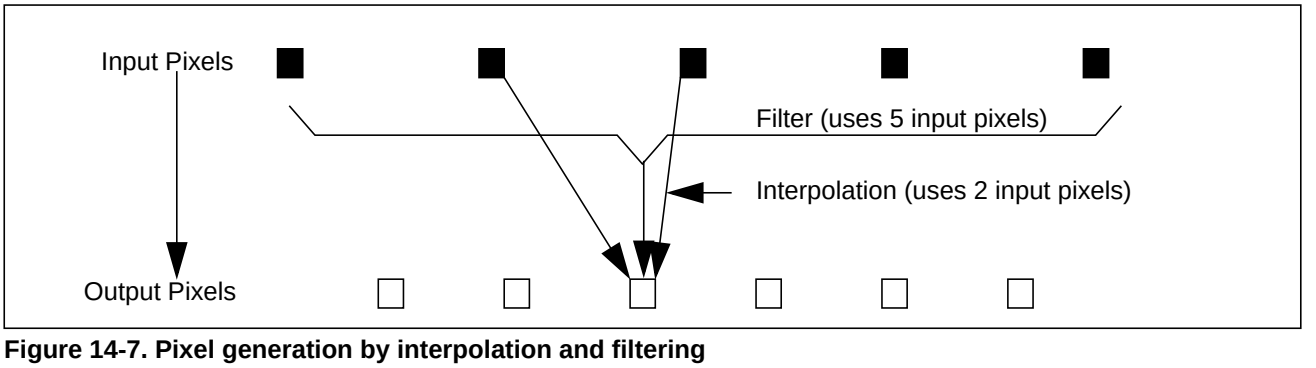


Figure 14-7. Pixel generation by interpolation and filtering

3. Multi-tap filtering, where the new in-between pixels are multi-pixel filtered version of the adjacent input pixels. This approach results in the best image.

The more general case is where the output image resolution is not an integral multiple or sub-multiple of the input image resolution, such as converting from 640 x 480 to 1024 x 768. In this case, the output pixels have differing positions relative to the input pixels in the horizontal or vertical dimensions. In converting from 640 to 1024, the first output pixel on a line corresponds to the first input pixel. The second output pixel is at 640/1024 of the distance between the first and second input pixels. The third output pixel is at (2*640)/1024 of the distance = 1280/1024 = 1+ 256/1024 = 256/1024 of the distance between the second and third input pixels, etc. The output pixels shift with respect to the input pixel grid as you move along the line in the horizontal or vertical dimensions. This is shown in Figure 14-8.

New pixels are generated by interpolation or filtering of the original pixels. Interpolation is the weighted average of the input pixels adjacent to the output pixel. Filtering extends interpolation to include input pixels beyond the input pair adjacent to the output pixel. The number of pixels used to generate the output defines the filter type. Interpolation is a 2-tap filter. A 4-tap filter would use the two pixels to the left and the two pixels to the right of the output pixel. A 5-tap filter identifies the single pixel nearest the output as the center pixel, and uses this pixel plus two to the left and two to the right to generate the output.

If the ratio of the output pixel count per line (in H or V) to input pixel count per line is the ratio of small integers, there is a repeating pattern in these relative positions of input to output pixel locations. For example, for 640 to 1024, the ratio is 8/5. The pattern repeats for every 8 output and every 5 input pixels. If the ratio is not a ratio of small integers, the pattern will take a long time to repeat. The worst case would be 640 to 641, for example. There would be no exact repetition for the whole line.

The interpolator or filter coefficients must be weighted according to the relative position of the new pixel relative to the old pixels. The weighting factor is between 0.0 and 1.0, corresponding to the relative position of the new pixel with respect to the old pixel grid. With a repeating pattern, fewer weighting factors are needed, and therefore fewer coefficients in the linear interpolator or filter generating the new pixels, since you can reuse them each time the pattern repeats. A filter with a repeating pattern is called polyphase, indicating a repeating pattern in the phase (offset position) of the output pixels relative to the input pixels.

Generating the output pixels: relating the output grid to the input grid

Scaling is a pixel transformation in which an array of output pixels is generated from an array of input pixels. The value of each pixel on the output pixel grid is calculated from the values of its adjacent pixels on the input grid. To find these adjacent pixels, you overlay the output grid on the input grid and align the starting pixels, X0Y0, of the two grids. To identify the adjacent input pixels for a given output pixel, you divide the output pixel X (pixel number along the output line) and Y (pixel line number within window) by their corresponding scaling factors:

$X_{in} = X_{out}$ / (horizontal scaling factor)
where: horizontal scaling factor = output length / input length

$Y_{in} = Y_{out}$ / (vertical scaling factor)
where: vertical scaling factor = output height / input height

Note that the resulting X_{in} and Y_{in} values will be real numbers because the output pixels will usually fall between the input pixels. The fractional portion indicates the fractional distance to the next pixel. To calculate the output pixel value, you use the value for the nearest pixel to the left and above and combine it with the value of the other adjacent pixel(s). For example, horizontal interpolation uses the starting pixel to the left interpolated with the next pixel to the right, with the fractional value used to determine the weighting for the interpolation.

ICP scaling output resolution

In the ICP, scaling is forced to have a repeating pattern by limiting the resolution of the new pixel position to 1/32; the new position is forced to be at a location n/32 in H and V relative to the position of the original pixel grid. This results in a worst case error of approximately 1.5% in amplitude relative to calculations using exact output pixel positions. This is comparable to the errors caused by quantizing the amplitude of the pixels. The additional quantization noise can be avoided by choosing an appropriate scale factor which, when inverted, results in fractional values which are expressed in 32nds, such as the 8/5 scaling factor in the 640 to 1024 example above. A diagram of the input to output pixel relationship and the

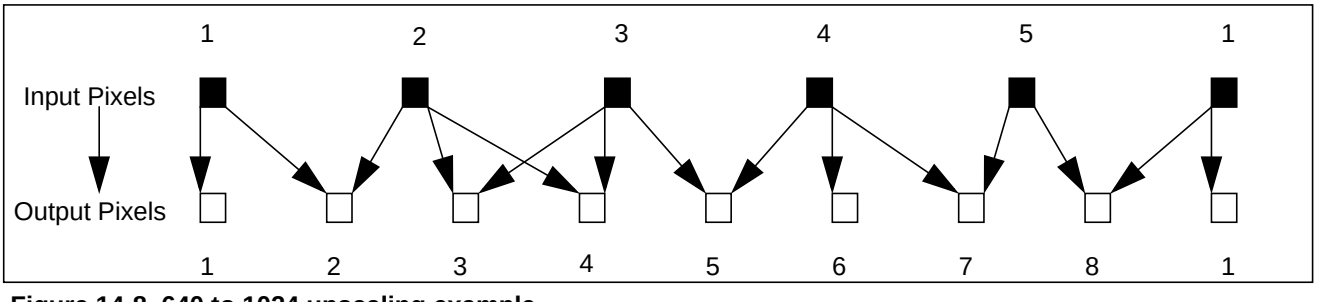


Figure 14-8. 640 to 1024 upscaling example

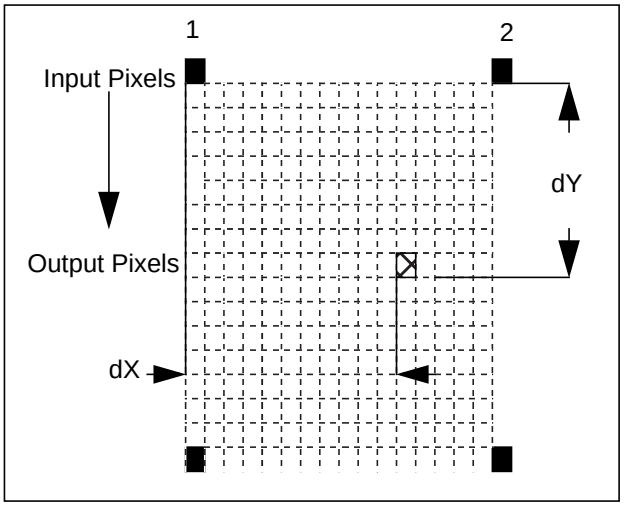


Figure 14-9. ICP 1/32 output resolution

output fractional X and Y subpixel offset is shown in Figure 14-9.

Output scaling calculation method

The output pixel distance in H and V in the ICP is calculated to high precision (16-bit fraction) even though the output resolution is fixed at 1/32 of the input grid. Each output pixel's location relative to the input pixel grid is given by:

X location of output pixel = X0 of input line + output pixel number / X Scale Factor

Y location of output pixel = Y0 of input window + output line number / Y scale factor

The X and Y locations may not be integer values, depending on the scale factor. The resulting X and Y pixel locations can be separated into an integer and a fractional part. The integer part of the X and Y location selects the pixel and line number closest to the output pixel, respectively. The fractional part gives the fractional distance of the output pixel to the next X and Y input pixel values. These fractional parts are the dX and dY values shown in Figure 14-9.

The output pixel value can be calculated by interpolation between the two input pixels or by 5-tap filtering using the 5 nearest pixels rather than the 2 nearest pixels. Interpolation or filtering uses the fractional position values, ΔX and ΔY, to select the appropriate filter coefficients. In the ICP, these values are limited to 5 bits for a resolution of 1/32, even though the actual position value has much higher resolution. The ICP uses fractional values centered around the center pixel with a range of -16/32 to +15/32.

To perform scaling, the X and Y locations of the output pixel relative to the input pixel grid must be generated. This includes both the integer part to locate the adjacent pixels and the fractional part to choose the filter coefficients which generate the output value from the adjacent pixels. This could be done by generating the output pixel X and Y numbers and dividing each by its associated scale factor. Since dividing is expensive in hardware and time, the ICP effectively multiplies the X and Y pixel numbers by the inverse of the X and Y scaling factors, resp. This is done by incrementing the X and Y input pixel counters by X and Y increment values that are the inverse of the X and Y scale factors, resp. For output pixel Xn, the inverse of the scale factor is added to the X input location n times. This is equivalent to multiplying n by the inverse of the scale factor.

The ICP uses a 16-bit integer and a16-bit fractional value for the X and Y increment values. This allows a fractional value resolution of 1/64K. Since the increment value will be added 1024 times in a 1024-pixel line, any error in an individual calculation will be multiplied by 1024. The high resolution of the calculation prevents an accumulation of error as you increment along the line.

Only the most significant 5 bits of the fractional value are used by the filter coefficient RAMs. However, the X and Y counters are incremented by the high-resolution X and Y increment values. The result of this truncation is a worst case error of approximately 1.5% in amplitude relative to arbitrary pixel output positions.

The error caused by discrete (1/32) resolution can be reduced to exactly zero if the output image size is adjusted to have a repeating pattern that fits on these 1/32 boundaries. For zero error, this implies that the scaling factor must be of the form of B/A, where B (the output pixel count factor) is a sub-multiple of 32 [i.e. 1, 2, 4, 8, 16, 32], and A (the input pixel count factor) is an integer determined by the nearest acceptable scale factor for a given B. In the 640 to 1024 conversion case, the B/A ratio was 8/5, meeting this requirement.

The integer values, if accumulated, would be equal to the total number of input pixels when scaling is complete. The integer values for each pixel define the number of pixels to read from memory and shift in to generate the next output pixel. For example, a scaling factor of 1.0 will result in one pixel shifted in for each output pixel generated. Upscaling will have integer increment values of less than one. This means that the integer value will be '0' for some pixels and '1' for others. For example, upscaling by 2.0 will result in integer values of '1' half the time and '0' for the other half, depending on the carry out from the fractional increment.

Pixel shift bypassing for large down scaling

Down scaling will have integer increment values of greater than one. In this case, the integer value indicates the number of pixels to read to obtain filter pixels for the next output pixels. There are two ways to read and shift in the pixels for down scaling: shift all and shift bypass. In the shift all mode (the default mode) all five pixels are shifted for each input value read and shifted in. Shift all mode uses the five input pixels nearest the output pixel, independent of scaling factor. In the shift bypass case, only the last pixel is shifted in. For example, in a down scaling of 10, nine pixels are read and the 10th pixel is shifted in to the filter. Shift bypass mode is used for large down scaling, i.e. down scaling factors of 2.0 or greater. The shift bypass mode is selected by setting the GETB bit in the parameter table. It uses input pixels that are nearest the output pixel and those nearest each of the four output

pixels adjacent to the output pixel. The shift bypass mode also forces the coefficient RAM inputs to '0', since interpolation between adjacent input pixels is no longer being performed.

Using scaling to convert from YUV 4:2:0 to YUV 4:2:2

YUV information in the 4:2:0 format has the UV pixels offset from the input grid in both X and Y. Also, the U and V pixels are at 1/2 of the horizontal and 1/2 of the vertical frequencies of the Y pixels. This means the UV pixels must be filtered and additionally scaled in both X and Y in order to line up with the output Y pixels even if no initial scaling is done. To generate 4:2:2 interspersed data, vertically up-scale U and V by a factor of 2 with a start offset of -1/4 pixel. Upscaling by 2 generates the additional lines required, and starting with a -1/4 pixel offset (relative to U, V space) moves the output up to the same line as the Y pixels. To generate 4:2:2 co-sited, then filter horizontally with no scaling factor but with a start offset of -1/4 pixel, moving the output left 1/4 pixel.

14.5.4 YUV to RGB Conversion

In the ICP, YUV to RGB conversion is done by sequentially processing triplets of Y, U, and V pixel data to convert the pixels to an internal YUV 4:4:4 format and applying the YUV to RGB conversion algorithm on the YUV 4:4:4 pixels. The results of this conversion normally go to the PCI bus but can also go back to SDRAM.

YUV to RGB conversion has two steps. First the Y, U and a V pixel data are used to generate an RGB pixel at the output location. When the Y,U, and V pixels are ready, YUV to RGB conversion is performed using the following algorithms:

R = Y + 1.375(V)= Y + (1 + 3/8)(V)
G = Y - 0.34375(U) - 0.703125(V)
= Y - (11/32)(U) - (45/64)(V)
B = Y + 1.734375(U)
= Y + (1 + 47/64)(U)

In CCIR601, the U and V values are offset by +128 by inverting the most significant bit of the 8-bit byte. This is the way the U and V values are stored in SDRAM. The above algorithms assume that the U and V values are converted back to normal signed two's complement values by inverting the MSB before being used.

14.5.5 Overlay and Alpha Blending

The ICP can add an overlay image to the main image when in the horizontal filter to RGB/YUV mode with PCI output. The overlay image is a user-defined rectangle within the main image. When the overlay is active, each overlay pixel is combined with each main image pixel to generate the resulting pixel to be displayed. Each pixel combination is controlled by an alpha value which determines the proportions of overlay and main image that contribute to the output pixel. The relation is given by:

Pout = (alpha) * Poverlay + (1-alpha) * Pmain =
(alpha) * (Poverlay-Pmain) + Pmain
where: alpha ranges from 0 to 1

In the ICP, the alpha value range is limited by the hardware to five values: {0.0, 0.25, 0.50, 0.75, 1.0}.

An alpha value is supplied for each overlay pixel. In the RGB 24+α overlay data format: an 8-bit alpha value is contained within the overlay data.

In all other overlay data formats (RGB 15+α, etc.), an alpha bit in the overlay data determines the alpha value. The alpha bit selects between two 8-bit values, alpha 1 and alpha 0, supplied by a pair of internal ICP registers. These registers are loaded from the parameter block when the ICP is started. When the alpha bit is '1', alpha 1 value is used as the alpha value; when the alpha bit is '0', alpha 0 is used as the alpha value. The two alpha registers allow translucent images and backgrounds while being restricted to one bit per pixel for alpha selection.

Alpha blending has several uses.

1. Alpha can be used to disable portions of the overlay, called keying. When the alpha for a pixel is '0', there is no overlay. When the alpha is '1', the overlay is 100%, replacing the image. This allows the user to put an irregular shaped object in an image without showing the bounding rectangle of the overlay.
2. Alpha blending allows translucent (smoky) backgrounds and/or translucent (ghostly) overlay images
3. Using alpha at the edges of small images such as font characters increases their effective visual resolution.

Chroma keying

The ICP also optionally provides a restricted form of chroma keying sometimes called color keying. When the overlay Y value is '0' (an illegal value in the YUV 4:2:2+α format) or the RGB values are all '0' (RGB15+α format), the alpha value is forced to '0' and no overlay or blending occurs. This provides three levels of overlay: none, alpha zero, and alpha one. This combination can be used to generate an irregularly shaped menu (an oval shape, for example) which is translucent (e.g. an alpha value of 50%) that contains opaque (alpha = 100%) letters. In a game, this could be a message written on a foggy background in an oval window. The chroma keying provides the definition of the oval shape, the alpha zero value defines the translucent foggy background and the alpha one value defines the opaque characters on the foggy background.

Chroma keying in the ICP is intended for computer generated or modified overlays. Chroma keying turns off the overlay process for selected pixels by forcing an alpha value of '0' for those pixels. Chroma keyed pixels use special codes to identify them. These codes must be computer generated in most cases. For example, the DSPCPU or other CPU would process an overlay image and convert the overlay pixels to be turned off into chroma keyed pixels by changing the data for those pixels to the chroma key code.

The ICP does not have full chroma keying. Full chroma keying has adjustable threshold values for the pixel components. Adjustable thresholds allow the user to automatically select an overlay sub-image from a larger overlay background, such as selecting an image of an actor

against a bright blue background while inhibiting the blue background.

14.5.6 Dithering

Short output codes, such as RGB 8, have few bits for output-value determination. RGB 8R has (2,3,3) bits for (R,G,B). The result is a coarse, patchy image if nothing is done to correct for the limited resolution. Dithering significantly improves the effective resolution of these images. For example, RGB 8 images dithering looks nearly as good as RGB 16.

Dithering works by adding a random dithering value to the pixel before it is truncated by the output formatter. The dither is added to the portion which will be truncated. The carry from this add will occasionally propagate into the most significant portion of the pixel before truncation. The carry from the add thus 'dithers' the displayed value.In the example shown in Figure 14-10, a random dither value is added to the original data before truncation. The dither value should have a range of from approximately 0 to 1 LSB of the truncated value. The dither value should be symmetrical around 1/2 the LSB of the quantizing error of the truncation. In the example shown, the dither signal has values of (1/8, 3/8, 5/8, 7/8). This set of values has a range of approximately 0 to 1 LSB, and it is symmetrical around 1/2 LSB.

In this example, the input signal has a value of 2.83. Without dithering, this value would be truncated to an output value of 2 in all cases. Averaging the un-dithered signal over four pixels still gives you a value of 2. By adding the dither signal, the output value is 2 or 3 depending on the value of the added dither signal. Averaging over four pixels, the average output value is 2.75, much closer to the input value than without the dither signal. The dither signal has significantly reduced the error when averaged over four pixels.

Two types of dithering are combined in the ICP: quad pixel and full image dithering. Quad pixel dithering, also known as ordered dithering, adds one of four dithering values to each pixel. The four dithering values correspond to four-pixel quads in the output image. The pixels in each quad have fixed positions in the input image, so the dither values are chosen on the bases of odd or even line number and odd or even pixel number in the line. The dither values of (0/4, 3/4, 2/4, 1/4) are added by line and pixel number: even line & even pixel, even line & odd pixel, odd line & even pixel, odd line & odd pixel. This gives a four value ordered function for four adjacent pixels in the image. The (0,3,2,1) pattern is chosen specifically to prevent pairs of high or low pixel values from clustering. Spatial dithering provides a significant improvement in effective resolution.

Full image dithering adds a single randomly generated number to every pixel of the image. The result is that the intensity and color accuracy increases as the size of the sample is enlarged. The random number has a long bit length to prevent repeating patterns in the image. The random number can be static or dynamic. In the static case, the random number generator starts with a fixed seed at the start of the image. The random number spatial pattern is fixed for the image even though the image data may change from frame to frame. In the dynamic case, the random number generator runs continuously, and the dithering pattern changes from frame to frame.

The ICP combines quad pixel dithering with full image dithering to provide the final dithering signal for each pixel. The quad pixel dither provides the two most significant bits of the dither signal, and the full image dither provides the least significant 4-bits of the dither signal. The combined dither signal is 6 bits.

From 1 to 6 bits of dither signal are used, depending on the output format. If fewer than 6 bits are needed, only the MSBs of the dither signal are used. For example in the RGB 8R output format, the R output value is 3 bits in size. The output uses the 3 MSBs of the R input value and truncates the 5 LSBs. The dither unit adds 5 bits of dither signal (the 5 MSBs) to the 5 LSBs of the R input value before truncation, and the RGB formatter truncates the result after adding.

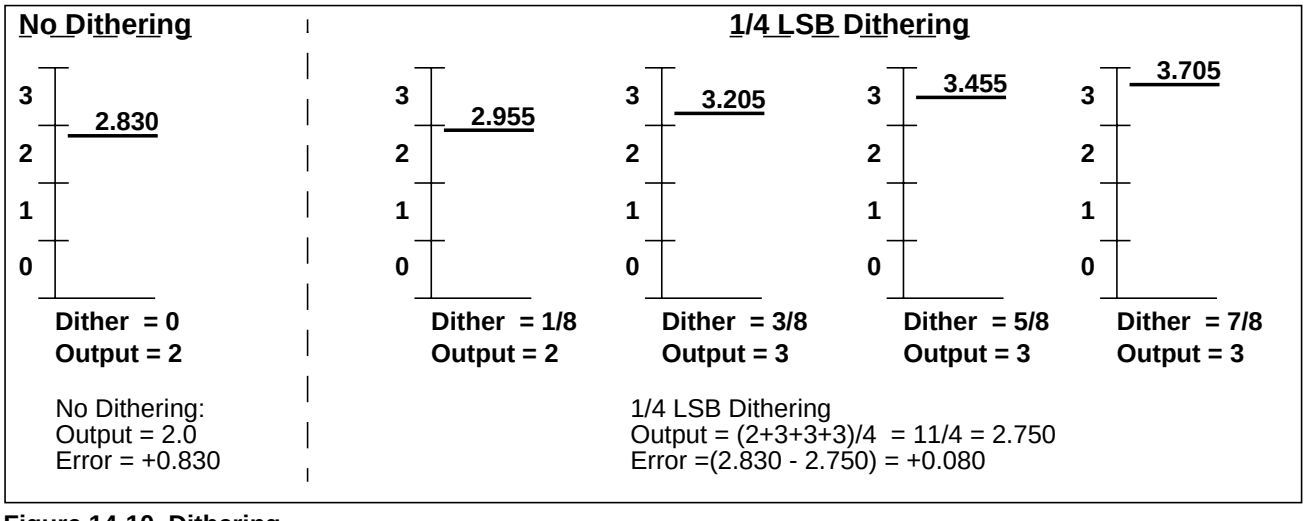


Figure 14-10. Dithering

14.5.7 Implementation Overview: Horizontal Scaling and Filtering

Figure 14-11 shows a data flow block diagram of the ICP horizontal scaling algorithm implementation. Blocks of pixels are provided by the input block buffer. Each block of pixels is transferred sequentially to the 5-tap filter. The filter does scaling and filtering of the data and puts the resulting pixels in the output buffer. Completed pixels in the output buffer are written back to SDRAM or to the PCI output. A bypass multiplexer allows the filter to be bypassed for SDRAM to SDRAM block moves.

Input pixel access is controlled by the Y Counter. The Y Counter selects the word and byte for the current pixel in the Y FIFO buffer. The Y Increment register, Y LSB Register and the Y MSB Counter control the increment of the Y Counter. If the Y MSB Counter contents is not '0', the Y Counter is incremented and the Y MSB register is decremented until the Y MSB Counter is '0'.

The Y MSB Counter is loaded with the integer portion of the results of the Y Counter Increment operation. Y Counter Increment involves adding the Y Increment fraction and integer values to the Y LSB register and Y MSB Counter, respectively. If there is no scaling (scaling factor = 1.0), the Y Increment integer value will be '1', and the Y Increment fractional value will be '0'. Each Y Counter Increment operation will increment the Y Counter by one in this case.

The Y Counter keeps track of horizontally indexed pixels sent to the filter. The Y Counter is incremented once (1.0 for no scaling) for each pixel. For a line of pixels beginning with X_a and ending with X_b, the Y Counter reads pixels from the block buffer beginning with X_{a-2} and ending with X_{b+2}. The extra pixels are required by the 5-tap filter, which uses a total of 5 pixels to generate each output pixel, two pixels before and two pixels after each pixel. The horizontal filter uses the current output from the block buffer and four delayed versions of it to generate the filter output as the weighted sum of the center pixel plus the two on either side. (For the case where the scaling factor = 1.0, the LSBs are always '0'.)

For up or down scaling, the Y Increment value is not 1.0, it is the inverse of the scaling factor (See "ICP scaling output resolution," on page 14-7). For up scaling by a factor of 2.0, the effective Y increment value is 0.5, for example. This means two output pixels are generated for each input pixel. The Y Counter effectively increments as 0.0, 0.5, 1.0, 1.5, 2.0, etc. The LSBs of the counter (i.e. the fractional part less than 1) in the Y LSB register are used by to the filter to generate the intermediate values. An LSB value of 0.5 indicates that the output pixel is half way between X_n and X_{n+1}. The filter contains a set of 5 filter parameter RAMs, one for each coefficient. The 5 most significant LSBs from the counter select the filter coefficients which will generate the correct value for the output pixel at the relative offset from 0.0 indicated by the LSBs.

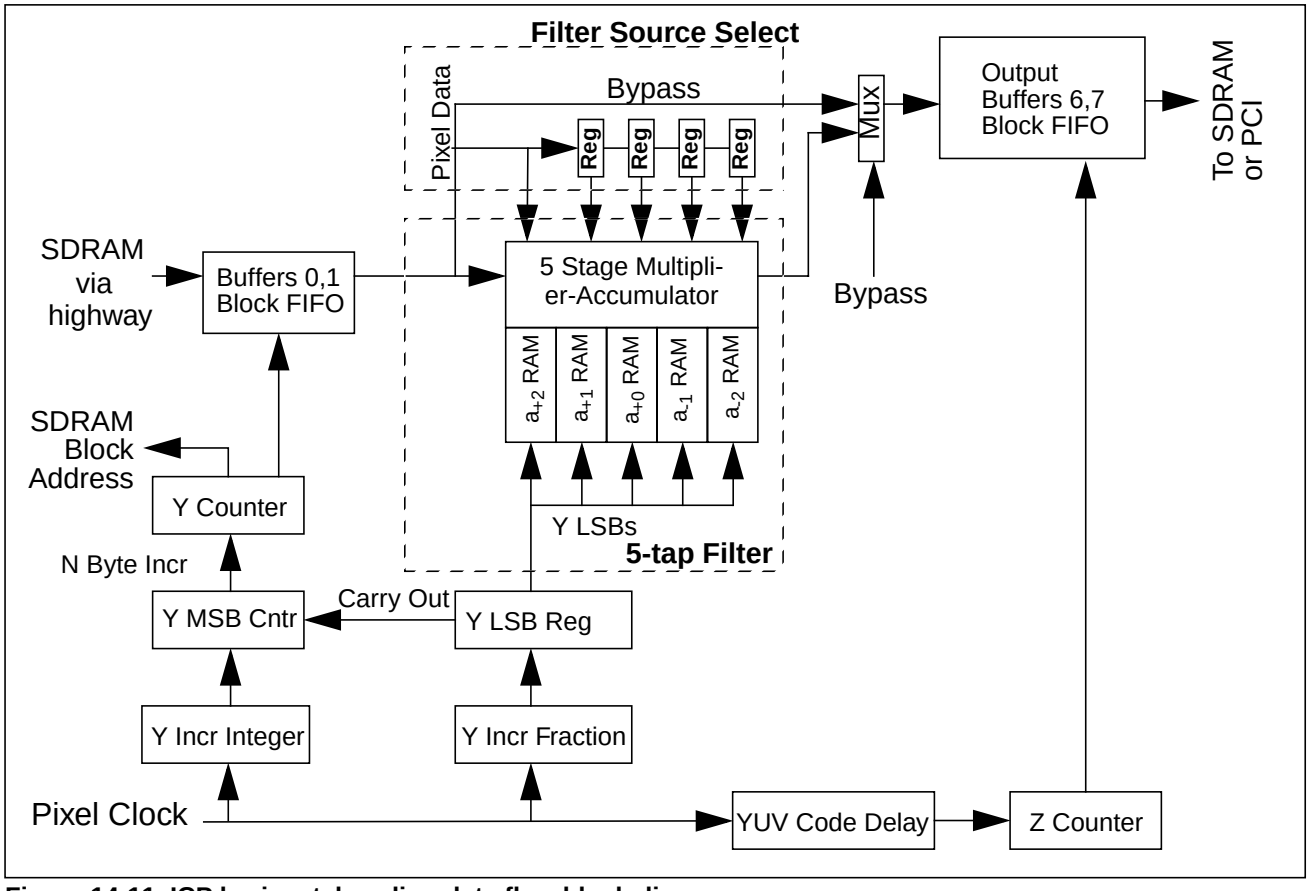


Figure 14-11. ICP horizontal scaling data flow block diagram

The Y Counter indicates the next pixel from the input buffer. A new pixel is clocked into the filter registers only when the Y Counter contents change, which happens when the Y MSB Counter is loaded with a value greater than '0'. Note that for Y increment values less than 1.0 (up scaling), the change will be caused by carry increment from the Y LSBs, and a new pixel will not be clocked into the filter shift register on every Y clock.

For increment values of 2.0 or for values of 1.0 or greater with carry in (down scaling), multiple new pixels will be clocked into the filter shift register before the filter inputs are ready. The number of new bytes needed for the next pixel is the sum of the Y Increment Integer value and the carry out of the Y LSB adder. This result is loaded into the Y MSB Counter. The filter clock is stalled until the inputs are ready. The integer value of the increment -- including carry -- defines the number of new pixels to be clocked through the shift register before the filter inputs are ready for use.

In this discussion, the Y Counter LSBs form a 16-bit binary number. The upper 5 bits of this 16-bit number form a 5-bit binary number between 0 and 31 representing a fractional distance between Y pixels between 0/32 and 32/31. If the new pixel relative distance is 31/32, it is nearest the right pixel of the two pixels it is between, and the right 2 pixels will be more heavily weighted than the left 3.

The horizontal filter shown in Figure 14-11 is pipelined to generate a pixel for every integer increment of the Y Counter. The filter input is always 5 clocks ahead of its output. The first stage generates the filter term $a_{n+2}X_{n+2}$ using the data from the input block and the a_{n+2} coefficient from the coefficient RAM driven by the Y LSBs. The second stage registers hold the data for X_{n+1} and its corresponding Y LSBs and generate $a_{n+1}X_{n+1}$. The last stage registers hold the data for X_{n-2} and the X_{n-2} LSBs and generate $a_{n-2}X_{n-2}$.

The LSB Register contents can change on every clock. In the 2:1 scaling example, the LSBs alternated between 0.0 and 0.5. The LSB Counter represents each output pixel's x offset value from the input pixel grid. The LSB Increment value is 16 bits long. The 5 upper bits go to the coefficient RAMs, and the 11 lower bits provide precision increment of the LSB Counter for precision in representing the scaling factor. The 11 lower bits of the LSB Increment value added to the 11 lower bits of the LSB Counter determine when to increment the 5 LSBs that drive the coefficient RAMs and when to clock a new Y pixel into the filter.

14.5.7.1 Loading the extra pixels in the filter

For a 5-tap filter, 4 more pixel inputs are needed to the filter than are generated at the filter output, two before the first pixel and two after the last pixel. In the worst case of a window that is exactly N blocks wide and starts at the first pixel of the first block, two extra blocks must be read - one at each end of the window - in order to get these 4 pixels! This is an unavoidable problem with a multi-tap filter. For an n-tap filter, n-1 extra pixels are needed. There are two techniques that avoid this efficiency hit of fetching extra blocks.

1. Move the window edges so they are not within 2 pixels of a 64 input pixel boundary.
2. Simulate the edge pixels, such as by mirroring the pair of pixels you have on the other side. This is the only solution to the problem of starting (or ending) at the edge of the image, where there are no pixels to the left (or right) of the image window.

The ICP uses automatic mirroring to supply these pixels. Mirroring is used in both horizontal and vertical filter modes.

14.5.7.2 Mirroring pixels at the ends of a line

A line may start and/or end at the edge of the input image. In this case, the two start and/or end pixels needed for the first and last pixels of the line, respectively, are missing. The start mirror uses the two pixels to the right of the first pixel, and the end mirror uses the two pixels to the right of the last pixel. These pixels are supplied by controlling the Y counter.

A mirror multiplexer in the 5-tap filter provides mirroring of one or two pixels at the filter inputs. This mirror multiplexer is used for both horizontal and vertical filtering. In horizontal filtering, the first and last two pixels in the line are mirrored. The mirror multiplexer is set to the appropriate mirror code for the first and last two pixels in the line. The first two pixels are mirrored for the first two clock pulses, and the last two pixels are detected using the pixel counter for the line.

Mirroring is optional, depending on whether the start or end of the line is on a window boundary. The DSPCPU or microprogram must detect this and enable start and/or end mirroring as required.

14.5.7.3 Horizontal filter SDRAM timing

Figure 14-13 shows a timing diagram for block data flow between the SDRAM and the filter for a scaling factor of 1.0. The bus block reads and writes are one fourth of the filter processing time because the filter processes data at 100 Mpix/sec, and the SDRAM reads and writes blocks of pixels at 400 Mpix/sec. The SDRAM logic reads the next block while the current block is being processed. This also provides the two pixels from the next block required to finish filtering the current block.

If the scaling factor is greater or less than 1.0. the SDRAM bus activity will be different. For scaling factors greater than 1.0, there will be fewer SDRAM reads for the same number of writes generated by the filter. For example, a scale factor of 2.0 means that it is necessary to read only half as many blocks to generate the same number of output blocks. For a scale factor less than one, there will be more reads for the same number of writes. For a scale factor of 0.5, two blocks must be read for every block of output. If the scale factor is less than 1/3, more time will be spent reading and writing SDRAM than filtering.

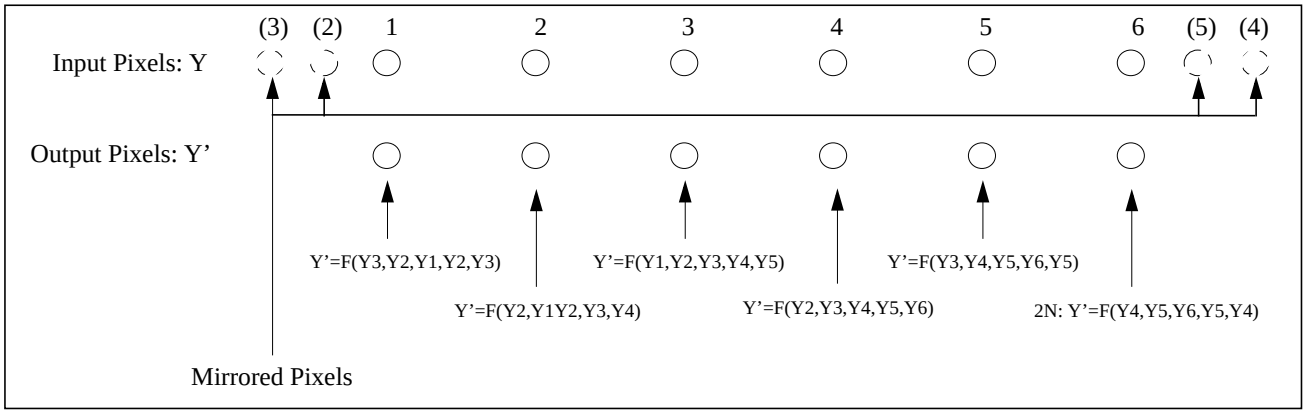


Figure 14-12. Horizontal Pixel Mirroring

14.5.8 Implementation Overview: Vertical Scaling and Filtering

Figure 14-14 shows a data flow block diagram of the ICP vertical scaling algorithm implementation. Blocks of pixels are loaded sequentially into five input block buffers, one for each of the 5 terms of the 5-tap filter. Each block of pixels is transferred sequentially to the 5-tap filter. The filter does scaling and filtering of the data and puts the resulting pixels in the output buffer. Completed pixels in the output buffer are written back to SDRAM.

In vertical scaling, five separate blocks of pixels, one for each line, are required because the pixels are stored in horizontal sequence in the SDRAM. The Y Counter steps through the 64 horizontal pixels of the five input blocks and writes the resulting pixels into the output block. Four of the five blocks are used on the next pass, so that one block of pixels in generates one block of pixels out except for end conditions. The image is processed in 64-pixel columns. Since the image to be filtered will not generally start or end on a block boundary, the number of horizontal pixels for the first and last columns will be less than 64 in these cases. Also, the data in the columns must be aligned vertically. This results in the requirement that the line-to-line address offset value must be a multiple of 64 bytes. Note that only the address offset value is modulo 64; the image to be filtered can start and stop anywhere. Block alignment is not required.

Vertical scaling and filtering processes five 64-pixel input line segments to generate one 64-pixel output segment. When input lines Y_{n-2} to Y_{n+2} have been processed to generate one 64-pixel output segment for output line Y_n, five new input segments are needed for the next output line segment in the 64-pixel column, Y_{n+1}. If the vertical scale factor is 1.0 (no scaling), line segments Y_{n-1} to Y_{n+2} are reused, a new block for Y_{n+3} is loaded and the block for line Y_{n-2} is discarded.

To load Y_{n+3}, the MCU adds the Y offset value to the block address (upper 26 bits) of the Y Counter, and the Y Counter selects the next Y block to be read from SDRAM. The Y Counter points to the line block address for last Y block loaded, and the Y offset value is the address difference between the start of one line and the start of the next, X0Y0 to X0Y1. The line offset is always an integral number of SDRAM blocks. The line offset value must be added to the current line address to get the next line address.

Up and down scaling use the U Counter and U Increment value. The U Counter is used to detect how many lines must be read (0 to 5) to generate the next output line and to generate the vertical offset fraction for the 5-tap filter for output lines that fall between the input lines. The U Counter is set to its starting value (typically '0') at the start of the column, and the U Increment value is added to the U Counter for each output line segment generated in the column. For a scaling factor of 1.0, the U Increment value is 1.0, and each line processed will generate a request for one block. If the scaling factor is 1/2, the increment value will be two, corresponding to moving down two lines. In this case, twice the line offset is added to the Y Counter value.

For up scaling by a factor of 2.0, the Y increment value is 0.5. This means two output lines are generated for each input line. The U Counter increments as 0.0, 0.5, 1.0, 1.5, 2.0, etc. The LSBs of the U Counter (i.e. the fractional part less than 1) are passed along to the filter to generate the intermediate values. An LSB value of 0.5 means that

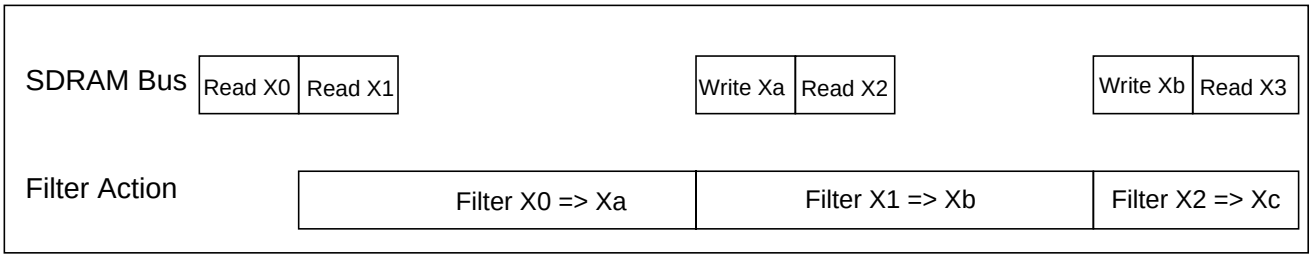


Figure 14-13. SDRAM and horizontal filter block timing

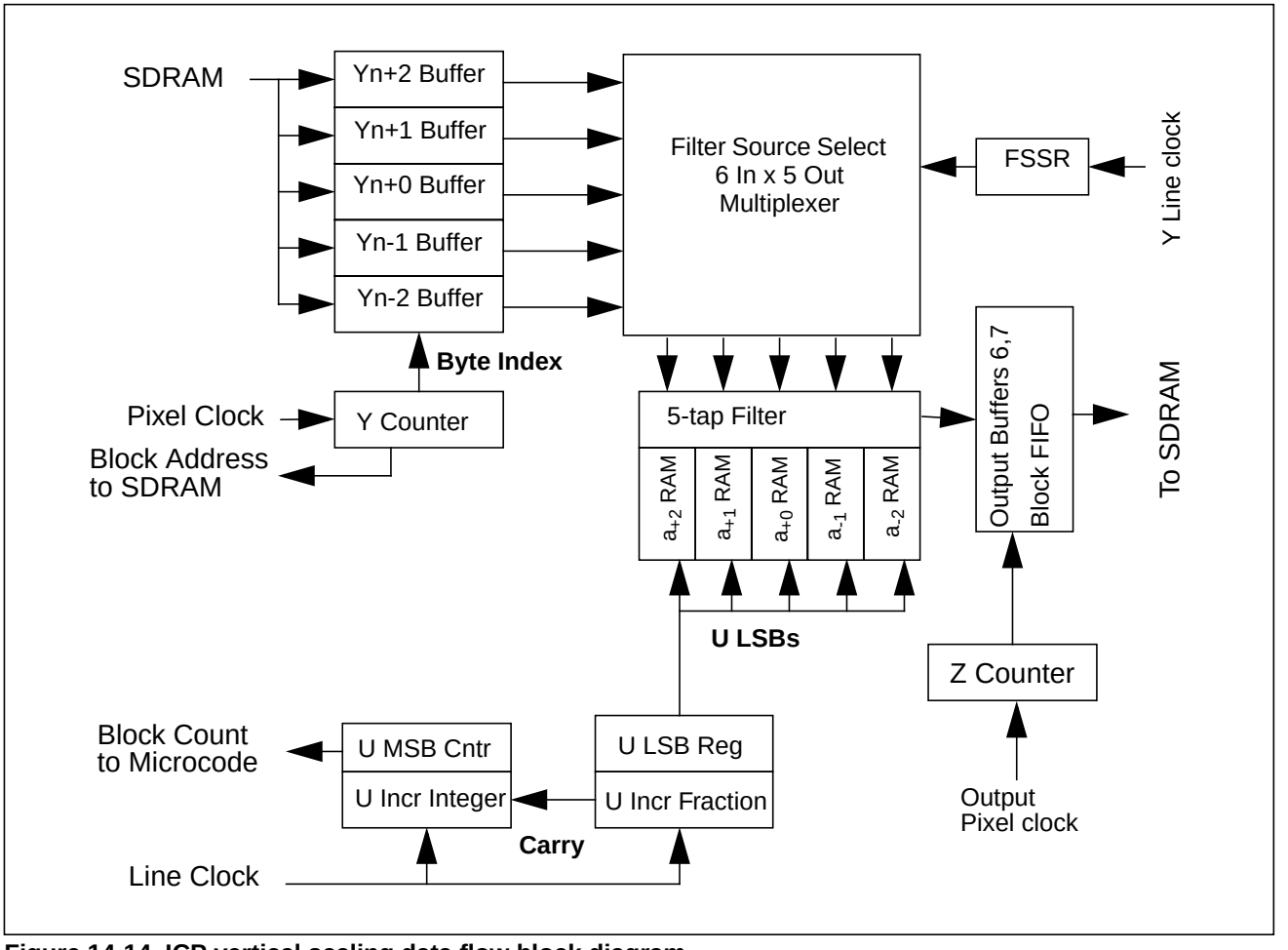


Figure 14-14. ICP vertical scaling data flow block diagram

the output line is half way between Y_n and Y_{n+1}. The filter contains a set of 5 filter parameter RAMs, one for each coefficient. The 5 most significant LSBs from the counter select the filter coefficients which will generate the correct value for the output pixel at the relative offset from 0.0 indicated by the LSBs.

For down scaling, the increment factor will be greater than one. If the increment factor is 2.0, two new blocks will have to be loaded before starting the next vertical filter pass. If the increment factor is 5 or greater, all five blocks must be loaded. The number of blocks to be loaded for the next line is equal to the integer increment value plus carry out from the LSB portion of the U Counter increment.

Note that the LSB adder carry out is available before the U Counter has been updated. This allows the current U Counter value LSB bits to be used for the filter coefficients while using the carry out for the next value to predict how many blocks to fetch. The integer value from the U increment value plus the carry in from the LSB portion of the Increment adder is the number of blocks to be loaded. These blocks must be sequentially loaded (and not skipped) so that the filter has the necessary 5 adjacent lines to perform the filtering. The contents of the integer portion of the U Counter (updated after the add) are not used.

Only one new block can be loaded while the current line is being processed. If two or more blocks are needed to process the next line, load one in overlap. Wait until the current line is done, then load the rest of the blocks. The microprogram only has to make two decisions for the next line: is the increment value ‘0’ or greater than ‘0’, and if greater than ‘0’, is it greater than five. If it is ‘0’, do nothing: you will reuse all five blocks. If it is 1-4, load the next block. If it is five or more, calculate the address of the first block -- by adding N times the address offset to the Y counter -- and fetch it.

When a new block is loaded and it is time to process the next line, the block which was Y_{n+2} becomes Y_{n+1}. The Y blocks, in effect, shift up one line as you scan down the image. This shifting action is implemented by shifting the block select codes in the Filter Source Select Register (FSSR). The FSSR contains six 3-bit register fields. These 3-bit fields are rotated by a shift command to the FSSR. The output of five of the FSSR fields go to the input multiplexer, which selects the next block combination and sends it to the filter. The output of the sixth field is the free block to be filled for the next line while the current line is being processed. The select code is also the block code (0 to 5), so the free block is identified by its block code in the FSSR. The FSSR codes for the six cases of vertical filtering are shown in Table 14-4

Table 14-4. FSSR codes for vertical filtering.

Case	Pn-2	Pn-1	Pn+0	Pn+1	Pn+2	IO Block
1	5	4	3	2	1	0
2	0	5	4	3	2	1
3	1	0	5	4	3	2
4	2	1	0	5	4	3
5	3	2	1	0	5	4
6	4	3	2	1	0	5

14.5.8.1 Mirroring lines at the ends of an image

A window may start and/or end at the edge of the input image. In this case, the two start and/or end lines needed for the first and last lines of the window, respectively, are missing. These pixels are supplied by the mirror multiplexer at the 5-tap filter which mirrors the input lines.The mirror multiplexer is controlled by the mirror counter and mirror end register in the same manner as in horizontal filtering. The mirror register in vertical filtering is incremented by the output line counter. Mirroring is performed on the first two and last two lines of the column. Mirroring is optional, depending on whether the start or end of the line is on a window boundary. The DSPCPU or microprogram must detect this and enable start and/or end mirroring as required.

14.5.8.2 Vertical filter SDRAM block timing

Figure 14-15 shows a timing diagram for block data flow between the SDRAM and the filter for a scaling factor of 1.0. The bus block reads and writes require one fourth of the filter processing time because the filter processes data at 100 Mpix/sec, and the SDRAM reads and writes blocks of pixels at 400 Mpix/sec (peak). The vertical filter starts by reading in the five blocks necessary to generate the next output block. While the current block is being processed, the next block is read from SDRAM to prepare for the next output block.

14.5.9 Horizontal Scaling and Filtering for RGB Output

Figure 14-16 shows a data flow block diagram of the ICP horizontal scaling to RGB output algorithm implementation. The six input block buffers are arranged as three block FIFOs, one each for Y, U and V pixel streams. These three streams are sequentially filtered, pixel by pixel by the 5-tap filter to generate a scaled output sequence of Y, U, V, Y, U, V, etc. This YUV stream is fed to the YUV to RGB converter where it is converted to one of several RGB output formats, blended with RGB overlay pixels supplied by the Overlay FIFO and masked by bit mask pixels from the bit mask block. The resulting scaled, converted, overlay blended and masked RGB stream is sent to the PCI interface -- typically to an RGB format frame buffer on the PCI bus -- or to SDRAM.

The input pixel streams from the input FIFOs are transferred sequentially to the 5-tap filter. Each stream has its own set of four-stage delay registers used to perform horizontal filtering on the stream. A pair of 3-way multiplexers switch the five filter data inputs and the 5-bit filter coefficient select codes to the 5-tap filter. This set of multiplexers is driven by the YUV Sequence counter, a 2-bit counter that provides the YUV processing sequence.

In horizontal scaling and filtering from SDRAM to SDRAM, each Y, U and V component is filtered separately as a complete image. In RGB output horizontal scaling and filtering, the image is processed as three interwoven streams of all three YUV components.

In the RGB output mode, the ICP normally generates RGB data and writes it into a frame buffer memory on the PCI bus or to the SDRAM. The frame buffer memory format is RGB with one R, one G and one B value per pixel. This could be called RGB 4:4:4. To generate this image, the ICP generates a YUV 4:4:4 image and converts it to RGB. This process is done one RGB output pixel at a time. The ICP generates a U pixel and saves it in a register, generates a V pixel and saves it in a register, then generates a Y pixel for output. The YUV to RGB converter combines each Y pixel as it is generated with the previously stored U and V pixels to generate the RGB output data. This process is repeated until the whole image has been converted and sent to the PCI bus or SDRAM.

14.5.9.1 YUV sequence counter in YUV 4:2:2 output Mode

For RGB output formats, the YUV data must be scaled to YUV 4:4:4 format before conversion to RGB. The YUV data in SDRAM is typically stored in YUV 4:2:2. This means that the U and V data must be upscaled by 2 relative to the Y data to generate the internal YUV 4:4:4 format required for RGB conversion.

For the YUV 4:2:2 output formats, the U and V data do not need to be up scaled to 4:4:4. The YUV 4:4:4 data would be upscaled only to be decimated back to YUV 4:2:2. For YUV 4:2:2 output, the U and V pixels are used twice. This is done by having a half-speed mode for the YUV Sequence Counter. In this mode, the sequence is U0, V0, Y0, Y1, U2, V2, Y2, Y3, etc. The U and V are not

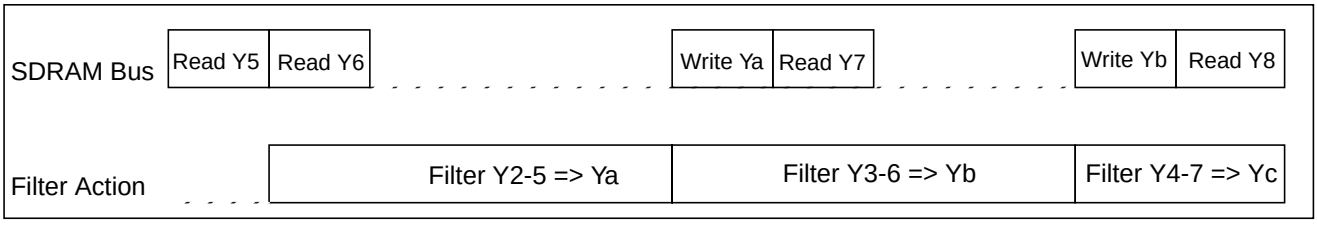


Figure 14-15. SDRAM and vertical filter block timing

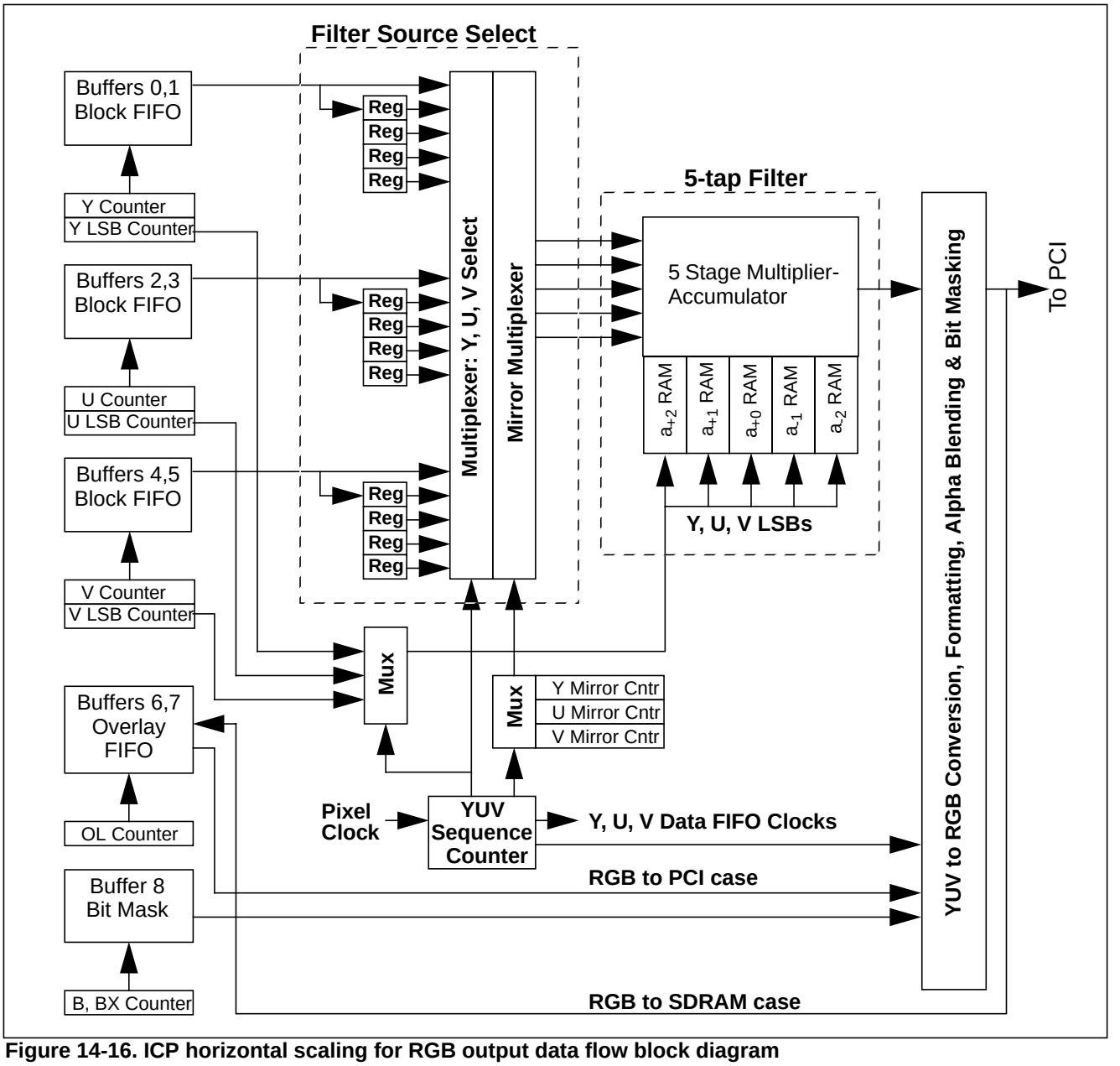


Figure 14-16. ICP horizontal scaling for RGB output data flow block diagram

up scaled by 2 relative to the Y component for YUV 4:4:4 output, although they could be up scaled as part of general up scaling of the image.

The YUV 4:2:2 output mode also provides higher processing bandwidth relative to YUV 4:4:4 up scaling. Half as many U and V pixels are processed.The output pixel rate is one pixel per 20 nanoseconds for the YUV 4:2:2 output mode versus one pixel per 30 for conversion to YUV 4:4:4. This can be used to provide some processing performance improvement for very large images at the expense of some chroma quality.

14.5.9.2 PCI output block timing

The ICP outputs pixels to the PCI interface at a peak rate of 33 Mpix/sec in RGB mode and 50 Mpix/second in the YUV mode using YUV sequencing. For one word per pixel output codes, such as RGB-24, this is a peak rate of 33 Mwords/sec or 132 Mpix/sec in the RGB sequencing mode. This is the same speed as the 132 MB/sec peak rate of the PCI interface. (At 50 Mpix/sec, the result would be 200 MB/sec.) The BIU control for the PCI interface has a FIFO for buffering data from the ICP, but this buffer is only 16 words deep. Therefore, the ICP will occasionally have to wait for the PCI to accept more data. In the PCI output mode, this stalls the ICP clock.

14.6 OPERATION AND PROGRAMMING

The ICP uses a combination of hardware and a Microprogram Control Unit (MCU) to implement its scaling, filtering and conversion functions. The microprogram is a

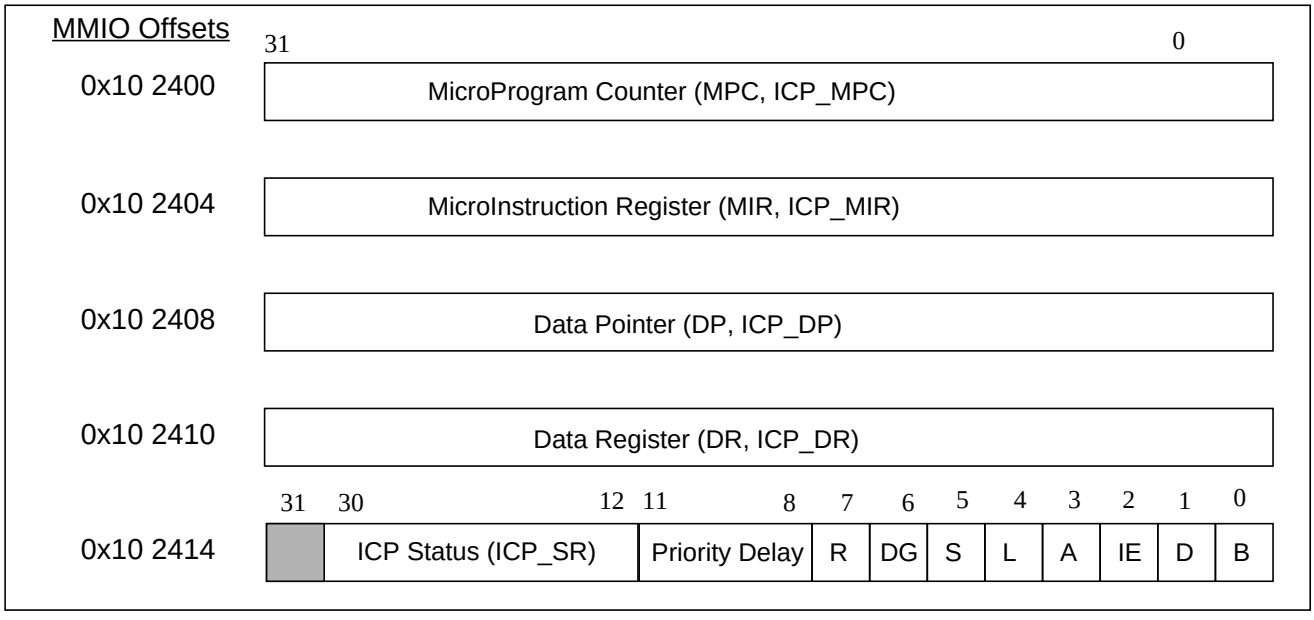


Figure 14-17. ICP MMIO Registers

factory-supplied state machine that resides in SDRAM. It is read each time the ICP executes an operation. Using an SDRAM-resident microprogram-controlled state machine minimizes hardware and provides flexibility in handling special conditions without additional hardware.

Important Note: You must set the ICP DMA Enable bit (IE) in the BIU_CTL register of the PCI interface for RGB output to PCI. This bit must be set before initiating RGB to PCI operations, or the ICP will stall waiting for the PCI to become ready. Refer to Section 11.7.5, “BIU_CTL Register.”

14.6.1 ICP Register Model

The ICP is controlled by the DSPCPU through five MMIO registers: the MicroProgram Counter (MPC), the Micro Instruction Register (MIR), the Data Pointer (DP), the Data Register (DR) and the ICP Status register (SR), as shown in Figure 14-17. The MPC, DP and SR are used in normal operations, and the MIR and DR are used in test and debug. Note that the MMIO registers should *never* be written while the ICP is executing microcode, i.e test the Busy bit in the SR register before writing any ICP MMIO register.

The MPC is the MCU instruction counter. It points to the next microinstruction to be executed. The entry point in the microprogram defines which ICP operation is to be executed.The DP points to the location in SDRAM of a table of parameters used by the ICP to process the image data, such as the image input and output start addresses, scaling factor, etc.

The SR has 13 active bits: Busy (B), Done (D), done Interrupt Enable (IE), ACK_DONE (A), Little Endian (L), Step (S), Diagnostic (DG), Reset (R), Priority Delay (PD, 4 bits). Bits 12 .. 30 are reserved.

- (B)usy indicates the ICP is busy executing microcode.
- (D)one indicates that the previous requested function is complete, and that the ICP clock is stopped.
- (D)one causes an interrupt to the DSPCPU when Interrupt Enable is set.
- (A)CK_DONE clears (D)one and the corresponding interrupt.
- (L)ittle Endian sets the highway endian swap multiplexer to little endian mode for data on the SDRAM bus.
- (S)tep causes the MCU to execute one microinstruction. Step is used for diagnostics to step the ICP through its microinstructions one clock step at a time. Writing a ‘1’ to Step sets Busy, which is reset at the end of execution of the next microinstruction.
- (DG) allows SDRAM operations in step mode.
- (R) is a write-only bit that resets ICP internal registers.
- (PD) sets a timer for bus activity that defines the minimum bus bandwidth available to the ICP.

The ICP Status Register contains 20 read-only status bits. The upper 16 bits of the Status Register can contain a 16-bit code returned by the microprogram upon completion. Bits 15 through 12 are reserved for error flags.

Important Note: You must set the ICP DMA Enable bit (IE) in the BIU_CTL register of the PCI interface for RGB output to PCI. This bit must be set before initiating RGB to PCI operations, or the ICP will stall waiting for the PCI to become ready. Refer to Section 11.7.5, “BIU_CTL Register.”

14.6.2 Power Down

The ICP block enters in power down state whenever TM1300 is put in global power down mode.

The ICP block can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. Refer to Chapter 21, “Power Management.”

It is recommended that ICP is in an idle state before block level power down is activated.

14.6.3 ICP Operation

The DSPCPU commands the ICP to perform an operation by loading the DP with a pointer to a parameter block, loading the MPC with a microprogram start address and setting Busy in the SR. For example to cause the ICP to scale and filter an image, set up a block of SDRAM with the image and filter parameters, load the MPC with the starting address of the appropriate microprogram entry point in SDRAM, load the DP with the address of the parameter block, and set Busy in the SR by writing a ‘1’ to it. When the filter operation is complete, the ICP will set Done and issue an interrupt. The DSPCPU clears the interrupt by writing a ‘1’ to ACK_DONE. Note: The interrupt should be set up as a ‘level triggered.’

When the DSPCPU sets Busy, the MCU begins reading the microprogram from SDRAM. The microinstructions are read in from SDRAM as required by the ICP, and internal pre-fetching is used to eliminate delays. Setting Busy enables the MCU clock, the first block of microinstructions is automatically read in, and the MCU begins instruction execution at the current address in the MPC. Clearing Busy stops the MCU clock. Busy can be cleared by hardware reset, by the MCU, or by the DSPCPU. Hardware reset clears the Status register, including Busy and Done, and internal registers, such as the TCR. When the MCU completes a microprogram operation, the microprogram typically clears Busy and sets Done, causing an interrupt if IE is enabled.

The DSPCPU performs a software reset by clearing (writing a ‘0’ to) Busy and by writing a ‘1’ to Reset. The DSPCPU can also set Done to force a hardware interrupt, if desired.

14.6.4 ICP Microprogram Set

The ICP comes with a factory-generated microprogram set which implements the functions of the ICP. The microprogram set includes the following functions:

1. Loading the filter coefficient RAMs.
2. Horizontal scaling and filtering from SDRAM to SDRAM of an input image to an output image. The input and output images can be of any size and position that fits in SDRAM. The scaling factors are, in general, limited only by input and output image sizes.
3. Vertical scaling and filtering from SDRAM to SDRAM of an input image to an output image. The input and output images can be of any size and position that fits in SDRAM. The scaling factors are, in general, limited only by input and output image sizes.
4. Horizontal scaling, filtering and YUV to RGB conversion of an input image from SDRAM to an output image to PCI or SDRAM, with an alpha-blended and chroma-keyed RGB overlay and a bit mask. The input and output images can be of any size and position that fit in SDRAM and can be output to the PCI bus or SDRAM. In general, scaling factors are limited only by input and output image sizes.

The microprogram is supplied with the ICP as part of the device driver. The entry point in the microprogram defines which ICP operation is to be done. The entry points are given below in terms of word offsets from the beginning of the microprogram:

Offset	Function
0	Load coefficients
1	Horizontal scaling and filtering
2	Vertical scaling and filtering
3	Horizontal scaling, filtering, YUV to RGB conversion, bit masking (PCI) and overlay (PCI) with alpha blending and chroma keying

14.6.5 ICP Processing Time

The processing time for typical operations on typical picture sizes has been measured.

Measurements were performed with the following configuration:

- CPU clock and SDRAM clock set to 100 MHz
- PCI clock set to 33MHz
- All measurement with PCI as pixel destination were done with an Imagine 128 Series II graphics card, which never caused a slowdown of the ICP operation.
- TRITON2 mother-board with SB82437UX and SB82371SB based Intel® Pentium™ chipset.
- TM1300 arbiter set to default settings
- TM1300 latency timer set to maximum value = 0xf8.
- Overlay sizes were the same as picture sizes.

Results are tabulated below for three different cases of available memory bandwidth:

1. No other load to SDRAM, i.e. full SDRAM bandwidth available for ICP. See Table 14-5

2. SDRAM memory loaded to 95% of its bandwidth by DCACHE traffic from DSPCPU. Priority delay = 1, i.e. ICP did wait one block time before competing for memory. See Table 14-6

3. SDRAM memory loaded to 95% of its bandwidth by DCACHE traffic from DSPCPU. Priority delay = 16, i.e. ICP did wait 16 block times before competing for memory. See Table 14-7

Note: A load of 95% of the memory bandwidth is very rarely found in a real system. So the results in these tables may be useful to estimate upper bounds for the computation time in a loaded system.

The priority delays were set to the minimum and maximum possible values, so the computation time for other priority delay values should be somewhere in between.

A simple linear model of computation time has been fitted to the tabular data and to corresponding measurements with half the number of pixels per line.

It was assumed that

processing time = (time per line start)* (number of lines)

+(time per pixel) * (number of pixels)

Table 14-8, Table 14-9, Table 14-10 give the time per line start and the time per pixel in this equation for the three memory bandwidth cases.

The maximum deviation between measured time and fitted model is on the order of 10% in the range W = 180 ... 1024, H = 240 ...768. The deviation is much less in most cases. The values were found by least squares fit to the measured data.

In some cases the cumulative time for line starts contributed so little to the total computation time that the value per line start could only be determined relatively inaccurately. In other words the pixel time portion dominated the equation so much that the line time portion was negligible, given the inaccuracies of the model.

Therefore the simple model is only thought to allow interpolation for other picture sizes within the range W = 180 ...1024, H = 240 ... 768. Extrapolation to picture sizes much outside this range should not be attempted using this data.

In some cases the real ICP performance may be much better than that predicted by the model, due to irregular behavior of the ICP.

For horizontal and vertical up/down-scaling operations use the larger W or H value occurring at input/output with the H/V filter times table or model.

This will lead to overestimation of processing time by up to 20%.

Table 14-5. Measured processing time in ms - no other load to SDRAM

W in pixels	360	640	720	720	800	800	1024
H in pixels	240	480	480	768	480	600	768
horizontal filter, 1 component	1.22	3.82	4.43	7.08	4.78	5.98	9.27
horizontal filter, 3 components YUV 4:2:2	2.68	8.18	9.29	14.86	10.08	12.60	19.35
vertical filter, 1 component	2.57	8.73	10.24	16.36	11.19	13.97	22.30
vertical filter, 3 components YUV 4:2:2	5.15	17.47	20.48	32.72	22.95	28.65	44.60
yuv to rgb8a, pci output	3.36	10.74	11.93	19.08	13.04	16.30	26.02
yuv to rgb15a, pci output	3.39	10.79	11.96	19.12	13.10	16.41	26.15
yuv to rgb24, pci output	3.72	12.24	13.52	21.62	14.85	18.59	29.98
yuv to rgb24a, pci output	4.34	14.52	16.04	25.02	17.58	21.63	35.01
yuv to rgb8a, sdram output	3.39	10.78	11.95	19.09	13.13	16.40	26.08
yuv to rgb15a, sdram output	3.46	11.04	12.26	19.60	13.46	16.82	26.87
yuv to rgb24, sdram output	3.62	11.69	13.06	20.88	14.43	18.03	28.71
yuv to rgb24a, sdram output	3.90	12.69	14.11	22.57	15.65	19.56	31.07
yuv to rgb8a, bitmask, pci output	3.37	11.42	12.49	19.97	13.61	17.01	27.83
yuv to rgb8a, RGB 15a overlay, pci output	3.67	11.72	12.92	20.67	14.23	17.79	28.23
yuv to rgb8a, RGB 24a overlay, pci output	4.23	13.57	15.32	24.51	16.93	21.15	33.15
yuv to rgb8a, yuv 422a overlay, pci output	3.67	11.72	12.92	20.67	14.23	17.79	28.23
yuv to rgb8a, 422 sequencing, pci output	2.52	7.77	8.57	13.70	9.32	11.65	18.40

Table 14-6. Measured processing time in ms - SDRAM loaded 95%, priority delay = 1

W in pixels	360	640	720	720	800	800	1024
H in pixels	240	480	480	768	480	600	768
horizontal filter, 1 component	2.01	6.37	7.60	12.16	8.02	10.02	16.02
horizontal filter, 3 components YUV 4:2:2	4.11	13.69	15.62	24.96	16.56	20.68	32.65
vertical filter, 1 component	2.60	8.79	10.34	16.50	11.25	14.05	22.43
vertical filter, 3 components YUV 4:2:2	5.20	17.59	20.66	32.96	23.15	28.89	44.87
yuv to rgb8a, pci output	3.51	11.08	12.17	19.46	13.51	16.88	26.56
yuv to rgb15a, pci output	3.52	11.11	12.22	19.51	13.47	16.82	26.65
yuv to rgb24, pci output	3.88	12.51	13.79	22.08	15.21	18.99	30.26

Table 14-6. Measured processing time in ms - SDRAM loaded 95%, priority delay = 1

W in pixels	360	640	720	720	800	800	1024
H in pixels	240	480	480	768	480	600	768
yuv to rgb24a, pci output	4.39	14.29	15.84	25.30	17.72	22.00	34.83
yuv to rgb8a, sdram output	3.69	11.67	12.75	20.39	14.20	17.80	27.95
yuv to rgb15a, sdram output	4.25	13.15	14.64	23.41	16.79	20.98	31.49
yuv to rgb24, sdram output	5.17	16.56	18.71	29.90	20.85	26.06	40.82
yuv to rgb24a, sdram output	5.82	18.64	21.02	33.62	23.23	29.03	45.34
yuv to rgb8a, bitmask, pci output	3.65	12.37	13.45	21.50	14.68	18.34	30.13
yuv to rgb8a, rgbl15a overlay, pci output	4.94	15.30	17.23	27.51	19.06	23.78	36.70
yuv to rgb8a, rgbl24a overlay, pci output	6.77	21.93	24.85	39.73	27.44	34.31	53.67
yuv to rgb8a, yuv422a overlay, pci output	4.95	15.30	17.22	27.51	19.06	23.80	36.70
yuv to rgb8a, 422sequencing, pci output	3.04	8.92	9.63	15.39	10.53	13.16	20.37

Table 14-7. Measured processing time in ms, SDRAM loaded 95%, priority delay = 16

W in pixels	360	640	720	720	800	800	1024
H in pixels	240	480	480	768	480	600	768
horizontal filter, one component	7.70	24.28	29.32	46.90	30.05	37.56	60.39
horizontal filter, 3 components YUV 4:2:2	15.28	52.00	60.08	96.10	63.13	78.90	123.29
vertical filter, one component	7.50	26.71	30.92	49.31	33.57	41.93	68.18
vertical filter, 3 components YUV 4:2:2	14.48	53.45	60.70	96.83	68.69	85.79	136.40
yuv to rgb8a, pci output	10.55	31.61	34.95	55.84	37.18	46.47	74.29
yuv to rgb15a, pci output	10.55	31.61	34.93	55.84	37.17	46.45	74.29
yuv to rgb24, pci output	10.39	31.71	34.93	55.84	37.25	46.54	73.58
yuv to rgb24a, pci output	10.49	31.95	35.06	55.98	37.15	46.46	74.10
yuv to rgb8a, sdram output	13.83	41.93	48.10	76.94	51.57	64.42	99.33
yuv to rgb15a, sdram output	17.58	55.55	60.95	97.49	65.82	82.24	137.71
yuv to rgb24, sdram output	20.25	65.46	74.67	119.44	81.74	102.12	158.43
yuv to rgb24a, sdram output	24.05	78.51	88.98	142.21	98.69	125.67	196.99
yuv to rgb8a, bitmask, pci output	11.05	35.04	37.75	60.37	40.15	50.19	85.13
yuv to rgb8a, rgbl15a overlay, pci output	18.19	57.11	62.60	100.04	70.84	88.26	136.03
yuv to rgb8a, rgbl24a overlay, pci output	24.81	80.19	91.86	145.57	100.72	125.00	198.15
yuv to rgb8a, uv422a overlay, pci output	18.20	57.11	62.60	100.04	70.00	88.28	135.98
yuv to rgb8a, 422sequencing, pci output	10.56	31.09	34.79	55.63	36.27	45.33	74.43

Table 14-8. Line start and pixel time for linear model, no other load on SDRAM

function	t/linestart (μs)	t/pixel (ns)
horizontal filter, 1 component	1.1	11
horizontal filter, 3 components YUV 4:2:2	3.2	22
vertical filter, 1 component	0.2	29
vertical filter, 3 components YUV 4:2:2	0.7	58
yuv to rgb8a, pci output	3.2	30
yuv to rgb15a, pci output	3.3	30
yuv to rgb24, pci output	3.7	34
yuv to rgb24a, pci output	5.3	40
yuv to rgb8a, sdram output	3.4	30
yuv to rgb15a, sdram output	3.3	31
yuv to rgb24, sdram output	3.1	33
yuv to rgb24a, sdram output	3.4	36
yuv to rgb8a, bitmask, pci output	2.5	32
yuv to rgb8a, rgbl15a overlay, pci output	3.8	32
yuv to rgb8a, rgbl24a overlay, pci output	4.0	39
yuv to rgb8a, yuv422a overlay, pci output	3.8	32
yuv to rgb8a, 422sequencing, pci output	3.2	20

Table 14-9. Line start and pixel time for linear model, SDRAM loaded 95%, priority delay = 1

function	t/linestart (μs)	t/pixel (ns)
horizontal filter, 1 component	0.9	20
horizontal filter,3 components YUV 4:2:2	2.8	40
vertical filter, 1 component	0.2	29
vertical filter, 3 components YUV 4:2:2	0.7	58
yuv to rgb8a, pci output	3.8	30
yuv to rgb15a, pci output	3.8	30
yuv to rgb24, pci output	4.5	34
yuv to rgb24a, pci output	6.0	39
yuv to rgb8a, sdram output	4.3	31
yuv to rgb15a, sdram output	4.9	36
yuv to rgb24, sdram output	4.6	47
yuv to rgb24a, sdram output	5.0	53
yuv to rgb8a, bitmask, pci output	3.2	34
yuv to rgb8a, rgbl15a overlay, pci output	5.5	42
yuv to rgb8a, rgbl24a overlay, pci output	5.8	63
yuv to rgb8a, yuv422a overlay, pci output	5.5	42
yuv to rgb8a, 422sequencing, pci output	4.9	21

Table 14-10. Line start and pixel time for linear model, SDRAM loaded 95%, priority delay = 16

function	t/linestart (μs)	t/pixel (ns)
horizontal filter, 1 component	2.9	77
horizontal filter, 3 components YUV422	8.7	154
vertical filter, 1 component	0.4	87
vertical filter, 3 components YUV 4:2:2	1.2	174
yuv to rgb8a, pci output	13.9	82
yuv to rgb15a, pci output	13.8	82
yuv to rgb24, pci output	13.7	82
yuv to rgb24a, pci output	14.0	82
yuv to rgb8a, sdram output	15.8	115
yuv to rgb15a, sdram output	18.5	151
yuv to rgb24, sdram output	17.5	187
yuv to rgb24a, sdram output	16.6	233
yuv to rgb8a, bitmask, pci output	14.3	91
yuv to rgb8a, rgbl15a overlay, pci output	20.7	153
yuv to rgb8a, rgbl24a overlay, pci output	21.6	232
yuv to rgb8a, yuv422a overlay, pci output	20.8	153
yuv to rgb8a, 422sequencing, pci output	14.0	80

14.6.6 Priority Delay and ICP Minimum Bus Bandwidth

The Priority Delay field in the Status register sets the time the ICP will wait for SDRAM service before changing from a low-priority bus request to a high-priority request. The ICP normally requests SDRAM bus service at the lowest-priority level, since it is a background processing device. In some cases, service to the ICP could be continuously delayed by other background devices, such as the VLD processor or by high-priority requests from the DSPCPU.

The PD field sets a timer on the currently active bus request. The timer is loaded with the PD value and started each time a bus request is submitted. The timer is incremented once each block time, the time required to load one block of 64 bytes. If the timer reaches 16 before the request is serviced, the ICP changes its bus request priority from low to high.

The resulting time delay until the ICP changes to high priority is:

timer delay = (16 - PD)*(block time)

One block time is 16 clock cycles.

Table 14-11 gives the delay in block times as a function of the PD field.

Table 14-11. ICP priority delay vs. PD code

PD Code	Delay block times
1111	1
1110	2
1101	3
1100	4
1011	5
1010	6
1001	7
1000	8
0111	9
0110	10
0101	11
0100	12
0011	13
0010	14
0001	15
0000	16

The priority delay mechanism in interaction with the arbiter mechanism allows the user to allocate enough bandwidth for the ICP to do its processing in the required frame time. For details of the arbiter mechanism see Chapter 20.

14.6.7 ICP Parameter Tables

Each microprogram in the microprogram set has an associated parameter table used by the ICP to process the image data, such as the image input and output start addresses, scaling factor, etc. The DP points to the location in SDRAM of the first word of the parameter table. The parameter table address must be word aligned. The parameter table can be more than one SDRAM block (16 32-bit words) long.

Note: In packed RGB24 to PCI operation the output address offset from the start of video memory must be a multiple of 6 bytes, i.e. on an even pixel boundary.

14.6.8 Load Coefficients

This routine loads the filter coefficient RAMs with coefficient data in the parameter table. A total of 32 sets of five 10-bit coefficients are loaded. Each set of five coefficients forms a 50-bit coefficient word. Two coefficients are stored in each 32-bit word in SDRAM. Three 32-bit words are used for each set of five coefficients that form a coefficient word. The parameter table is 96 words (6 SDRAM blocks) long. Each coefficient is stored as the 10 LSBs of each 16-bit half word of the 32-bit word.

The parameter table for the coefficient load function contains the coefficient data directly, as shown below. The parameter table is 96 words long.

Table 14-12. Load coefficients parameter table

Parameter Word		Description
Upper 2 bytes	Lower 2 bytes	
a+2	a+1	RAM Coefficient word 0
a+0	a-1	
a-2	0	
a+2	a+1	RAM Coefficient word 1
a+0	a-1	
a-2	0	
a+2	a+1	RAM Coefficient word 31
a+0	a-1	
a-2	0	

14.6.9 Horizontal Filter - SDRAM to SDRAM

This routine performs horizontal scaling and filtering of one component (Y, U or V) of an N x M image from one location in SDRAM to another.

14.6.9.1 Algorithms

The routine reads image data from SDRAM using the Y address counter, then scales and filters the data in the horizontal direction and writes it back to the SDRAM using the Z address counter. The 5-tap filter scales and filters the data. The LSB Increment value supplied by the parameter table determines the scaling. The routine reads and writes a line at a time until the full image is transferred. The filter mirrors the ends of each line to provide the extra pixels needed by the filter at the ends of each line.

14.6.9.2 Parameter table

The parameter table, shown in Table 14-13, supplies the input and output starting addresses and offsets, the image height in lines and width in pixels, and the increment value, which is derived from the scale factor.

The input and output addresses are the byte addresses of their respective tables. They do not need to be word- or block-aligned.

The input and output line offsets define the difference in bytes from the address of the first pixel in the first line to the address of the first pixel in the second line for their respective blocks. The line offset must be constant for all lines in each table. The line offset allows some space between the end of one line and the start of the next line. It also allows the ICP to scale and filter a subset of an existing image, such as magnifying a portion of an image. There are no restrictions on line offset values other than they must be 16-bit, two's complement integer values. (Note that this allows negative offsets. You can use this to flip an image vertically.)

The input and output image height and width values are the height in lines and width in pixels per line for their re-

Table 14-13. Horizontal filter parameter table

Parameter Word		Description
Upper 2 bytes	Lower 2 bytes	
Input image start address		Start address of X0Y0 (byte address)
Y counter Start fraction	Input image Line offset	Starting value: may be 0.5, etc. for interspersed convert; Line offset from X0Y0 to X0Y1
Fraction increment	Integer increment	Increment value for Y = 1/scale factor
Input image height	Input image Width	Height and width in input lines and pixels
Output image start address		Start address of X0Y0 (byte address)
Control	Output Image Line offset	Control bits; Line offset from X0Y0 to X0Y1
Output image height	Output image width	Height and width in output lines and pixels

spective images. The height and width are 16-bit positive binary numbers between 0 and 64K-1.

The Integer increment and Fraction increment values are the scaling parameters. The Integer value is a 16-bit integer, and the Fraction value is a positive binary fraction between 0 and 0.99999+. For up scaling (output image bigger), the increment value is the inverse of the scaling value. If you are upscaling by a factor of 2.5, the increment value will be the inverse of 2.50 = 0.40. The Integer increment value will be 0 and the Fraction increment value will be 0.40. For down scaling, the increment value is equal to the scaling value. If you are down scaling by 2.5 (output image smaller), the Integer increment value will be 2, and the Fraction increment value will be 0.500.

To perform scaling, the Integer and Fractional increment values must be generated and placed in the parameter table. The simplest way to generate these values in common computer languages such as C is as follows:

1. Generate the Increment Value as a floating point number = Input Width / Output Width
2. Multiply the Increment Value by 65536
3. Convert the result to a Long Integer (32 bits). The upper 16 bits of the Long integer will be the Integer increment value, and the lower 16 bits will be the Fractional value.
4. Store the 32-bit Long integer in the parameter table as the combined Integer and Fractional increment values.

The Start Fraction defines the starting value in the scaling counter for each line. It is a 16-bit, two's complement fractional value between -0.500 and +0.49999. The Start Fraction allows the input data to be offset by up to half a pixel, referred to the input pixel grid. It is '0' for Y and for UV co-sited data, and set to '-0.25' (C000h) for interspersed to co-sited conversion of U and V data. The '-0.25' value effectively shifts the U and V data toward the start of the line by 1/4 pixel, the amount required for conversion.

14.6.9.3 Control word format

The Control word provides bit fields which affect the horizontal filtering operation. The format of the Control word is as follows.

Bit	Name	Function
15	Bypass	Bypass filter. Picks nearest input pixel and passes it to output unfiltered. When Bypass is set & scale factor is 1.0, this results in an image block move
9	GETB	Large down-scaling bit. Picks nearest input pixels and passes them to filter. Equivalent to bypass + 5-tap filter of output pixels. LSB value = 0 for filtering.

The Bypass bit causes the data to bypass the 5-tap filter. The scaling operation selects the center pixel, and this pixel is passed to the filter output. No filtering or interpolation is provided. If the scaling factor is '1.0', the result is an image block move where the image is moved from one part of SDRAM to another without modification. If the scaling factor is other than '1.0', the effective algorithm is pixel picking, where the input pixel nearest the output pixel location is used as the output pixel.

The GETB bit is an optional bit for large (> 4) down scaling. When GETB is '0' (normal operation), the 5-tap filter receives the pixel nearest the output pixel as its center pixel plus the two adjacent input pixels on either side of this pixel to form the five filter inputs. When GETB is set, the filter receives the pixel nearest the output pixel as its center pixel plus the two pixels nearest the adjacent output pixels on either side of this pixel to form the five filter inputs. The effective algorithm is pixel picking plus 5-tap filtering of the result. GETB also forces the scaling LSB value to '0', since output pixels are being filtered and no

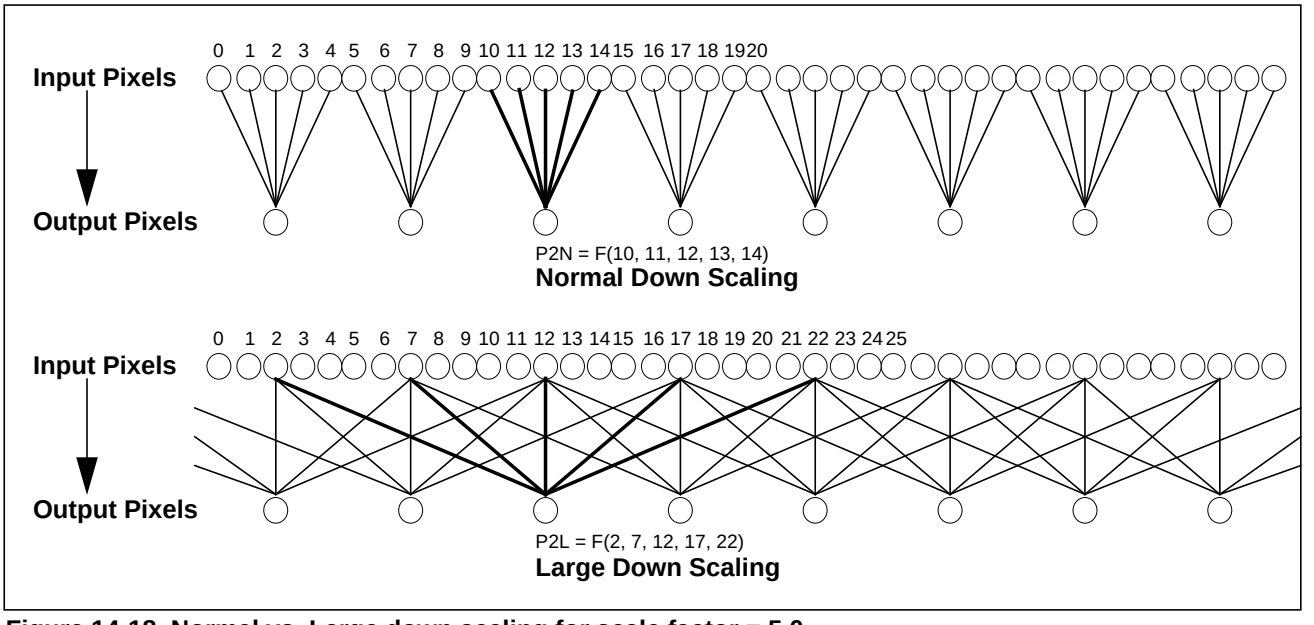


Figure 14-18. Normal vs. Large down scaling for scale factor = 5.0

interpolation is used. (See Section 14.5.2, “Filtering”) This is shown in Figure 14-18.

14.6.10 Vertical Filter - SDRAM to SDRAM

This routine performs vertical scaling and filtering of one component (Y, U or V) of an N x M image from one location in SDRAM to another.

14.6.10.1 Algorithms

The routine reads image data from SDRAM using the Y address counter, scales and filters the data in the vertical direction, and writes it back to the SDRAM using the Z address counter. The 5-tap filter scales and filters the data. The U LSB register is used as the scaling coefficient register. The U LSB Increment value supplied by the parameter table determines the scaling. Lines at the top and bottom of the image are mirrored to provide the extra line data needed by the 5-tap filter.

The routine reads and writes data in 64-byte (one SDRAM block) columns of pixels until the entire image is transferred. For each column, line segments of 64 pixels are processed until the entire column has been processed. Each 64-pixel line segment generated requires five vertically adjacent 64-pixel line segments as input to the 5-tap filter. The routine processes the image in pixel columns to eliminate redundant read of input pixel data: each new line segment typically requires reading only one new 64 byte line segment.

The routine processes data in 64-pixel blocks, corresponding to the input block buffer sizes. Five buffers are used in processing the current line segment, while the sixth buffer reads in the next line segment in overlap with current processing.

14.6.10.2 Parameter table

The parameter table, as shown in Figure 14-19, supplies the input and output starting addresses and offsets, the image height in lines and width in pixels, and the scale factor.

Figure 14-19. Vertical filter parameter table

Parameter Word		Description
Upper 2 bytes	**Lower 2 bytes**	
Input image start address		Start address of X0Y0 (byte address)
U counter Start fraction	Input image Line offset	Starting value: may be 0.5, etc. for interspersed convert; Line offset from X0Y0 to X0Y1
Fraction increment	Integer increment	Increment value for U = 1/scale factor
Input image height	Input image width	Height and width in input lines and pixels
Output image start address		Start address of X0Y0 (byte address)
Control	Output image Line offset	Control Word; Line offset from X0Y0 to X0Y1
Output image height	Output Image Width	Height and width in output lines and pixels

The input and output addresses are the byte addresses of their respective tables. The input and the output address need to be 64-byte aligned.

The input and output line offsets define the difference in bytes from the address of the first pixel in the first line to the address of the first pixel in the second line for their respective blocks. The line offset must be constant for all lines in each table. It allows some space between the end of one line and the start of the next line. It also allows the ICP to scale and filter a subset of an existing image, such as magnifying a portion of an image. Offset values are 16-bit, two's complement integer values.

Vertical filtering has a restriction on input and output line offset values: they must be positive, and they must be multiples of 64. Note that this only applies to the line-to-line spacing. Even with this restriction, input images may be any height and any width and may start at any byte address. Also, image subsets of arbitrary height and width can be used. As long as the original image has a line offset which is a multiple of 64, all subsets of that image will also automatically have a line offset, which is a multiple of 64 - the same as the original image. All images should have line offsets which are multiples of 64 as good programming practice, even though this restriction only applies to vertical filtering. If an image does not have a multiple of 64 line offset, it can be converted to that by using horizontal filtering in the image block move mode with the output offset value being a multiple of 64.

The input and output image height and width values are the height in lines and width in pixels per line for their respective images. The height and width are 16-bit positive binary numbers between 0 and 64K-1.

The Integer increment and Fraction increment values are the scaling parameters. The Integer value is a 16-bit integer, and the Fraction value is a positive binary fraction between 0 and 0.99999+. For up scaling (output image bigger), the increment value is the inverse of the scaling value. If you are upscaling by a factor of 2.5, the increment value will be the inverse of 2.50 = 0.40. The Integer increment value will be 0 and the Fraction increment value will be 0.40. For down scaling, the increment value is equal to the scaling value. If you are down scaling by 2.5 (output image smaller), the Integer increment value will be 2, and the Fraction increment value will be 0.500.

To perform scaling, the Integer and Fractional increment values must be generated and placed in the parameter table. The simplest way to generate these values in common computer languages such as C is as follows:

1. Generate the Increment Value as a floating point number = Input Height / Output Height
2. Multiply the Increment Value by 65536
3. Convert the result to a Long Integer (32 bits). The upper 16 bits of the Long integer will be the Integer increment value, and the lower 16 bits will be the Fractional value.
4. Store the 32-bit Long integer in the parameter table as the combined Integer and Fractional increment values.

The Start Fraction defines the starting value in the scaling counter for each line. It is a 16-bit, two's complement fractional value between -0.500 and 0.49999+. This value is placed in the Start Fraction allows the input data to be offset by up to half a line, referred to the input pixel grid. It is set to '0' for all conventional YUV input data.

14.6.10.3 Control word format

The Control word provides bit fields which affect the vertical filtering operation. The format of the Control word is as follows.

Bit	Name	Function
15	Bypass	Bypass filter. Picks nearest input line and passes it to output unfiltered. When Bypass is set & scale factor is 1.0, this results in an image block move

The Bypass bit causes the data to bypass the 5-tap filter. The scaling operation selects the center line, and this line is passed to the filter output. No filtering or interpolation is provided. If the scaling factor is 1.0, the result is an image block move where the image is moved from one part of SDRAM to another without modification. If the scaling factor is other than 1.0, the effective algorithm is line picking, where the input line nearest the output line location is used as the output line.

14.6.11 Horizontal Filter with RGB/YUV Conversion to PCI or SDRAM

This routine moves an N x M image in YUV 4:2:2, YUV 4:2:0 or YUV 4:1:1 format from SDRAM to the PCI bus or to SDRAM. The image is scaled and filtered in the horizontal direction during the move. Optional bit masking and/or RGB overlay can be used during the move when PCI output is specified.

14.6.11.1 Algorithms

The routine reads image data from SDRAM using the Y, U, and V address counters, scales and filters the data in the horizontal direction and writes it to the PCI interface or SDRAM. The 5-tap filter scales and filters the data. The LSB Increment value for each of the Y, U and V components supplied by the parameter table determines the scaling. Separate scaling factors allows YUV 4:2:2 interspersed to co-sited transformation as the data is being filtered. The scaled and filtered data is converted to RGB or YUV format before being sent to the PCI interface or to SDRAM. In the PCI output case, overlay data with alpha blending and chroma keying can be added to the output image, and the output image can be gated by a bit mask before it is sent to the PCI interface.

The routine reads and writes a line at a time until the full image is transferred. The filter mirrors the ends of each line to provide the extra pixels needed by the filter at the ends of each line.

14.6.11.2 Parameter table

The parameter table, shown in Table 14-14, supplies the input and output starting addresses and offsets for Y, U, V, OL, B and Z, the image height in lines and width in pixels, and the scale factors for each component.

The input and output addresses are the byte addresses of their respective tables. They do not need to be word or block aligned. Note the following restriction: in packed RGB24 to PCI operation the output address offset from the start of video memory must be a multiple of 6 bytes, i.e. on an even pixel boundary.

The input and output line offsets define the difference in bytes from the address of the first pixel in the first line to the address of the first pixel in the second line for their respective blocks. The line offset must be constant for all lines in each table. The line offset allows some space between the end of one line and the start of the next line. It also allows the ICP to scale and filter a subset of an existing image, such as magnifying a portion of an image. There are no restrictions on line offset values other than they must be 16-bit, two's complement integer values. (Note that this allows negative offsets. You can use this to flip an image vertically.)

The input and output image height and width values are the height in lines and width in pixels per line for their respective images. The height and width are 16-bit positive binary numbers between 0 and 64K-1.

Table 14-14. Horizontal filter to RGB output parameter table

Parameter Word		Description
Upper 2 bytes	**Lower 2 bytes**	
Input image Y start address		Y Start address of X0Y0 (byte address)
Y Counter Start fraction	Input image Y line offset	Starting value: may be 0.5, etc. for interspersed convert; Y Line offset from X0Y0 to X0Y1
Y fraction increment	Y integer increment	Increment value for U = 1/scale factor
Y input image height	Y input image width	Y Height and width in pixels
Input image U start address		U Start address of X0Y0 (byte address)
U counter Start fraction	Input image U line offset	Starting value: may be 0.5, etc. for interspersed convert; U Line offset from X0Y0 to X0Y1
U fraction increment	U integer increment	Increment value for Y = 1/scale factor
U input image height	U input image Width	U Height and width in pixels
Input image V start address		V Start address of X0Y0 (byte address)
V Counter Start fraction	Input image V line offset	Starting value: may be 0.5, etc. for interspersed convert; V Line offset from X0Y0 to X0Y1
V fraction increment	V integer increment	Increment value for V = 1/scale factor
V Input image height	V input image width	V Height and width in pixels
Output image start address		Start address of X0Y0 (byte address)
Control	Output image Line offset	Input & output formats & control bits; Line offset from X0Y0 to X0Y1
Output image height	Output image width	Height and width in output pixels
Bit Map image start address		Start address of X0Y0 (byte address)
0	Bit map image Line offset	Line offset from X0Y0 to X0Y1
RGB overlay start address		Start address of X0Y0 (byte address)
Alpha 1 & Alpha 0	Overlay Line offset	Alpha 1 & Alpha 0 blend code for RGB15+α, etc.; Line offset from X0Y0 to X0Y1
Overlay end pixel	Overlay start pixel	Start and end pixels along line
Overlay end Line	Overlay start line	Start and end lines in frame

The Integer increment and Fraction increment values are the scaling parameters. There is a separate scaling parameter for each of the Y, U and V input components. The Integer value is a 16-bit integer, and the Fraction value is a positive binary fraction between 0 and 0.99999+. For up scaling (output image bigger), the increment value is the inverse of the scaling value. If upscaling by a factor of 2.5, the increment value will be the inverse of 2.50 = 0.40. The Integer increment value will be '0' and the Fraction increment value will be '0.40'. For down scaling, the increment value is equal to the scaling value. If you are down scaling by 2.5 (output image smaller), the Integer increment value will be '2', and the Fraction increment value will be '0.500'.

To perform scaling, the Integer and Fractional increment values must be generated and placed in the parameter

table. The simplest way to generate these values in common computer languages such as C is as follows:

1. Generate the Increment Value as a floating point number = Input Width / Output Width
2. Multiply the Increment Value by 65536
3. Convert the result to a Long Integer (32 bits). The upper 16 bits of the Long integer will be the Integer increment value, and the lower 16 bits will be the Fractional value
4. Store the 32-bit Long integer in the parameter table as the combined Integer and Fractional increment values

For YUV 4:2:2 or YUV 4:2:0 input data and RGB output data, the scaling factor for U and V must be twice the scaling factor for Y, unless YUV4:2:2 sequencing is used for speed. In YUV 4:2:2 or YUV 4:2:0 data, the horizontal components of U and V are half those of Y. The U and V must be upscaled by 2 to generate a YUV 4:4:4 format internally for YUV to RGB conversion. For YUV 4:1:1 input data, the U and V components must be upscaled by a factor of 4 to generate the required internal YUV 4:4:4 format.

The Start Fraction defines the starting value in the scaling counter for each line. It is a 16-bit, two's complement fractional value between -0.500 and 0.49999+. The Start Fraction allows the input data to be offset by up to half a pixel, referred to the input pixel grid. It is '0' for Y and for UV co-sited data, and is set to '-0.25' (C000) for interspersed to co-sited conversion of U and V data. The '-0.25' value effectively shifts the U and V data toward the start of the line by 1/4 pixel, the amount required for conversion.

The Alpha 1 and Alpha 0 values are 8-bit fields within the 16-bit Alpha field. These values are loaded into the Alpha 1 and Alpha 0 registers, resp., for use by RGB 15+α and YUV 4:2:2+α overlay formats in alpha blending.

The Overlay start and end pixels and lines define the start and end pixels and lines within the output image for the overlay. The first pixel of the overlay image will be blended with the pixel at the Overlay Start Pixel and Overlay Start Line in the output image.

14.6.11.3 Control word format

The Control word provides bit fields which affect the horizontal filtering operation. The format of the Control word is as follows.

Bits	Name	Function
15	Bypass	Normally set to 0 to enable filtering. Can be set to 1 to accomplish data move without filtering.
14	422SEQ	4:2:2 Sequence bit. Used with YUV 4:2:2 output
13	YUV420	YUV 4:2:0 input format
12	OEN	Overlay enable. Valid only for PCI output
11	PCI	PCI output enable. Otherwise SDRAM output
10	BEN	Bit mask enable. Valid only for PCI output
9	GETB	Large down scaling bit. Picks five input pixels nearest 5 output pixels and passes to filter. Equivalent to filter bypass + 5-tap filter of output pixels. LSB value = 0 for filtering.
8	OLLE	Overlay little endian enable
7-6	OFRM	Overlay format 0 = RGB 24+α 1 = RGB 15+α 2 = YUV 4:2:2+α
5	CHK	Chroma keying enable
4	LE	RGB output little endian enable
3-0	RGB	RGB Output Code 0 = YUV 4:2:2+α 1 = YUV 4:2:2 2 = RGB 24+α 3 = RGB 24 packed 4 = RGB 8A (RGB 233) 5 = RGB 8R (RGB 332) 6 = RGB15+α 7 = RGB 16

The 422SEQ bit controls the internal sequencing of the YUV to RGB operation. It is set to '1' when YUV 4:2:2 output is selected. When 422SEQ is '0', normal RGB output is assumed. In this mode, the input is YUV 4:2:2 or YUV 4:2:0, and the output is RGB. To generate the RGB output, the YUV 4:2:2 or YUV 4:2:0 input must be upscaled to YUV 4:4:4 before conversion to RGB. This means the scaling factor for U and V must be twice the scaling factor for Y. The internal sequencing of the filter in this case is UVY, UVY, UVY to generate RGB, RGB, RGB. For YUV 4:2:2 output formats, no upscaling of U and V is required. In this case, the 422SEQ bit is set to one, and the filter sequence is UVYY, UVYY, UVYY.

The 422SEQ bit can be set in RGB output mode to decrease the processing time for the image at the expense of color bandwidth and some corresponding decrease in picture quality. If the 422SEQ bit is set for RGB output, the filter will perform the UVYY sequence. In this case, the U and V components are not upscaled by 2, and the YUV to RGB converter updates its U and V components every other pixel. In the normal case (422SEQ=0), it takes 6 clock cycles to generate two RGB pixels. In the 422SEQ=1 case, it takes 4 clock cycles to generate two RGB pixels, reducing processing time by 33%.

The YUV420 bit indicates that the input data is in YUV 4:2:0 format. In YUV 4:2:0 format, the U and V components are half the width and half the height of the Y data. YUV 4:2:0 data is normally converted to YUV 4:2:2 data by a separate vertical upscaling by a factor of 2.0 for best quality. The YUV420 bit allows using YUV 4:2:0 data directly but with some quality degradation. When YUV420 is set, the ICP up scales the data vertically by line duplication. Each U and V input line is used twice. The sepa-

rate vertical scaling step is eliminated at the expense of some quality since the lines are simply duplicated rather than being fully scaled and filtered.

The OEN bit enables overlay. Set it to ‘1’ if an overlay is used, ‘0’ if not. Overlays are only valid for PCI output.

The PCI bit selects PCI as the output port for the ICP data. A ‘1’ selects PCI output; a ‘0’ selects SDRAM output.

The BEN bit enables bit masking. Set it to ‘1’ if bit masking is used, ‘0’ if not. Bit masking is only valid for PCI output.

The GETB bit is an optional bit for large (> 4) down scaling. When GETB is ‘0’ (normal operation), the 5-tap filter receives the pixel nearest the output pixel as its center pixel plus the two adjacent input pixels on either side of this pixel to form the five filter inputs. When GETB is set, the filter receives the pixel nearest the output pixel as its center pixel plus the two adjacent output pixels on either side of this pixel to form the five filter inputs. The effective algorithm is pixel picking plus 5-tap filtering of the result. GETB also forces the scaling LSB value to ‘0’, since output pixels are being filtered and no interpolation is used.

The OFRM bit field selects the overlay data format, as shown in the Control word format list.

The CHK bit enables chroma keying. Set it to ‘1’ if chroma keying is used, ‘0’ if not.

The OLLE bit sets the endian-ness of the overlay data input. Set it to ‘1’ if the overlay data is little-endian, ‘0’ if big endian. This bit is normally set to the same value as the LE bit in the Status register.

The LE bit sets the endian-ness of the RGB/YUV output data. Set it to ‘1’ if the output data is little-endian, ‘0’ if big endian. The LE bit is normally set to the same value as the LE bit in the Status register.

The RGB field defines the output data format, as shown in the Control word format list.

Important Note: The ICP DMA Enable bit (IE) in the BIU_CTL register of the PCI interface must be set for RGB output to PCI. This bit must be set before initiating RGB to PCI operations, or the ICP will stall waiting for the PCI to become ready.

Variable Length Decoder Chapter 15

by Gene Pinkston and Selliah Rathnam

15.1 VLD OVERVIEW

The variable length decoder (VLD) unit Huffman-decodes MPEG-1 and MPEG-2 (Main Profile) video bitstreams[1-3]. This chapter describes a programmers view of the VLD.

The VLD reads an MPEG stream from SDRAM, decodes the bitstream under the control of DSPCPU and outputs two data streams. The output data streams contain macroblock header information and the run-length encoded DCT coefficients. The output data streams are stored in the SDRAM buffers.

The VLD unit, operates independently during the slice decoding process. The remaining decoding of the MPEG stream is carried out by the DSPCPU.

15.2 VLD OPERATION

Enabled by the DSPCPU, the VLD unit can be initialized by hardware or software reset operations. Hardware reset is provided by the external TRI_RESET# pin. Software reset is provided by one of the VLD commands. The DSPCPU controls the VLD through the VLD command register. There are five commands supported by the VLD:

- Shift the bitstream by some number of bits (a maximum of 15-bit shift)
- Search for the next start code
- Reset the VLD
- Parse some number of macroblocks
- Flush VLD output buffers to SDRAM

The normal mode of operation will be for the DSPCPU to request that the VLD to parse some number of macroblocks. Once the VLD has begun parsing macroblocks, it may stop for any one of the following reasons:

- The command was completed with no exceptions
- A start code was detected
- An error was encountered in the bitstream

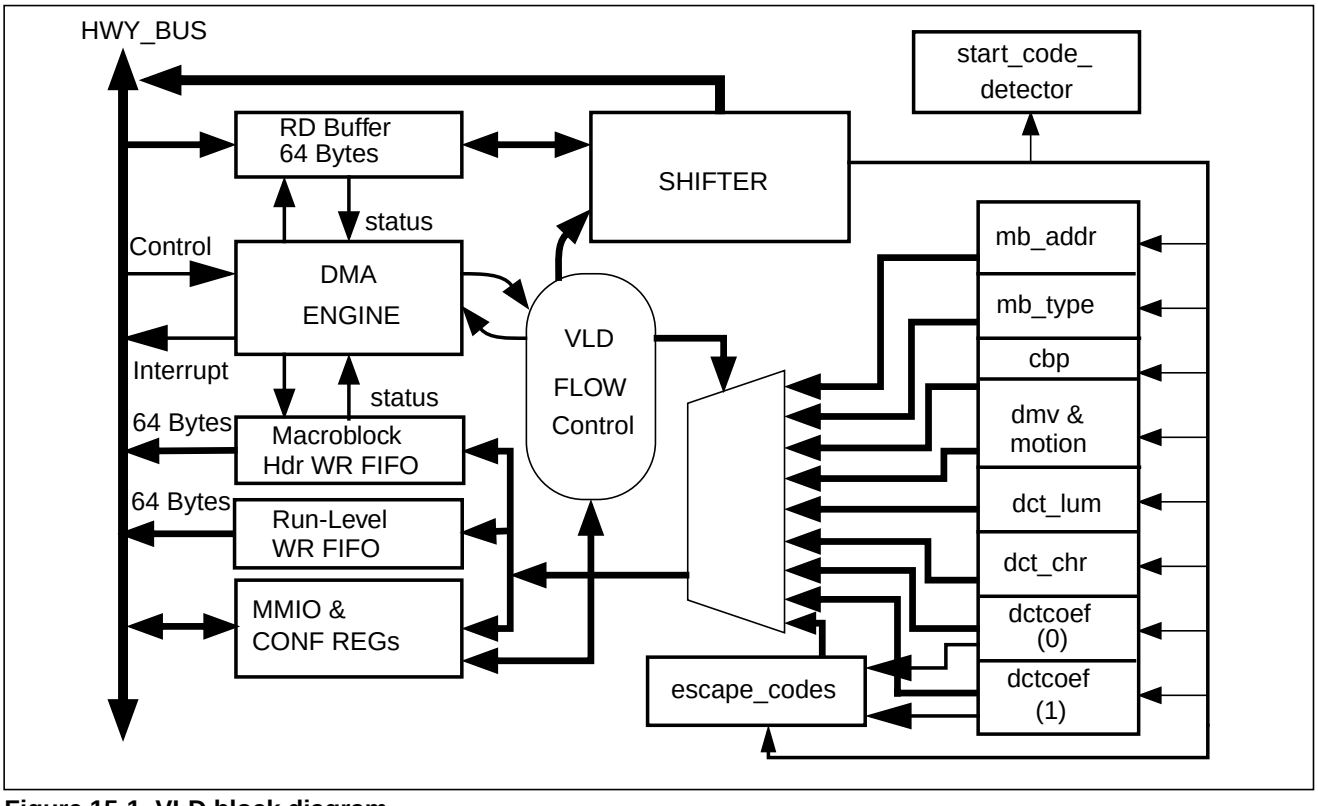


Figure 15-1. VLD block diagram

- The VLD input DMA completed, and the VLD is stalled waiting for more data
- One of the VLD output DMAs has completed and the VLD is stalled because the output FIFO is full

The DSPCPU can be interrupted whenever the VLD halts.

Consider the case in which the VLD has encountered a start code. At this point, the VLD will halt and set the status flag to indicate that a start code has been detected. This event will generate an interrupt to the DSPCPU (if corresponding interrupt is enabled). Upon entering the interrupt routine, the DSPCPU will read the VLD status register to determine the source of the interrupt. Once it has determined that a start code was encountered, the CPU will read 8 bits from the VLD shift register to determine the type of start code encountered. If it a 'slice' start code, the DSPCPU reads from the shift register the slice quantization scale and any extra slice information. The slice quantization scale is then written back to the VLD quantizer-scale register. Before exiting the interrupt routine, the DSPCPU will clear the start code detected status bit in the status register and issue a new command to process the remaining macroblocks.

15.3 DECODING UP TO A SLICE

MPEG decoding up to the slice layer is carried out by the DSPCPU and the VLD. The VLD is controlled by the DSPCPU for the decoding of all parameters up to the slice-start code. During this process, the DSPCPU reads from the VLD_SR register which contains the next 16 bits of the bitstream. The DSPCPU also issues shift commands to the VLD in order to advance the contents of the shift register by the specified number of bits. The DSPCPU may also command the VLD to advance to the next start code. Refer to Table 15-6 for a complete list of VLD commands and their functions. Once at the slice layer, the VLD operates independently for the entire slice decoding. The slice decoding starts once the DSPCPU issues a *parse* command.

15.4 VLD INPUT

Input to the VLD is controlled by the VLD input DMA engine. The input DMA engine is programmed by the DSPCPU to read from SDRAM. The DSPCPU programs this DMA engine by writing the address and the length of the SDRAM buffer containing the MPEG stream. The address of the buffer is written to the VLD_BIT_ADR register. The length, in bytes, of the buffer is written to the VLD_BIT_CNT register.

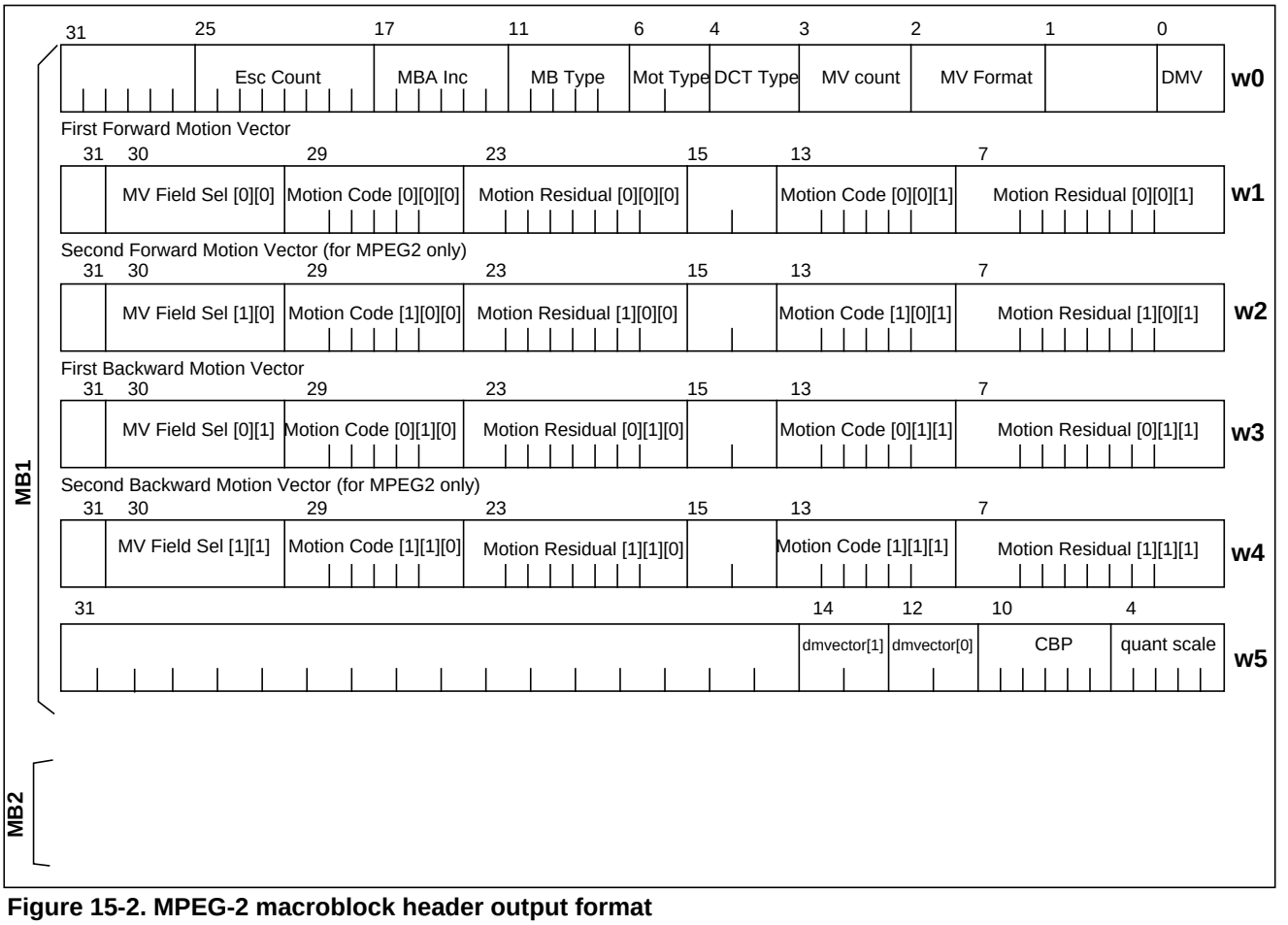


Figure 15-2. MPEG-2 macroblock header output format

The VLD reads data from SDRAM into an internal 64-byte FIFO. The VLD processing engine then reads data from the FIFO as needed. Once this internal FIFO is empty the VLD reads more data from SDRAM. The VLD_BIT_ADR and VLD_BIT_CNT registers are updated after each read from main memory. The content of the VLD_BIT_ADR register reflects the next address from which the bitstream data will be fetched. The content of the VLD_BIT_CNT register reflects the number of bytes remaining to be read before the current transfer is complete. When the number of bytes remaining to be read from SDRAM is zero, a status flag is set and an interrupt can be generated to the DSPCPU. The DSPCPU will provide the new bitstream buffer address and the number of bytes in the bitstream buffer to the VLD.

Table 15-1. References for the MPEG-2 macroblock header data

Item	Default value	References from MPEG-2 Video Standard, IS 13818-2 document
Esc count	0	Section 6.2.5
MBA inc	-	Section 6.2.5 and Table B-1
MB type	undefined	Section 6.2.5.1 and Tables B-2, B-3, and B-4; Only 5 Msb bits from the tables are used
Mot type	undefined	Section 6.2.5.1; Field or Frame motion type will be decided by the user
DCT type	undefined	Section 6.2.5.1
MV count	undefined	Tables 6-17 and 6-18. The MV Count value is one less than the value from the tables.
MV format	undefined	Tables 6-17 and 6-18
DMV	undefined	Tables 6-17 and 6-17
MV field Sel[0]0] to MV field Sel[1][1]	undefined	Section 6.2.5 and 6.2.5.2
Motion code[0][0][0] to Motion code[1][1][1]	undefined	Section 6.2.5.2.1 and Table B-10
Motion Residual[0][0][0] to Motion Residual[1][1][1]	undefined	Section 6.2.5.2.1; the corresponding rsize bits are extracted from the bitstream and stored as left justified; to get the final value shift the given number by 8 (corresponding rsize). The rsize values are stored in VLD_PI register
dmvector[1] and dmvector[0]	undefined	Section 6.2.5.2.1 and Table B-11; signed 2-bit integer from Table B11.
CBP	-	Section 6.2.5, 6.2.5.3 and Table B-9
Quant scale	-	Section 6.2.5; 5-bit from bitstream and use Table 7-6 to compute the quant scale value.

15.5 VLD OUTPUT

The VLD outputs two data streams which are written back to main memory by two output DMA engines. These DMA engines are programmed by the DSPCPU. One of the output streams contains macroblock header information and the other contains run-length encoded DCT coefficients. Each DMA engine contains a 64-byte FIFO which is transferred to main memory once it is full. The main memory address and count for the macroblock header output are contained in the VLD_MBH_ADR and VLD_MBH_CNT registers respectively. The main memory address and count for the DCT coefficient output are contained in the VLD_RL_ADR and VLD_RL_CNT registers respectively. The counts for both the macroblock header and coefficient data are expressed in terms of 32-bit (4 bytes) words.

15.5.1 Macroblock Header Output Data

For each MPEG-2 macroblock parsed by the VLD, six 32-bit words of macroblock header information will be output from the VLD. Figure 15-2 pictures the layout of the VLD output, the fields are described in Table 15-1. Note that these fields may or may not be valid depending upon the MPEG-2 video standard[2]. For example, motion vectors are not valid for intra coded macroblocks. Similarly, 'DCT Type' is not valid for field pictures.

For each MPEG-1 macroblock parsed by the VLD, four 32-bit words of macroblock header information will be output from the VLD. Figure 15-3 pictures the layout of the VLD output, while the fields are described in Table 15-2. Note that these fields may or may not be valid depending upon the MPEG-1 video standard[1].

Table 15-2. References for the MPEG-1 macroblock header data

Item	Default value	References from IS 11172-2 document
Esc count	0	Section 2.4.3.6
MBA inc	-	Section 2.4.3.6
MB type	undefined	Section 2.4.3.6 and Tables B-2a to B2d
Motion code[0][0][0] to Motion code[0][1][1]	undefined	Section 2.4.2.7 and Table B-4
Motion residual[0][0][0] to Motion residual[0][1][1]	undefined	Section 2.4.2.7;the corresponding rsize bits are extracted from the bitstream and stored as left justified; to get the final value shift the given number by (8 - corresponding rsize). The rsize values are stored in VLD_PI register.
CBP	-	Section 2.4.3.6 and Table B-3
Quant scale	-	Section 2.4.2.7

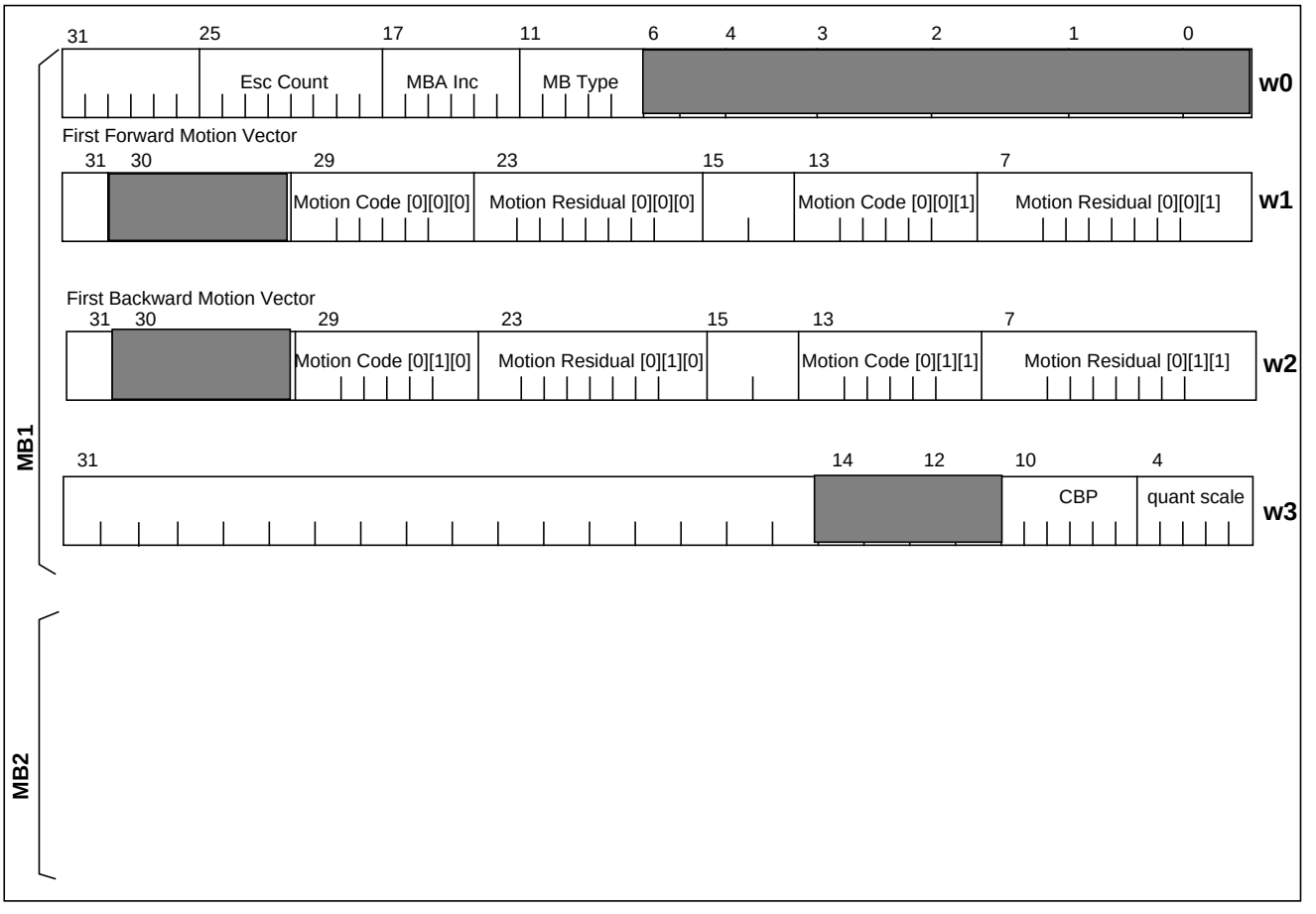


Figure 15-3. MPEG1 Macroblock Header Output Format

15.5.2 Run-Level Output Data

The DCT coefficients associated with the macroblock are output to a separate memory area and each DCT coefficient is represented as one 32-bit quantity (16 bits of run and 16 bits of level). For intra blocks, the DC term is expressed as 16 bits of DC size and a 16-bit value whose most significant bits (the number of bits used for DC level is determined by DC size) represent the DC level. Each block of DCT coefficients is terminated by a run value of '0xff'.

15.6 VLD TIME SHARING

The TM1300 VLD is targeted for a single bitstream decode and there is no provision to decode more than one bitstream at a time by using the VLD in time multiplexed mode. However internal development has shown that up to 4 simultaneous MPEG1 bitstreams can be decoded. This procedure is beyond the scope of this databook but can be discussed further by contacting customer support.

15.7 MMIO REGISTERS

To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as '0's.

15.7.1 VLD Status (VLD_STATUS)

This register contains the current status information most pertinent to the normal operation of an MPEG video decode application. VLD status description is detailed in Table 15-3 and pictured in Figure 15-4. Default value (after hardware reset) is '0'.

Interrupts can be enabled for any of the defined status bits (see following VLD_IMASK description). Acknowledgment of the interrupt is done by writing a '1' to the corresponding bit in VLD_STATUS register. Writing a one to the bits one through five clears the corresponding bits. However bit 0 (COMMAND_DONE) is cleared only by issuing a new command. Writing a '0' to bit 0 of the status register will result in undefined behavior of the VLD. Note that several status bits may be asserted simultaneously. Thus it is recommended to use level triggered interrupts (see Section 3.5.3.6 on page 3-11) and carefully acknowledge the interrupt.

15.7.2 VLD Interrupt Enable (VLD_IMASK)

This register allows the DSPCPU to control the initiation of the interrupt for the corresponding bits in the VLD Status Register. Writing a '1' into any of the defined VLD_IMASK bits enables the interrupt for the corresponding bit in the status register (VLD_STATUS). Default value (after hardware reset) is '0'.

Table 15-3. VLD_STATUS register

Name	Size (bits)	Description
COMMAND_DONE	1	Indicates successful completion of current command
STARTCODE	1	VLD encountered 0x000001 while executing *parse* or *next start code* command
ERROR	1	VLD encountered an illegal Huffman code or an unexpected start code
DMA_IN_DONE	1	DMA transfer of given SDRAM buffer has completed and VLD is stalled waiting on more main memory input data; DSPCPU is responsible to provide the new SDRAM buffer to VLD
MBH_OUT_DONE	1	Macroblock Header DMA transfer has completed
RL_OUT_DONE	1	Run-level DMA transfer complete

15.7.3 VLD Control (VLD_CTL)

The VLD_CTL register has one bit indicating the endianness of the VLD unit. Little-Endian = '1', Big-Endian = '0'. Default value (after hardware reset) is '0'.

Table 15-4. VLD control (R/W)

Name	Size (bits)	Description
Reserved	1	
Little Endian	1	Forces VLD to operate in Little Endian Mode when set to 1.

15.8 VLD DMA REGISTERS

There are one input DMA engine and two output DMA engines in the VLD block. Each of the three DMA engines (or channels) for the VLD is controlled by two MMIO registers. The address register always contains the address of the next SDRAM transaction. The count register always indicates the amount of data to be transferred to or from main memory. A DMA completes when its count reaches zero. Once a DMA count register becomes zero, a bit is set in the status register and the DSPCPU can be interrupted. The DSPCPU sets a non-zero value to a DMA count register to initiate a new DMA transaction. The input count register always contains number of bytes to be fetched from the main memory. The output count registers always contain the number of words (4 bytes) to be written to the main memory.

Note that both of the DMA output engines write only to 64-byte aligned addresses and they always write 64 bytes. When flushing the DMA output FIFOs there may not be 64 bytes of valid data at the time the flush command is received. In this case, 64 bytes are still written to the main memory. The valid bytes can be determined from the count register value before issuing the flush command. The valid data always resides in the first N bytes while the last 64-N bytes will contain random data and should be ignored.

15.8.1 DMA Input

The bitstream input to the VLD is controlled by VLD_BIT_ADR and VLD_BIT_CNT MMIO registers. VLD_BIT_ADR contains the main memory address for the next read from the main memory to the VLD input FIFO. VLD_BIT_CNT register contains the number of bytes remaining to be read before the current DMA is completed.

The VLD input address is byte aligned.

15.8.2 Macroblock Header Output DMA

The macroblock header output of the VLD is controlled by VLD_MBH_ADR and VLD_MBH_CNT registers. VLD_MBH_ADR contains the address of the next write of macroblock header data to the main memory. VLD_MBH_CNT contains the remaining number of words (4 bytes) to write before the current DMA expires.

The macroblock header output address is 64-byte aligned.

15.8.3 Run-Level Output DMA

The run-level output of the VLD is controlled by VLD_RL_ADR and VLD_RL_CNT. VLD_RL_ADR contains the address of the next write of macroblock header data to the main memory. VLD_RL_CNT contains the number of 4-byte writes remaining before the current DMA expires.

The run-level buffer address is 64-byte aligned.

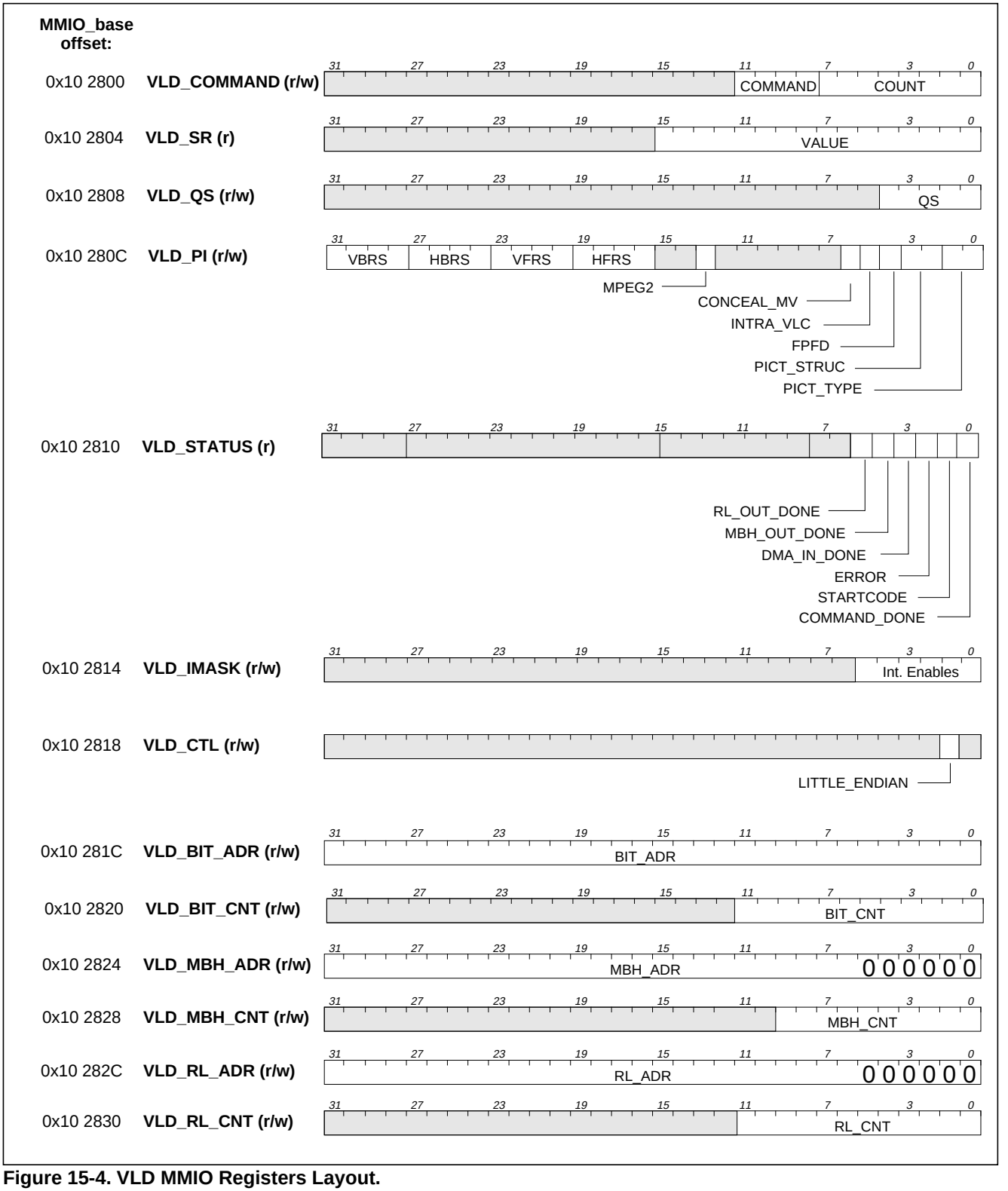


Figure 15-4. VLD MMIO Registers Layout.

15.9 VLD OPERATIONAL REGISTERS

15.9.1 VLD Command (VLD_COMMAND)

This register indicates the next action to be taken by the VLD. Some commands have an associated count which resides in the least significant 8 bits of this register. There are currently five commands recognized by the VLD block:

- Shift the bitstream by ‘count’ bits (‘count’ must be less than or equal to 15)
- Parse ‘count’ un-skipped macroblocks
- Search for the next start code
- Reset the VLD
- Flush the VLD output buffers

The DSPCPU must wait for the VLD to halt before the next command can be issued. Note that there are several ways in which a command may complete. Only a successful completion is indicated by the COMMAND_DONE bit in the status register. A command may complete unsuccessfully if a start code or an error is encountered before the requested number of items has been processed. Note also that expiration of a DMA count does not constitute completion of a command. When a DMA count expires the VLD is stalled as it waits for a new DMA to be initiated. It is not halted. Default value (after hardware reset) is ‘0’. VLD_COMMAND fields are described in Table 15-5 and the different commands explained in Table 15-6.

Table 15-5. VLD Command Register

Name	Size (bits)	Description
COUNT	8	Count for current command
COMMAND	4	VLD command to be executed

Table 15-6. VLD Commands

Command	Field coding	Flags Set after Completion of the Command	Description
Shift the bitstream by ‘count’ bits	1	COMMAND_DONE or DMA_IN_DONE	VLD shifts the number of bits in its internal shift register. The shift register value is available in the VLD_SR register. The DMA_IN_DONE flag will be set when VLD runs out of data from input FIFO. The flag is reset by issuing the new command.
Search for the next start code	3	STARTCODE or COMMAND_DONE or DMA_IN_DONE	VLD search for a start code. The search code has 0x000001 prefix and an additional 8-bit value. The DMA_IN_DONE flag will be set when VLD runs out of data from input FIFO. The STARTCODE detected flag is reset by writing a ‘1’ value to the flag. The COMMAND_DONE flag is reset by issuing the new command.
Reset the VLD	4	None	Refer section 15.12 for more details
Parse for a given number of macroblocks	2	COMMAND_DONE or STARTCODE or ERROR or DMA_IN_DONE	VLD parses for a given number of un-skipped macroblocks and the associated run-level values. COUNT will indicate the remaining macroblocks to parse. Note that this number is slightly inaccurate since a parsed macroblock can still be in internal 64-byte FIFO. If VLD encounters a start code, the parsing action will be terminated and VLD sets only the STARTCODE detected flag. If VLD parses the given number of un-skipped macroblocks without encountering a start code, VLD will set the COMMAND_DONE flag. The ERROR flag will be set when VLD encounters an error while parsing the bitstream. The DMA_IN_DONE flag will be set when VLD runs out of data from input FIFO. The STARTCODE detected flag is reset by writing a ‘1’ value to the flag. The COMMAND_DONE flag is reset by issuing the new command.
Flush the VLD output buffer	8	COMMAND_DONE	VLD flushes the remaining macroblock header data and the remaining run-level data to SDRAM. The highway byte-enables will be used in order to write only the valid data to SDRAM. Only the valid word count values written to SDRAM will be subtracted from the VLD_MBH_CNT and the VLD_RL_CNT registers.

15.9.2 VLD Shift Register (VLD_SR)

This read only register is a shadow of the VLD’s operational shift register. Tt allows the DSPCPU to access the bitstream through the VLD. Bits 0 through 15 are the current contents of the VLD shift register. Bits 16 to 31 are RESERVED and should be treated as undefined by the programmer.

15.9.3 VLD Quantizer Scale (VLD_QS)

This 5-bit register contains the quantization scale code (from the slice header) to be output by the VLD until it is overridden by a macroblock quantizer scale code. The quantizer scale code is part of the macroblock header output.

15.9.4 VLD Picture Info (VLD_PI)

This 32-bit register contains the picture layer information necessary for the VLD to parse the macroblocks within that picture. Again, the values for each of these fields are determined by the appropriate standard (MPEG [1-3]).

Table 15-7. VLD picture info register (r/w)

Name	Size (bits)	Description
PICT_TYPE (picture type)	2	I, P, or B picture
PICT_STRUC (picture structure)	2	field or frame picture
FPFD (frame prediction frame dct)	1	specifies that this picture uses only frame prediction and frame dct
INTRA_VLC	1	Use DCT table zero or one
CONCEAL_MV	1	concealment vectors present in the bitstream
reserved	6	Reserved for future expansion
MPEG2 mode	1	Switches VLD between MPEG-1 and MPEG-2 decoding. Value '1' = MPEG-2 mode
reserved	2	reserved
HFRS (horizontal forward rsize)	4	size of residual motion vector
VFRS (vertical forward rsize)	4	size of residual motion vector
HBRS (horizontal backward rsize)	4	size of residual motion vector
VBRS (vertical backward rsize)	4	size of residual motion vector

15.10 ERROR HANDLING

Upon encountering a bitstream error, the VLD will set the bitstream-error flag (ERROR) in the VLD_STATUS register and interrupt the DSPCPU, if the interrupt is enabled. Note that if a start code is present (in the VLD shift register) when an error is detected, then both the start code and the error bits will be set. A separate flush command is required to flush any valid data in the run-level and macroblock header output buffers.

The DSPCPU de-asserts the ERROR flags by writing a '1' to the ERROR flag.

15.11 INTERRUPT

The interrupt source number for the VLD is 14 and it should be set in level sensitive mode (see Section 3.5.3.6 on page 3-11).

15.12 RESET

The VLD block is reset by a hardware reset or a software reset. The hardware reset signal is generated from the external pin TRI_RESET#. The software reset is initiated by writing a 'Reset VLD' command in the VLD_COMMAND register. Refer Table 15-8 for the details on the software reset procedure.

Table 15-8. Software reset procedure

Cycle no.	Action	Remarks
i	DSPCPU issues the 'Reset the VLD' command by writing the required value in the VLD_COMMAND register.	
i to j	VLD will complete the pending, if any, highway transactions.	Any highway transactions, once started, will not be aborted in the middle
j+1	VLD will perform the full reset.	All status and control registers are reset and all the buffers are made empty. MMIO Registers initialized to zero includes VLD_STATUS.

15.13 ENDIAN-NESS

VLD supports little-endian and big-endian modes of operations. Refer to Appendix C for the endian-ness specification of the VLD input and output data.

15.14 POWER DOWN

The VLD block can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. For a description of powerdown, see Chapter 21, "Power Management."

The VLD block should not be active when applying block powerdown.

If the block enters power-down state while it is enabled, its behavior upon power-up is undefined.

15.15 REFERENCES

[1] ISO/IEC IS 13818-2, International Standard (1994), MPEG-2 Video.

[2] ISO/IEC IS 11172-2, International Standard (1992), MPEG-1 Video.

[3] MPEG Video Compression Standard, by Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, Didier J. LeGall; ITP publication.

I²C Interface

Chapter 16

by Essam Abu-ghoush, Robert Nichols

16.1 I²C OVERVIEW

TM1300 includes an I²C interface which can be used to control many different multimedia devices such as:

- DMSDs - Digital multi-standard decoders
- DENCs - Digital encoders
- Digital cameras
- I²C - Parallel I/O expanders

The key features of the I²C interface are:

- Supports I²C single master mode
- I²C data rate up to 400 kbits/sec
- Support for the 7-bit addressing option of the I²C specification
- Provisions for full software use of I²C interface pins for implementing software I²C or similar protocols

Note that the I²C pins are also used to load the initial boot parameters and/or code from a serial EEPROM as described in Section 13, “System Boot”. The boot logic is only active upon TM1300 hardware reset and quiescent afterwards.

A typical system using the I²C interface is presented in Figure 16-1. The TM1300 is connected as a master to a series of slave devices through SCL and SDA. Note that the bus has one pullup resistor for each of the clock and data lines. The pullup should be set to a voltage no higher than VREF_PERIPH.

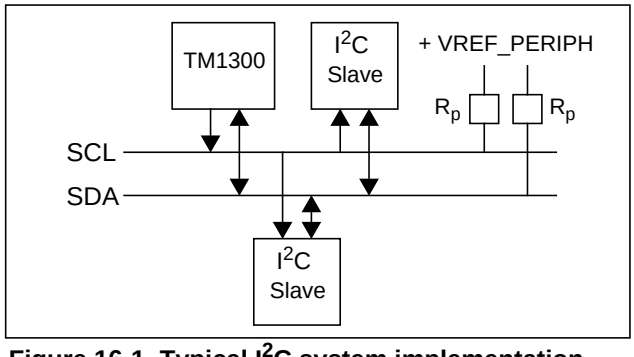


Figure 16-1. Typical I²C system implementation

16.2 NEW IN TM1300

The following are the main I²C differences from TM1000:

- The SEX bit is removed. Endian-ness is fixed.
- The I²C clock rate is closer to 100/400 kHz
- The GDI bit now correctly indicates write-completion
- Clock stretching is always enabled.

16.3 EXTERNAL INTERFACE

The I²C external interface is composed of two signals as shown in Table 16-1.

Table 16-1. I²C External interface

Signal	Type	Description
IIC_SDA	I/O	I²C serial data
IIC_SCL	O	I²C clock

16.4 I²C REGISTER SET

The I²C user interface consists of four registers visible to the programmer. The registers are mapped into the MMIO address space and are fully accessible to the programmer. Figure 16-2 shows the I²C register set. To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as ‘0’s.

16.4.1 IIC_AR Register

The IIC_AR is the I²C address register and is used in both master receive and transmit modes. This register is written with the address(es) of the I²C slave device and the bytecount for transmit/receive. Table 16-2 lists the bitfield definitions for the IIC_AR register.

Table 16-2. IIC_AR Register

Bits	Field Name	Definition
31:25	ADDRESS	7-bit slave device address.
24	DIRECTION	Read/Write control bit
23:16	reserved	must be written to ‘0’
15:8	COUNT	Byte count of requested transfer
7:0	reserved	Read as ‘0’

ADDRESS must be programmed to contain the 7 bits of the desired slave address

The DIRECTION bitfield controls read/write operation on the I²C interface. The bit definition is:

- DIRECTION = 0 –> I²C write

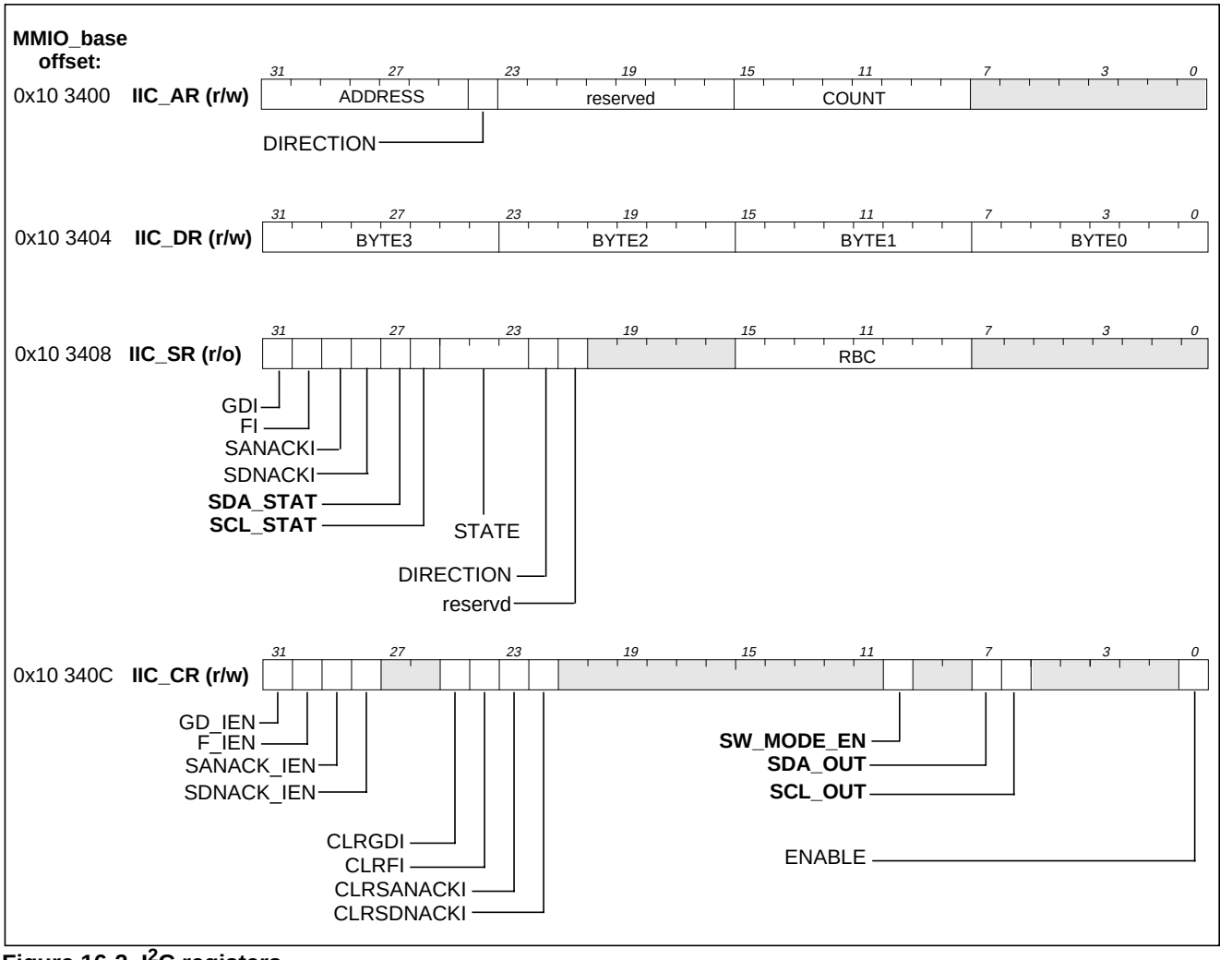


Figure 16-2. I²C registers

- DIRECTION = 1 –> I²C read

The COUNT field must contain the desired bytecount for the current transfer. The COUNT field will decrement by one for each data byte transferred across I²C. The remaining bytecount for the current transfer can be read from the COUNT field at any time. Note that the DSPCPU must refrain from rewriting the IIC_AR register until the current transfer completes to avoid corrupting the bytecount or address fields.

Note: *For writes, the byte count decrements before the byte is actually transferred over the I²C bus. However, the last byte is saved in an internal register and the DSPCPU can write a new word when COUNT = 0.*

16.4.2 IIC_DR Register

The IIC_DR register contains the actual data transferred during I²C operation. For a master transmit operation, data transfer will be initiated when data is written to this register. Transmission will begin with the transfer of the address byte in the IIC_AR register followed by the data bytes that were written to the IIC_DR register, byte3 first and byte0 last. The I²C interface will interrupt for more transmit data to be written to the IIC_DR until the transfer bytecount COUNT in the IIC_AR register is reached.

In master receive operation, one or more data bytes received are placed in the IIC_DR register by the I²C interface. Data bytes received are loaded into the IIC_DR register starting with byte3, then byte2, byte1 and byte0.:

The number of bytes the DSPCPU requests for a transfer is written into the COUNT bitfield of the IIC_AR register. The transfer completes when the I²C interface receives the number of bytes indicated by the COUNT bitfield of the IIC_AR register.

16.4.3 IIC_SR Register

The I²C status register contains status information regarding the transfer in progress and the nature of interrupts associated with I²C operation.

The IIC_SR register is read only and is intended as the primary source of status regarding current I²C operation. The IIC_SR register must be used in conjunction with the IIC_CR register. The interrupt sources of the IIC_SR register are individually enabled by writing to the appropriate enable bit in the IIC_CR register. The bitfield definitions

Table 16-3. IIC_SR register

Bits	Field Name	Definition
31	GDI	Good Data Interrupt. This is the normal transfer complete interrupt flag. This interrupt may be asserted without the IIC_SR.FI interrupt bit at the end of an I^2C transfer or after master abort of an I^2C transfer.
30	FI	Full Interrupt. This interrupt indicates the condition of the IIC_DR register dependent upon whether the I^2C interface is in receive or transmit mode.
29	SANACKI	Slave Address No Acknowledge Interrupt.
28	SDNACKI	Slave Data No Acknowledge Interrupt.
27	SDA_STAT	This bit is used to examine the state of the external I^2C SDA data pin. Bit polarity is: 1 = SDA pad is low 0 = SDA pad floated high
26	SCL_STAT	This bit is used to examine the state of the external I^2C SCL clock pin. Bit polarity is: 1 = SCL pad is low 0 = SCL pad floated high
25:23	STATE	The STATE field indicates the microactivity of the I^2C bus.
22	DIRECTION	Direction of current data transfer.
21	Reserved	Read as '0'
15:8	RBC	Remaining Byte Count.
7:0	Reserved	Read as '0'

of the IIC_SR register are presented in Table 16-3. The IIC_SR provides four sources of interrupts. Note: the interrupt should be set up as level triggered interrupt.

- **GDI** interrupt — The GDI bit together with the FI bits provide status about I^2C transfer completion. The interpretation of GDI/FI bit combinations are different depending on whether the I^2C interface is in master transmit or master receive mode. Refer to Table 16-4 and Table 16-6 for GDI/FI interpretation.
- **FI** interrupt — See GDI bit definition and GDI/FI transmit and receive definitions in Table 16-4 and Table 16-6.
- **SANACKI** interrupt — This interrupt flag bit indicates that a slave address was transmitted but no slave on the I^2C bus acknowledges the address to claim the transaction. This is an error condition. Once the I^2C interface has set this interrupt flag, the interface is idle. The DSPCPU should clear this interrupt flag by writing a '1' to IIC_CR.CLRSANACKI before reattempting this transfer or starting another I^2C transfer.
- **SDNACKI** interrupt — This interrupt flag bit indicates that an addressed slave receiver device has refused to acknowledge the current byte of data for an ongoing transfer. This is an error condition. Once the I^2C interface has set this interrupt flag, the interface is idle. The DSPCPU should clear this interrupt flag by writing a '1' to IIC_CR.CLRSDNACKI before retrying this transfer or starting another.

Table 16-4. Master transmit mode GDI/FI status

GDI	FI	Description
0	0	Message is not complete. The IIC_DR is not empty. No interrupt.
0	1	Message is not complete. The IIC_DR is empty and the requested transmit byte count is not equal to 0. The DSPCPU must write additional bytes of the current transfer to the IIC_DR register.
1	X	Message transmission has completed. The IIC_DR is empty. The byte transmit count = 0.

Table 16-5. STATE field values

STATE	Meaning
000	I^2C Interface is idle.
001	RESERVED FOR FUTURE USE
010	IDLE (MSG is done, awaiting clear GDI to go to 000 state)
011	Address phase is being processed
100	BYTE3 (first byte) is being processed
101	BYTE2 is being processed
110	BYTE1 is being processed
111	BYTE0 (last) is being processed

Table 16-6. Master receive GDI/FI conditions

GDI	FI	Description
0	0	Message is not complete. IIC_DR is not full. No interrupt.
0	1	IIC_DR contains received data and needs to be read serviced. More data bytes are expected since the receive byte count is not equal to 0.
1	X	The transfer has been completed and the receive byte count is equal to 0. 0 to 4 valid bytes are in the IIC_DR register awaiting read servicing by the DSPCPU.

The SDA_STAT and SCL_STAT bits indicate the current state of the SDA and SCL signals. The STATE field indicates the microactivity of the I^2C interface. The field values and their meanings are presented in Table 16-5 The DIRECTION status bit indicates if the I^2C interface is in transmit or receive mode.

- if DIRECTION = 0 then I^2C is a transmitter.
- if DIRECTION = 1 then I^2C is a receiver.

The RBC bitfield indicates the remaining bytecount for an I^2C transfer in progress. The IIC_SR.RBC bitfield serves as a read-only 'shadow register' for the IIC_AR.COUNT bitfield. During I^2C transfer, the RBC bitfield will reflect the remaining bytecount. To avoid corrupting an I^2C

transfer, the DSPCPU must refrain from writing to the IIC_AR.COUNT bitfield until a message is complete. Completion is indicated by the RBC bitfield decrementing to zero.

16.4.4 IIC_CR Register

The I^2C control register contains control information required for enabling I^2C transfers. This register is used to enable and clear interrupt sources which normally occur during I^2C operation. The four interrupt sources described in the section on the IIC_SR register are enabled and cleared through the IIC_CR register. The enable bitfields are:

Table 16-7. IIC_CR Register

Bits	Field Name	Definition
31	GD_IEN	Enable for normal transfer complete interrupt
30	F_IEN	Enable for IIC_DR data service request interrupt
29	SANACK_IEN	Enable for slave address not acknowledged interrupt
28	SDNACK_IEN	Enable for slave data not acknowledged interrupt. An addressed slave receiver has refused to accept the last byte transmitted to it
27:26	Reserved1	Always write ‘0’s to these bits. (See Note1)
25	CLRGDI	Clear bit for the GDI interrupt in the IIC_SR register. Writing a ‘1’ to this bit clears the GDI interrupt
24	CLRFI	Clear bit for the FI interrupt in the IIC_SR register. Writing a ‘1’ to this bit clears the FI interrupt
23	CLRSANACKI	Clear bit for the SANACKI interrupt in the IIC_SR register. Writing a ‘1’ to this bit clears the SANACKI interrupt.
22	CLRSDNACKI	Clear bit for the SDNACKI interrupt in the IIC_SR register. Writing a ‘1’ to this bit clears the SDNACKI interrupt.
21:6	Reserved2	Always write ‘0’s to these bits. (See Note1)
10	SW_MODE_EN	0 (power-on/reset default) - Normal I2C hardware operating mode. 1 - Enable software operating mode. The I^2C pins are entirely controlled by user writes to the ‘sda_out’ and ‘scl_out’ register bits.
7	SDA_OUT	Enabled by sw_mode_en. This bit is used by sw to manually control the external I^2C SDA data pin. Bit polarity is: 1 = SDA pad pulled low 0 = SDA pad left open drain
6	SCL_OUT	Enabled by sw_mode_en. This bit is used by sw to manually control the external I^2C SCL clock pin. Bit polarity is: 1 = SCL pad pulled low 0 = SCL pad left open drain
5:2	Reserved3	Always write ‘0’s to these bits. (See Note1)
1	Reserved4	Always write ‘0’s to these bits. (See Note1)
0	ENABLE	I^2C serial interface enable

- **GD_IEN** — Enable for normal transfer complete interrupt.
- **F_IEN** — Enable for IIC_DR data service request interrupt.
- **SANACK_IEN** — Enable for slave address not acknowledged interrupt. This is an error interrupt.
- **SDNACK_IEN** — Enable for slave data not acknowledged interrupt. An addressed slave receiver has refused to accept the last byte transmitted to it. This is handled as an error interrupt.

In addition to the interrupt enable bits, the IIC_CR contains interrupt clear bits associated with each of the interrupt sources in the IIC_SR register. These IIC_CR interrupt clear bits are defined as:

- **CLRGDI** — Clear bit for the GDI interrupt in the IIC_SR register. Writing a ‘1’ to this bit clears the GDI interrupt.
- **CLRFI** — Clear bit for the FI interrupt in the IIC_SR register. Writing a ‘1’ to this bit clears the FI interrupt.
- **CLRSANACKI** — Clear bit for the SANACKI interrupt in the IIC_SR register. Writing a ‘1’ to this bit clears the SANACKI interrupt.
- **CLRSDNACKI** — Clear bit for the SDNACKI interrupt in the IIC_SR register. Writing a ‘1’ to this bit clears the SDNACKI interrupt.

The remaining bitfield of the IIC_CR register is:

- **ENABLE** — Master enable for I^2C serial interface. ENABLE must be set equal to ‘1’ to transfer any bits from the I^2C interface block. Writing a ‘0’ to the ENABLE bit effectively resets the entire I^2C interface, including all status and interrupt flag bits. A transfer in progress is aborted and the byte currently transferred is lost.

  Note: *For writes, Reserved1, 2, 3 and 4 bitfields MUST always be written with ‘0’s.*

16.5 I^2C SOFTWARE OPERATION MODE

I^2C software operation mode is intended for use by software I^2C or similar algorithm implementations. In this case, the SCL and SDA pins are fully controlled and ob-

served by software, and the hardware I²C interface is disconnected from the SCL and SDA pins. Refer to Figure 16-3 for a clarification of the principles involved. Software mode is by default disabled after boot. Software mode is enabled by writing a '1' to IIC_CR.SW_MODE_EN. At that point, the SCL and SDA pins can be controlled by the IIC_CR SDA_OUT and SCL_OUT bits. Writing a '1' to either bit causes the corresponding pin to become active, i.e. be pulled low. The SDA and SCL lines are open-collector outputs, and can hence also be pulled low by external devices. The actual pin state can be observed by software by examining IIC_SR SDA_STAT and SCL_STAT bits. A 1 in these MMIO bits indicates that the corresponding pin is currently pulled low.

By appropriate software, possibly using a timer interrupt, full I²C functionality can be implemented using this mechanism.

16.6 I²C HARDWARE OPERATION MODE

Hardware operation of I²C is the default mode after boot. The TM1300 I²C hardware interface operates in one of two modes:

1. Master-transmitter (to write data to a slave)
2. Master-receiver (to read data from a slave)

As a master, the I²C logic will generate all the serial clock pulses and the START and STOP bus conditions. The START and STOP bus conditions are shown in Figure 16-4. A transfer is ended with a STOP condition or a repeated START condition. Since a repeated START condition is also the beginning of the next serial transfer, the I²C bus will not be released.

Note: *The I²C interface on TM1300 will operate as a master ONLY!*

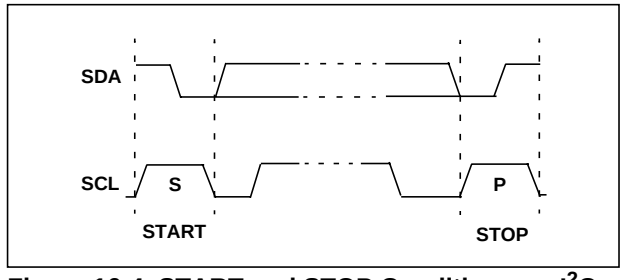


Figure 16-4. START and STOP Conditions on I²C

The number of bytes transferred between the START and STOP conditions from transmitter to receiver is not limited. Each 8-bit data byte is followed by one acknowledge bit. The transmitter releases the SDA line which will pull-up to a HIGH level during the acknowledge bit time. The receiver acknowledges by pulling the data line LOW

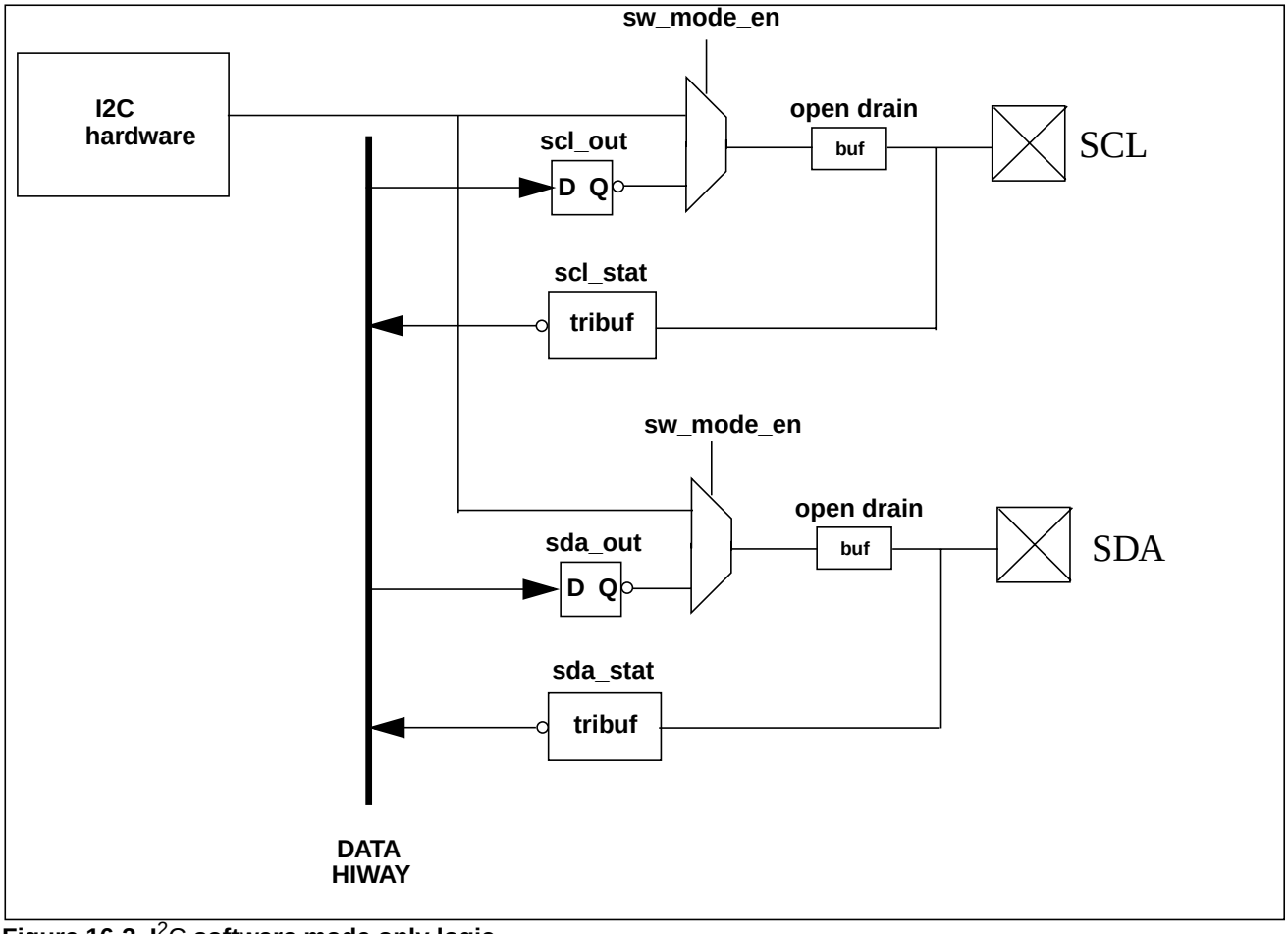


Figure 16-3. I²C software mode only logic

during this acknowledge period. The master must always generate the SCL transitions for the acknowledge bit time.

Two types of data transfers are supported by the TM1300 I^2C interface:

- Data transfer from a master transmitter to a slave receiver, also called a WRITE operation. The master first transmits a 1-byte slave address, then the desired number of data bytes. The slave receiver returns an acknowledge bit after each byte. The master terminates the transaction by a STOP after the last byte.
- Data transfer from slave transmitter to master receiver, also called a READ operation. The first byte (the slave address) is transmitted by the master and acknowledged by the slave. The selected slave transmits successive data bytes which are each acknowledged by the master, except the last byte desired by the master, for which the master generates a 'notack' condition. This causes the slave to terminate byte transmission. The slave transmitter then must release the bus so that the master may generate a STOP condition.

The type of transaction is indicated by the LSbit of the address byte. Data transfer from a master transmitter to a slave receiver is called a WRITE. It is signified by a '0' in the LSbit of the address byte. Data transfer from a slave transmitter to a master receiver is called a READ. It is signified by a '1' in the LSBit of the address byte.

Example steps for successful programming of the I^2C interface on TM1300 are outlined as follows for both reads and writes. Enable the I^2C interface prior to attempting any accesses to external I^2C devices.

To enable the interface:

- Set bit IIC_CR.ENABLE (0x10340c) = 1

For write addressing mode:

1. On entry, clear any possible I^2C interrupt sources by writing IIC_CR bits [25:22] = '1111'. (Note that programmers must mask and enable high-level interrupt sources through the VIC facility in the DSPCPU. See the appropriate TM1300 databook chapter).
2. Enable desired I^2C interrupt sources by setting IIC_CR[31:28] bits appropriately.
3. Simultaneously load IIC_AR[31:25] with 7-bit slave address, IIC_AR.DIRECTION = 0 and IIC_AR[15:8] with the appropriate bytecount for the transfer.
4. Load IIC_DR[31:0] with data for the write. Note that writing this register triggers the transfer across the I^2C bus.Up to 4 bytes will be transferred after writing, dependent on bytecount in IIC_AR[8:15}.Transfers of more than 4 bytes have to be done by breaking them down into a sequence of 4-byte transfers and a last transfer which may be less than 4 bytes. This is done by repeatedly reloading the register until the bytecount is fulfilled. Transfer is done high byte first, proceeding to low byte.
5. Detect I^2C resulting condition code in IIC_SR[31:28] and respond - OR - Detect I^2C high level interrupt and respond. (Note that this last step is dependent upon system software requirements).
6. If transfer count is not yet fulfilled, clear GDI and FI bits and proceed with step iv) until all data is written.

For read addressing mode:

1. On entry, clear any possible I^2C interrupt sources by writing IIC_CR bits [25:22] = '1111'. (Note that programmers must mask and enable high level interrupt sources through the VIC facility in the DSPCPU. See the appropriate databook chapter).
2. Enable desired I^2C interrupt sources by setting IIC_CR[31:28] bits appropriately.
3. Simultaneously load IIC_AR[31:25] with 7-bit slave address, IIC_AR.DIRECTION = 1 and IIC_AR[15:8] with the appropriate bytecount for the transfer. Note that writing this register triggers the read across the I^2C bus.
4. Detect I^2C resulting condition in IIC_SR[31:28] and respond - OR - Detect I^2C interrupt and respond. (Note that this last step is dependent upon system software requirements.)
5. Clear GDI and FI bits and read the contents of IIC_DR. Up to 4 bytes will be available in IIC_DR, fever if the remaining bytecount was less than 4. Bytes are stored high byte first, proceeding to low byte.
6. Proceed with step iv) until all data is read, i.e bytecount is fulfilled.

16.6.1 Slave NAK

If a slave device does not generate an ACK where required, this is considered a NAK. Upon receipt of a NAK after transmitting a device address or data byte, the master takes the following actions:

- the I^2C state becomes IDLE (STATE = 000)
- a STOP condition is issued on the bus
- no more data is sent

16.7 I²C CLOCK RATE GENERATION

The I²C hardware block diagram is shown in Figure 16-5 below. In hardware operating mode, the IIC__SCL external clock is derived by division from the BOOT_CLK pin on TM1300. The BOOT_CLK pin is normally connected to TRI_CLKIN. The IIC__SCL clock divider value is determined at boot time and cannot be changed thereafter. The value chosen depends on the first byte read from the EEPROM, as described in Section 13.3.1, “Boot Procedure Common to Both Autonomous and Host-Assisted Bootstrap.”

The TM1300 I²C block is able to ‘stretch’ the SCL clock in response to slaves that need to slow down byte transfer. This mechanism of slowing SCL in response to a slave is called ‘clock stretching.’ This clock stretching is accomplished by the slave by holding the SCL line ‘low’ after completion of a byte transfer and acknowledge sequence. Clock stretching is always enabled.

Table 16-8. I²C speed and EEPROM byte 0

BOOT_CLK bits	EEPROM speed bit	divider value	actual I²C speed
00 (100 MHz)	0 (100 kHz)	1008	99.2 kHz
00	1 (400 kHz)	256	390.6 kHz
01 (75 MHz)	0 (100 kHz)	752	99.7 kHz
01	1 (400 kHz)	192	390.6 kHz
10 (50 MHz)	0 (100 kHz)	512	97.6 kHz
10	1 (400 kHz)	128	390.6 kHz
11 (33 MHz)	0 (100 kHz)	336	98.2 kHz
11	1 (400 kHz)	96	343.8 kHz

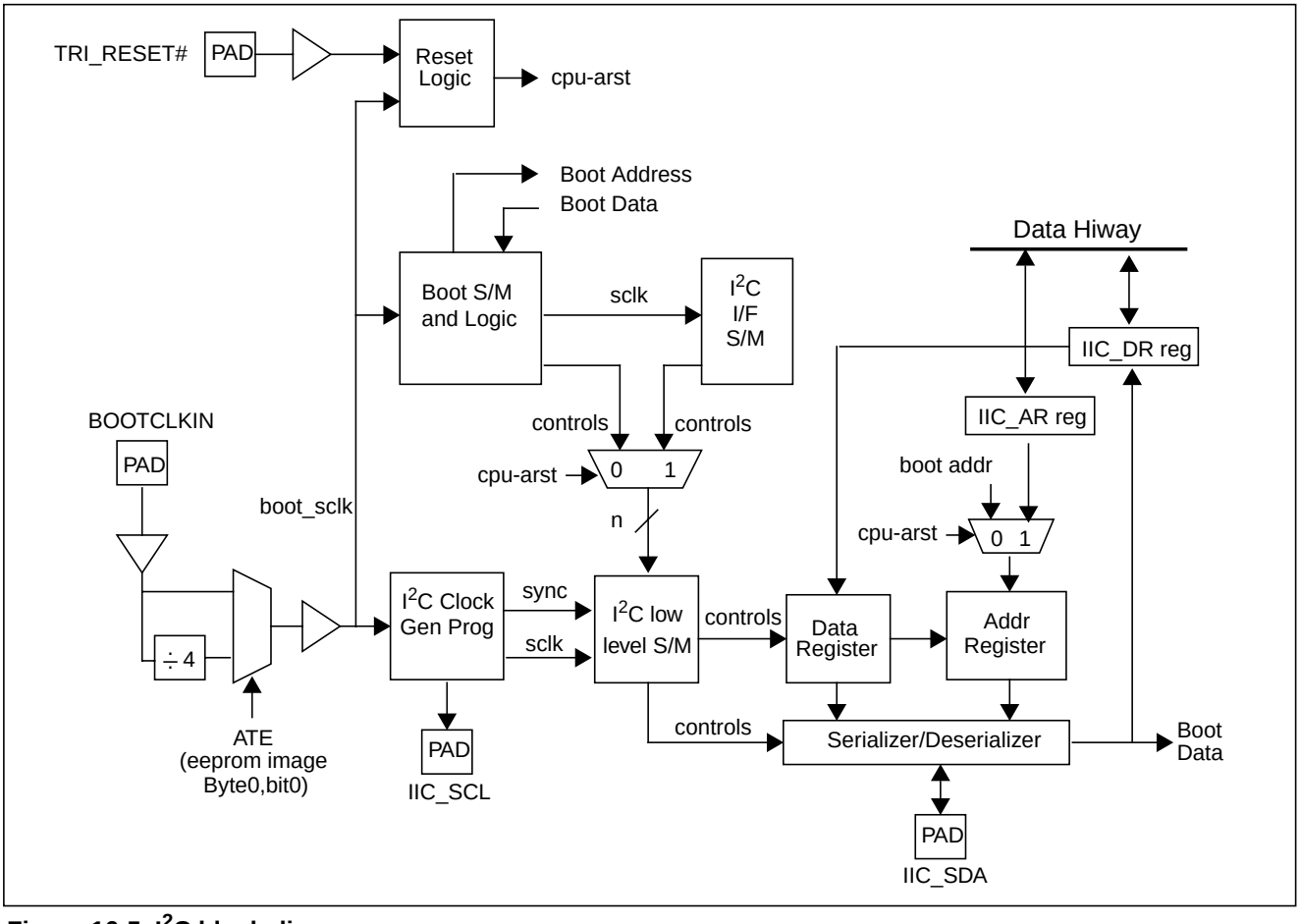


Figure 16-5. I²C block diagram

17.1 SYNCHRONOUS SERIAL INTERFACE OVERVIEW

The TM1300 synchronous serial interface (SSI) unit interfaces to an off-chip modem analog front end (MAFE) subsystem, network terminator, ADC/DAC or codec through a flexible bit-serial connection. The hardware performs full-duplex serialization/deserialization of a bit stream from any of these devices. Any such front end device connected must support transmitting, receiving of data, and initialization via a synchronous serial interface.

Since the communication algorithm is implemented in software by the TM1300 DSPCPU and the analog interface is off chip, a wide variety of modem, network and/or FAX protocols may be supported.

The SSI hardware includes:

- A 16-bit receive shift register (RxSR), synchronized by an external receive frame synchronization pulse (SSI_RxFSX) and clocked by an external clock (RxCLK)
- A 32-bit MMIO receive data register (SSI_RxDR) to provide data access from the DSPCPU
- 32-entry deep,16-bit wide receive buffer (RxFIFO), to buffer between the receive shift register (RxSR) and MMIO receive data register (SSI_RxDR)
- A 16-bit transmit shift register (TxSR), synchronized by an external or internal transmit frame synchronization pulse and clocked by an external clock (either SSI_IO1 or SSI_RxCLK)
- A 32-bit MMIO transmit data register (SSI_TxDR) to transmit data from the DSPCPU.
- 30-entry deep, 16-bit wide transmit buffer (TxFIFO), to buffer between the MMIO transmit data register (SSI_TxDR) and transmit shift register (TxSR)
- Transmit frame sync pulse generation logic
- Control and status logic
- Interrupt generation logic

The SSI unit is not a hiway bus master. All I/O is completed through DSPCPU MMIO cycles. FIFOs are used to increase allowable interrupt response time and decrease interrupt rate.

17.2 INTERFACE

The external interface consists of the 6 pins described in

Table 17-1. Synchronous serial interface pins

Name	Type	Description
SSI_RxCLK	IN-5	Serial interface clock signal; provided by an external communication device.
SSI_RxFSX	IN-5	Frame synchronization reference signal; provided by an external communication device.
SSI_RxDATA	IN-5	Receive serial data signal; provided by the receive channel of an external communication device.
SSI_TxDATA	OUT	Transmit serial data signal output.
SSI_IO1	I/O-5	Transmit clock input or general purpose I/O pin.
SSI_IO2	I/O-5	Transmit Frame synchronization signal input or output or general purpose I/O pin.

17.3 BLOCK DIAGRAM

The main block diagram of the SSI unit is illustrated in Figure 17-1.

The I/O block is used for control of the I/O pins and for selecting the transmit clock and transmit frame synchronization signals.

The frame synchronization block can be used for generating an internal synchronization signal derived from receive clock input (SSI_RxCLK) or from an I/O pin (SSI_IO1).

The SSI transmit block buffers and transmits the bits using the generated frame synchronization signal (TxFSX) and the transmit clock. The transmit clock is either the receive clock or the clock present on SSI_IO1.

The SSI receive block receives and buffers the bits on the SSI_RxDATA line, using the receive clock (SSI_RxCLK) and the receive frame synchronization signal (SSI_RxFSX).

Each of the blocks will be described in detail in the next subsections.

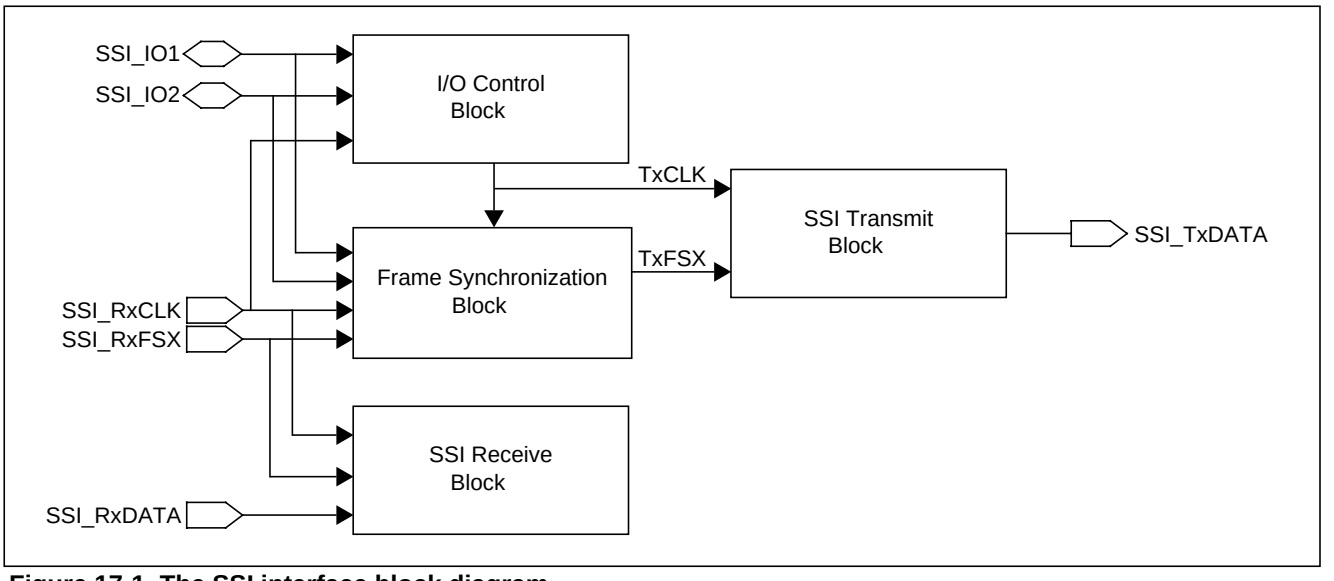


Figure 17-1. The SSI interface block diagram

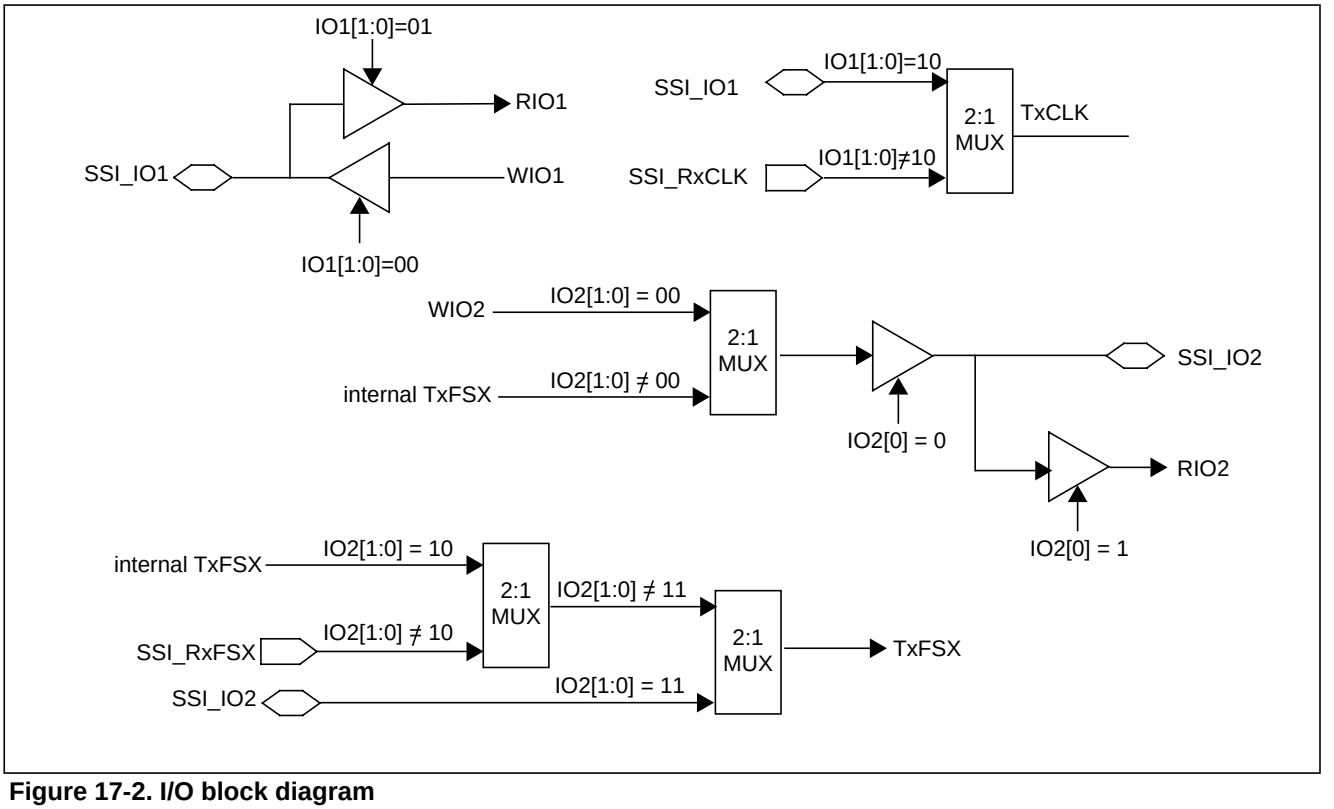


Figure 17-2. I/O block diagram

17.3.1 General Purpose I/O

Figure 17-2 illustrates the functionality of the general purpose I/O pins. The SSI_IO1 and SSI_IO2 external pins may be used as general purpose I/O by proper configuration of the SSI_CTL register, or they may be used as transmit clock input and as transmit framing signal input or output. The SSI_CTL.IO1 and SSI_CTL.IO2 Mode Select fields control the direction and functionality of these two pins.

A hardware reset or a software reset of the transmitter through SSI_CTL.TXR command sets the SSI_CTL.IO1 and SSI_CTL.O2 fields to 11b, a conflict-free initial pin state.Table 17-2 shows the effect of SSI_CTL.IO1 on pin SSI_IO1, Table 17-3 shows the effect of SSI_CTL.IO2 on SSI_IO2. Note: If SSI_IO1 is not selected as transmit clock input, the transmit clock is taken from the receive clock signal instead. If SSI_IO2 is not selected as transmit framing signal input or output, the transmit framing signal is taken from the receive framing signal instead.

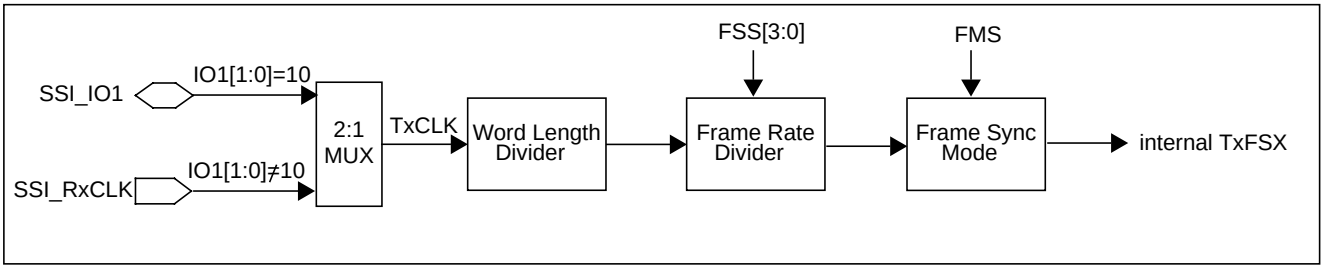


Figure 17-3. Frame synchronization generation block diagram

Table 17-2 Effect of SSI_CTL.IO1 on SSI_IO1

IO1[0:1]	Function of SSI_IO1
00	general purpose output with positive logic polarity, reflecting the value in SSI_CTL.WIO1
01	general purpose input, with optional change detector function. The input state can be read from SSI_CSR.RIO1. The change detector is clocked by the highway bus. The change detector may optionally generate an interrupt, under the control of CDE bit of SSI_CTL.
10	Transmit clock (TxCLK) input
11	tri-state, input signal value ignored

Table 17-3 Effect of SSI_CTL.IO2 on SSI_IO2

IO2[0:1]	Function of SSI_IO2
00	General purpose output with positive logic polarity, reflecting the value in SSI_CTL.WIO2
01	General purpose input. The input state can be read in from SSI_CSR.RIO2. No change detector is provided for this pin.
10	Internal transmit framing signal (TxFSX) output.
11	Transmit framing signal (TxFSX) input.

17.3.2 Frame Synchronization

The internal frame synchronization logic is illustrated in Figure 17-3. An internal Frame Synchronization signal (TxFSX) is being generated from the transmit or receive clock selected by SSI_CTL.IO1. The Clock is divided by the word length (16) and a Frame Rate Divider which is controlled by the FSS[3:0] bits in the SSI_CTL register. FMS determines the Frame Mode operation, whether the frame sync pulse is word-length or bit-length. The transmit framing signal is selected depending on SSI_CTL.IO2, as shown in Table 17-4.

Table 17-4. Effect of SSI_CTL.IO2 on transmit framing signal

IO2[0:1]	Source of transmit framing signal
00	taken from RxFSX
01	taken from RxFSX
10	internally generated
11	taken from SSI_IO2 pin

17.3.3 SSI Transmit

The transmitter control block diagram is illustrated in Figure 17-4. The transmitter clock can be selected from two sources, i.e. SSI_IO1 or SSI_RxCLK by programming IO1[1:0] bits in the SSI_CTL register (see Figure 17-2). A transfer takes place on either the rising or falling edge of the clock, which can be configured with SSI_CTL.TCP.

The transmitter has a 30-entry deep, 16-bit transmit buffer that buffers the data between the 32-bit SSI_TXDR register and the 16-bit transmit shift register (TxSR).

The TxSR is a 16-bit transmit shift register. It can be configured to shift out MSB or LSB first with SSI_CTL.TSD.

A detailed description of the configuration of the transmitter can be found in the SSI_CTL and SSI_CSR register description (17.10.1 and 17.10.2)

SSI_TxDR is a 32-bit MMIO transmit register.

17.3.4 SSI Receive

The receiver control block diagram is illustrated in Figure 17-5. The receiver clock, frame synchronization and data signal are always taken from the external pins.

The receiver has a 32-entry deep, 16-bit receive buffer that buffers the data between the 16-bit receive shift register (RxSR) and the 32-bit SSI_RXDATA register.

The input pin SSI_RxDATA provides serial shift in data to the RxSR. The RxSR is a 16-bit receive shift register. RxSR can be configured to shift in from MSB or LSB first using SSI_CTL.RSD. A transfer takes place on either the rising or falling edge of the receiver clock, which can be configured with the SSI_CTL.RCP.

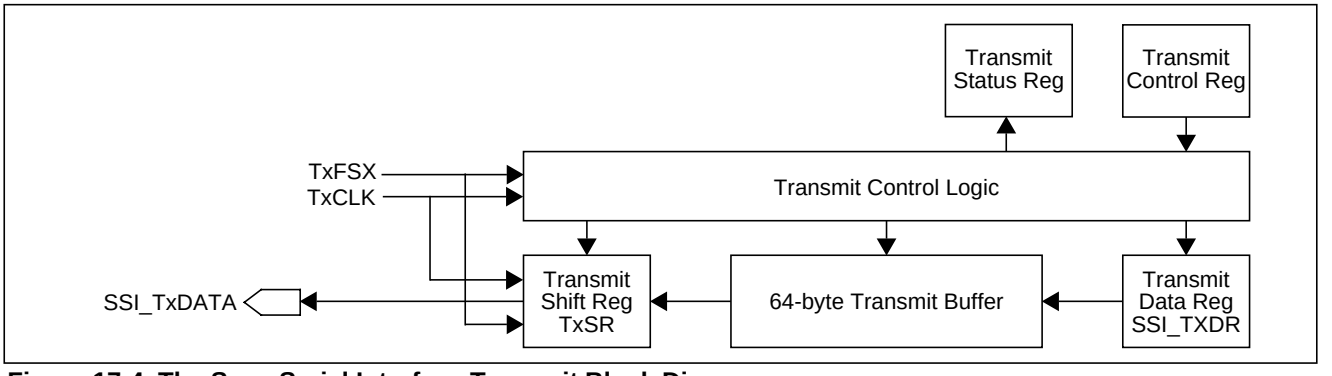


Figure 17-4. The Sync Serial Interface Transmit Block Diagram

A detailed description of the configuration of the receiver can be found in the SSI_CTL and SSI_CSR register description (17.10.1 and 17.10.2)

SSI_RxDR is a 32-bit MMIO receive data register.

Due to the possibility of speculative reading of the SSI_RxDR, the read itself can not be implemented to acknowledge the data as a side effect. For this reason an explicit acknowledge mechanism is provided by the SSI_RxACK register.

The SSI_RxACK is a 1-bit MMIO register that is used to signal the SSI receiver state machine that a word has been successfully read from the SSI_RxDR.

Writing a '1' to this register initiates updating of the internal state. Writing a '0' has no effect.

The register cannot be read, its effect may be observed in the WAR field of the SSI_CSR.

The status fields of the SSI_CSR will update within 1 highway clock cycle after writing to the SSI_RXACK register.

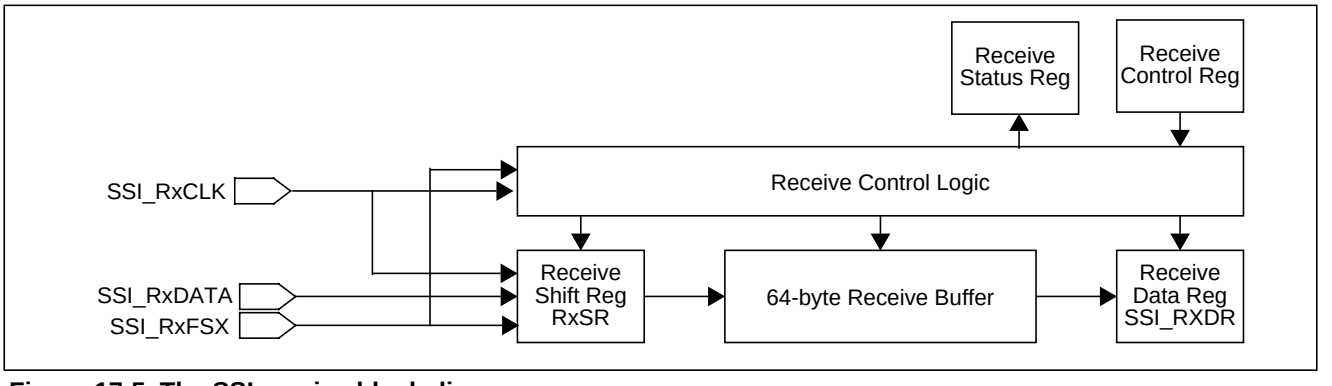


Figure 17-5. The SSI receive block diagram

17.4 SSI TRANSMIT OPERATION

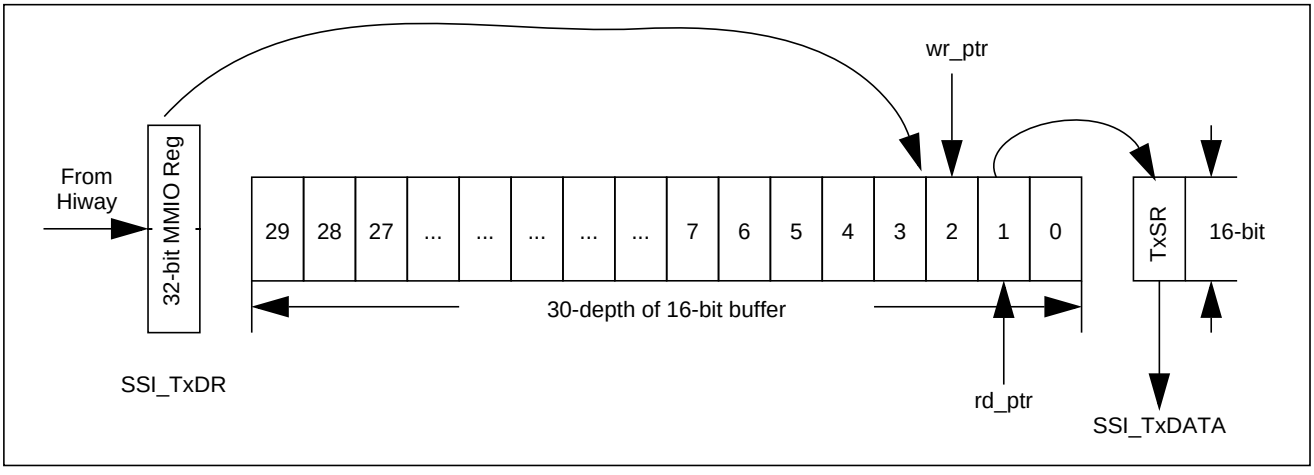


Figure 17-6. The transmit buffer operation

17.4.1 Setup SSI_CTL

Write the SSI_CTL to reset and enable the transmitter. Both the transmitter and receiver must be reset simultaneously. This will set all registers and internal logic to be same as after a power-up reset. The recommended procedure is to set up all transmitter-related control bits before performing a TXE assert. In particular, fields TCP, RSD, IO1, IO2, FMS, FSP, MOD and TMS should NOT be changed after enabling the transmitter until after the next transmitter reset.

The TxCLK is taken from the SSI_IO1 pin or from the receive clock, dependent on SSI_CTL.IO1. The direction of shift in the TxSR and the clock edge on which to shift must also be configured in SSI_CTL. If the DSPCPU does not poll the SSI status registers, it should enable the transmitter interrupt and set the ILS field by writing to the SSI_CTL to allow interrupt driven servicing of the SSI. Note that both transmit and receive use the same ILS field. Set the framing controls, slot size, and mode required according to the external communication circuit's requirements by writing the SSI_CTL. Finally, set the interrupt level to respond to empty levels in the TxFIFO. Note that the Rx and Tx machines share the framing and clock divide controls. They cannot be set to different values for Rx and Tx.

If the RxCLK used to derive the TxCLK needs a divide by two, this is done by setting SSI_CSR.CD2.

17.4.2 Operation Details

The transmit state machine will wait for transmit data to be written to the SSI_TxDR register. (see also Figure 17-6) As soon as SSI_TxDR is written, it's value will be propagated through two entries of the TxFIFO (TxFIFO is 16-bit and SSI_TxDR is 32-bit) and transferred to TxSR, synchronized to TxFSX. The order of transferring the two 16-bit parts in the 32-bit SSI_TxDR can be configured by the endian bit SSI_CTL.EMS. Data will begin shifting out of TxSR, one bit for each active edge of the TxCLK, from either bit 15 (MSB first SSI_CTL setting) or from bit 0 (LSB first) until TxSR is empty. For endian control and shift direction see also subsection 17.8. When the shift register is empty, the transmit state machine will load the value from the next available TxFIFO location and begin shifting out that data. The transmission continues until the transmit state machine is disabled or reset.

If the last available TxFIFO has **not** been updated at the appropriate time to reload TxSR, the last transmitted frame is retransmitted and a transmit underrun error is indicated in the transmitter status SSI_CSR.TUE

17.4.3 Interrupt and Status

The refill status of the SSI_TxDR register is stored in SSI_CSR. As the transmit state machine loads a TxFIFO register to the TxSR, it sets the associated status bits. The SSI will generate an internal interrupt when the number of empty words in the TxFIFO rises above the level set by SSI_CSR.ILS. If the transmit state machine attempts to read a TxFIFO while the last available TxFIFO has not been updated, it will set the transmit underrun bit. This can cause a protocol error in the transmission.

The number of available word buffers (SSI_CSR.WAW) and transmitter data register empty (SSI_CSR.TDE) information is updated automatically by the SSI block.

17.5 SSI RECEIVE OPERATION

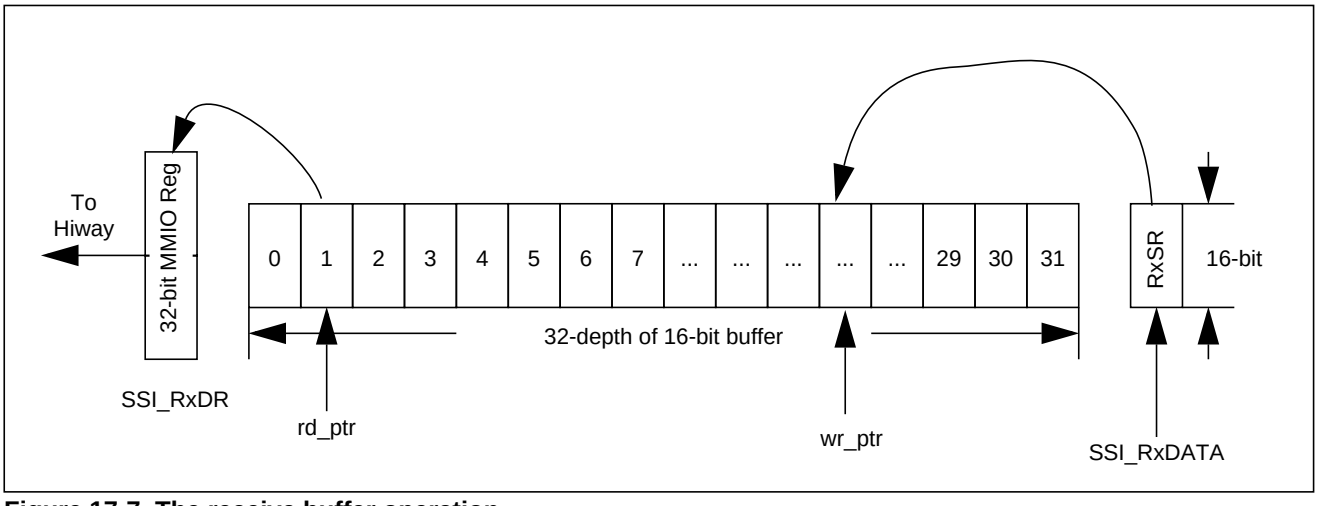


Figure 17-7. The receive buffer operation

17.5.1 Setup SSI_CTL

Write the SSI_CTL to reset and enable the receiver. Both the transmitter and receiver must be reset simultaneously. This will set all registers and internal logic the same as after a power-up reset. The recommended procedure is to set up all receiver related control bits before performing a RXE assert. In particular, fields TCP, RSD, IO1, IO2, FMS, FSP, MOD and TMS should NOT be changed after enabling the receiver until after the next receiver reset.

The direction of shift in the RxSR, mode, and the clock edge polarity must also be configured in SSI_CTL. Set the framing controls according to the external communication circuit's requirements. Note that the Rx and Tx machines share the framing and clock divide controls.

If the DSPCPU does not poll the SSI status registers, it should enable the receiver interrupt and set the ILS field by writing to the SSI_CTL to allow interrupt driven servicing of the SSI receiver. Note that both transmit and receive use the same ILS field.

If the RxCLK is double the frequency of the data rate on the SSI bus, SSI_CSR.CD2 can be used to divide the receive clock by two.

17.5.2 Operation Details

The receive state machine will begin shifting SSI_RxDATA into the RxSR on the first active edge of SSI_RxCLK received after the receiver is enabled (see also Figure 17-7). When full, the RxSR is parallel transferred to the first available RxFIFO entry and possibly SSI_RxDR. Reception continues and when RxSR is full again, a parallel load of the next available RxFIFO entry from RxSR is accomplished. This continues until the receiver is disabled or reset. If the receive state machine must transfer RxSR into one of the RxFIFO entries and none of the RxFIFO entries is available, the value will be lost and the receive overrun bit will be set.

17.5.3 Interrupt and Status

The status of the RxFIFO is visible in SSI_CSR. WAR is the number of 32-bit words available for read; it is more than ILS (RDF). As the receive state machine loads RxFIFO from the RxSR, it sets the associated status bit. The SSI will generate an internal interrupt when the number of full entries in RxFIFO is more then SSI_CTL.ILS. If the receive state machine attempts to load RxFIFO while none of the RxFIFO entries is available, it will set the receive overrun bit and generate an interrupt.

Due to the possibility of speculative reading of the SSI_RxDR, the DSPCPU must explicitly indicate a successful read of SSI_RxDR by writing a '1' in the LSB to the SSI_RxACK register. The status fields of the SSI_CSR will update within 1 highway clock cycle after completion of writing to SSI_RXACK register.

17.6 FRAME TIMING

The frame timing can be controlled by the FSS and VSS fields in the SSI_CTL register.

The FSS[3:0] bits control the divide ratio for the programmable frame rate divider used to generate the frame sync pulses. The valid value ranges from 1 to 16 slots of 16 bit each, e.g. a value of 5 indicates that a frame contains 5 slots of 16 bits each. Note: the value '16' is accomplished by storing a '0' in this field. If a codec is connected which generates 6 slots and the SSI block is programmed to 5 slots a framing error is indicated in SSI_CSR.FES; and if TIE or RIE is enabled, an interrupt is generated.

For an example of a frame timing diagram see Figure 17-11 and Figure 17-12.

The VSS[3:0] bits control the number of valid slots in the frame, starting from slot 1. For example, if the VSB[3:0] bits are if set to 4 and FSS set to 5, slots 1, 2, 3 and 4 in the frame contain valid data from the transmitter FIFO and slot 5 will contain non-valid data. The receiver will only accept data in slot 1, 2, 3 and 4.

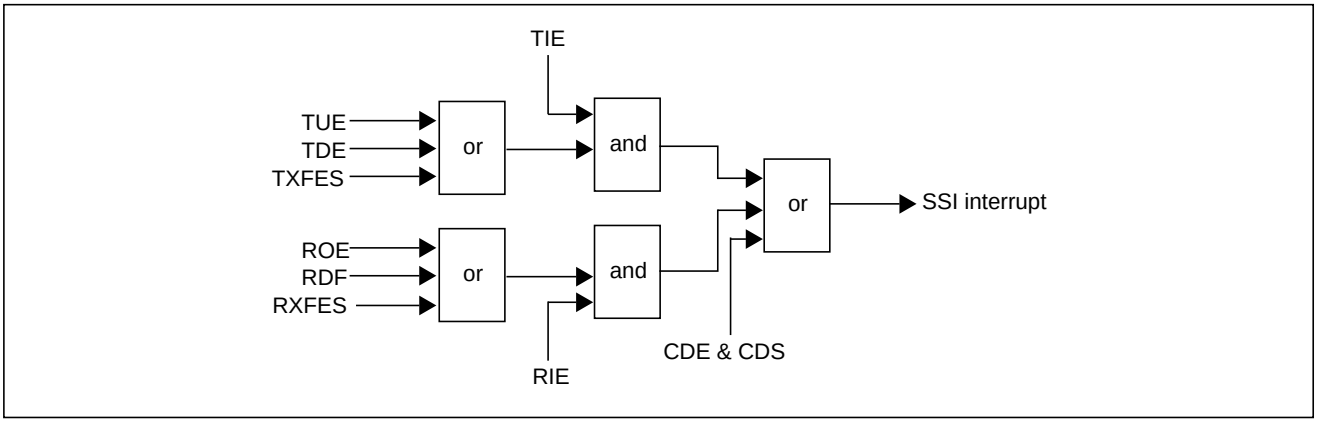


Figure 17-8. Interrupt generation logic.

17.7 INTERRUPT GENERATION

Depending on the settings of the TIE, RIE and CDE bits in the SSI_CTL register, the SSI unit can generate interrupts. This is best illustrated by Figure 17-8. Note: RXFES and TXFES are the internal receive and transmit framing error conditions. When an SSI interrupt is detected, the interrupt service routine should check all status bits.The interrupts should be set up as level-triggered interrupts.

17.8 16-BIT ENDIAN-NESS AND SHIFT DIRECTION

The SSI unit supports both access orders for the 16-bit halves of a machine word. In addition, the shift direction can be controlled to select MSB or LSB shifting first. The SSI_CTL.EMS bit controls the 16-bit endian mode, and the TSD and RSD bits control transmit and receive shift direction.

When EMS is set, the first data word received in a frame will be transferred to bit 15-0 of the SSI_RxDR, the second word will be transferred to bits 31-16 of the SSI_RxDR. EMS = '0' reverses the order of the halves of SSI_RxDR. Likewise in the transmitter, when EMS is set, the first data word transmitted in a frame will be bits 15-0 of SSI_TxDR, the second word transferred will be bits 31-16 of SSI_TxDR.

TSD and RSD control the shift direction of transmit and receive shift registers (TxSR and RxSR). Transmit data is transmitted MSB first when TSD is '0' or LSB first otherwise. Receive data is received MSB first when RSD equals '0', LSB first otherwise.

For an example of the transmit operation see Figure 17-9. Receive works the same, only that data is shifted in.

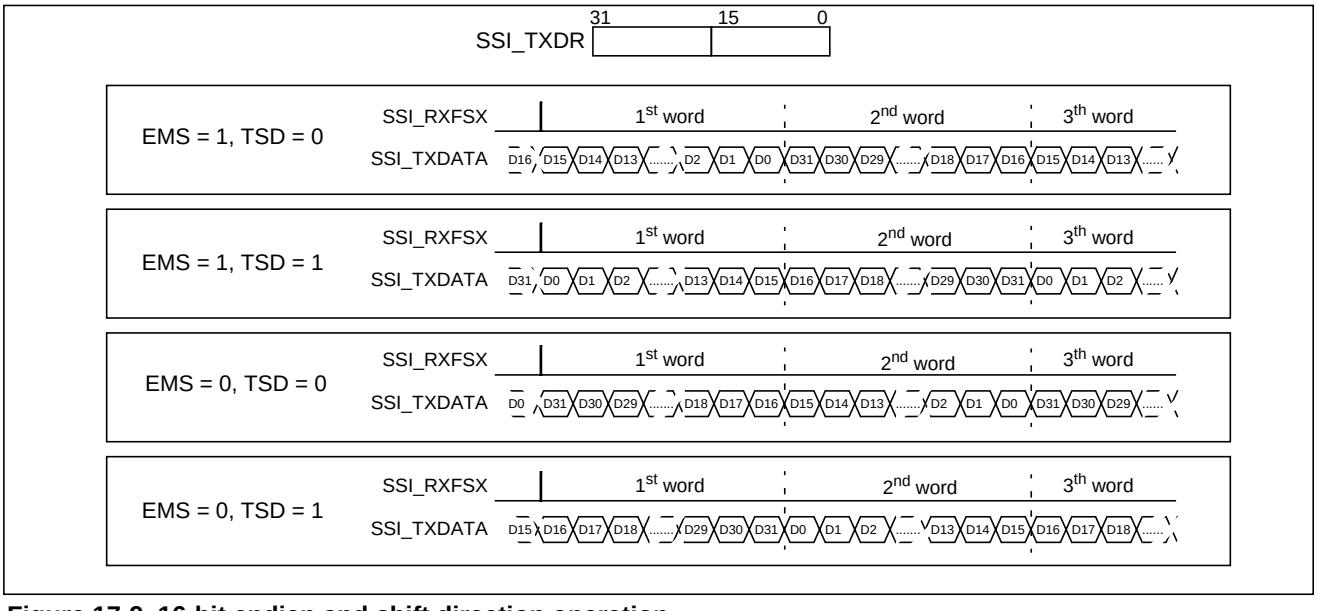


Figure 17-9. 16-bit endian and shift direction operation.

17.9 SSI TEST MODES

The SSI unit has two test modes which can be controlled by setting SSI_CSR.TMS. A remote and a local loop back testmode are supported (see also Table 17-9).

17.9.1 Remote Loopback

This test mode allows a remote transmitter to test itself, the intervening transmission media, and its associated receiver. In this mode, the data received on the SSI_RxDATA pin is buffered and transmitted on the SSI_TxDATA pin. The data is not transferred to SSI_TxDR/TxFIFO and the DSPCPU is never interrupted. The transmitter is clocked by the SSI_RxCLK pin with a combinatorial clock delay.

17.9.2 Local Loopback

This test mode allows the DSPCPU to run local checks of the SSI. Data written to the TxFIFO is serialized and passed to the receiver via an internal serial connection. The receiver deserializes the data and passes it to the RxFIFO register. Interrupts will be generated if enabled. During local loop back mode, the data on the SSI_RxDATA pin is ignored and the SSI_TxDATA pin is tristated. An external CLK must be provided during local loop back mode or no transmission or reception will occur.

17.10 MMIO REGISTERS

The MMIO Control and Status registers are shown in Figure 17-10. The register fields are described in Table 17-5, Table 17-6, Table 17-7, Table 17-8, and Table 17-9. To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as ‘0’s.

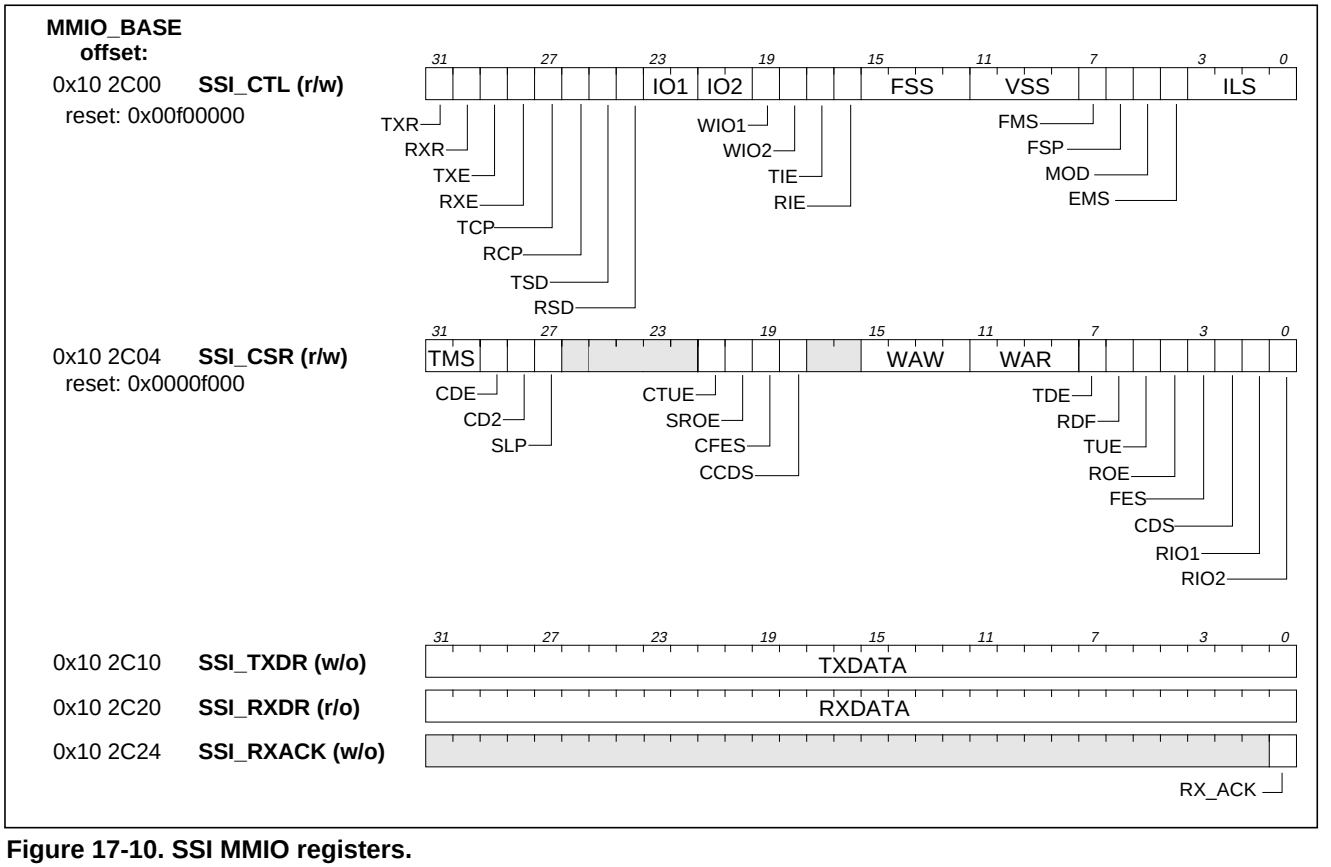


Figure 17-10. SSI MMIO registers.

17.10.1 SSI Control Register (SSI_CTL)

SSI_CTL is a 32-bit read/write control register used to direct the operation of the SSI. The value of this register after a hardware reset is 0x00F00000.

Table 17-5. SSI control register (SSI_CTL) fields.

Field	Description
TXR	Transmitter Software Reset (Bit 31). Setting TXR performs the same functions as a hardware reset. Resets all transmitter functions. A transmission in progress is interrupted and the data remaining in the TxSR is lost. The TxFIFO pointers are reset and the data contained will not be transmitted, but the data in the SSI_TxDR and/or TxFIFO are not explicitly deleted. The transmitter status and interrupts are all cleared. This is an action bit. This bit always reads '0'. Writing a '1' in combination with writing a '1' in the RXR field will initiate a reset for the SSI module. Note: this bit is always set together with RXR because a separate transmitter or receiver reset is not implemented.
RXR	Receiver Software Reset (Bit 30). Setting RXR performs the same functions as a hardware reset. Resets all receiver functions. A reception in progress is interrupted and the data collected in the RxSR is lost. The RxFIFO pointers are reset, and the SSI will not generate an interrupt to DSPCPU to retrieve data in the SSI_RxDR and/or RxFIFO. The data in the SSI_RxDR and/or RxFIFO is not explicitly deleted. The receiver status and interrupts are all cleared.This is an action bit.This bit always reads '0'. Writing a '1' in combination with writing a '1' in the TXR field will initiate a reset for the SSI module. Note: this bit is always set together with TXR, because a separate transmitter or receiver reset is not implemented.
TXE	Transmitter Enable (Bit 29). TXE enables the operation of the transmit shift register state machine. When TXE is set and a frame sync is detected, the transmit state machine of the SSI is begins transmission of the frame. When TXE is cleared, the transmitter will be disabled after completing transmission of data currently in the TxSR. The serial output (SSI_TxDATA) is three-stated, and any data present in SSI_TxDR and/or TxFIFO will not be transmitted (i.e., data can be written to SSI_TxDR with TXE cleared; TDE can be cleared, but data will not be transferred to the TxSR). Status fields updated by the Transmit state machine are not updated or reset when an active transmitter is disabled.
RXE	Receive Enable (Bit 28). When RXE is set, the receive state machine of the SSI is enabled. When this bit is cleared, the receiver will be disabled by inhibiting data transfer into SSI_RxDR and/or RxFIFO. If data is being received while this bit is cleared, the remainder of that 16-bit word will be shifted in and transferred to the SSI RxFIFO and/or SSI_RxDR. Status fields updated by the Receive state machine are not updated or reset when an active receiver is disabled.
TCP	Transmit Clock Polarity (Bit 27). The TCP bit value should only be changed when the transmitter is disabled. TCP controls on which edge of TxCLK data is output. TCP=0 causes data to be output at rising edge of TxCLK, TCP=1 causes data to be output at falling edge of TxCLK.
RCP	Receive Clock Polarity (Bit 26). RCP controls which edge of RxCLK samples data. The data is sampled at rising edge when RCP = '1' or falling edge when RCP = '0'.
TSD	Transmit Shift Direction (Bit 25). TSD controls the shift direction of transmit shift register (TxSR). Transmit data is transmitted MSB first when TSD = '0' or LSB first otherwise. The operation of this bit is explained in more detail in section 17.8.
RSD	Receive Shift Direction (Bit 24). The RSD bit value should only be changed when the receiver is disabled. RSD controls the shift direction of receive shift register (RxSR). Receive data is received MSB first when RSD = '0', LSB first otherwise. The operation of this bit is explained in more detail in section 17.8.
IO1	Mode Select SSI_IO1 pin (Bit 23-22). The IO1 field value should only be changed when the transmitter and receiver are disabled. The IO1[1:0] bits are used to select the function of SSI_IO1 pin. The function may be selected as listed in table Table 17-6.
IO2	Mode Select SSI_IO2 pin (Bit 21-20). The IO2 field value should only be changed when the transmitter and receiver are disabled. The IO2[1:0] bits are used to select the function of SSI_IO2 pin. The function may be selected according to Table 17-7
WIO1	Write IO1 (Bit 19). Value written here appears on the SSI_IO1 pin when the pin is configured to be a general purpose output.
WIO2	Write IO2 (Bit 18). Value written here appears on the SSI_IO2 pin when this pin is configured to be a general purpose output.
TIE	Transmit Interrupt Enable (Bit 17). Enables interrupt by the TDE flag in the SSI status register (transmit needs refill) Also enables interrupt of the TUE (transmitter underrun error) and TXFES (transmit framing error)
RIE	Receive Interrupt Enable (Bit 16). When RIE is set, the DSPCPU will be interrupted when RDF in the SSI status register is set (receive complete). It will also be interrupted on ROE (receiver overrun error) and on RXFES (receive framing error).
FSS	Frame Size Select (Bits 15-12). The FSS[3:0] bits control the divide ratio for the programmable frame rate divider used to generate the frame sync pulses. The valid setup value ranges from 1 to 16 slot(s). The value '16' is accomplished by storing a 0 in this field.

Table 17-5. SSI control register (SSI_CTL) fields.

Field	Description
VSS	Valid Slot Size (Bit 11-8). The VSS[3:0] bits control the valid slot size (starting from slot 1) for different modem analog front end devices. The valid setup value ranges from 1 to 16 slot(s). The value 16 is accomplished by storing a '0' in this field.
FMS	Frame Sync Mode Select (Bit 7). The FMS bit value should only be changed when the transmitter and receiver are disabled. FMS selects the type of frame sync to be recognized by both Rx and Tx. When FMS = '1', frame sync is word-length bit clock. When this bit = '0', frame sync is a 1-bit clock.
FSP	Frame Sync Polarity (Bit 6). The FSP bit value should only be changed when the transmitter and receiver are disabled. FSP controls which edge of frame sync is the active edge for both Rx and Tx. This bit causes frame signal to be active at rising edge when FSP = '0' , or falling edge when FSP = '1'.
MOD	Mode Select (Bit 5). The MOD bit value should only be changed when the transmitter and receiver are disabled. MOD selects the operational mode of the SSI for ISDN functionality. When MOD is set, the SSI is configured as a U-interface for ISDN NT. Otherwise, set to '0'. Setting MOD bit and CD2 supports the MC145574 and MC145572 ISDN interface transceivers.
EMS	Endian Mode Select (Bit 4). Selects the big- or little-endian mode operation. See Section 17.8 for more detail.
ILS	Interrupt Level Select (Bit 3-0). Sets the point where an interrupt is generated for normal data buffer servicing. The number ranges from 1 to 15. This field controls interrupt level of both transmit and receive functions.

Table 17-6. IO1 mode select

Bit	Mode
00	**General Purpose Output:** Configures the SSI_IO1 pin for general purpose output. The pin follows the state of the WIO1 field of the SSI_CTL.
01	**General Purpose Input:** Change detector may be used. Value can be read in from the RIO1 field of the SSI_CSR.
10	**Enable External TxCLK:** Allows for use of an externally generated TxCLK. The clock is provided via the TxCLK pin. All general purpose I/O functions are unavailable.
11	**Disable:** Pin is not used. Output buffer is tristated and the input is ignored. (RESET default)

Table 17-7. IO2 mode select

Bit	Mode
00	**General Purpose Output:** Configures the SSI_IO2 pin as a general purpose output. The pin follows the state of the WIO2 field of the SSI_CTL.
01	**General Purpose Input:** Value can be read in from RIO2 field of the SSI_CSR.
10	**Frame Signal TxFSX (Output):** Outputs the frame signal generated by the internal frame signal generation logic.
11	**Frame Signal TxFSX (Input):** Allows for use of an externally generated TxFSX. The frame sync signal is provided via TxFSX pin. All general purpose I/O functions are unavailable. (RESET default)

17.10.2 SSI Control/Status Register (SSI_CSR)

SSI_CSR is a 32-bit read/write register that controls the SSI unit and shows the current status of the SSI module. The default value after hardware reset is 0x0000F000.

Table 17-8. SSI control/status register (SSI_CSR) fields

Field	Description
TMS	Test Mode Select (Bit 31-30). Value should only be changed when the transmitter and receiver are disabled. See Table 17-9.
CDE	Change Detector Enable (Bit 29). CDE enables the change detector function on the SSI_IO1 pin. When CDE is set, the DSPCPU will be interrupted when CDS in the SSI status register is set. When CDE is cleared, this interrupt is disabled. However, the CDS bit will always indicate the change detector condition. When the change detector is enabled, the CLK samples SSI_IO1. The CDS bit will be set for either a '0' –> '1' or a '1' –> '0' change between the current value and the stored value.
CD2	RXCLK Divider (Bit 28). When CD2 = '1', the internal RxCLK is divided by two. In the divide by 2 mode, the clock edge that samples the asserted Frame Sync Pulse will resync the RxCLK divider to be a data capture edge. Data samples will occur every other clock thereafter until the end of the valid slots in the frame.
SLP	Sleepless (Bit 27). When set, this bit allows the SSI to ignore the global power down signal. If cleared, assertion of the global power down signal will cause the SSI transmitter to finish transmission of the current 16-bit word, then enter a state similar to transmitter disabled, (SSI_CTL.TXE = '0'). In the receiver, a 16-bit word currently being transmitted to RxSR will complete reception and be transferred to the RxFIFO. The receiver will then enter a state similar to receiver disabled, (SSI_CTL.RXE = '0').
CTUE	Clear Transmitter Underrun Error (Bit 21). A control bit written by the DSPCPU to indicate that the transmitter underrun error flag should be cleared. This is an action bit. Writing a '1' clears SSI_CSR.TUE. The bit always reads '0'.
CROE	Clear Receiver Overrun Error (Bit 20). A control bit written by the DSPCPU to indicate that the receiver overrun error flag should be cleared. This is an action bit. Writing a '1' clears SSI_CSR.TOE. The bit always reads '0'.
CFES	Clear Framing Error Status (Bit 19). A control bit written by the DSPCPU to indicate that the receiver's framing error flag should be cleared. This is an action bit. Writing a '1' clears SSI_CSR.FES. The bit always reads '0'.
CCDS	Clear Change Detector Status (Bit 18). A control bit written by the DSPCPU to indicate that the change detector status on IO1 flag should be cleared. This is an action bit. Writing a '1' clears SSI_CSR.CDS. The bit always reads '0'.
WAW	Word buffers Available for Write (Bit 15-12). The WAW[3:0] bits provide the number of 32-bit words available for write in the transmit buffer (TxFIFO). The SSI can store 15 words in the transmit FIFO. When the FIFO is empty, WAW = '15'. When the FIFO is full, WAW = '0' and the SSI will ignore any further attempts to add words to the FIFO. Note: The fill routine should check that WAW is nonzero, before writing data.
WAR	Word buffers Available for Read (Bit 11-8). The WAR[3:0] bits provide the number of 32-bit word available for read in the receive buffer (RxFIFO). The SSI can store 16 words in the receive FIFO. However, the maximum value indicated by the WAR register = '15' (because it's a 4-bit register field). When the FIFO is empty, WAR = '0'. When the FIFO is full, WAR = '15' and the SSI will generate an overrun error if more data is received.
TDE	Transmit Data register Empty (Bit 7). In normal operation, this bit will be set when the number of empty words in the TxFIFO is greater than the Interrupt Level Select value, SSI_CTL.ILS. If SSI_CTL.TIE is set, the SSI will generate an interrupt. When set, it indicates that the SSI_TxDR/TxFIFO registers require DSPCPU service for refilling after normal transmission. As the DSPCPU refills the TxFIFO during the interrupt service routine, this bit will be cleared by the SSI when the number of empty slots drops below the value of SSI_CTL.ILS.
RDF	Receive Data register Full (Bit 6). In normal operation, this bit will be set when the number of words in the RxFIFO is greater than SSI_CTL.ILS. If SSI_CTL.RIE is set, the SSI will generate an interrupt. When set, this bit indicates that normal received data resides in SSI_RxDR register and RxFIFO buffer for reading. DSPCPU must service the RxFIFO before a receiver overrun occurs.
TUE	Transmitter Underrun Error (Bit 5). No current data was available from the TxFIFO when a load of the TxSR was scheduled. The transmitted message may have been corrupted. Generates interrupt if enabled by TIE.
ROE	Receive Overrun Error (Bit 4). No RxFIFO slot in which to store received data. These bits have been lost and the message stream is incomplete. Generates an interrupt if enabled by RIE.
FES	Frame Error (Bit 3). A frame sync pulse has been detected where not expected or did not occur as expected during transmit or receive. Received data may be invalid. Transmit data have been sent out of sync. Receive frame error RXFES generates an interrupt if enabled by RIE. Transmit frame error TXFES generates an interrupt if enabled by TIE
CDS	Change Detector Status (Bit 2). The input change detector on SSI_IO1 pin has detected a change in state.
RIO1	Read IO1 (bit 1). RIO1 reflects the value on the SSI_IO1 pin.
RIO2	Read IO2 (bit 2). RIO2 reflects the value on the SSI_IO2 pin.

Table 17-9. Test mode select

Bit	Mode
0X	**Normal Operation.**
10	**Remote Loopback Test:** Direct connection of receiver serial data to transmitter serial data. Transmitter is clocked with RxCLK. No data loaded to the SSI_RxDR register or RxFIFO buffer and no CPU interrupt is generated. Useful to allow remote device to test the communication medium and the Rx and Tx front ends.
11	**Local Loopback Test:** Feedback is after SSI_TxDR and SSI_RxDR register and serializer/deserializer. Allows DSPCPU to test the bulk of the Rx and Tx circuits. During Local Loopback Test, an external clock on SSI_RXCLK should be present to clock the SSI unit.

17.11 TIMING DIAGRAMS

Figure 17-11 and Figure 17-12 illustrate the timing of the data signals and the frame timing.

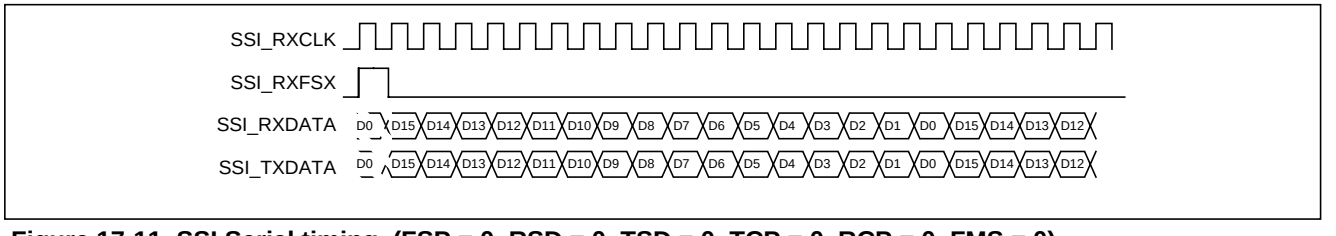


Figure 17-11. SSI Serial timing. (FSP = 0, RSD = 0, TSD = 0, TCP = 0, RCP = 0, FMS = 0)

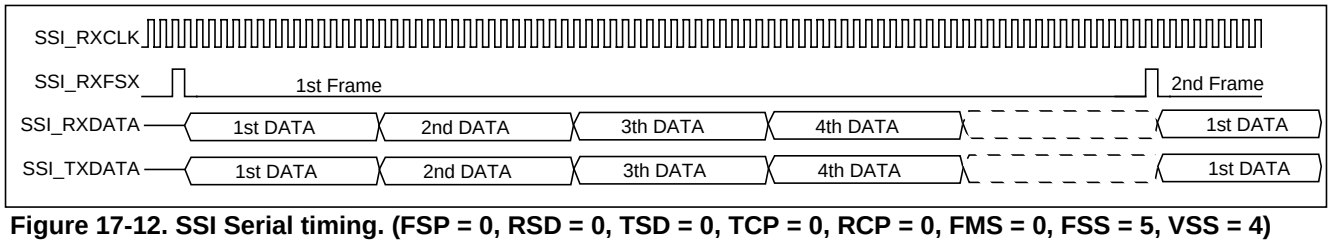


Figure 17-12. SSI Serial timing. (FSP = 0, RSD = 0, TSD = 0, TCP = 0, RCP = 0, FMS = 0, FSS = 5, VSS = 4)

17.12 POWER DOWN

SSI block can be separately powered down by setting a bit in the BLOCK_POWER_DOWN register. For a description of powerdown, see Chapter 21, “Power Management.” The SSI block should not be active when applying block powerdown.

If the block enters power-down state while transmission is enabled, behavior upon power-up is undefined.

JTAG Functional Specification Chapter 18

by Renga Sundararajan, Hans Bouwmeester and Frank Bouwman

18.1 OVERVIEW

The IEEE 1149.1 (JTAG) standard can be used for various purposes including testing connections between integrated circuits on board level, controlling the testing of the internal structures of the integrated circuits, and monitoring and communicating with a running system.

The JTAG standard defines on-chip test logic, four or five dedicated pins collectively called the Test Access Port (TAP) and a TAP controller.

The JTAG standard defines instructions that must always be implemented by a TAP controller in order to guarantee correct behavior on board level. Apart from mandatory instructions, the standard also allows user-defined and private instructions. In TM1300, user defined and private instructions exist for debug purposes and for production test. For debug there is communication between a debug monitor running on the TM1300 DSPCPU and a debugger front-end running on a host computer. This will be explained in chapter Section 18.3

18.2 TEST ACCESS PORT (TAP)

The Test Access Port includes three or four dedicated input pins and one output pin:

- TCK (Test Clock)
- TMS (Test Mode Select)
- TDI (Test Data In)
- TRST (Test Reset, optional!)
- TDO (Test Data Out)

TRST is *not* present on TM1300.

TCK provides the clock for test logic required by the standard. TCK is asynchronous to the system clock. Stored state devices in JTAG controller must retain their state indefinitely when TCK is stopped at 0 or 1.

The signal received at TMS is decoded by the TAP controller to control test functions. The test logic is required to sample TMS at the rising edge of TCK.

Serial test instructions and test data are received at TDI. The TDI signal is required to be sampled at the rising edge of TCK. When test data is shifted from TDI to TDO, the data must appear without inversion at TDO after a number of rising and falling edges of TCK determined by the length of the instruction or test data register selected.

TDO is the serial output for test instructions and data from the TAP controller. Changes in the state of TDO must occur at the falling edge of TCK. This is because devices connected to TDO are required to sample TDO at the rising edge of TCK. The TDO driver must be in an inactive state (i.e., TDO line HIghZ) except when data scanning is in progress.

18.2.1 TAP Controller

The TAP controller is a finite state machine; it synchronously responds to changes in TCK and TMS signals. The TAP instructions and data are serially scanned into the TAP controller's instruction and data registers via the common input line TDI. The TMS signal tells the TAP controller to select either the TAP instruction register or a TAP data register as the destination for serial input from the common line TDI. An instruction scanned into the instruction register selects a data register to be connected between TDI and TDO and hence to be the destination for serial data input.

TAP controller state changes are determined by the TMS signal. The states are used for scanning in/out TAP instruction and data, updating instruction and data registers, and for executing instructions.

The controller state diagram (Figure 18-1) shows separate states for 'capture', 'shift' and 'update' of data and instructions. The reason for separate states is to leave the contents of a data register or an instruction register undisturbed until serial scan-in is finished and the update state is entered. By separating the shift and update states, the contents of a register (the parallel stage) is not affected *during* scan in/out.

The TAP controller must be in Test Logic Reset state after power-up. It remains in that state as long as TMS is held at '1'. It transitions to Run-Test/Idle state when TMS = '0'. The Run-Test/Idle state is an idle state of the controller in between scanning in/out an instruction/data register. The 'Run-Test' part of the name refers to start of built-in tests. The "Idle" part of the name refers to all other cases. Note that there are two similar sub-structures in the state diagram, one for scanning in an instruction and another for scanning in data. To scan in/out a data register, one has to scan in an instruction first.

An instruction or data register must have at least two stages, a shift register stage and a parallel input/output stage. When an n-bit data register is to be 'read', the register is selected by an instruction. The registers contents are 'captured' first (loaded in parallel into shift register stage), n bits are shifted in and at the same time n bits are shifted out. Finally the register is 'updated' with the new n bits shifted in.

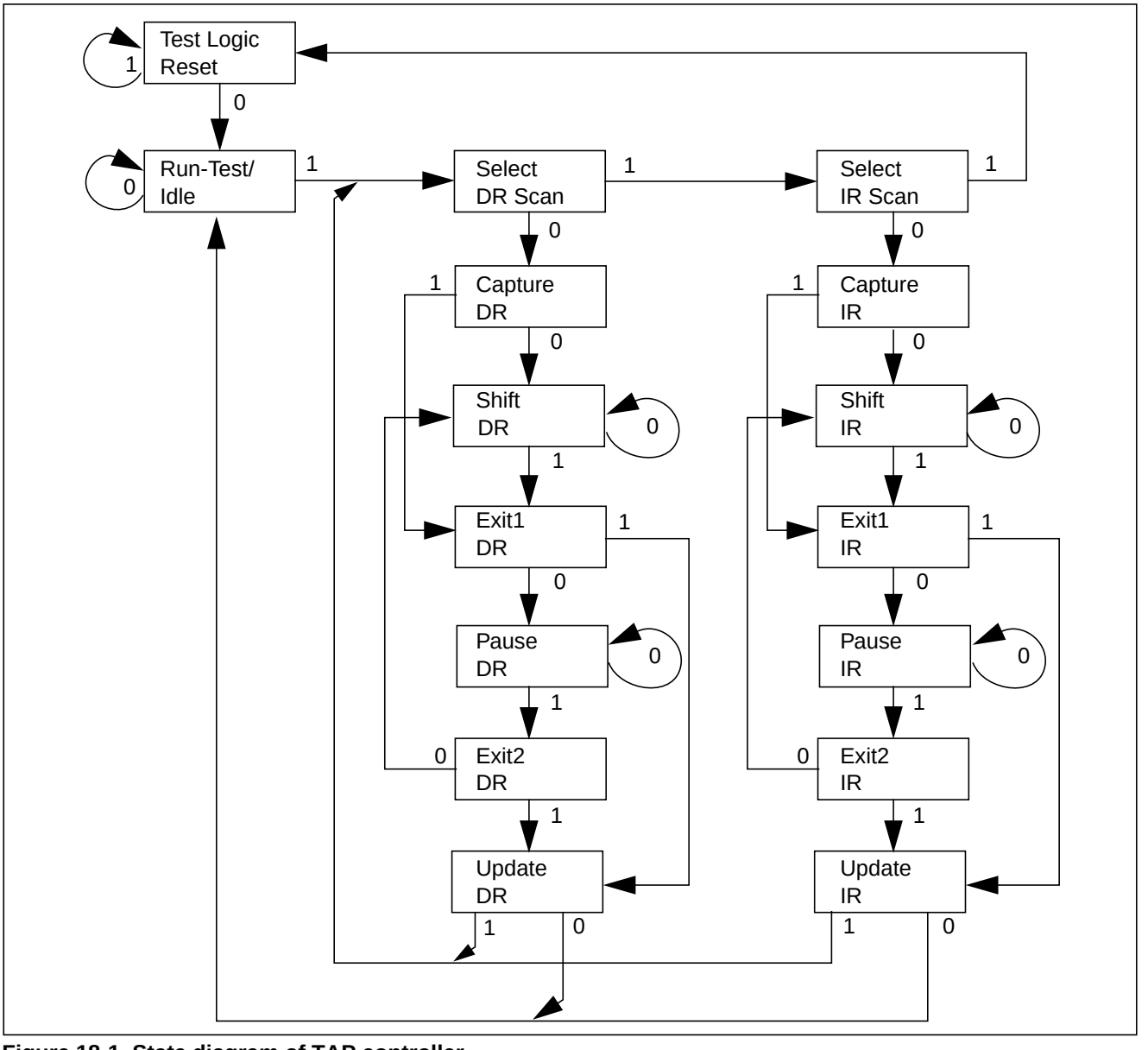


Figure 18-1. State diagram of TAP controller

Note: when a register is scanned, its old value is shifted out of TDO. The new value shifted in via TDI is written to the register at the update state. Hence, scan in/out involve the same steps. This also means that reading a register via JTAG destroys its contents *unless otherwise stated.* We can specify some registers as *read-only* via JTAG so that when the controller transitions to update state for the read-only register, the update has no effect. Sometimes, read-write registers are needed (for example, control registers used for handshake) which can be read non-destructively. In such cases, the value shifted in determines whether the old value is 'remembered' or something else happens.

18.2.2 TM1300 JTAG Instruction Set

TM1300 uses a 5-bit instruction register. The unspecified opcodes are private and their effects are undefined. Table 18-1 lists the JTAG instructions.

Table 18-1. JTAG instruction encoding

Encoding	Instruction name	Action
00000	EXTEST	Select (dummy) boundary scan register
00001	SAMPLE/PRELOAD	Select (dummy) boundary scan register
11111	BYPASS	Select bypass register
10000	RESET	Reset TriMedia to power on state
10001	SEL_DATA_IN	Select DATA_IN register
10010	SEL_DATA_OUT	Select DATA_OUT register
10011	SEL_IFULL_IN	Select IFULL_IN register
10100	SEL_OFULL_OUT	Select OFULL_OUT register
10101	SEL_JTAG_CTRL	Select JTAG_CTRL register

Table 18-1. JTAG instruction encoding

Encoding	Instruction name	Action
11110	MACRO	Hardware test mode select
01010	BURNIN	Private
01110	PASS_C_S	Private

The JTAG instructions EXTEST, SAMPLE/PRELOAD, and BYPASS are standard instructions and are not discussed here. The MACRO, BURNIN, and PASS_C_S instructions are used during hardware test mode, and are also not discussed here. All other instructions are discussed in Section 18.3

18.3 USING JTAG FOR TM1300 DEBUG

Figure 18-2 shows an overview of the JTAG access path from a host machine to a target TriMedia system and a simplified block diagram of the TriMedia processor. The JTAG Interface Module shown separately in the diagram may be a PC add-on card such as PC-1149.1/100F Boundary Scan Controller Board from Corelis Inc. or a similar module connected to a PC serial or parallel port. The JTAG interface module is necessary only for TriMedia systems that are not plugged into a PC. For PC-hosted TriMedia systems, the host based debugger front-end can communicate with the target resident debug monitor via the PCI bus.

The enhancements to the standard functionality of JTAG test logic provides a handshake mechanism for transferring data to and from a TriMedia processor's MMIO registers reserved for this purpose, for posting an interrupt, and for resetting processor state. The actual interpretation of the contents of the MMIO registers is determined by a software protocol used by the debug monitor running on the TriMedia processor and the debug front-end running on a host machine.

The communication between a host computer and a target TriMedia system via JTAG requires, at a high level of abstraction, the following components.

- **A host computer with a serial or parallel interface.**

  The host computer transfers data to and from the JTAG interface module, preferably in word-parallel fashion. A JTAG interface device driver is also needed to access and modify the registers of the JTAG interface module.

- **A JTAG interface module (hardware) that asynchronously transfers data to and from the host computer.**

  The interface module synchronously transfers data to and from the JTAG TAP on a TriMedia processor, and supplies the test clock, TCK, and other signals to the TriMedia JTAG controller. The interface module may be a PC plug-in board.

  This module may transfer data from and to the host computer in bit-serial or word-parallel fashion. It transfers data from and to the JTAG registers on a TriMedia processor in bit-serial fashion in accordance with the IEEE 1149.1 standard. The JTAG interface

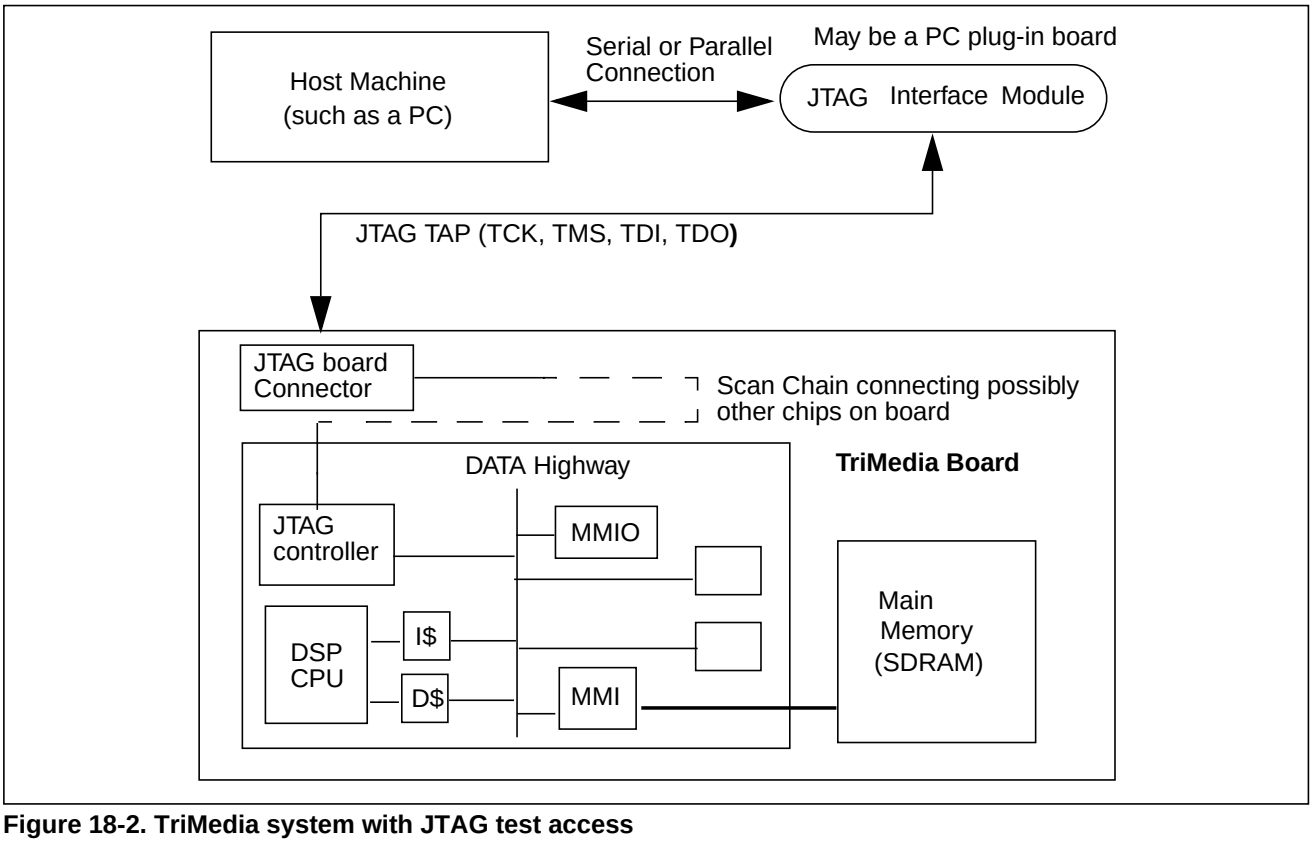


Figure 18-2. TriMedia system with JTAG test access

module connects to a 4-pin JTAG connector on a TriMedia board which provides a path to the JTAG pins on a TriMedia processor. It is the responsibility of the interface module to scan data in and out of the TriMedia processor into its internal buffers and make them available to the host computer.

- **A JTAG controller on the TriMedia processor which provides a bridge between the external JTAG TAP and the internal system.**

  The controller transfers data from/to the TAP to/from its scannable registers asynchronous to the internal system clock. A monitor running on a TriMedia processor and the debugger front-end running on a host computer exchange data via JTAG by reading/writing the MMIO registers reserved for this purpose, including a control register used for the handshake.

18.3.1 JTAG Instruction and Data Registers.

Table 18-2. MMIO Register Assignments

MMIO Offset	JTAG Register
0x 10 3800	JTAG_DATA_IN
0x 10 3804	JTAG_DATA_OUT
0x 10 3808	JTAG_CTRL

TM1300 has two JTAG data registers and one JTAG control register (see Figure 18-3) in MMIO space and a number a JTAG instructions to manipulate those registers. Table 18-2 lists the MMIO addresses of the JTAG data and control registers. The addresses are offsets from MMIO_BASE. All references to instruction and data registers below are JTAG instructions and data registers and not TriMedia instruction or data registers.

- **Two 32-bit data registers, JTAG_DATA_IN and JTAG_DATA_OUT in MMIO space.** Both registers can be connected in between TDI and TDO like the standard Bypass and Boundary Scan registers of JTAG (not shown in Figure 18-3).

  The JTAG_DATA_IN register can be read or written to via the JTAG port. The JTAG_DATA_OUT register is read-only via the JTAG port, so that scanning out JTAG_DATA_OUT is non-destructive.

  The JTAG_DATA_IN and JTAG_DATA_OUT are readable/writable from the TriMedia processor via the usual load/store operations.

- **An 8-bit control register JTAG_CTRL in MMIO space.** The JTAG_CTRL register is used for handshake between a debug monitor running on a TriMedia and a debugger front-end running on a host.

  JTAG_CTRL.ofull = '1' means that JTAG_DATA_OUT has valid data to be scanned out. On power-on reset of the TriMedia processor, JTAG_CTRL.ofull = '0'. JTAG_CTRL.ofull is both readable and writable via JTAG tap. Writing 0 to JTAG_CTRL.ofull via JTAG is a 'remember' operation, i.e., JTAG_CTRL.ofull retains its previous state. Writing a '1' to JTAG_CTRL.ofull via JTAG is a 'clear' operation, i.e., JTAG_CTRL.ofull becomes '0'.

  JTAG_CTRL.ifull = '0' means that the JTAG_DATA_IN register is empty. JTAG_CTRL.ifull = 1 means that JTAG_DATA_IN has valid data and the debug monitor has not yet copied it to its private area. On power-on reset of the TriMedia processor, JTAG_CTRL.ifull = 0. JTAG_CTRL.ifull is readable and writable via JTAG. Writing a '0' to JTAG_CTRL.ifull via JTAG is a remember operation, i.e., JTAG_CTRL.ifull retains it previous state. Writing a '1' to JTAG_CTRL.ifull posts an interrupt on hardware line 18.

  The peripheral blocks on a TriMedia processor may enter a 'power down' state to reduce power consumption. The JTAG_CTRL.sleepless bit determines if the JTAG block participates in a power down state. In the power-on RESET state, JTAG_CTRL.sleepless bit is '1' meaning the JTAG block does not power down. It can be read and written to by the TriMedia processor via load/store operations and by the debugger front-end running on a host by scan in/out.

- **Two virtual registers, JTAG_IFULL_IN and JTAG_OFULL_OUT.** The first virtual register JTAG_IFULL_IN connects the registers JTAG_CTRL.ifull and JTAG_DATA_IN in series. Likewise, the virtual register JTAG_OFULL_OUT connects JTAG_CTRL.ofull and JTAG_DATA_OUT in series.

  The reason for the virtual registers is to shorten the time for scanning the JTAG_DATA_IN and

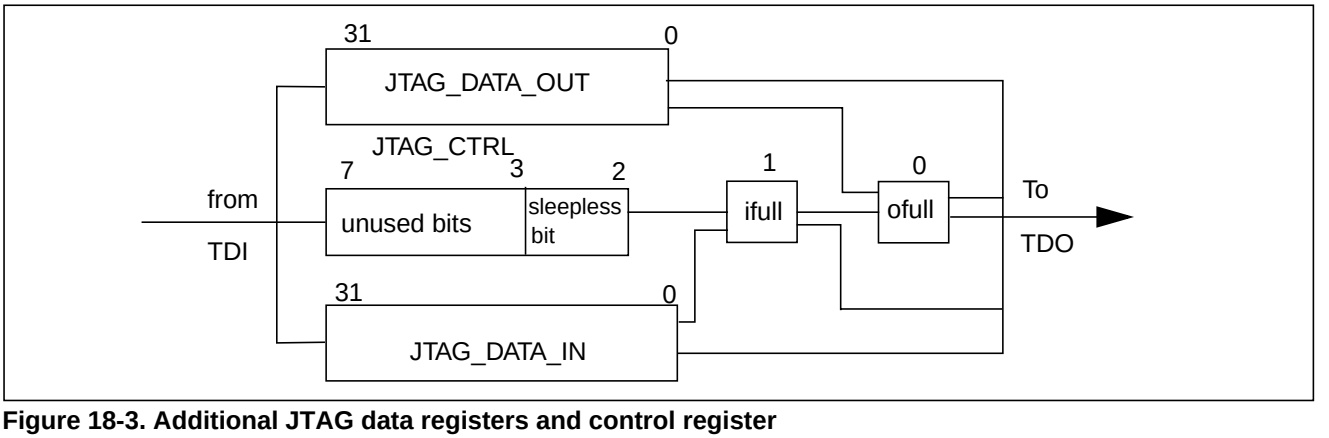


Figure 18-3. Additional JTAG data registers and control register

JTAG_DATA_OUT registers. Without virtual registers, we must scan in an instruction to select JTAG_DATA_IN, scan in data, scan an instruction to select JTAG_CTRL register and finally scan in the control register. With virtual register, we can scan in an instruction to select JTAG_IFULL_IN and then scan in both control and data bits. Similar savings can be achieved for scan out using virtual registers.

- **Five JTAG instructions**
  - 5 instructions, SEL_DATA_IN, SEL_DATA_OUT, SEL_IFULL_IN, SEL_OFULL_OUT, and SEL_JTAG_CTRL, for selecting the registers to be connected between TDI and TDO for serial input/output.
  - An instruction RESET for resetting the TriMedia processor to power on state.
- In the capture-IR state of the TAP controller, the least 2 significant bits (bits 0 and 1) of the shift register stage must be loaded with the '01' as required in the standard. The standard allows the remaining bits of the IR shift stage to be loaded with design specific data. The bits 2, 3 and 4 of the IR shift stage are loaded with bits 0, 1 and 2 of the JTAG_CTRL register. This means that shifting in any instruction allows the 3 least significant bits of the JTAG_CTRL register to be inspected. This reduces the polling overhead for data transfer.

Race Conditions

Since the JTAG data registers live in MMIO space and are accessible by both the TriMedia processor and the JTAG controller at the same time, race conditions must not exist either in hardware or in software. The following communication protocol uses a handshake mechanism to avoid software race conditions.

18.3.2 JTAG Communication Protocol

The following describes the handshake mechanism for transferring data via JTAG.

- **Transfer from debug front-end to debug monitor**

  The debugger front-end running on a host transfers data to a debug monitor via JTAG_DATA_IN register. It must poll JTAG_CTRL.ifull bit to check if JTAG_DATA_IN register can be written to. If the JTAG_CTRL.ifull bit is clear, the front-end may scan data into JTAG_DATA_IFULL_IN register. Note that data and control bits may be shifted in with SEL_IFULL_IN instruction and the bit shifted into JTAG_CTRL.ifull register must be '1'. This action triggers an interrupt. The debug monitor must copy the data from JTAG_DATA_IN register into its private area when servicing the interrupt and then clear JTAG_CTRL.ifull bit thus allowing JTAG interface module to write to JTAG_DATA_IN register the next piece of data.
- **Transfer from monitor to front-end**

  The monitor running on TriMedia must check if JTAG_CTRL.ofull is clear and if so, it can write data to JTAG_DATA_OUT. After that, the monitor must set the JTAG_CTRL.ofull bit. The debugger front-end polls the JTAG_CTRL.ofull bit. When that bit is set, it can scan out JTAG_DATA_OUT register and clear JTAG_CTRL.ofull bit. Since JTAG_DATA_OUT is *read-only* via JTAG, the update action at the end of scan out has no effect on JTAG_DATA_OUT. The JTAG_CTRL.ofull bit, however, must be cleared by shifting in the value '1'.
- **Controller States**

  In the power-on reset state, JTAG_CTRL.ifull and JTAG_CTRL.ofull must be cleared by the JTAG controller.

18.3.3 Example Data Transfer Via JTAG

Scanning in a 5-bit instruction will take 12 TCK cycles from the Run-Test/Idle state: 4 cycles to reach Shift-IR state, 5 cycles for actual shifting in, 1 cycle to exit1-IR state, 1 cycle to Update-IR state, and 1 cycle back to Run-Test/Idle state. Likewise, scanning in a 32 bit data register will take 38 TCK cycles and transferring an 8-bit JTAG_CTRL data register will take 14 TCK cycles from Idle state. However, if a data transfer follows instruction transfer, then the transition to DR scan stage can be done without going through Idle state, saving 1 cycle.

18.3.3.1 Transferring data to TriMedia via JTAG

Poll control register to check if input buffer is empty. Scan in data when it is empty and set the ifull control bit to '1' triggering an interrupt. Note that scanning in any instruction automatically scans out the 3 least significant bits (including ifull and ofull bits) of the JTAG_CTRL register.

Table 18-3. Transfer of Data in via JTAG

Action	Number of TCK cycles
IR shift in SEL_IFULL_IN instruction	12
While JTAG_CTRL.ifull = 1, scan in SEL_IFULL_IN instruction	11+
DR scan 33 bits of register JTAG_IFULL_IN	38
TOTAL	61+ cycles

18.3.3.2 Transferring data from TriMedia via JTAG

Poll control register to check if output buffer is full. Scan out data when it is full and clear the ofull control bit. Note that scanning in any instruction automatically scans out the 3 least significant bits (including ifull and ofull bits) of JTAG_CTRL register.

Table 18-4. Transfer of Data out via JTAG

Action	Number of TCK cycles
IR shift in SEL_OFULL_OUT instruction	12
While JTAG_CTRL.ofull = 0, scan in SEL_OFULL_OUT instruction	11+
DR scan 33 bits of register JTAG_OFULL_OUT	38
TOTAL	61+ cycles

Note that the above timings do not include the overheads of the JTAG software driver for JTAG interface module plugged into a PC.

18.3.4 JTAG Interface Module

It is expected that the interface module will be a programmable JTAG interface module. One end of the module should be connected to a JTAG tap and the other end to a host computer via a serial or parallel line or plugged into a PC. It is up to the JTAG driver software on a host computer to program the JTAG interface module via the serial/parallel interface for transferring data to/from the target. The transfer rates will depend on the interface module.

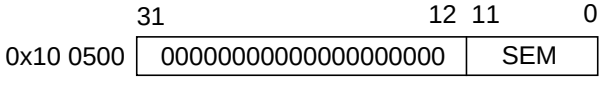
TM1300 has a simple MP semaphore-assist device. It is a 32-bit register, accessible through MMIO by either the local TM1300 CPU or by any other CPU on PCI through the aperture made available on PCI. The semaphore, SEM, is located at MMIO offset 0x10 0500.

SEM operation is as follows: each master in the system constructs a personal nonzero 12 bit ID (see below). To obtain the global semaphore, a master does the following action:

```
write ID to SEM (use 32 bit store, with ID in 12 LSB)
retrieve SEM    (use 32 bit load, it returns 0x00000nnn)
if (SEM = ID) {
    “performs a short critical section action”
    write 0 to SEM
}
else “try again later, or loop back to write”
```

19.1 SEM DEVICE SPECIFICATION

SEM is a 32-bit MMIO location. The 12 LSB consist of storage flip-flops with surrounding logic, the 20 MSBs always return a ‘0’ when read.



SEM is RESET to ‘0’ by power up reset.

When SEM is written to, the storage flip-flops behave as follows:

```
if (cur_content == 0)        new_content = write_value;
else if (write_value == 0)   new_content = 0;
/* ELSE NO ACTION ! */
```

19.2 CONSTRUCTING A 12-BIT ID

A TM1300 processor can construct a personal, nonzero 12-bit ID in a variety of ways. Below are some suggestions.

PCI configspace PERSONALITY entry. Each TM1300 receives a 16-bit PERSONALITY value from the EEPROM during boot. This PERSONALITY register is located at offset 0x40 in configuration space. In a MP system, some of the bits of PERSONALITY can be individualized for each CPU involved, giving it a unique 2/3/4-bit ID, as needed given the maximum number of CPUs in the design.

In the case of a host-assisted TM1300 boot, the PCI BIOS assigns a unique MMIO_BASE and DRAM_BASE to every TM1300. In particular, the 11 MSBs of each MMIO_base are unique, since each MMIO aperture is 2 MB in size. These bits can be used as a personality ID. Set bit 11 (MSB) to '1' to guarantee a nonzero ID#.

19.3 WHICH SEM TO USE

Each TM1300 in the system adds a SEM device to the mix. The intended use is to treat one of these SEM devices as THE master semaphore in the system. Many methods can be used to determine which SEM is master SEM. Some examples below:

Each DSPCPU can use PCI configuration space accesses to determine which other TM1300s are present in the system. Then, the TM1300 with the lowest PERSONALITY number, or the lowest MMIO_base is chosen as the TM1300 containing the master semaphore.

19.4 USAGE NOTES

To avoid contention on the master SEM device, it should only be used for inter-processor semaphores. Processes running on a single CPU can use regular memory to implement synchronization primitives.

The critical section associated with SEM should be kept as short as possible. Preferably, SEM should only be used as the basis to make multiple memory-resident simple semaphores. In this case, the non-cacheable DRAM area of each TM1300 can be used to implement the semaphore data structures efficiently.

As described here, SEM does not guarantee starvation-free access to critical resources. Claiming of SEM is purely stochastic. This should work fine as long as SEM is not overloaded. Utmost care should be taken in SEM access frequency and duration of the basic critical sections to keep the load conditions reasonable.

Arbiter Chapter 20

by Eino Jacobs, Luis Lucas, Chris Nelson, Allan Tzeng, Gert Slavenburg

20.1 ARBITER FEATURES

The TM1300 internal highway bus conveys all the memory and MMIO traffic. The on-chip peripheral units described in this databook are connected to this internal highway bus. Accesses to the bus are controlled by a central arbiter. Figure 2-1 on page 2-1 shows the whole system where the arbiter is embedded in the main memory interface (MMI) block. The traffic includes the memory requests issued by most of the on-chip units as well as the MMIO transactions issued by the DSPCPU or PCI block and responded to by the peripherals.

The arbiter was designed to make TM1300 a true real-time system by providing a highly programmable bus bandwidth allocation scheme. The primary characteristics are:

- round robin arbitration
- hierarchical organization
- programmable allocation of highway bandwidth
- dual priorities with priority raising mechanism

These features are explained in the next sections of this chapter. The arbiter is programmed through two MMIO registers:

- ARB_RAISE
- ARB_BW_CTL

The default values (after hardware RESET) stored in these two MMIO registers are suitable for most of the applications. If these default settings introduce violations of real-time constraints in units like Video In (VI), Video Out (VO), Audio In (AI) and Audio Out (AO) (each of these units has a Highway Bandwidth Error detection mechanism), the ARB_BW_CTL register should be programmed to 0x090A9. This setting gives almost maximum priority to real-time units but may slow down the CPU.

Fine tuning of the arbiter settings is described in the following sections.

20.2 DUAL PRIORITIES WITH PRIORITY RAISING MECHANISM

The best CPU performance is obtained if cache misses can take priority over peripheral requests on the highway. However, peripherals need to have a maximum guaranteed latency low enough to satisfy the real-time constraints of I/O units.

TM1300 provides this feature with the following priority-raising mechanism.

Peripheral unit requests can have 2 priorities: low and high. Within each class there is fair, round-robin arbitration (Section 20.3). Requests with high priority take precedence over requests with low priority.

Units can indicate the priority of their requests to be low or high.

A unit may initially post a request with low priority. If the request is not serviced within a particular waiting time, the unit can raise the priority of the request to high. This can be done when the worst case latency at high priority approaches the real-time constraint of the unit. Thus, the unit uses only spare bandwidth without slowing down the CPU unless real-time constraints require it to claim high priority.

In TM1300, only the ICP unit has its own priority raising logic (i.e. it controls the low to high transition of the request). Refer to Chapter 14, “Image Coprocessor,” for more information.

Priority raising for the VLD, PCI, VI and VO units is handled by the arbiter central priority raising mechanism. The central priority raising mechanism settings are controlled from the DSPCPU with the ARB_RAISE MMIO register (see Table 20-1). The delay is the amount of time for which the arbiter handles the request at low priority.

The delay is defined by a 5-bit field (dedicated per unit) and is counted in CPU clock cycles. The granularity of the delay is 16 cycles, so the maximum time spent at low priority for each request can be programmed from 0 to 496 cycles, inclusive, in increments of 16 cycles.

Table 20-1. ARB_RAISE register layout

Offset	Name	Bits	Fields
0x10010C	ARB_RAISE	19:15	VLD_delay[4:0]
		14:10	PCI_delay[4:0]
		9:5	VI_delay[4:0]
		4:0	VO_delay[4:0]

The default value for the entire ARB_RAISE register is ‘0’. This causes all requests from VLD, PCI, VI and VO to be handled as high-priority requests until the ARB_RAISE register contents has been changed for the application requirements.

Corner-case note: There is some risk in setting the delay high, then lowering it, as the last request submitted with the high delay might violate the latency constraints of the new real-time domain. However this should not happen since this register should be set before the application starts.

The other units (AI, AO and BTI (boot block)) and the CPU will always have their requests considered as high priority. High priority for the CPU will give maximum possible performance.

AO and AI requests are happening at very low rate. Hence, the probability that they take time away from the CPU is negligible.

20.3 ROUND ROBIN ARBITRATION

In addition to the dual priority mechanism, a round-robin arbitration is used to schedule the requests with same priority. The purpose is to ensure, for every unit with a high-priority request, a maximum latency for gaining access to the highway and/or a minimum share of the available bandwidth.

Round-robin arbitration ensures that no starvation of requests can occur and therefore requests with real-time constraints can be handled in time.

The round robin arbitration algorithm is as follows.

Requests are granted according to a dynamic priority list. Whenever a unit request is granted, it will be moved to the last position in the priority list and another unit will be moved to the first position in the priority list. Priorities are rotated. A unit with a waiting request will eventually reach the first place in the priority list.

As an example, Figure 20-1 shows a state diagram of an arbitration state machine with 2 requesters. The nodes A and B indicate states A and B. In state A, requester A has ownership of the highway, in state B requester B has ownership. The arc from state A to state B indicates that if the current state is state A and a request from requester B is asserted, then a transition to state B occurs, i.e. ownership of the highway passes from requester A to requester B.

When, in a particular state, none of the arcs leaving from that node has its condition fulfilled, the state machine remains in the same state.

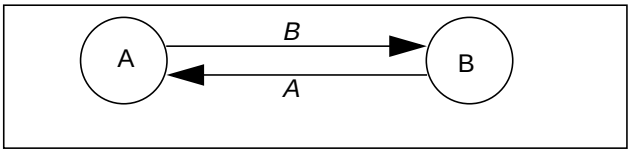


Figure 20-1. State diagram of round robin arbitrator with 2 requesters.

When both requester A and B have requests asserted, then ownership of the highway switches between A and B, creating fair allocation of ownership.

Figure 20-2 pictures a state diagram that allocates fair arbitration with 3 requesters.

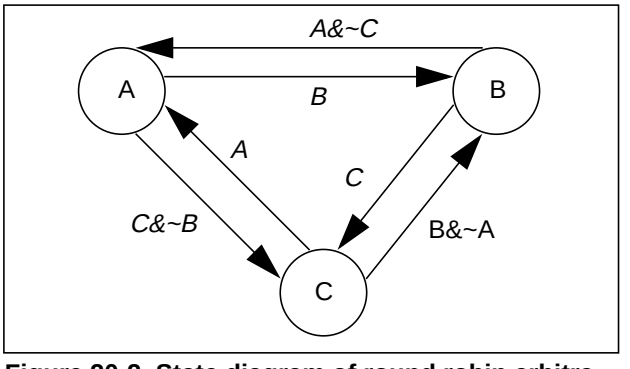


Figure 20-2. State diagram of round robin arbitrator with 3 requesters.

20.3.1 Weighted Round Robin Arbitration

Not all units need to have equal latency and bandwidth. It is preferred to allocate bandwidth to units according to their needs. This is achieved with weighted round-robin and can be illustrated in the following examples.

Figure 20-3 pictures a state machine with two requesters A and B with double weight given to requester A. There are now 2 states A1 and A2 where requester A has ownership of the highway. When both A and B requests are asserted, requester A will have ownership of the highway twice as often as requester B.

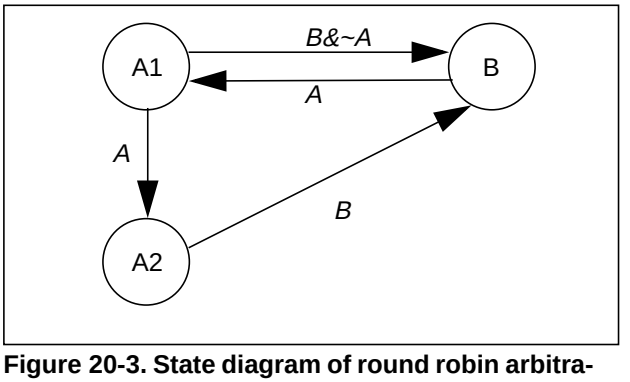


Figure 20-3. State diagram of round robin arbitrator with 2 requesters; A has double weight.

Figure 20-4 shows a state machine with 3 requesters in which double weight is given to requester A.

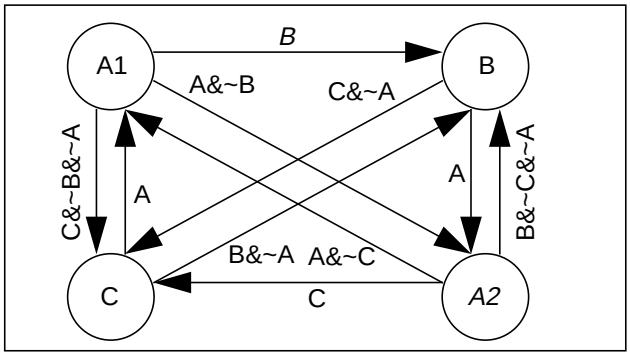


Figure 20-4. State diagram of round robin arbitrator with 3 requesters; A has double weight.

Such state machines can become very complex and cannot be implemented for a large system like TM1300 with 9 requesters. Hierarchy or arbitration levels are used to overcome this problem.

20.3.2 Arbitration Levels

The arbitration is split into multiple levels of hierarchy. Each level of hierarchy has an independent arbitration state machine. At the bottom of the hierarchy, the arbitration is performed between a group of units. Whichever of these units 'wins' is passed to the next level of hierarchy, where the selected unit competes with other units at that level for highway access.This is continued until the highest level of arbitration.

By splitting arbitration into multiple levels it is easy to support a large number of highway units while the complexity of the arbitration state machines at each level of hierarchy remains modest.

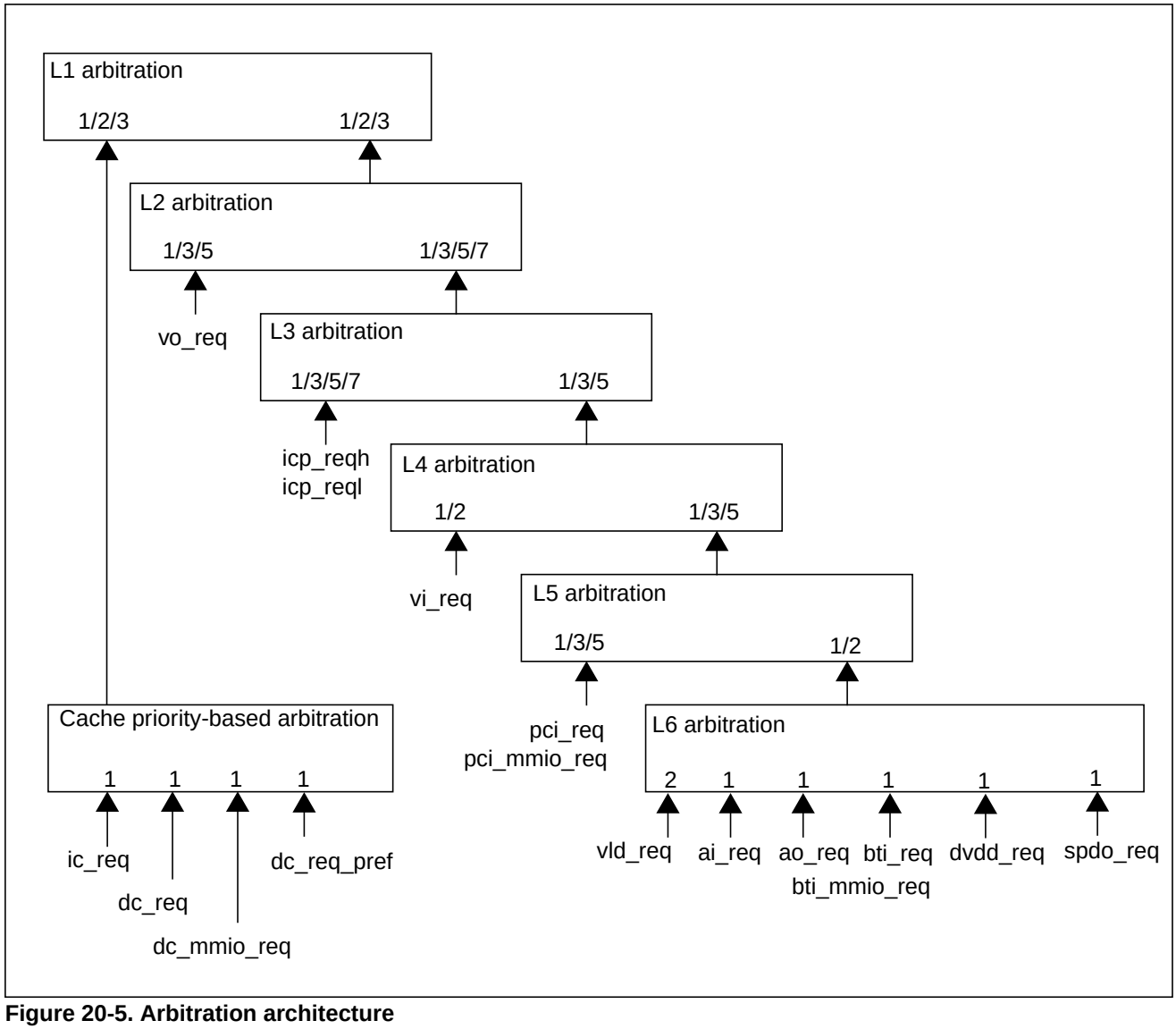


Figure 20-5. Arbitration architecture

Hierarchy also makes it easy and natural to allocate bus bandwidth or latency to a group of units. Most bandwidth or latency-demanding units are located at the top of the hierarchy while the less demanding are at the bottom and get a small amount of overall bandwidth.

Table 20-2. Minimum bandwidth allocation between CPU caches and peripheral units.

weight of CPU and caches	weight of level 2	bandwidth at level 1	bandwidth at level 2
3	1	75%	25%
2	1	67%	33%
3	2	60%	40%
1	1	50%	50%
2	3	40%	60%
1	2	33%	67%
1	3	25%	75%

20.4 ARBITER ARCHITECTURE

In addition to the dual priority mechanism described in Section 20.2, TM1300 supports an arbitration architecture made of 6 fixed levels of hierarchy. This is combined with a programmable weighted round robin algorithm per level, as pictured in Figure 20-5.

Table 20-3. Arbitration weights at each level

Level	Arbitration Weights
level 1:	CPU MMIO, Dcache, Lcache are arbitrated with fixed priorities between each other and together have a programmable weight of 1, 2 or 3. Level 2 has a programmable weight of 1, 2 or 3.
level 2:	VO unit has a programmable weight of 1, 3 or 5. Level 3 has a programmable weight of 1, 3, 5 or 7.
level 3:	The ICP unit has a programmable weight of 1,3,5 or 7. Level 4 has a programmable weight of 1,3 or 5.
level 4	The VI unit has a programmable weight of 1 or 2. Level 5 has a programmable weight of 1,3 or 5.
level 5:	The PCI unit has a programmable weight of 1,3 or 5. Level 6 has a programmable weight of 1 or 2.
level 6:	Level 6 contains several lower bandwidth and/or latency-tolerant units. The VLD has a weight of 2. AI, AO, DVDD and the boot block (only active during booting) have a weight of 1.

The weights can be adjusted by software to allocate bandwidth and latency depending on application requirements. Within a level of hierarchy the units can have equal weights, giving them an equal share of bandwidth. Alternatively, they can have different weights, giving them an unequal share of the bandwidth for that level.

The arbitration weights at each level are described in Table 20-3 and illustrated in Figure 20-5.

Table 20-2 presents the minimum bandwidth allocation at Level 1 between the DSPCPU and the peripherals (level 2) according to the different weight values that can be programmed. Note that programming a weight of 3/3 or 2/2 instead of 1/1 is legal and results in the same allocation.

Note: The different types of requests from the DSPCPU caches are arbitrated locally before sending a single CPU request to the arbiter. The PCI bus also performs local arbitration before sending a system request to the arbiter.

The weight programming is done by setting the MMIO register ARB_BW_CTL. Register offset as well as field description and coding is provided in Table 20-4.

The hardware RESET value of ARB_BW_CTL is 0, resulting in a weight of 1 for all requests.

Note that each media processor application needs to carefully review its arbiter settings.

Table 20-4. ARB_BW_CTL MMIO register

Offset	level of arbitration	field	bits	allowed values
0x100104	n/a	RESERVED	25:18	
	level 1	CPU weight	17:16	00 = weight 1 01 = weight 2 10 = weight 3
	level 1	L2 weight	15:14	00 = weight 1 01 = weight 2 10 = weight 3
	level 2	VO weight	13:12	00 = weight 1 01 = weight 3 10 = weight 5
	level 2	L3 weight	11:10	00 = weight 1 01 = weight 3 10 = weight 5 11 = weight 7
	level 3	ICP weight	9:8	00 = weight 1 01 = weight 3 10 = weight 5 11 = weight 7
	level 3	L4 weight	7:6	00 = weight 1 01 = weight 3 10 = weight 5
	level 4	VI weight	5	0 = weight 1 1 = weight 2
	level 4	L5 weight	4:3	00 = weight 1 01 = weight 3 10 = weight 5
	level 5	PCI weight	2:1	00 = weight 1 01 = weight 3 10 = weight 5
	level 5	L6 weight	0	0 = weight 1 1 = weight 2

20.5 ARBITER PROGRAMMING

The TM1300 arbiter accepts programmable bandwidth weights to directly control the percentage of bandwidth allocated to each unit. In the worst case all bandwidth is used. If not all of the bandwidth is used, then all units eventually get their desired bandwidth (as the bus becomes free) **regardless of the weights**. However, the weights still indirectly guarantee each unit a worst-case latency, which is important for the real-time behavior.

There are two basic types of TM1300 coprocessor and peripheral units. The first type is units which have hard real-time constraints, i.e. VO, VI, AO and AI. To ensure multimedia functionality, these units must be able to acquire the bus within a fixed amount of time in order to fill or empty a buffer before it over- or underflows.

The second type, the CPU, PCI, ICP, VLD and DVDD units, can absorb long latencies but performance is enhanced (there are fewer stall cycles or waiting cycles) if latency is short. The bandwidth requirement is usually known and depends on the application. It is especially well known that ICP and VLD or DVDD have a fixed bandwidth requirements in multimedia applications.

For the TM1300 DSPCPU, latency is of prime importance. CPU performance reduces as average latency increases. The design of the arbiter guarantees that the DSPCPU gets all unused bus bandwidth with lowest possible latency. Optimal operation is achieved if the arbiter is set in such a way that the DSPCPU has the best possible latency given the required latency and bandwidth of units active in the application.

To pick programmable weights and priority raising delays, the following procedure is recommended:

1. Try to keep CPU weight as high as possible through the remaining steps.
2. Pick weights sufficient to guarantee latency to hard real-time peripherals (see Section 20.5.1).
3. Pick weights for remaining peripherals in order to give enough bandwidth to each (see Section 20.5.2). Step 2 above has priority, because bandwidth can be acquired as the bus becomes free and because the hard real-time units use a known amount of bandwidth.
4. If latency and bandwidth slack remains, increase priority raise delays in order to improve average CPU latency.

20.5.1 Latency Analysis

In the following, ceil(X) is the least integral value greater than or equal to X.

Latency is defined in each real-time unit chapter through this databook. Refer to the related sections to find out the latency requirement according to the mode and clock speed at which the unit is operating.

This latency value has to be larger than the maximum latency L_x (in nanoseconds) guaranteed by the arbiter.

For a unit x the arbiter guarantees a latency of:

$L_x = L_{x,sc}$ * (SDRAM cycle time in ns)

where

$L_{x,sc} = (D_x * T) + E + ceil(D_x * T / K_d) * K + ceil(16*R_x/C)$

is the latency in SDRAM clock cycles.

Latency in CPU clock cycles is defined by:

$L_{x,cc} = ceil(L_{x,sc} * C)$

The symbols are defined as follows:

T = 20 cycles (transaction length, assuming worst case pattern alternating reads and writes).

E = 10 cycles (extra delay in case the first transaction made by the CPU requires a different bank order to satisfy the critical word first.

K = 19 cycles (refresh transaction length).

K_d is the programmed refresh interval (see Section 12.11 on page 12-6).

C is the CPU/SDRAM ratio (i.e. 5/4, 4/3, 3/2, 2/1 or 1 as explained in Section 12.6.2 on page 12-3).

R_x is the priority raise delay of unit x as stored in MMIO register ARB_RAISE (see Section 20.2).

R_x = 0 for units other than VO, VI, PCI or VLD.

D_x is the worst case number of requests that the arbiter allows before the request from unit x goes through.

D_x includes the transaction from unit x (the unit which needs the data) as well as the internal implementation delays that occur in the transaction.

D_x is derived from the arbiter settings as follows:

$$D_{CPU} = ceil\left(\frac{CPU_{weight} + L2_{weight}}{CPU_{weight}}\right)$$

$$D_{VO} = ceil\left(\frac{VO_{weight} + L3_{weight}}{VO_{weight}}\right) \times D_2 + 1$$

$$D_{ICP} = ceil\left(\frac{ICP_{weight} + L4_{weight}}{ICP_{weight}}\right) \times D_3 + 1$$

$$D_{VI} = ceil\left(\frac{VI_{weight} + L5_{weight}}{VI_{weight}}\right) \times D_4 + 1$$

$$D_{PCI} = ceil\left(\frac{PCI_{weight} + L6_{weight}}{PCI_{weight}}\right) \times D_5 + 1$$

$$D_{VLD} = ceil\left(\frac{2 + 1 + 1 + 0 + 1 + 1}{2}\right) \times D_6 + 1$$

$$D_{AI} = ceil\left(\frac{2 + 1 + 1 + 0 + 1 + 1}{1}\right) \times D_6 + 1$$

$$D_{AO} = ceil\left(\frac{2 + 1 + 1 + 0 + 1 + 1}{1}\right) \times D_6 + 1$$

$$D_{DVDD} = ceil\left(\frac{2 + 1 + 1 + 0 + 1 + 1}{1}\right) \times D_6 + 1$$

$$D_{SPDO} = ceil\left(\frac{2 + 1 + 1 + 0 + 1 + 1}{1}\right) \times D_6 + 1$$

Where

$$D_2 = ceil\left(\frac{CPU_{weight} + L2_{weight}}{L2_{weight}}\right)$$

$$D_3 = ceil\left(\frac{VO_{weight} + L3_{weight}}{L3_{weight}}\right) \times D_2$$

$$D_4 = ceil\left(\frac{ICP_{weight} + L4_{weight}}{L4_{weight}}\right) \times D_3$$

$$D_5 = ceil\left(\frac{VI_{weight} + L5_{weight}}{L5_{weight}}\right) \times D_4$$

$$D_6 = ceil\left(\frac{PCI_{weight} + L6_{weight}}{L6_{weight}}\right) \times D_5$$

As an example, if CPU_{weight} is 3, $L2_{weight}$ is 2, VO_{weight} is 3 and $L3_{weight}$ is 7, then

- D_2 is ceil[(3 + 2) / 2] = 3,
- D_{VO} is ceil[(3 + 7) / 3] * 3 +1 = 13.

If CPU/SDRAM ratio is 5/4 (for example memory frequency is 80 MHz and CPU frequency is 100 MHz), refresh interval K_d is 1220 cycles, and R_x is 2, then the maximum latency for VO is:

- $L_{VO,sc}$ = 13 * 20 + 10 + ceil[13 * 20 / 1220] * 19 + ceil(16 * 2 / (5 / 4)] = 315 SDRAM cycles
- $L_{VO} = L_{VO,sc}$ * 12.5 = 3937.5 ns

Note: Average latency is normally much lower than worst case latency because on rare occasions many units will issue requests at exactly the same time (this is assumed when evaluating the maximum latency).

Note: All real-time units have a special exception notification flag that is raised if an overflow or underflow occurs while operating.

Note: To compute the latency L_x when a unit is not enabled, its weight has to be set to '0' in the $D_{\{2,3,4,5,6\}}$ equations and in $D_{\{AI,AO,VLD\}}$ for AI, AO or VLD.

These equations are not accurate for all the weights, but give an upper bound of the worst case (which is usually too pessimistic).

A much more accurate number could be found by simulating the arbiter, e.g. if the settings are: CPU_{weight}=1, $L2_{weight}$=2, VO_{weight}=1 and $L3_{weight}$=1, then

D_{VO} = ceil[(1 + 1) / 1] * ceil[(1 + 2) / 2]

giving 4 requests. But actually the worst case grant requests order is: CPU, L3, VO - resulting in 3 requests only.

20.5.2 Bandwidth Analysis

In the following, ceil(x) means the least integral value greater than or equal to x.

Minimum allocated bandwidth, B_x for a unit x, by the arbiter is defined as follows:

$B_x = (M_{cycles} - K_k) * S / [T * E_x + (16 * R_x / C)]$

Where:

M_{cycles} is the total amount of SDRAM cycles available in a period P in which the bandwidth is computed. For example, if the period is 1 second and SDRAM runs at 80 MHz then M_{cycles} is 80,000,000.

K_k is the amount of SDRAM cycles used by the refresh during the same period P.

If P is in seconds it could be expressed as:

K_k = ceil(4096 * P / .064) * K

For example, if P is 1 second then K_k is

ceil(4096 * 1 / .064) * 19 = 1216000 SDRAM cycles.

S is the size of the transaction on the bus.

For TM1300, S is equal to 64 (bytes).

E_x is the ratio of requests available for a unit x according to the arbiter settings.

It means the unit x will get 1 / E_x out of the total requests. E_x is derived from the arbiter settings as follows:

$$E_{CPU} = \frac{CPU_{weight} + L2_{weight}}{CPU_{weight}}$$

$$E_{VO} = \frac{VO_{weight} + L3_{weight}}{VO_{weight}} \times E_2$$

$$E_{ICP} = \frac{ICP_{weight} + L4_{weight}}{ICP_{weight}} \times E_3$$

$$E_{VI} = \frac{VI_{weight} + L5_{weight}}{VI_{weight}} \times E_4$$

$$E_{PCI} = \frac{PCI_{weight} + L6_{weight}}{PCI_{weight}} \times E_5$$

$$E_{VLD} = \frac{2 + 1 + 1 + 0 + 1 + 1}{2} \times E_6$$

$$E_{AI} = \frac{2 + 1 + 1 + 0 + 1 + 1}{1} \times E_6$$

$$E_{AO} = \frac{2 + 1 + 1 + 0 + 1 + 1}{1} \times E_6$$

$$E_{DVDD} = \frac{2 + 1 + 1 + 0 + 1 + 1}{1} \times E_6$$

$$E_{SPDO} = \frac{2 + 1 + 1 + 0 + 1 + 1}{1} \times E_6$$

Where:

$$E_2 = \frac{CPU_{weight} + L2_{weight}}{L2_{weight}}$$

$$E_3 = \frac{VO_{weight} + L3_{weight}}{L3_{weight}} \times E_2$$

$$E_4 = \frac{ICP_{weight} + L4_{weight}}{L4_{weight}} \times E_3$$

$$E_5 = \frac{VI_{weight} + \mathrm{L5}_{weight}}{\mathrm{L5}_{weight}} \times E_4$$

$$E_6 = \frac{PCI_{weight} + \mathrm{L6}_{weight}}{\mathrm{L6}_{weight}} \times E_5$$

For example, with the same settings as in the example of Section 20.5.1, then

- E_2 is (3 + 2) / 2 = 2.5
- E_{VO} is (3 + 7) / 3 * 2.5 = 8.33,

which gives

- B_{VO} = (80 - 1.216) * 64 / [20*8.33 + 16*2 / (5/4)]

resulting in 26.23 million B/sec corresponding to 25.01 MB/sec.

Note: In order to compute the latency B_x when a unit is not enabled, its weight has to be considered as '0' in the $E_{\{2,3,4,5,6\}}$ equations and in $E_{\{AI,AO,VLD\}}$ for AI, AO or VLD.

The maximum amount of requests, A_x, for unit x allowed during M_{cycles} period is:

A_x = floor(B_x / S)

Where floor(X) is the greatest integral value less than or equal to X.

Note: This number does not take into account the worst case pattern for request acknowledgment. Thus if the period is too small A_x is not accurate.

20.6 EXTENDED BEHAVIOR ANALYSIS

The following sections describes a more accurate behavior of the TM1300 arbitration system.

20.6.1 Extended Bandwidth Analysis

The **minimum** bandwidth allocation derived from the arbiter settings is accurate if one of the two following conditions are true:

- The units emit requests all the time (i.e. do back-to-back requests)
- After a request has been acknowledged, the unit emits a new request before the new arbitration point. The arbitration is decided around every 16 cycles. This time depends on the direction of the transactions (read/write).

In TM1300, the only unit almost able to sustain back-to-back requests is the data cache. The other units will post a request and wait for the data before the next request is posted. This behavior makes the bandwidth computation:

- almost accurate if the unit is down in the arbiter hierarchy (true if the units placed above are enabled).
- rather inaccurate if large weights are used for a unit.

Since no back-to-back requests are implemented, the worst case is that a unit can only get one request out of three if all the others are asking. This limits the use of large weights for other units than data cache.

However some units might be able to catch one request out of two. This depends on the way requests interleave, since the arbitration point is dependent on the type of the request (read or write) as well as on the CPU ratio.

This makes it almost impossible to describe the behavior precisely.

The **exact** bandwidth necessary for units like VO, VI, AO or AI are well known (see dedicated sections in each corresponding chapter). If the arbiter settings allocate more bandwidth for these units than they can use, the extra bandwidth can be used by units that are located below these units (VO, VI) or at the same level as (AO and AI) in the arbiter hierarchy.

As an example, with the default settings, VO gets 25% of the available bandwidth and the CPU gets 50%. If the SDRAM clock speed is 100 MHz, then 100 MB/sec are allocated to VO. If VO runs at 27 MHz (NTSC or PAL mode), then VO will not use all this allocated bandwidth. Thus any of the units that are below VO in the arbiter hierarchy can potentially use the remaining allocated bandwidth.

In other words - even if only 10% are allocated to one unit like the CPU, PCI or the ICP, it may use more.

20.6.2 Extended Latency Analysis

Some units (VO and VI) have a latency/bandwidth requirement and their behavior needs to be simulated in order to find out the correct settings. For example the requirement for VO (in image mode 4:2:2 or 4:2:0 without up scaling, overlay disabled) is:

- During 128 VO clock cycles, VO block needs to have 2 requests acked ([2 Ys, one U and one V]/2).

The default value '0' for ARB_BW_CTL leads to a bus allocation of 50% for CPU, 25% for VO and 25% for L3 blocks.

The worst case arbitration for VO is then: CPU L3 CPU VO, CPU L3 CPU VO to which the refresh (K), internal delays (T) and E for the first CPU request need to be added.

The first VO request will require 129 SDRAM cycles (D_{VO} = 5 or from the worst case pattern 19 + 10 + 20 + 4 * 20).

The arbitration pattern shows that the following request will require (in the worst case) an extra 4 * 20 SDRAM cycles. Thus VO clock speed cannot be greater than 61.24% (128 / [129 + 80]) of the SDRAM clock speed.

By changing the settings to 33% for the CPU, 33% for VO and 33% for L3 blocks (i.e. CPU_{weight} = '1', $L2_{weight}$ = '2', VO_{weight} = '1', $L3_{weight}$ = 1), the new SDRAM/VO clock percentage becomes 75.74% (128 / [109 + 60]) corresponding to a worst case arbitration pattern of CPU L3 VO, CPU L3 VO.

Before changing the settings the minimum SDRAM speed required to run VO at 74.25 MHz (high definition

speed) was 122 MHz. After the new allocation 100 MHz is fine. Note that here D_{VO} remains equal to '5'.

When VO is running in image mode 4:2:2 or 4:2:0 without upscaling and overlay enabled, the requirements become:

- During the first 64 VO clock cycles at least one request must be acked (the OL (overlay) data).
- During 128 VO clock cycles, VO block requires that 4 requests be acked ([4 OLs, two Ys one V and one U]/2).

If the settings are 33% for the CPU, 33% for VO and 33% for L3 blocks then the worst case arbitration pattern is CPU L3 VO, CPU L3 VO, etc.

The first requirement limits the VO/SDRAM ratio to

(64 / [19 + 10 + 20 + 3 * 20]) = 58.7%.

The second requirement gives a VO/SDRAM ratio of 44.29% (128 / [19 + 10 + 20 + 3 * 20 + 3 * 20 * 3]).

Thus if VO clock speed is supposed to be 54 MHz (progressive scan) the SDRAM must run at least at 122 MHz.

By setting the arbiter to 25% for the CPU, 37.5% for VO and 37.5% for VI (CPU_{weight} = 1, $L2_{weight}$ = 3, VO_{weight} = 1, $L3_{weight}$ = 1, assuming only VO and VI are enabled) the arbitration pattern becomes CPU VI VO VI CPU VO VI VO CPU VI VO.

Now both VI and VO are able to catch one request out of two, thanks to the read / write overlap. This leads to a VO/SDRAM ratio of 47.5% or a 113 MHz SDRAM.

20.6.3 Raising Priority

If VO is running at 27 MHz (NTSC or PAL) without overlay and CPU_{weight} is set to '3' while all the other weights are set to '1', then the worst case latency derived from 20.5.1 for VO is:

$L_{VO,sc}$ = (ceil[(1 + 1) / 1] + ceil[(3 + 1) / 1] + 1) * 20 + 10 + 19 = 169 SDRAM cycles (assumes R_{VO} = '0').

The latency for VO is 1 request in 64 VO clock cycles. If SDRAM is running at 80 MHz, then the maximum latency tolerated by VO is floor(64 / (27 / 80)) = 189 SDRAM cycles.

This means that VO requests can remain at low priority for 189 - 169 = 20 SDRAM cycles.

If the CPU clock speed is 100 MHz (ratio is 5 / 4) then the ARB_RAISE register can be programmed to:

floor(20 * (5 / 4) / 16) = 1.

VO requests will stay at low priority for 16 cycles allowing slightly better average CPU performance.

20.6.4 Conclusion

There is no obvious way to set the best weights for latency or bandwidth allocation since the behavior of each block cannot be easily described with equations. Practical results obtained by running applications showed that once the arbiter is weighted to meet latencies the remaining weight settings do not allow much improvement.

The best way to tune the weights is by experiment, running the application.

The only accurate computation is the maximum worst case latency, which ensures that the hard real-time units work properly. This computation gives an upper bound and can be too pessimistic - but it still gives the right order of magnitude. Refer to Table 20-5 for the recommended allocation method.

Table 20-5. Recommended Allocation Method

Video In	allocate required latency
Video Out	allocate required latency
Audio In	allocate required latency
Audio Out	allocate required latency
SPDIF Out	allocate required latency
ICP	allocate bandwidth
PCI	allocate bandwidth
VLD	allocate bandwidth/latency
DVDD	allocate bandwidth/latency

Power Management Chapter 21

by Eino Jacobs and Hani Salloum

21.1 OVERVIEW

TM1300 supports power management in two ways:

- In global power-down mode, most clocks on the chip are shut down and the SDRAM main memory is brought into low-power self-refresh mode. The power of all on-chip peripheral blocks except for BTI (boot and I^2C blocks), D$,I$, PCI, timers and VIC blocks is shut off. Some peripherals can be selectively prevented from participating in the global power down.
- A block power down mechanism allows power down of select peripheral blocks

21.2 ENTERING AND EXITING GLOBAL POWER DOWN MODE

Power management is software controlled and is initiated by writing to the MMIO register POWER_DOWN. During execution of this MMIO operation, the system is powered down without completing the MMIO operation. When the system wakes up from power down mode, the MMIO operation is completed.

This means that during program execution on the DSPCPU the moment of power down is defined exactly: any instruction before the instruction that contains the MMIO operation is completed before entering power down mode. The instruction containing the MMIO operation and all subsequent instructions are completed after wake up from power down mode.

Wake-up from power down mode is effected by receiving an interrupt (any interrupt) that passes the acceptance criteria of the interrupt controller.

There is also wake-up from power down if a peripheral unit asserts a memory request signal on the highway.

During power down mode the whole chip is powered down, except the PLLs, the interrupt logic, the timers, the wake-up logic in the MMI, and any logic in the peripheral units and PCI bus interface that is not participating in the power down.

Note: Writing to the global POWER_DOWN register (at offset 0x100108) has no effect on the contents of the BLOCK_POWER_DOWN register (at offset 0x103428), and vice versa.

21.3 EFFECT OF GLOBAL POWER DOWN ON PERIPHERALS

The on-chip peripheral units participate in global power down. This can be a programmable option for selected peripherals. These selected peripherals have a programmable MMIO control bit, the SLEEPLESS bit, that can be used to prevent it from participating in the global power down mode. By default every peripheral unit must participate in power down.

The following peripheral units have the SLEEPLESS bit: Video In, Video Out, Audio In, Audio Out, SPDO, SSI, and JTAG.

The following peripherals do not have the SLEEPLESS bit and always participate in power down: VLD, boot/I^2C and ICP.

The following peripherals do not participate in global power down, although they must power themselves down when they are inactive: VIC, PCI.

When a peripheral does not participate in global power down, it can still do regular main memory traffic. Every time a peripheral unit asserts the highway request signal, the MMI will initiate a wake-up sequence. The CPU must execute software that initiates a new power down of the system. This software can be the wait-loop of the RTOS.

Programmer's note: Since the system is awakened each time there is a transaction on the highway, it may be interesting to make a software loop that does the activation of the POWER_DOWN mode. Then the activation is conditional and most of the time done using a global variable, usually set by a handler. It then becomes mandatory to be sure that there are no interruptible jumps between the time the value of the global variable is fetched and compared by the DSPCU and the time the conditional write to the MMIO is performed (it is the classical semaphore or test and set issue). Thus it is recommended that a separate function be used with the address of the variable as a parameter. This function needs then to be compiled specifically without interruptible jumps.

The wake-up from power down mode takes approximately 20 SDRAM clock cycles. This amount of time is added to the worst case latency for memory requests compared to the situation when the system is not in power down mode.

21.4 DETAILED SEQUENCE OF EVENTS FOR GLOBAL POWER DOWN

The sequence of events to power down TM1300 is as follows:

- Issue a MMIO write to the POWER_DOWN register
- The main memory interface (MMI) waits till the completion of the current SDRAM transfer, if there is one still busy.
- The MMI brings SDRAM into the self refresh state, goes into a wait state, and asserts the global signal global_power_down.
- All units that participate in the power down, respond to the global_power_down signal by disabling their clocks.
- Only the PLL, interrupt controller, timers, wake-up logic, the PCI bus interface, and any peripherals that have their SLEEPLESS bit control bit set continue to be clocked. The SDRAM clock continues.
- An interrupt is detected by the interrupt controller or a unit that didn't participate in the power down requests a memory transfer.
- The MMI de-asserts the global_power_down signal, activating all blocks on the chip.
- The MMI recovers SDRAM from self-refresh.
- The MMI causes completion of the MMIO operation that initiated the power down sequence.
- When software takes an interruptible branch operation, the interrupt that caused the wake-up will be serviced (if the wake-up was initiated by an interrupt).

21.5 MMIO REGISTER POWER_DOWN

The register POWER_DOWN has an offset 0x100108 in the MMIO aperture and has no content. Writing to this register has the side-effect of powering down the chip. Reading from this register returns an undefined value and has no side-effect.

21.6 BLOCK POWER DOWN

This feature is new in TM1300. It selectively shuts off a particular block or a set of blocks based on software programming.

This type of power down can be used in applications where certain blocks will never participate in the operation of the chip. The objective of having this type of power down is saving on power consumption.

Each peripheral unit which can participate in the global power down can be selectively powered down.

This is done by setting a control bit in MMIO register BLOCK_POWER_DOWN specifically for the block. The BLOCK_POWER_DOWN register is located at MMIO offset 0x103428. See Figure 21-1 below.

Setting a particular bit to '1' in this register has the effect of shutting off the corresponding block. Writing '0' to this bit, enables the power for the block again.

A block should not be powered down if it is active. Enable bit should be set to '0' before deciding to power down the block.

Note: The unassigned bits of this register have to be written to '0' and read as '0'.

Note: Writing to the global POWER_DOWN register (at offset 0x100108) has no effect on the contents of the BLOCK_POWER_DOWN register (at offset 0x103428), and vice versa.

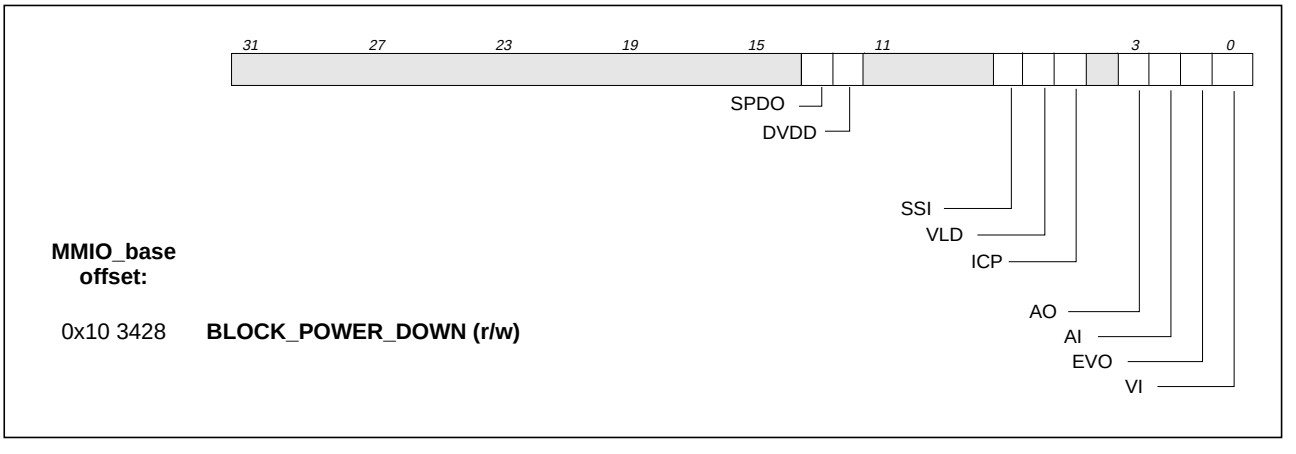


Figure 21-1. Power down register BLOCK_POWER_DOWN

PCI-XIO External I/O Bus Chapter 22

By David Wyland

22.1 SUMMARY FUNCTIONALITY

The TM1300 PCI-XIO bus allows glueless connection to PCI peripherals, 8-bit microprocessor peripherals and 8-bit memory devices. All these device types can be intermixed in a single TM1300 system.

The PCI-XIO bus provides the following features:

- All PCI 2.1 features (32-bit, 33 MHz)
- Simple, non-multiplexed, 8-bit data, 24-bit address XIO bus with control signals for 68K and x86 style devices
- Glueless connection to ROM, EPROM, flash EEPROM, UARTs, SRAM, etc.
- Programmable internal or external bus clock source
- 0-7 programmable wait states for XIO devices
- Support for single byte read, single byte write, DMA read or DMA write
- The 16 MB of XIO device space is visible as 16 MWords (64 MBytes) in the DSPCPU memory map

22.1.1 Description

The XIO logic that implements the protocol for 8-bit devices appears as a on-chip PCI target device to the rest of the TM1300. It only responds when it is addressed by the TM1300 as initiator and never responds to external PCI masters. When it is addressed by the TM1300 as an initiator, it responds to the TM1300 PCI BIU as a normal slave device, activating PCI_DEVSEL#.

The XIO logic serves as a bridge between the PCI bus and XIO devices such as ROMs, flash EPROMs and I/O device chips. The TM1300 addresses XIO devices on the PCI-XIO bus in the same way as registers or memory in any other PCI slave device. The XIO logic supplies the PCI_TRDY# signals to the PCI bus and also supplies the chip-select, read, write and data-strobe signals to XIO devices attached to the PCI-XIO Bus. A *conceptual only* block diagram of the PCI-XIO Bus is shown in Figure 22-2. The real hardware uses the PCI_AD[0:30] signals and PCI_C/BE#[0:3] signals for both PCI and XIO devices, as shown in Figure 22-3.

The XIO logic is activated when the Enable bit in the XIO_CTL register is asserted and whenever the TM1300 (as initiator) addresses the PCI-XIO bus address range, as defined by a 6-bit address field in the XIO Bus Control Register. This 6-bit field defines the 6 most significant bits of the XIO Bus address space. When the TM1300 sends out an address as an initiator, the upper 6 bits of the address are compared with this field. If they match, the PCI-XIO bus logic is activated. The PCI_INTB# output is asserted to indicate that the PCI-XIO Bus is active. It becomes active at PCI data phase time. When XIO is enabled, the PCI_INTB# signal becomes dedicated as XIO bus chip-select, and turns from an open-drain output into a normal logic output. PCI_INTB# serves as a global chip select for all XIO Bus chips. When XIO is disabled, PCI_INTB# is available for PCI-specific use or as a general purpose software I/O pin with open-drain behavior as in TM1000.

The Address field bits in the XIO Bus Control register serve as a base address register in PCI terms. The XIO Bus Control register is not a PCI configuration register. It does not need to be a PCI configuration register because the PCI-XIO Bus can only be addressed by the TM1300. It will not respond to requests by any other external PCI device.

When the XIO-PCI Bus controller logic is activated, it generates PCI_DEVSEL# as a response to the PCI bus. When PCI_IRDY# has been received from the BIU, it asserts an external PCI_INTB# signal as the global chip select. It also reconfigures the PCI address/data pins for 8-bit byte transfers. When the PCI-XIO Bus is active, the lower 24 bits of the external 32-bit PCI bus are used to output a 24-bit address for all transfers, read or write. The upper 8 bits of the external PCI bus are unchanged and transfer data normally. This is shown in Figure 22-3.

The 24-bit address on the XIO Bus pins is the word address for the PCI transfer, which is the lower 26 bits of the PCI transfer address with the two least significant bits ignored. One word is transferred to or from the PCI bus for each byte read or written on the XIO bus. In writes to the XIO bus, a 32-bit word is transferred from the PCI BIU to the XIO Bus controller, but the lower 24 bits and the PCI byte enables are ignored. In reads from the PCI bus, a 32-bit word is transferred from the XIO Bus controller to the PCI BIU with the data in the upper 8 bits and the 24-bit address in the lower 24 bits. Note that the 24-bit address returned in a read is the lower 26 bits of the PCI transfer address with the two least significant bits truncated. For example, a PCI transfer address of 44 hexadecimal would return a value of 11 hexadecimal as the lower 24 bits of the 32-bit data in a read. The 24-bit XIO Bus address is generated by an address counter in the XIO Bus controller. This counter is loaded with the PCI word address at PCI frame time at the start of the PCI transfer and is incremented for each PCI word transferred.

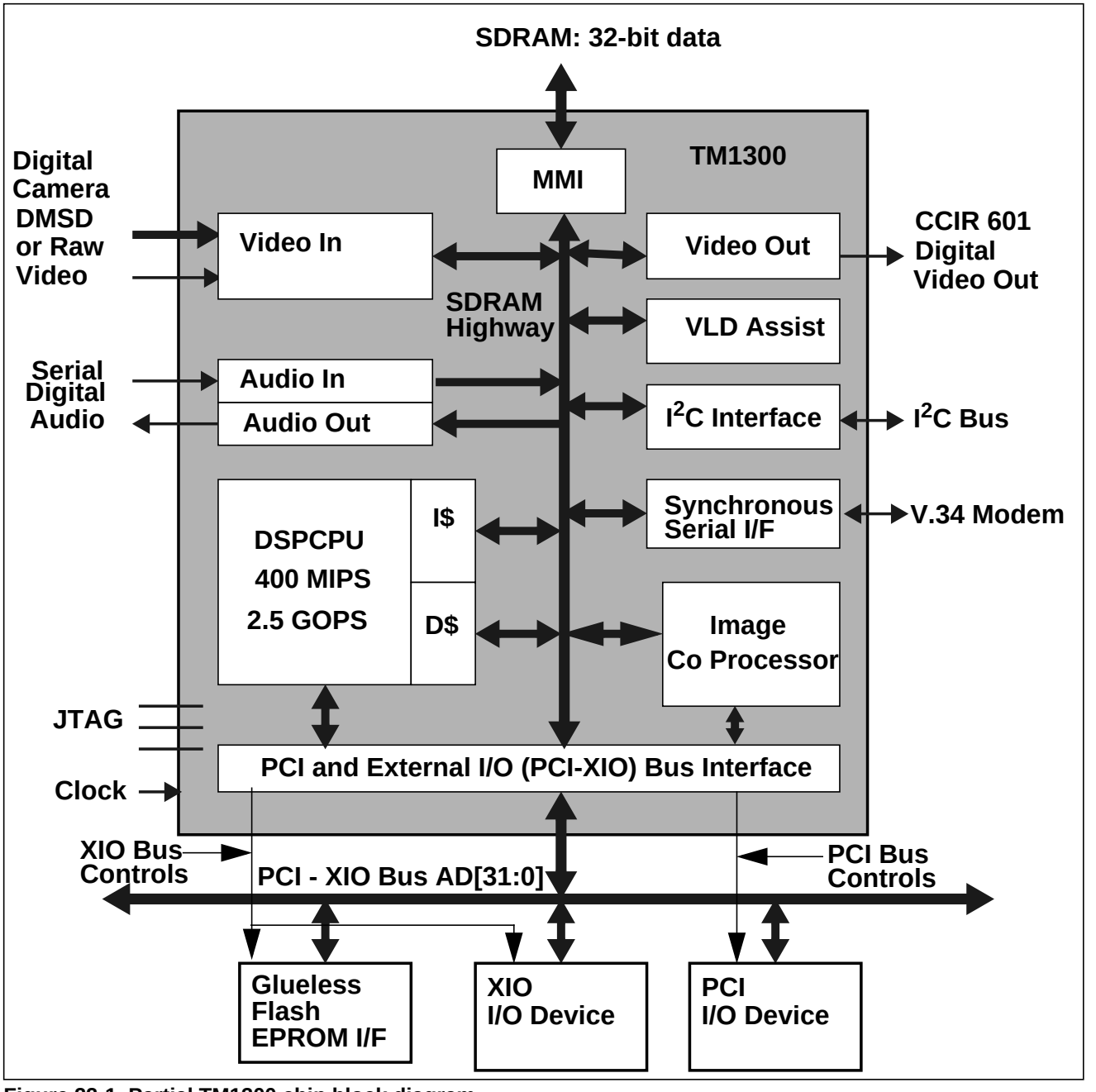


Figure 22-1. Partial TM1300 chip block diagram

The XIO Bus does not generate parity during XIO Bus write transfers or check parity during XIO Bus read transfers. This allows the XIO Bus to interface to standard 8-bit devices without having to add parity-generation and check logic. While the XIO Bus is active, the XIO Bus logic inhibits parity checking and drives the PCI Parity and Parity Error pins so that they do not float.

Word transfer is used to transfer the bytes to and from the PCI bus for hardware simplicity. The primary intended use of the PCI-XIO Bus is for slow devices, ROMs, flash EPROMs and I/O. Because the PCI-XIO bus is so much slower than the TM1300, there is time available for the TM1300 to pack and unpack the words. In the case of ROMs and flash EPROMs, the data is typically compressed, requiring the TM1300 CPU to both unpack and decompress the data.

The PCI-XIO Bus Controller logic reconfigures the byte enables as control signals for the attached XIO Bus chips during XIO Bus transfers. It also drives the PCI_TRDY# signal to the PCI Bus for each transfer. The PCI Bus byte enables are reconfigured to generate XIO Bus timing signals: Read (IORD), Write (IOWR) and Data Strobe (DS). These signals allow ROM, flash EPROM, 68K and x86 devices to be gluelessly interfaced to the XIO Bus. For a single device, the PCI_INTB# line is used as the global chip enable. If more than one device is to be added, an external decoder, such as a 74FCT138, can be used to decode the upper bits of the 24-bit transfer address, with

the PCI_INTB# line used as a global chip enable to the decoder.

The PCI-XIO Bus controller has a wait state generator to provide timing for slow devices. The wait state generator allows the addition of up to 7 wait states for slow chip access and write times. The wait state generator logic generates the PCI_TRDY# signal to the PCI bus.

The XIO Bus controller contains a clock generator for standalone systems. The PCI-XIO Bus uses the PCI clock. This clock is normally supplied by a PCI Bus central resource outside the TM1300 chip. In standalone or low-cost systems, the internal clock generator can be used. The internal clock generator divides the TM1300 highway clock by a 5-bit number in a prescaler. This allows setting bus clocks from 4 MHz to 66 MHz in a 133 MHz system. The internal clock generator programming is described in Section 22.5, “XIO_CTL MMIO Register.”

22.2 BLOCK DIAGRAM

Figure 22-2 shows a *conceptual* block diagram of the PCI-XIO Bus as a slave device on the PCI Bus. The XIO Bus Controller generates an XIO Bus, which is an 8-bit bus with a 24-bit address. Devices attached to the XIO Bus appear as memory locations in the 16 MB address space of the XIO Bus.

Figure 22-3 shows an implementation block diagram of the PCI_XIO Bus. To conserve pins, the XIO Bus Controller uses the PCI I/O pins as XIO Bus pins during XIO Bus data transfers. It reconfigures the 32 PCI address/data pins as 8 XIO Bus data pins and 24 XIO Bus address pins, and it reconfigures the byte enable pins as XIO Bus timing signals. By changing the functions of the pins during the transfer, 36 pins are saved which would otherwise be required to drive the XIO Bus devices. By reconfiguring the PCI pins only during the data phase of the XIO Bus transfers, the PCI-XIO bus retains its PCI Bus compatibility.

Figure 22-4 shows a more detailed block diagram of the PCI-XIO Bus controller.

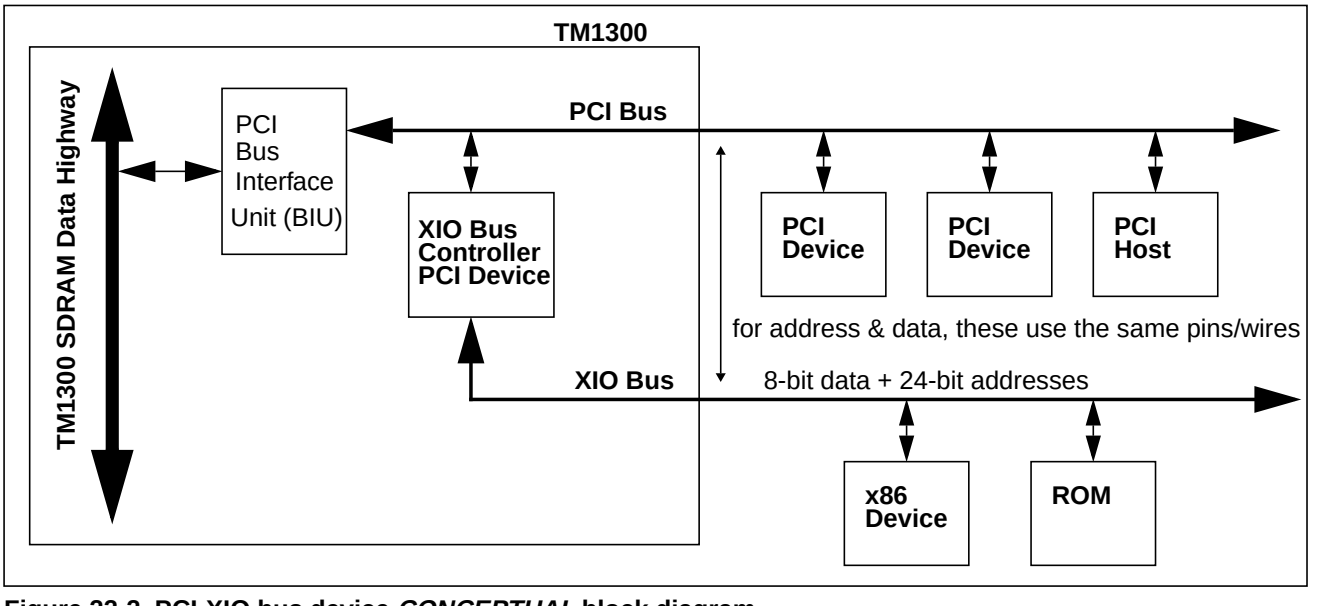


Figure 22-2. PCI-XIO bus device *CONCEPTUAL* block diagram

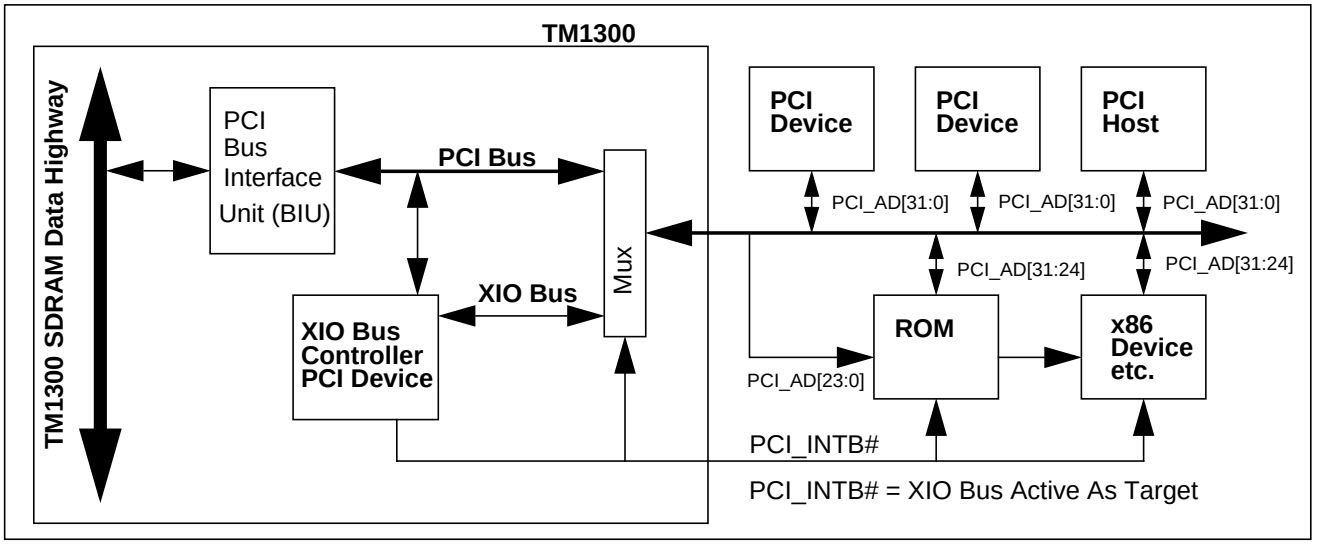


Figure 22-3. PCI-XIO Bus device implementation block diagram

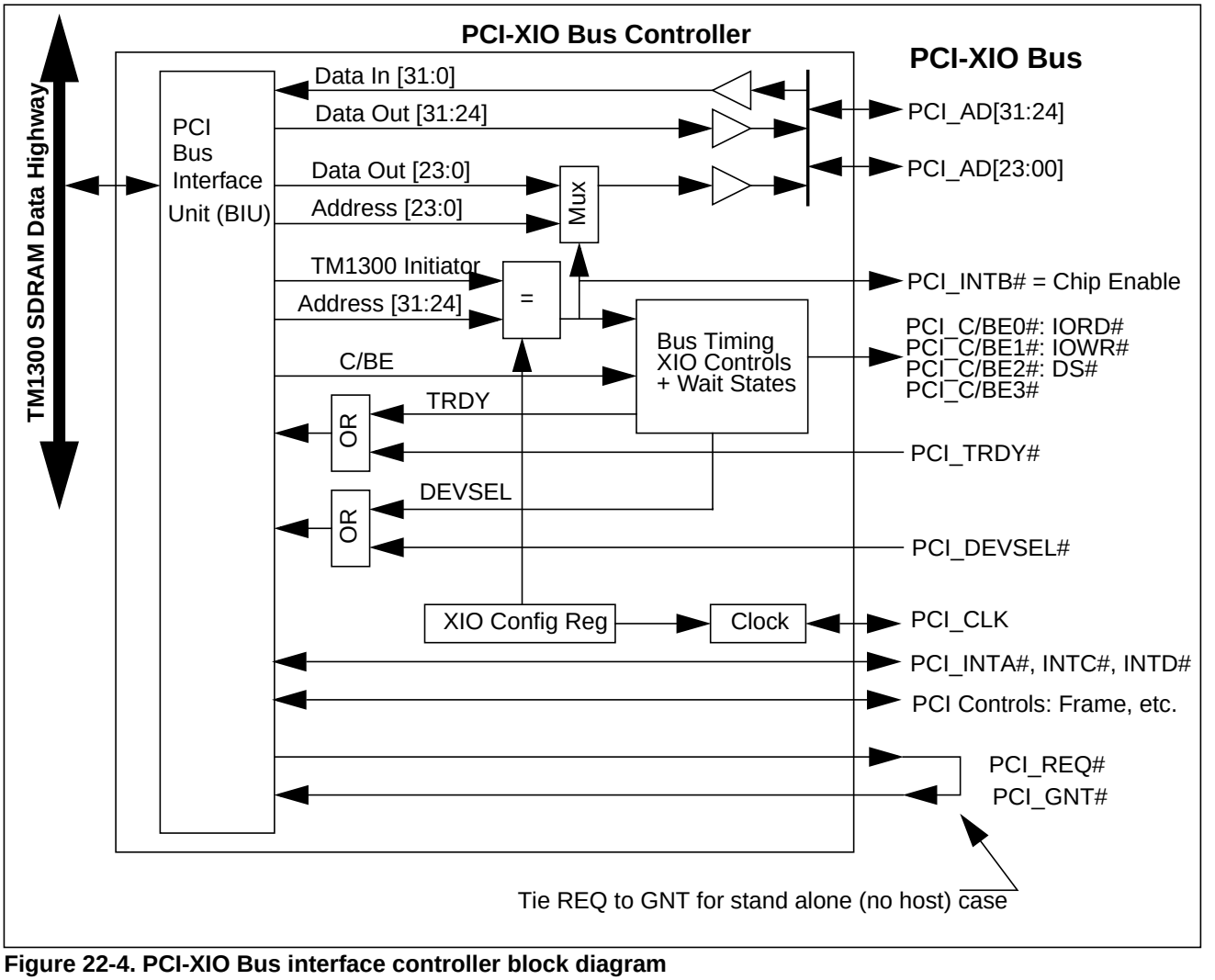


Figure 22-4. PCI-XIO Bus interface controller block diagram

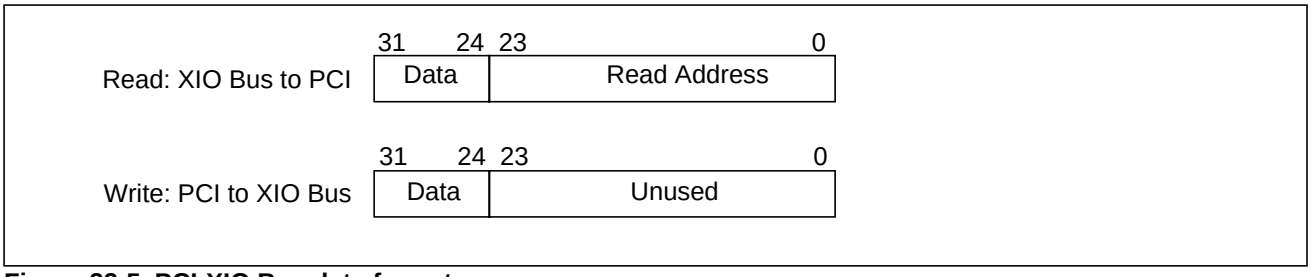


Figure 22-5. PCI-XIO Bus data formats

22.3 DATA FORMATS

The data transfer formats for the PCI-XIO bus are shown in Figure 22-5. The 8-bit data field is the data transferred to or from the PCI-XIO Bus. The read address is the 24-bit address on the PCI-XIO Bus address lines when the read transfer takes place.

22.4 INTERFACE

Table 22-1. PCI-XIO Bus signal definitions

TM1300 PCI Signal	Pins	I/O	PCI Function	XIO Function
PCI_INTB#	1	O	PCI-XIO Bus Enable = XIO Bus Active As Target Device	
PCI_AD[23:0]	24	I/O	PCI Address/Data	Address bus: 16 MB
PCI_AD[31:24]	8	I/O		Data bus: 8 bits
PCI_PAR	1	O	Even Parity for AD & C/BE	
PCI_C/BE0#	1		Command/Byte Enables On XIO read, BE[3:0] = 0110b'4 On XIO write, BE[3:0] = 0111b'4	IORD# = Read Enable
PCI_C/BE1#	1			IOWR# = Write Enable
PCI_C/BE2#	1			DS# = Data Strobe
PCI_C/BE3#	1			unused
PCI_CLK	1	I/O	33 MHz PCI Clock: can optionally be generated by TM1300 on board osc	
PCI_FRAME#	1	I/O	PCI Address/Command Strobe + Transfer In Progress	
PCI_DEVSEL#	1	I/O	Device Select Valid	Asserted by TM1300 = XIO Active
PCI_IRDY#	1	I/O	Initiator Ready = Transfer In Progress	
PCI_TRDY#	1	I/O	Target Ready	Asserted by TM1300 = XIO Transfer Timing
PCI_STOP#	1	I/O	Target Requests Stop of Transaction	
PCI_IDSEL#	1	I	Chip Select for PCI Config Writes	
PCI_REQ#	1	O	TM1300 Requesting PCI Bus	
PCI_GNT#	1	I	TM1300 Is Granted PCI Bus	
PCI_PERR#	1	I	Parity Error to TM1300	
PCI_SERR	1	O	System Error from TM1300	
PCI_INTA#	1	I/O	General Purpose I/O	
PCI_INTB#	1	I/O	General Purpose I/O	XIO Bus Active = Global Chip Select
PCI_INTC#	1	I/O	General Purpose I/O	
PCI_INTD#	1	I/O	General Purpose I/O	

22.4.1 PCI-XIO Bus Interface Design

The PCI-XIO Bus can accommodate a variety of different devices and bus protocols. The following are examples of devices interfaced to the PCI-XIO Bus.

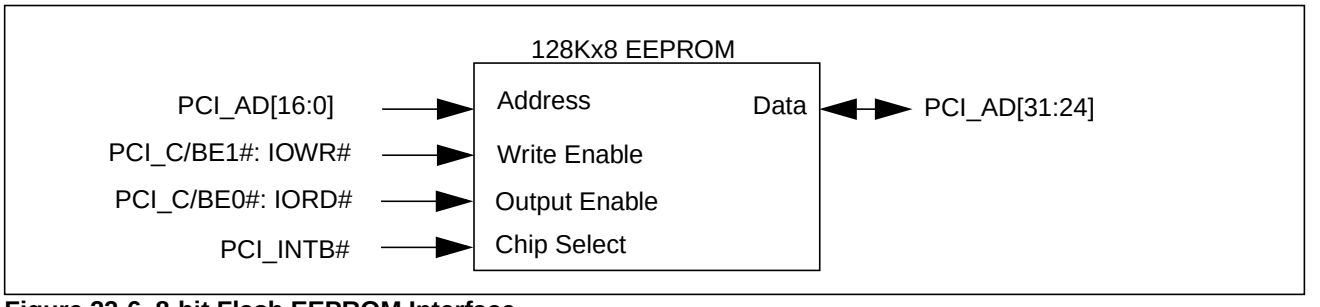


Figure 22-6. 8-bit Flash EEPROM Interface

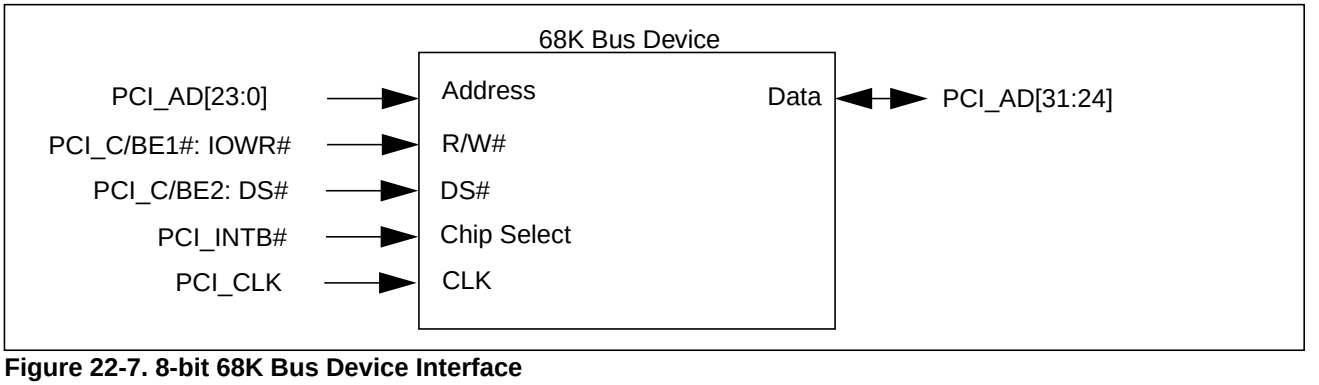


Figure 22-7. 8-bit 68K Bus Device Interface

22.4.1.1 Flash EEPROM

Figure 22-6 shows an 8-bit flash EEPROM interfaced to the PCI-XIO Bus. Examples of these devices are the Micron MT28F200C1 and the AMD 29LV400.

22.4.1.2 68K Bus I/O device

Figure 22-7 shows a 68K bus I/O device interfaced to the PCI-XIO Bus. Example devices are the Motorola MC68HC681 DUART and the MC68HC901 Multi-Function Peripheral.

22.4.1.3 x86/ISA Bus I/O device

Figure 22-8 shows an x86 or ISA bus I/O device interfaced to the PCI-XIO Bus. An example device is the Intel 82091 Advanced Integrated Peripheral (AIP).

22.4.1.4 Multiple Flash EEPROM

Figure 22-9 shows two 8-bit flash EEPROMs interfaced to the PCI-XIO Bus. A 74FCT138 logic chip decodes upper bits PCI_AD[19-17] of the XIO bus address to generate the chip selects for the two EEPROMs. These bits decode the address space into blocks of 128 KB. The address range of each enable is shown on the enable lines. Six spare chip selects are available for attaching up to six more EEPROMs or to attach other devices. The 74FCT138 provides both decode of the address bits and the AND function for the PCI_INTB# global chip enable

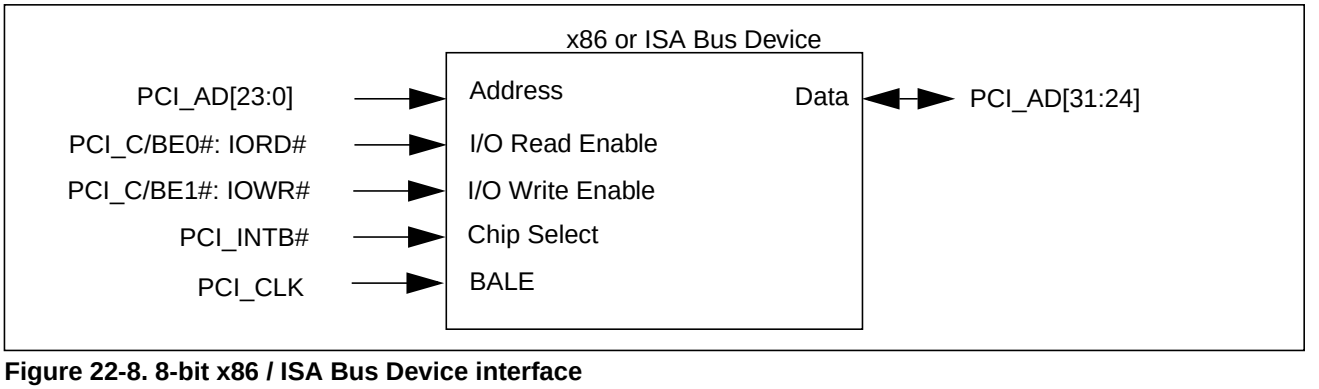


Figure 22-8. 8-bit x86 / ISA Bus Device interface

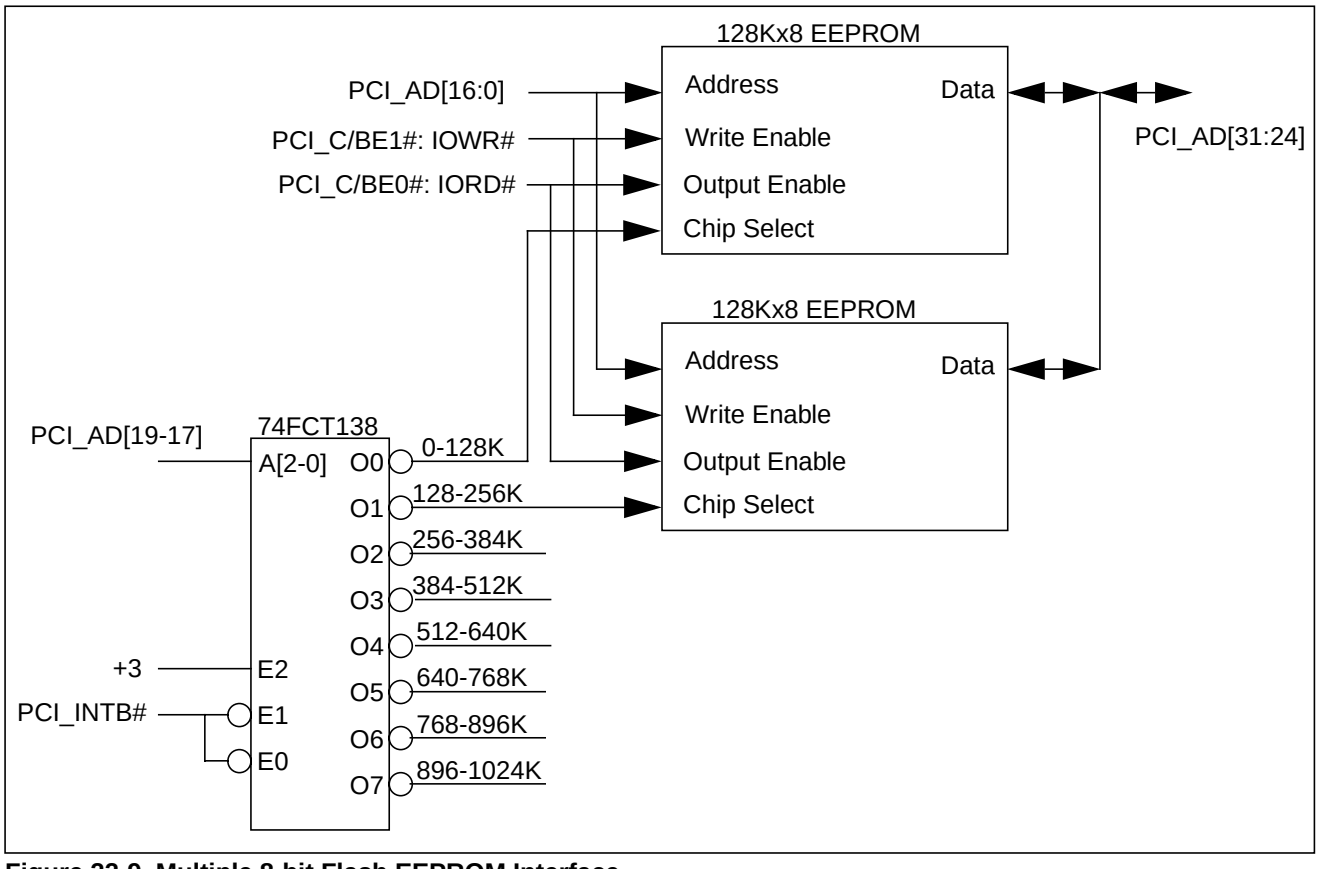


Figure 22-9. Multiple 8-bit Flash EEPROM Interface

signal so that only one EEPROM chip enable signal is active at global chip enable time.

22.5 XIO_CTL MMIO REGISTER

The PCI-XIO Bus Controller has one programmer visible MMIO register: XIO_CTL. Its format is shown in Table 22-2. To ensure compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as '0's.

Table 22-2. XIO_CTL Register Fields: MMIO Address 0x10 3060

Field	Bits	Function	Reset Value
Address	31:26	XIO address space	undefined
	25:11	unused	0
Wait States	10:8	Wait states	0
Enable	7	Enable XIO Bus operation	0 = disabled
	6:5	unused	
Clock Frequency	4:0	Clock divider	0x1f

22.5.1 PCI_CLK Bus Clock Frequency

PCI_CLK, the clock for the PCI and PCI-XIO bus can be supplied externally or internally. This is determined at boot time, by the 'enable internal PCI_CLK generator' bit, bit 6 of byte 9 in the boot EEPROM. Refer to Section 13.2. If this bit = '0', PCI_CLK acts compatible with TM1000 and normal PCI operation, i.e. PCI_CLK is an input pin that takes the PCI clock from the external world. If this bit = '1', an on-chip clock divider in the XIO logic becomes the source of PCI_CLK, and the PCI_CLK pin is configured as an output. In the latter case, the PCI_CLK frequency can be programmed to a divider of the TM1300 highway clock by setting the XIO_CTL register 'Clock Frequency' divider value.

Table 22-3. PCI_CLK frequencies for 133.0 MHz TM1300 highway clock

Clock Frequency (use odd values)	TM1300 Clocks	PCI-XIO Clock Period, ns	Frequency, MHz
0	illegal	illegal	illegal
1	2	15	66.5
2	3	22.5	44.33
3	4	30	33.25
...	...	...	...
30	31	233	4.29
31	32	241	4.16

A table of PCI-XIO Bus Clock frequencies versus Clock field values is shown in Table 22-3. Note that the PCI_CLK operating frequency should be set to observe the frequency limits given in the AC/DC timing characterization data for TM1300. Odd values of 'Clock Frequency' are recommended, resulting in an even divider, which generates a 50% duty cycle PCI_CLK.

22.5.2 Wait State Generator

The XIO Bus controller has an automatic wait state generator to allow for read and write cycle times of devices on the XIO bus.

Table 22-4. Wait state generator codes

Code	Wait States
0	0
1	1
2	2
...	...
7	7

22.6 PCI-XIO BUS TIMING

The timing for the PCI-XIO bus is shown below: Note that the 'fat' lines indicate active drive by TM1300. Thin lines indicate areas where the TM1300 is not actively driving. (In these areas, pull-up resistors retain the signal high for control signals, PCI_AD lines are left floating.) Figure 22-10 shows the timing for a single byte read transfer. Figure 22-11 shows the timing for a single byte read transfer with wait states. Figure 22-14 shows the timing for a DMA burst read transfer of 2 bytes, and Figure 22-16 shows the timing for a DMA burst write transfer of 2 bytes. The DMA burst transfers are shown at maximum rate, with zero wait states. DMA burst transfers with wait states insert wait states between the transfers. In the read case, the IORD# enable and DS# are extended by the wait states. In the write case, the IOWR# enable and DS# are delayed by the wait states.

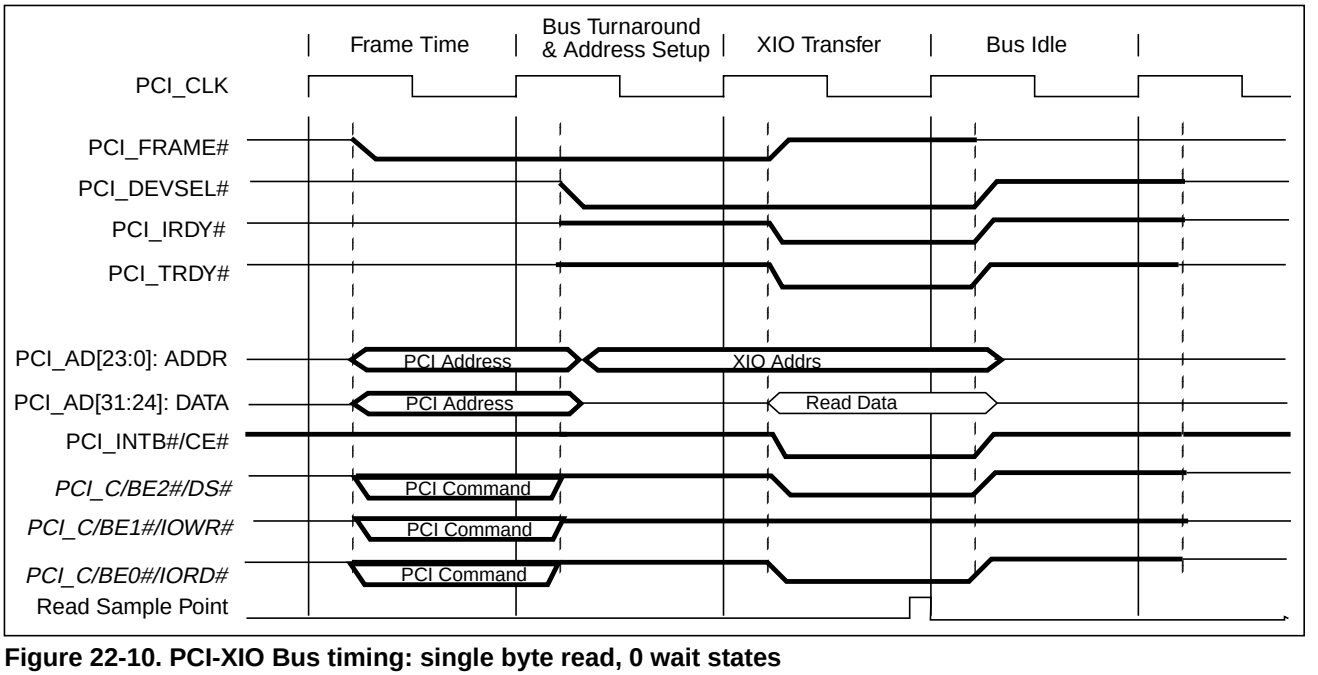


Figure 22-10. PCI-XIO Bus timing: single byte read, 0 wait states

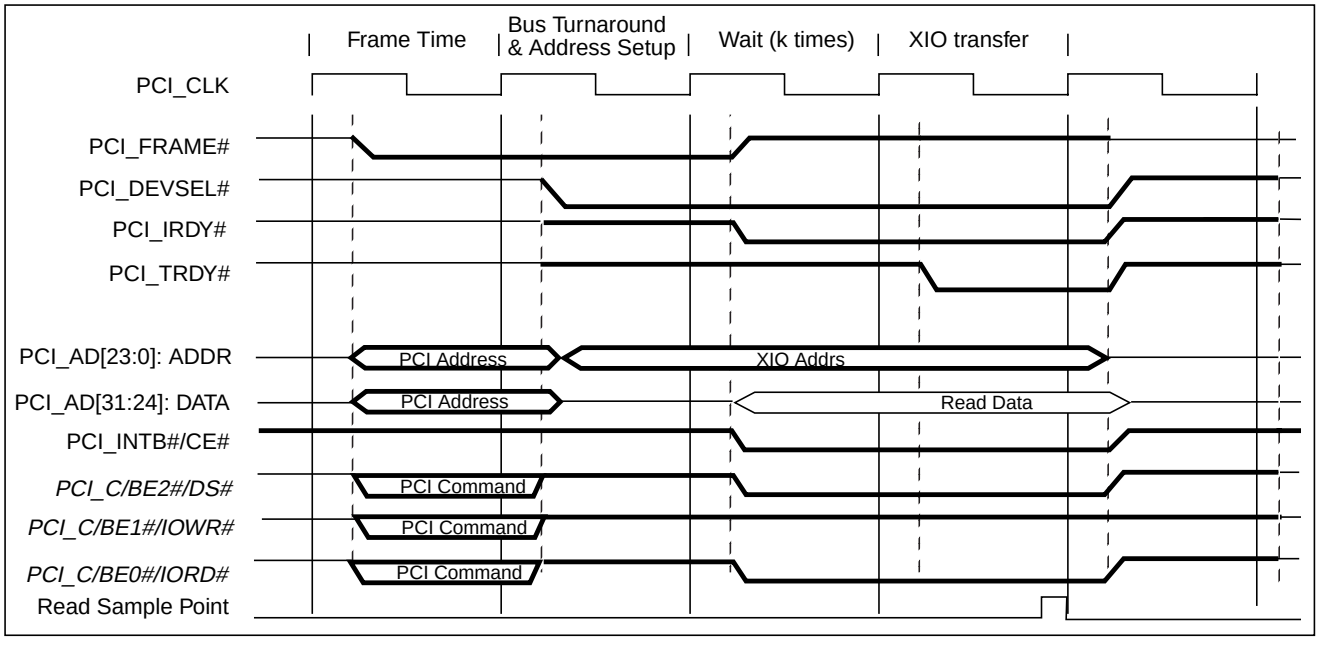


Figure 22-11. PCI-XIO Bus timing: single byte read, 1 or more wait states

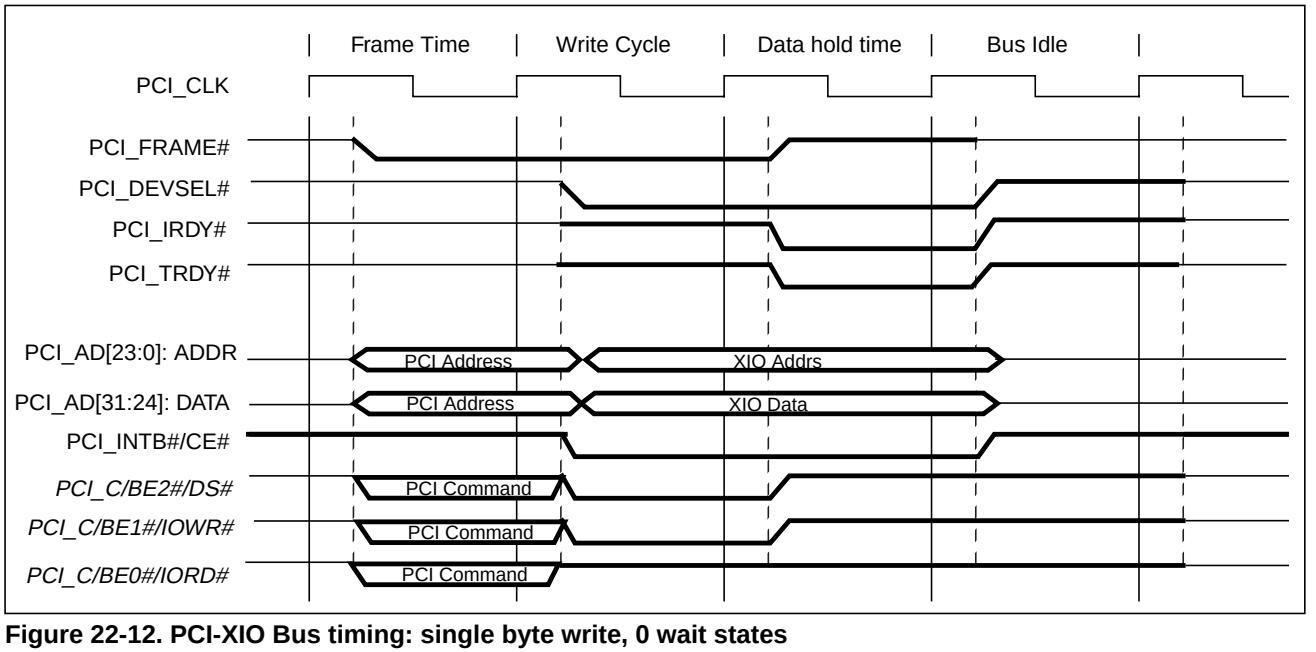


Figure 22-12. PCI-XIO Bus timing: single byte write, 0 wait states

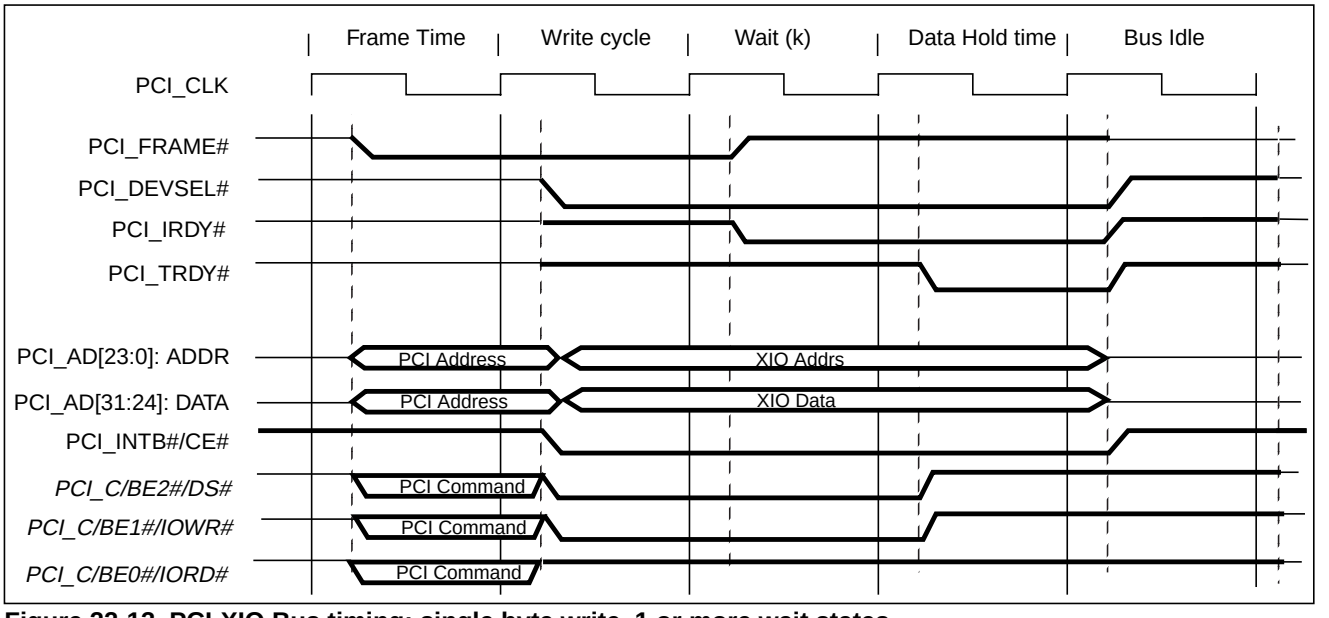


Figure 22-13. PCI-XIO Bus timing: single byte write, 1 or more wait states

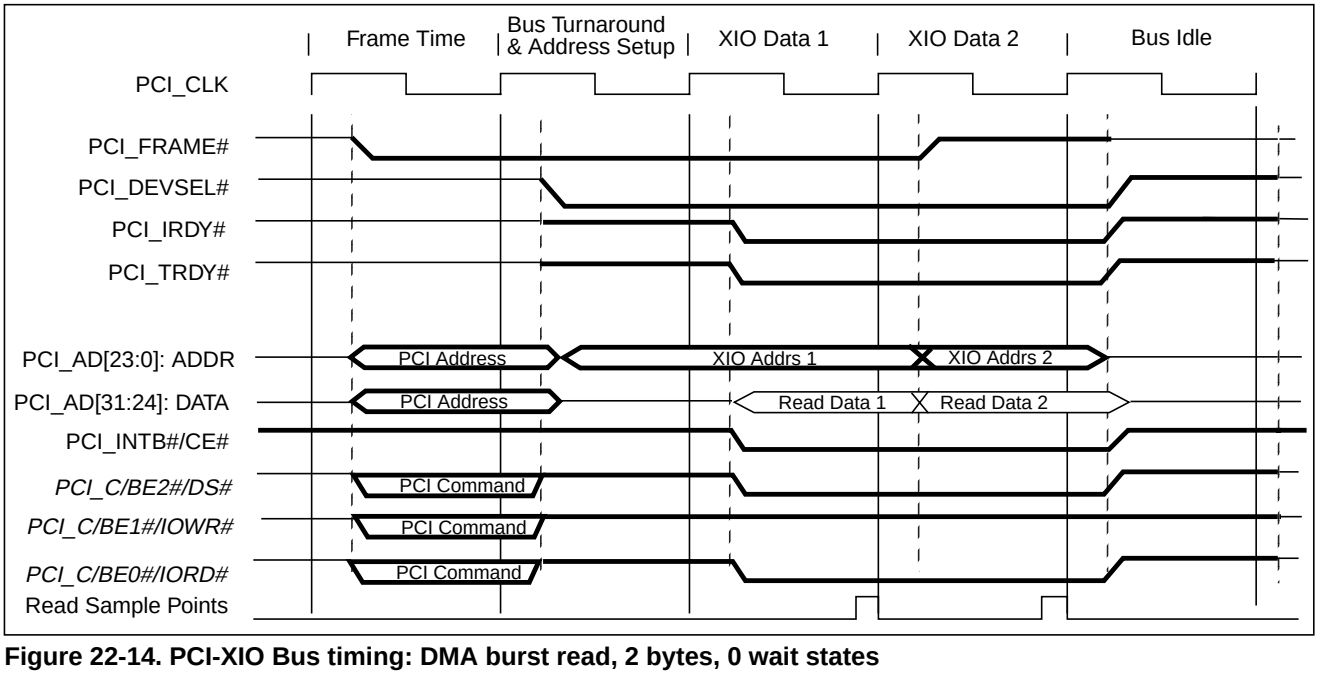


Figure 22-14. PCI-XIO Bus timing: DMA burst read, 2 bytes, 0 wait states

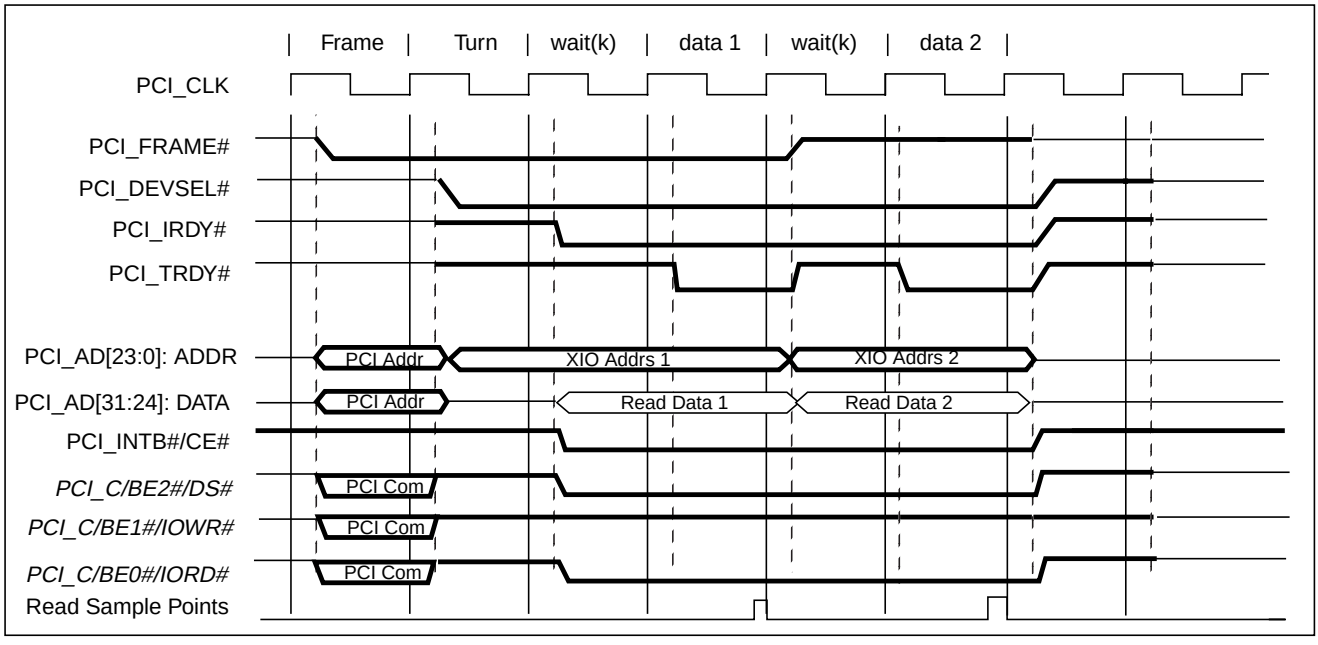


Figure 22-15. PCI-XIO Bus timing: DMA burst read, 2 bytes, 1 or more wait states

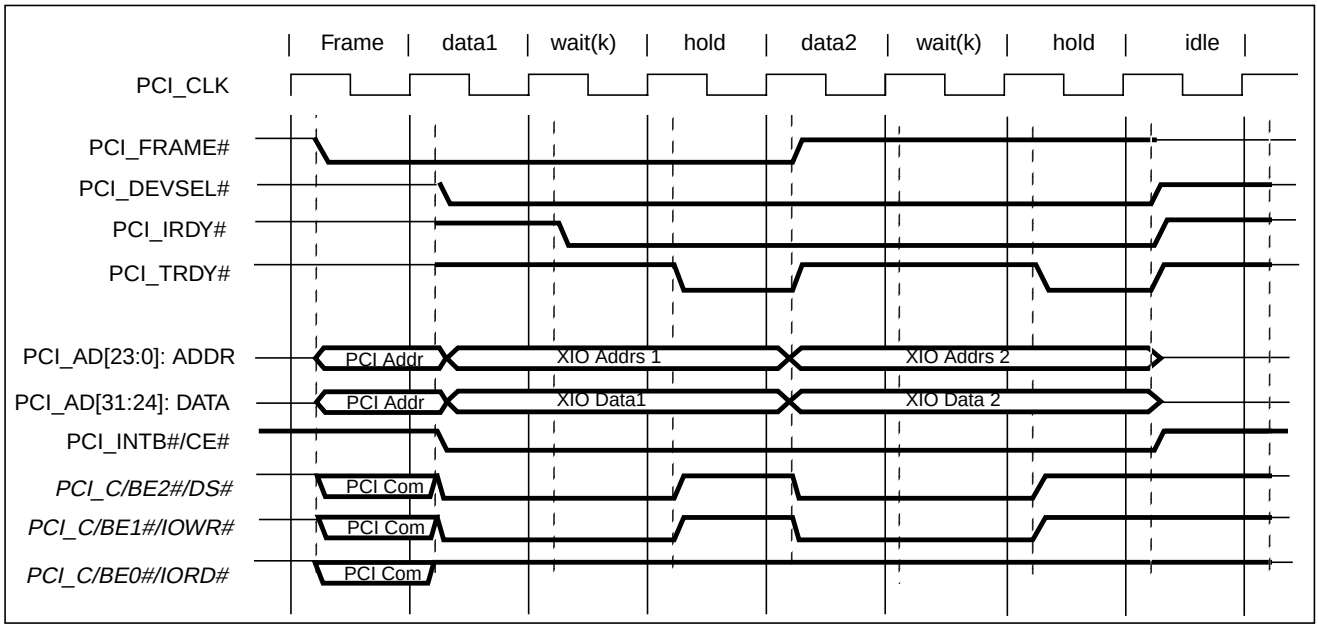


Figure 22-16. PCI-XIO Bus timing: DMA burst write, 2 bytes, 1 or more wait states

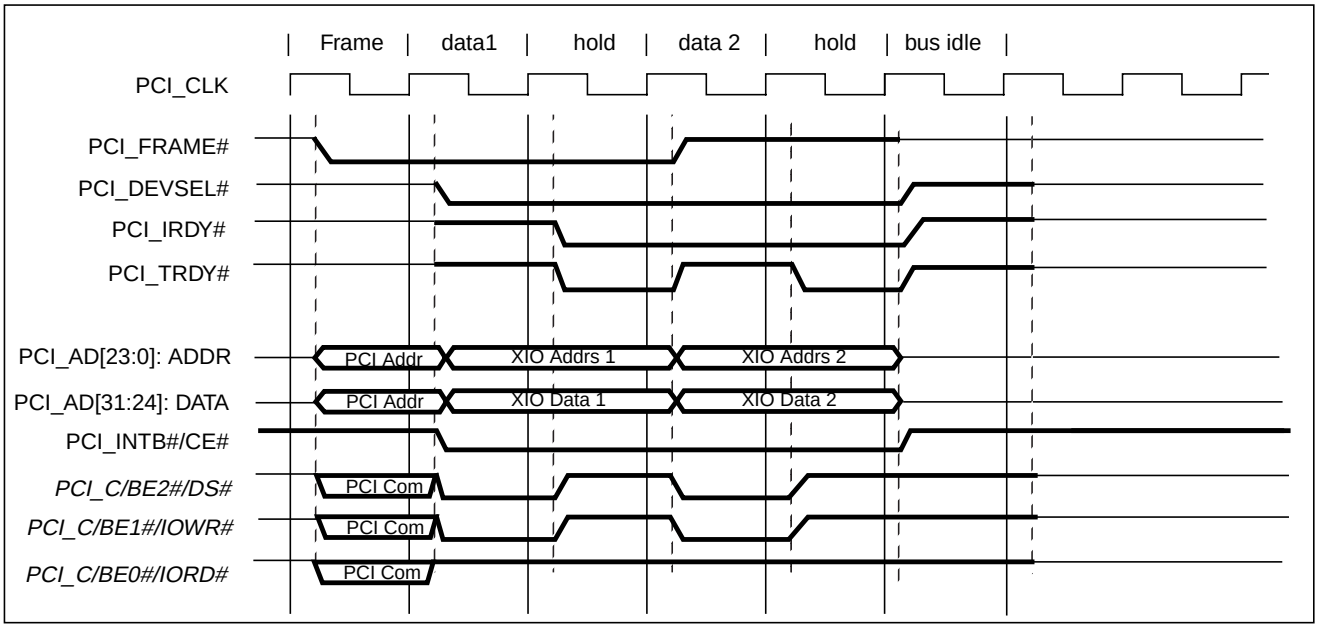


Figure 22-17. PCI-XIO Bus timing: DMA burst write, 2 bytes, 0 wait states

22.7 PCI-XIO BUS CONTROLLER OPERATION AND PROGRAMMING

The PCI-XIO Bus is a PCI target device. All valid PCI transfers with TM1300 as the initiator are allowed, including single word and DMA transfers. When data is read from the PCI-XIO Bus, it reads as a 32-bit word with the 8 bits of data as the most significant byte and the 24-bit XIO Bus transfer address as the least significant bytes. When data is written to the PCI-XIO Bus, it is written as a word, but only the most significant byte of the data is transferred to the bus. The lower 24 bits are ignored as they are replaced by the lower 24 bits of the transfer address before being placed on the bus.

Before the PCI-XIO Bus can be used, the PCI-XIO Bus Control Register must be set up. This register must be loaded with the base address for the PCI-XIO bus and the control fields for clock frequency, wait states per transfer and PCI-XIO Bus enable.

To read a single byte to a PCI-XIO Bus device, first define the 24-bit address for the device. This might be the address in an EPROM for the desired byte. Multiply this device address by four to convert it to a word address and add the XIO Bus base address. The combined address is the PCI transfer address. Use this address as the transfer address for a single word DSPCPU load. Table 22-5 shows examples of this address conversion. At the completion of the load, the data received will consist of 8 bits of data and the 24-bit device address. To write a byte, use the same transfer address and write a word to this address with the desired data as the most significant byte of the word written.

To transfer data between the XIO-PCI bus and the SDRAM using the PCI DMA capability, set the

Table 22-5. PCI to XIO Bus address conversion examples

XIO Bus Address in Hex	PCI Word Address in Hex	XIO-PCI Base Address in Hex	PCI Transfer Address in Hex
11	44	5800 0000	5800 0044
0123	048C	5800 0000	5800 048C
11 0012	44 0048	5800 0000	5844 0048

PCI_SRC_ADR or the PCI_DEST_ADR register to the PCI-XIO Bus transfer address, depending on the direction of the transfer. The PCI-XIO Bus transfer address is four times the starting address as seen on the PCI-XIO Bus address pins plus the PCI-XIO Bus controller base address. This is the starting address for the PCI-XIO Bus transfer. Set the other address, destination or source, to the desired starting address in SDRAM. Set the PCI_DMA_CTL register for the desired direction and set the transfer count to the four times number of PCI-XIO Bus bytes to be transferred. The transfer count is four times the PCI-XIO Bus bytes to be transferred because the PCI-XIO Bus transfers one word to or from the PCI bus for each byte transferred to or from devices on the PCI-XIO Bus.

Word transfer is used to transfer the bytes to and from the PCI bus for hardware simplicity. Additional hardware could be added to pack and unpack bytes, but this is an unnecessary complication given the speed of the PCI-XIO Bus relative to the speed of the TM1300 bus and CPU. The primary intended use of the PCI-XIO Bus is for ROMs, flash EPROMs and I/O devices. Because the PCI-XIO bus is so much slower than the TM1300, there

is time available for the TM1300 to pack and unpack the words. At three PCI-XIO bus wait states, at least 120 nanoseconds are required for each byte transferred. This corresponds to 12 CPU instructions at 100 MHz. The CPU may need to process each byte of data anyway. In the case of ROMs and flash EPROMs, the data is typically compressed, requiring the TM1300 CPU to both unpack and decompress the data.

DSPCPU Operations for TM1300 Appendix A

by Gert Slavenburg, Marcel Janssens

A.1 ALPHABETIC OPERATION LIST

The following table lists the complete operation set of TM1300's DSPCPU. Note that this is not an instruction list; a DSPCPU instruction contains from one to five of these operations.

A.2 OPERATION LIST BY FUNCTION

Allocate a cache block pseudo-op for allocd(0)

alloc

SYNTAX

```
[ IF rguard ] alloc(d) rsrc1
```

ATTRIBUTES

Function unit	dmemspec
Operation code	213
Number of operands	1
Modifier	-
Modifier range	-
Latency	-
Issue slots	5

FUNCTION

if *rguard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache block with [(rsrc1 + 0) & cache_block_mask] address
}

SEE ALSO

allocd allocr allocx

DESCRIPTION

The alloc operation is a pseudo operation transformed by the scheduler into an allocd(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The alloc operation allocate a cache block with the address computed from **[(rsrc1 + 0) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsibility of the programmer to update the allocated cache block by store operations.

Refer to the ‘cache architecture’ section for details on the cache block size.

The alloc operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the alloc operation. If the LSB of rguard is 1, alloc operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`alloc r10`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 alloc r10`	since guard is false, alloc operation is not executed
r10 = 0xac0f, r11 = 1, cache_block_size = 0x40	`IF r11 alloc r10`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

allocd

Allocate a cache block with displacement

SYNTAX

```
[ IF rguard ] allocd(d) rsrc1
```

FUNCTION

if r*guard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache block with [(rsrc1 + d) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	213
Number of operands	1
Modifier	7 bits
Modifier range	-255..252 by 4
Latency	-
Issue slots	5

SEE ALSO

allocr allocx

DESCRIPTION

The allocd operation allocate a cache block with the address computed from **[(rsrc1 + d) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsibility of the programmer to update the allocated cache block by store operations.

Refer to the ‘cache architecture’ section for details on the cache block size.

The `allocd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocd operation. If the LSB of r*guard* is 1, allocd operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`allocd(0x32) r10`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 allocd(0x32) r10`	since guard is false, allocd operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 allocd(0x4) r10`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

Allocate a cache block with index

allocr

SYNTAX

```
[ IF rguard ] allocr rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
    cache_block_mask = ~(cache_block_size -1)]
    allocate adata cache block with [(rsrc1 + rsrc2) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	214
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

allocd allocx

DESCRIPTION

The allocr operation allocate a cache block with the address computed from **[(rsrc1 + rscr2) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsibility of the programmer to update the allocated cache block by store operations.

Refer to the ‘cache architecture’ section for details on the cache block size.

The `allocr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocr operation. If the LSB of r*guard* is 1, allocr operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0x32 cache_block_size = 0x40	`allocr r10 r12`	Allocates a cache block for the address space from 0xabc0 to 0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, r12=0x32, cache_block_size = 0x40	`IF r11 allocr r10 r12`	since guard is false, allocr operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x4, cache_block_size = 0x40	`IF r11 allocr r10 r12`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

allocx

Allocate a cache block with scaled index

SYNTAX

```
[ IF rguard ] allocx rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache blockwith [(rsrc1 + 4 x rsrc2) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	215
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

allocd allocr

DESCRIPTION

The allocx operation allocate a cache block with the address computed from **[(rsrc1 + 4 x rscr2) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsibility of the programmer to update the allocated cache block by store operations.

Refer to the ‘cache architecture’ section for details on the cache block size.

The `allocx` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocx operation. If the LSB of r*guard* is 1, allocx operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xc cache_block_size = 0x40	`allocx r10 r12`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, r12=0xc, cache_block_size = 0x40	`IF r11 allocx r10 r12`	since guard is false, allocx operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x4, cache_block_size = 0x40	`IF r11 allocx r10 r12`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

Arithmetic shift left

asl

SYNTAX

```
[ IF rguard ] asl rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   n ← rsrc2<4:0>
   rdest<31:n> ← rsrc1<31–n:0>
   rdest<n–1:0> ← 0
   if rsrc2<31:5> != 0 {
      rdest <- 0
   }
}
```

ATTRIBUTES

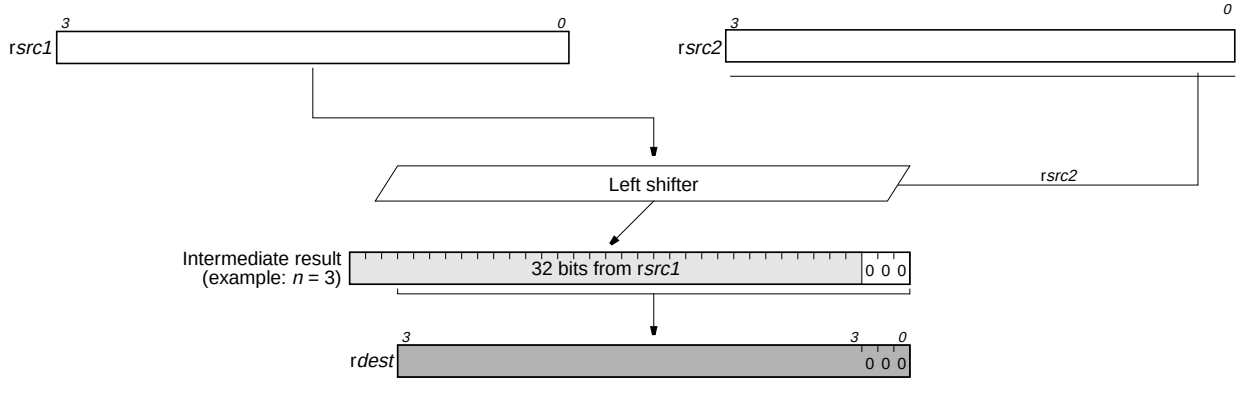
Function unit	shifter
Operation code	19
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asli asr asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the asl operation takes two arguments, r*src1* and r*src2*. R*src2* specify an unsigned shift amount, and r*dest* is set to r*src1* arithmetically shifted left by this amount. If the rsrc2<31:5> value is not zero, then take this as a shift by 32 or more bits. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The asl operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	asl r60 r30 → r90	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	IF r10 asl r60 r30 → r100	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	IF r20 asl r60 r30 → r110	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	asl r70 r40 → r120	r120 ← 0xfffffff0
r80 = 0xe, r50 = 0xfffffffe	asl r80 r50 → r125	r125 ← 0x00000000 (shift by more than 32)
r30 = 0x7008000f, r60 = 0x20	asl r30 r60 → r111	r111 ← 0x00000000
r30 = 0x8008000f, r45 = 0x80000000	asl r30 r45 → r100	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x23	asl r30 r45 → r100	r100 ← 0x00000000

asli

Arithmetic shift left immediate

SYNTAX

```
[ IF rguard ] asli(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:*n*> ← r*src1*<31–*n*:0>
    r*dest*<*n*–1:0> ← 0
}

ATTRIBUTES

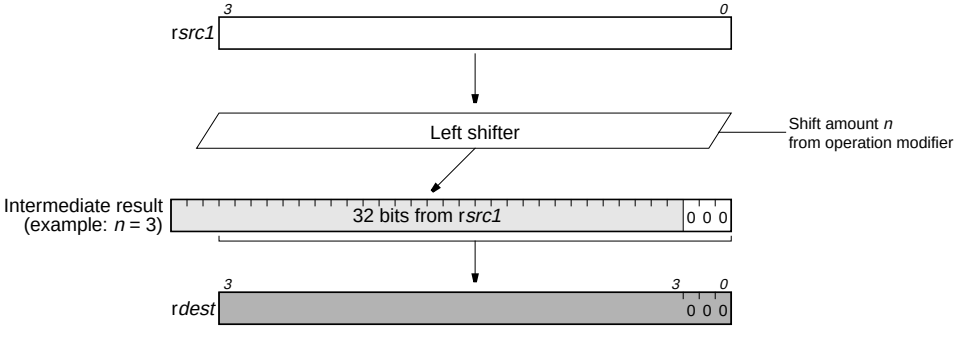
Function unit	shifter
Operation code	11
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asr asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the asli operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* arithmetically shifted left by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The asli operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	asli(3) r60 → r90	r90 ← 0x100
r10 = 0, r60 = 0x20	IF r10 asli(3) r60 → r100	no change, since guard is false
r20 = 1, r60 = 0x20	IF r20 asli(3) r60 → r110	r110 ← 0x100
r70 = 0xfffffffc	asli(2) r70 → r120	r120 ← 0xfffffff0
r80 = 0xe	asli(30) r80 → r125	r125 ← 0x80000000

Arithmetic shift right

asr

SYNTAX

```
[ IF rguard ] asr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   n ← rsrc2<4:0>
   rdest<31:31–n> ← rsrc1<31>
   rdest<30–n:0> ← rsrc1<30:n>
   if rsrc2<31:5> != 0 {
      rdest <- rsrc1<31>
   }
}
```

ATTRIBUTES

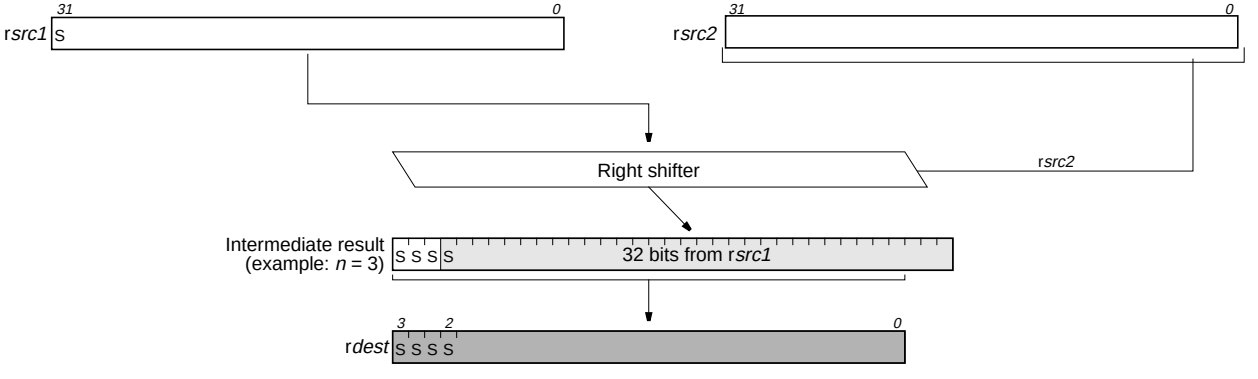
Function unit	shifter
Operation code	18
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the asr operation takes two arguments, r*src1* and r*src2*. R*src2* specifies an unsigned shift amount, and r*src1* is arithmetically shifted right by this amount. If the rsrc2<31:5> value is not zero, then take this as a shift by 32 or more bits. The MSB (sign bit) of r*src1* is replicated as needed to fill vacated bits from the left.



The asr operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f, r20 = 1	asr r30 r20 → r50	r50 ← 0x38040007
r30 = 0x7008000f, r42 = 2	asr r30 r42 → r60	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f, r44 = 4	IF r10 asr r30 r44 → r70	no change, since guard is false
r20 = 1, r30 = 0x7008000f, r44 = 4	IF r20 asr r30 r44 → r80	r80 ← 0x07008000
r40 = 0x80030007, r44 = 4	asr r40 r44 → r90	r90 ← 0xf8003000
r30 = 0x7008000f, r45 = 0x1f	asr r30 r45 → r100	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x1f	asr r30 r45 → r100	r100 ← 0xffffffff
r30 = 0x7008000f, r45 = 0x20	asr r30 r45 → r100	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x20	asr r30 r45 → r100	r100 ← 0xffffffff
r30 = 0x8008000f, r45 = 0x23	asr r30 r45 → r100	r100 ← 0xffffffff

asri

Arithmetic shift right by immediate amount

SYNTAX

```
[ IF rguard ] asri(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:31–*n*> ← r*src1*<31>
    r*dest*<30–*n*:0> ← r*src1*<31:*n*>
}

ATTRIBUTES

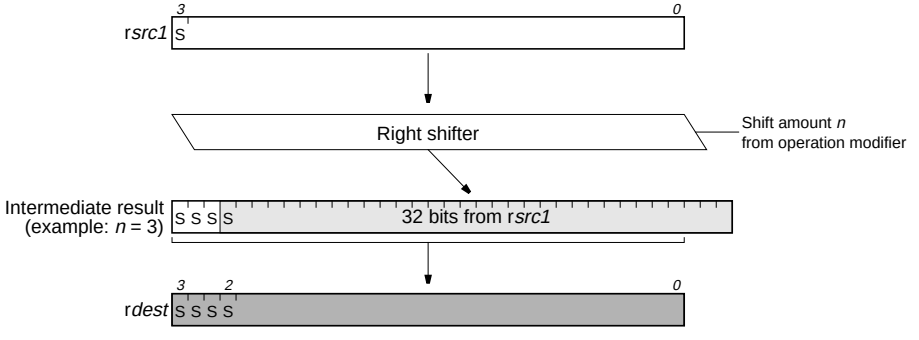
Function unit	shifter
Operation code	10
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the asri operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* that is equal to r*src1* arithmetically shifted right by *n* bits. The value of *n* must be between 0 and 31, inclusive. The MSB (sign bit) of r*src1* is replicated as needed to fill vacated bits from the left.



The asri operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f	asri(1) r30 → r50	r50 ← 0x38040007
r30 = 0x7008000f	asri(2) r30 → r60	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f	IF r10 asri(4) r30 → r70	no change, since guard is false
r20 = 1, r30 = 0x7008000f	IF r20 asri(4) r30 → r80	r80 ← 0x07008000
r40 = 0x80030007	asri(4) r40 → r90	r90 ← 0xf8003000
r30 = 0x7008000f	asri(31) r30 → r100	r100 ← 0x00000000
r40 = 0x80030007	asri(31) r40 → r110	r110 ← 0xffffffff

Bitwise logical AND

bitand

SYNTAX

```
[ IF rguard ] bitand rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← r*src1* & r*src2*

ATTRIBUTES

Function unit	alu
Operation code	16
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitor bitxor bitandinv

DESCRIPTION

The `bitand` operation computes the bitwise, logical AND of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitand` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitand r30 r40 → r90`	r90 ← 0xf3100000
r10 = 0, r50 = 0x88888888	`IF r10 bitand r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitand r30 r50 → r100`	r100 ← 0x80008888
r60 = 0x11119999, r50 = 0x88888888	`bitand r60 r50 → r110`	r110 ← 0x00008888
r70 = 0x55555555, r30 = 0xf310ffff	`bitand r70 r30 → r120`	r120 ← 0x51105555

bitandinv

Bitwise logical AND NOT

SYNTAX

```
[ IF rguard ] bitandinv rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← r*src1* & ~r*src2*

ATTRIBUTES

Function unit	alu
Operation code	49
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitor bitxor

DESCRIPTION

The `bitandinv` operation computes the bitwise, logical AND of the first argument, r*src1*, with the 1's complement of the second argument, r*src2*. The result is stored in the destination register, r*dest*.

The `bitandinv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitandinv r30 r40 → r90`	r90 ← 0x0000ffff
r10 = 0, r50 = 0x88888888	`IF r10 bitandinv r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitandinv r30 r50 → r100`	r100 ← 0x73107777
r60 = 0x11119999, r50 = 0x88888888	`bitandinv r60 r50 → r110`	r110 ← 0x11111111
r70 = 0x55555555, r30 = 0xf310ffff	`bitandinv r70 r30 → r120`	r120 ← 0x04450000

Bitwise logical NOT

bitinv

SYNTAX

```
[ IF rguard ] bitinv rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ~r*src1*

ATTRIBUTES

Function unit	alu
Operation code	50
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`bitand` `bitandinv` `bitor` `bitxor`

DESCRIPTION

The `bitinv` operation computes the bitwise, logical NOT of the argument r*src1* and writes the result into r*dest*.

The `bitinv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff	`bitinv r30 → r60`	r60 ← 0x0cef0000
r10 = 0, r40 = 0xffff0000	`IF r10 bitinv r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0xffff0000	`IF r20 bitinv r40 → r100`	r100 ← 0x0000ffff
r50 = 0x88888888	`bitinv r50 → r110`	r110 ← 0x77777777

bitor

Bitwise logical OR

SYNTAX

```
[ IF rguard ] bitor rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← r*src1* | r*src2*

ATTRIBUTES

Function unit	alu
Operation code	17
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitandinv bitinv bitxor

DESCRIPTION

The `bitor` operation computes the bitwise, logical OR of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitor` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitor r30 r40 → r90`	r90 ← 0xffffffff
r10 = 0, r50 = 0x88888888	`IF r10 bitor r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitor r30 r50 → r100`	r100 ← 0xfb98ffff
r60 = 0x11119999, r50 = 0x88888888	`bitor r60 r50 → r110`	r110 ← 0x99999999
r70 = 0x55555555, r30 = 0xf310ffff	`bitor r70 r30 → r120`	r120 ← 0xf755ffff

Bitwise logical exclusive-OR

bitxor

SYNTAX

```
[ IF rguard ] bitxor rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← r*src1* ⊕ r*src2*

ATTRIBUTES

Function unit	alu
Operation code	48
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitandinv bitinv bitor

DESCRIPTION

The `bitxor` operation computes the bitwise, logical exclusive-OR of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitxor` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitxor r30 r40 → r90`	r90 ← 0x0cefffff
r10 = 0, r50 = 0x88888888	`IF r10 bitxor r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitxor r30 r50 → r100`	r100 ← 0x7b987777
r60 = 0x11119999, r50 = 0x88888888	`bitxor r60 r50 → r110`	r110 ← 0x99991111
r70 = 0x55555555, r30 = 0xf310ffff	`bitxor r70 r30 → r120`	r120 ← 0xa645aaaa

borrow

Compute borrow bit from unsigned subtract

pseudo-op for ugtr

SYNTAX

```
[ IF rguard ] borrow rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 < rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ugtr carry

DESCRIPTION

The `borrow` operation is a pseudo operation transformed by the scheduler into an `ugtr` with reversed arguments. (Note: pseudo operations cannot be used in assembly source files.)

The `borrow` operation computes the unsigned difference of the first and second arguments, r*src1*–r*src2*. If the difference generates a borrow (if r*src2* > r*src1*), 1 is stored in the destination register, r*dest*; otherwise, r*dest* is set to 0.

The `borrow` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 2, r30 = 0xfffffffc	`borrow r70 r30 → r80`	r80 ← 1
r10 = 0, r70 = 2, r30 = 0xfffffffc	`IF r10 borrow r70 r30 → r90`	no change, since guard is false
r20 = 1, r70 = 2, r30 = 0xfffffffc	`IF r20 borrow r70 r30 → r100`	r100 ← 1
r60 = 4, r30 = 0xfffffffc	`borrow r60 r30 → r110`	r110 ← 1
r30 = 0xfffffffc	`borrow r30 r30 → r120`	r120 ← 0

Compute carry bit from unsigned add

carry

SYNTAX

```
[ IF rguard ] carry rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if (r*src1*+r*src2*) < 2^{32} **then**
    r*dest* ← 0
  else
    r*dest* ← 1
}

ATTRIBUTES

Function unit	alu
Operation code	45
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

borrow

DESCRIPTION

The `carry` operation computes the unsigned sum of the first and second arguments, r*src1*+r*src2*. If the sum generates a carry (if the sum is greater than 2^{32}-1), 1 is stored in the destination register, r*dest*; otherwise, r*dest* is set to 0.

The `carry` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 2, r30 = 0xfffffffc	`carry r70 r30 → r80`	r80 ← 0
r10 = 0, r70 = 2, r30 = 0xfffffffc	`IF r10 carry r70 r30 → r90`	no change, since guard is false
r20 = 1, r70 = 2, r30 = 0xfffffffc	`IF r20 carry r70 r30 → r100`	r100 ← 0
r60 = 4, r30 = 0xfffffffc	`carry r60 r30 → r110`	r110 ← 1
r30 = 0xfffffffc	`carry r30 r30 → r120`	r120 ← 1

curcycles

Read current clock cycle counter, least-significant word

SYNTAX

```
[ IF rguard ] curcycles → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← CCCOUNT<31:0>

ATTRIBUTES

Function unit	fcomp
Operation code	162
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`cycles hicycles writepcsw`

DESCRIPTION

Refer to Section 3.1.6, “CCCOUNT—Clock Cycle Counter” for a description of the CCCOUNT operation. The `curcycles` operation copies the current low 32 bits of the master Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*.. The master CCCOUNT increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The `curcycles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	`curcycles → r60`	r30 ← 0x12345678
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 curcycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 curcycles → r100`	r100 ← 0x12345678

Read clock cycle counter, least-significant word

cycles

SYNTAX

```
[ IF rguard ] cycles → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← CCCOUNT<31:0>

ATTRIBUTES

Function unit	fcomp
Operation code	154
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`hicycles` `curcycles` `writepcsw`

DESCRIPTION

Refer to Section 3.1.6, “CCCOUNT—Clock Cycle Counter” for a description of the CCCOUNT operation. The `cycles` operation copies the low 32 bits of the slave register of Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*. The contents of the master counter are transferred to the slave CCCOUNT register only on a successful interruptible jump and on processor reset. Thus, if `cycles` and `hicycles` are executed without intervening interruptible jumps, the operation pair is guaranteed to be a coherent sample of the master clock-cycle counter. The master counter increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The `cycles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	`cycles → r60`	r30 ← 0x12345678
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 cycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 cycles → r100`	r100 ← 0x12345678

dcb

Data cache copy back

SYNTAX

```
[ IF rguard ] dcb(d) rsrc1
```

FUNCTION

```
if rguard then {
   addr ← rsrc1 + d
   if dcache_valid_addr(addr) && dcache_dirty_addr(addr) then {
      dcache_copyback_addr(addr)
      dcache_reset_dirty_addr(addr)
   }
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	205
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

dinvalid

DESCRIPTION

The dcb operation causes a block in the data cache to be copied back to main memory if the block is marked dirty and valid, and the block's dirty bit is reset. The target block of dcb is the block in the data cache that contains the byte addressed by r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

A valid copy of the target block remains in the cache. Stall cycles are taken as necessary to complete the copy-back operation. If the target block is not dirty or if the block is not in the cache, dcb has no effect and no stall cycles are taken.

dcb has no effect on blocks that are in the non-cacheable SDRAM aperture. dcb does not change the replacement status of data-cache blocks.

dcb ensures coherency between caches and main memory by discarding all pending prefetch operations and by causing all non-empty copyback buffers to be emptied to main memory.

The dcb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls if the operation is carried out or not.If the LSB of r*guard* is 1, the operation is carried out; otherwise,it is not carried out.

EXAMPLES

Initial Values	Operation	Result
	`dcb(0) r30`	
r10 = 0	`IF r10 dcb(4) r40`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 dcb(8) r50`	

Invalidate data cache block

dinvalid

SYNTAX

```
[ IF rguard ] dinvalid(d) rsrc1
```

FUNCTION

```
if rguard then {
   addr ← rsrc1 + d
   if dcache_valid_addr(addr) then {
      dcache_reset_valid_addr(addr)
      dcache_reset_dirty_addr(addr)
   }
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	206
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

dcb

DESCRIPTION

The `dinvalid` operation resets the valid and dirty bit of a block in the data cache. Regardless of the block's dirty bit, the block is not written back to main memory. The target block of `dinvalid` is the block in the data cache that contains the byte addressed by r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

Stall cycles are taken as necessary to complete the invalidate operation. If the target block is not in the cache, `dinvalid` has no effect and no stall cycles are taken.

`dinvalid` has no effect on blocks that are in the non-cacheable SDRAM aperture. `dinvalid` does clear the valid bits of locked blocks. `dinvalid` does not change the replacement status of data-cache blocks.

`dinvalid` ensures coherency between caches and main memory by discarding all pending prefetch operations and by causing all non-empty copyback buffers to be emptied to main memory.

The `dinvalid` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls if the operation is carried out or not. If the LSB of r*guard* is 1, the operation is carried out; otherwise, it is not carried out.

EXAMPLES

Initial Values	Operation	Result
	`dinvalid(0) r30`	
r10 = 0	`IF r10 dinvalid(4) r40`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 dinvalid(8) r50`	

dspiabs

Clipped signed absolute value

pseudo-op for h_dspiabs

SYNTAX

```
[ IF rguard ] dspiabs rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 >= 0 then
      rdest ← rsrc1
   else if rsrc1 = 0x80000000 then
      rdest ← 0x7fffffff
   else
      rdest ← –rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	65
Number of operands	1
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

h_dspiabs h_dspidualabs dspiadd dspimul dspisub dspuadd dspumul dspusub

DESCRIPTION

The dspiabs operation is a pseudo operation transformed by the scheduler into an h_dspiabs with a constant first argument zero and second argument equal to the dspiabs argument. (Note: pseudo operations cannot be used in assembly source files.)

The dspiabs operation computes the absolute value of rsrc1, clips the result into the range [2^{31}–1..0] (or [0x7fffffff..0]), and stores the clipped value into r*dest*. All values are signed integers.

The dspiabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	dspiabs r30 → r60	r60 ← 0x00000001
r10 = 0, r40 = 0x80000001	IF r10 dspiabs r40 → r70	no change, since guard is false
r20 = 1, r40 = 0x80000001	IF r20 dspiabs r40 → r100	r100 ← 0x7fffffff
r50 = 0x80000000	dspiabs r50 → r80	r80 ← 0x7fffffff
r90 = 0x7fffffff	dspiabs r90 → r110	r110 ← 0x7fffffff

Clipped signed add

dspiadd

SYNTAX

```
[ IF rguard ] dspiadd rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspalu
Operation code	66
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

FUNCTION

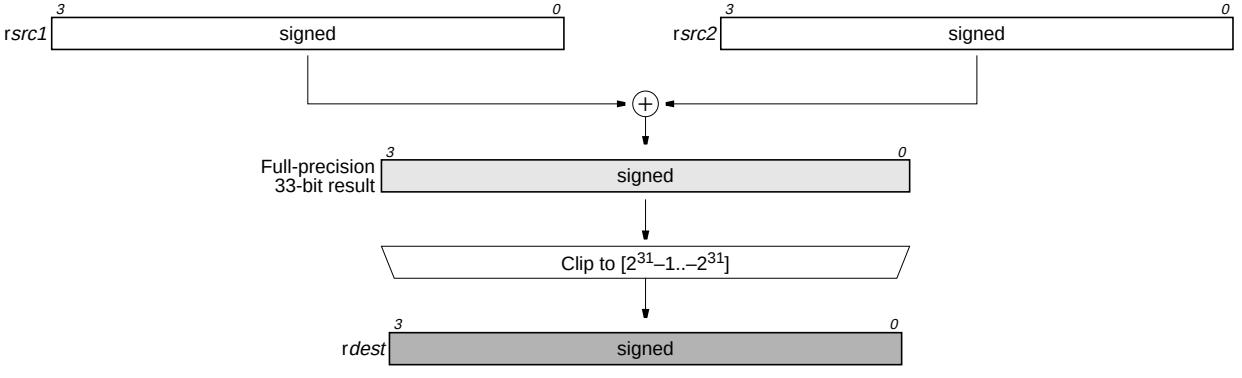
```
if rguard then {
   temp ← sign_ext32to64(rsrc1) + sign_ext32to64(rsrc2)
   if temp < 0xffffffff80000000 then
      rdest ← 0x80000000
   else if temp > 0x000000007fffffff then
      rdest ← 0x7fffffff
   else
      rdest ← temp
}
```

SEE ALSO

dspiabs dspimul dspisub dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspiadd operation computes the sum r*src1*+r*src2*, clips the result into the 32-bit signed range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into r*dest*. All values are signed integers.



The dspiadd operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	dspiadd r30 r40 → r60	r60 ← 0x12ff
r10 = 0, r30 = 0x1200, r40 = 0xff	IF r10 dspiadd r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	IF r20 dspiadd r30 r40 → r100	r100 ← 0x12ff
r50 = 0x7fffffff, r90 = 1	dspiadd r50 r90 → r110	r110 ← 0x7fffffff
r70 = 0x80000000, r80 = 0xffffffff	dspiadd r70 r80 → r120	r120 ← 0x80000000

dspidualabs

Dual clipped absolute value of signed 16-bit halfwords

pseudo-op for h_dspidualabs

SYNTAX

```
[ IF rguard ] dspidualabs rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  temp1 ← sign_ext16to32(r*src1*<15:0>)
  temp2 ← sign_ext16to32(r*src1*<31:16>)
  if temp1 = 0xffff8000 **then** temp1 ← 0x7fff
  if temp2 = 0xffff8000 **then** temp2 ← 0x7fff
  if temp1 < 0 **then** temp1 ← –temp1
  if temp2 < 0 **then** temp2 ← –temp2
  r*dest*<31:16> ← temp2<15:0>
  r*dest*<15:0> ← temp1<15:0>
}

ATTRIBUTES

Function unit	dspalu
Operation code	72
Number of operands	1
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

h_dspidualabs dspiabs dspidualadd dspidualmul dspidualsub

DESCRIPTION

The `dspidualabs` operation is a pseudo operation transformed by the scheduler into an `h_dspidualabs` with a constant zero as first argument and the `dspidualabs` argument as second argument. (Note: pseudo operations cannot be used in assembly source files.)

The `dspidualabs` operation performs two 16-bit clipped, signed absolute value computations separately on the high and low 16-bit halfwords of r*src1*. Both absolute values are clipped into the range [0x0..0x7fff] and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.

The `dspidualabs` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0032	`dspidualabs r30 → r60`	r60 ← 0x00010032
r10 = 0, r40 = 0x80008001	`IF r10 dspidualabs r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0x80008001	`IF r20 dspidualabs r40 → r100`	r100 ← 0x7fff7fff
r50 = 0x0032ffff	`dspidualabs r50 → r80`	r80 ← 0x00320001
r90 = 0x7fffffff	`dspidualabs r90 → r110`	r110 ← 0x7fff0001

Dual clipped add of signed 16-bit halfwords

dspidualadd

SYNTAX

```
[ IF rguard ] dspidualadd rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  temp1 ← sign_ext16to32(r*src1*<15:0>) + sign_ext16to32(r*src2*<15:0>)
  temp2 ← sign_ext16to32(r*src1*<31:16>) + sign_ext16to32(r*src2*<31:16>)
  if temp1 < 0xffff8000 **then** temp1 ← 0x8000
  if temp2 < 0xffff8000 **then** temp2 ← 0x8000
  if temp1 > 0x7fff **then** temp1 ← 0x7fff
  if temp2 > 0x7fff **then** temp2 ← 0x7fff
  r*dest*<31:16> ← temp2<15:0>
  r*dest*<15:0> ← temp1<15:0>
}

ATTRIBUTES

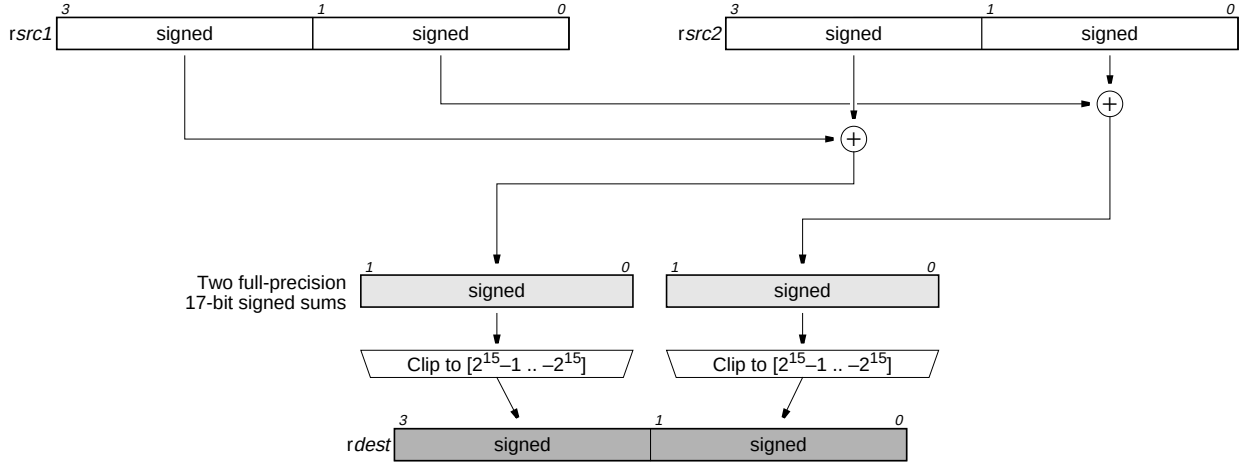
Function unit	dspalu
Operation code	70
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualabs dspidualmul dspidualsub dspiabs

DESCRIPTION

As shown below, the dspidualadd operation computes two 16-bit clipped, signed sums separately on the two pairs of high and low 16-bit halfwords of r*src1* and r*src2*. Both sums are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.



The dspidualadd operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12340032, r40 = 0x00010002	dspidualadd r30 r40 → r60	r60 ← 0x12350034
r10 = 0, r30 = 0x12340032, r40 = 0x00010002	IF r10 dspidualadd r30 r40 → r70	no change, since guard is false
r20 = 1, r30 = 0x12340032, r40 = 0x00010002	IF r20 dspidualadd r30 r40 → r100	r100 ← 0x12350034
r50 = 0x80000001, r80 = 0xffff7fff	dspidualadd r50 r80 → r90	r90 ← 0x80007fff
r110 = 0x00017fff, r120 = 0x7fff7fff	dspidualadd r110 r120 → r125	r125 ← 0x7fff7fff

dspidualmul — Dual clipped multiply of signed 16-bit halfwords

SYNTAX

```
[ IF rguard ] dspidualmul rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspmul
Operation code	95
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

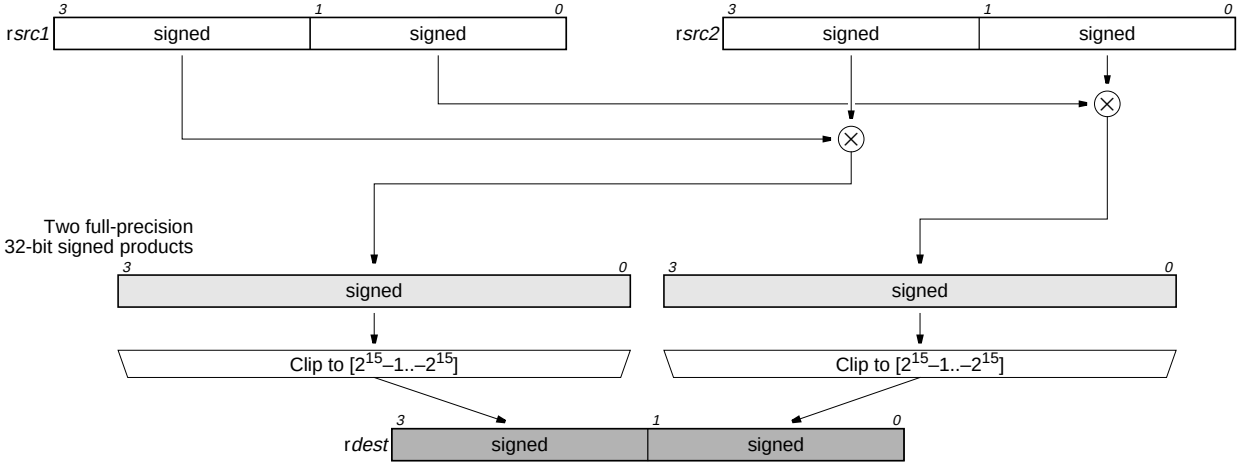
dspidualabs dspidualadd dspidualsub dspiabs

FUNCTION

if r*guard* **then {**
  temp1 ← sign_ext16to32(r*src1*<15:0>) × sign_ext16to32(r*src2*<15:0>)
  temp2 ← sign_ext16to32(r*src1*<31:16>) × sign_ext16to32(r*src2*<31:16>)
  if temp1 < 0xffff8000 **then** temp1 ← 0x8000
  if temp2 < 0xffff8000 **then** temp2 ← 0x8000
  if temp1 > 0x7fff **then** temp1 ← 0x7fff
  if temp2 > 0x7fff **then** temp2 ← 0x7fff
  r*dest*<31:16> ← temp2<15:0>
  r*dest*<15:0> ← temp1<15:0>
}

DESCRIPTION

As shown below, the `dspidualmul` operation computes two 16-bit clipped, signed products separately on the two pairs of high and low 16-bit halfwords of r*src1* and r*src2*. Both products are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.



The `dspidualmul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0020010, r40 = 0x00030020	`dspidualmul r30 r40 → r60`	r60 ← 0x00060200
r10 = 0, r30 = 0x0020010, r40 = 0x00030020	`IF r10 dspidualmul r30 r40 → r70`	no change, since guard is false
r20 = 1, r30 = 0x0020010, r40 = 0x00030020	`IF r20 dspidualmul r30 r40 → r100`	r100 ← 0x00060200
r50 = 0x80000002, r80 = 0x00024000	`dspidualmul r50 r80 → r90`	r90 ← 0x80007fff
r110 = 0x08000003, r120 = 0x00108001	`dspidualmul r110 r120 → r125`	r125 ← 0x7fff8000

Dual clipped subtract of signed 16-bit halfwords

dspidualsub

SYNTAX

```
[ IF rguard ] dspidualsub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
  temp1 ← sign_ext16to32(rsrc1<15:0>) – sign_ext16to32(rsrc2<15:0>)
  temp2 ← sign_ext16to32(rsrc1<31:16>) – sign_ext16to32(rsrc2<31:16>)
  if temp1 < 0xffff8000 then temp1 ← 0x8000
  if temp2 < 0xffff8000 then temp2 ← 0x8000
  if temp1 > 0x7fff then temp1 ← 0x7fff
  if temp2 > 0x7fff then temp2 ← 0x7fff
  rdest<31:16> ← temp2<15:0>
  rdest<15:0> ← temp1<15:0>
}
```

ATTRIBUTES

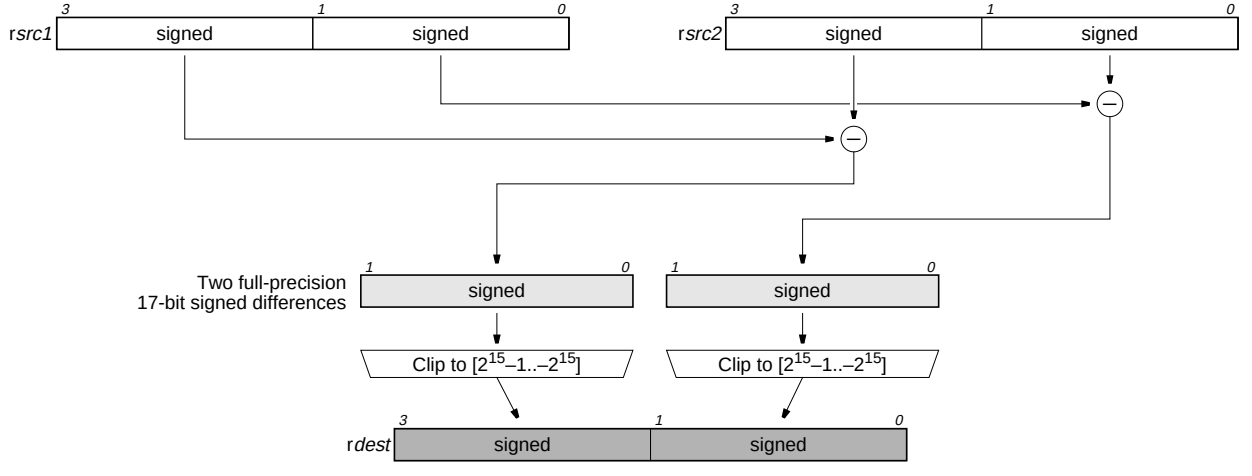
Function unit	dspalu
Operation code	71
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualabs dspidualadd dspidualmul dspiabs

DESCRIPTION

As shown below, the dspidualsub operation computes two 16-bit clipped, signed differences separately on the two pairs of high and low 16-bit halfwords of r*src1* and r*src2*. Both differences are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.



The dspidualsub operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12340032, r40 = 0x00010002	dspidualsub r30 r40 → r60	r60 ← 0x12330030
r10 = 0, r30 = 0x12340032, r40 = 0x00010002	IF r10 dspidualsub r30 r40 → r70	no change, since guard is false
r20 = 1, r30 = 0x12340032, r40 = 0x00010002	IF r20 dspidualsub r30 r40 → r100	r100 ← 0x12330030
r50 = 0x80000001, r80 = 0x00018001	dspidualsub r50 r80 → r90	r90 ← 0x80007fff
r110 = 0x00018001, r120 = 0x80010002	dspidualsub r110 r120 → r125	r125 ← 0x7fff8000

dspimul

Clipped signed multiply

SYNTAX

```
[ IF rguard ] dspimul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  temp ← sign_ext32to64(r*src1*) × sign_ext32to64(r*src2*)
  if temp < 0xffffffff80000000 **then**
    r*dest* ← 0x80000000
  else if temp > 0x000000007fffffff **then**
    r*dest* ← 0x7fffffff
  else
    r*dest* ← temp<31:0>
}

ATTRIBUTES

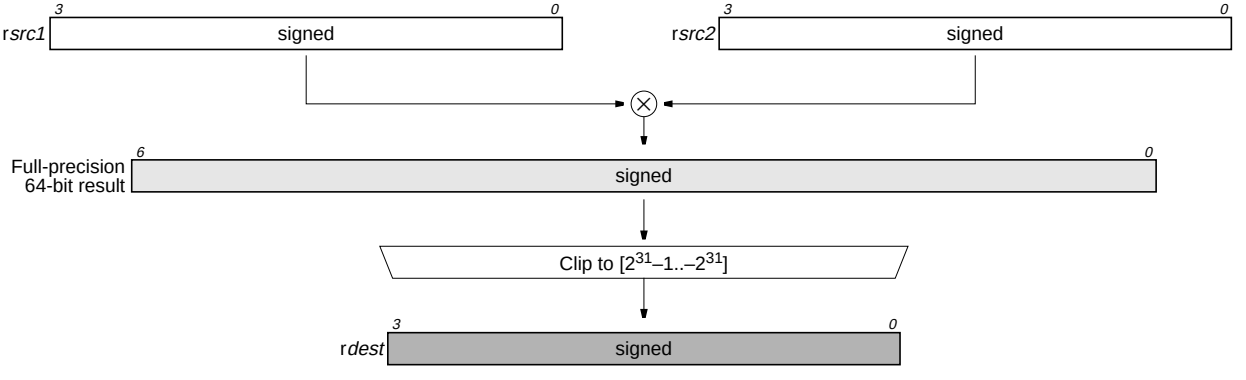
Function unit	ifmul
Operation code	141
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

dspiabs dspiadd dspisub dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspimul operation computes the product r*src1*×r*src2*, clips the result into the 32-bit range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into r*dest*. All values are signed integers.



The dspimul operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x10, r40 = 0x20	dspimul r30 r40 → r60	r60 ← 0x200
r10 = 0, r30 = 0x10, r40 = 0x20	IF r10 dspimul r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x10, r40 = 0x20	IF r20 dspimul r30 r40 → r100	r100 ← 0x200
r50 = 0x40000000, r90 = 2	dspimul r50 r90 → r110	r110 ← 0x7fffffff
r80 = 0xffffffff	dspimul r80 r80 → r120	r120 ← 0x1
r70 = 0x80000000, r90 = 2	dspimul r70 r90 → r120	r120 ← 0x80000000

Clipped signed subtract

dspisub

SYNTAX

```
[ IF rguard ] dspisub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   temp ← sign_ext32to64(rsrc1)  – sign_ext32to64(rsrc2)
   if temp < 0xffffffff80000000 then
      rdest ← 0x80000000
   else if temp > 0x000000007fffffff then
      rdest ← 0x7fffffff
   else
      rdest ← temp<31:0>
}
```

ATTRIBUTES

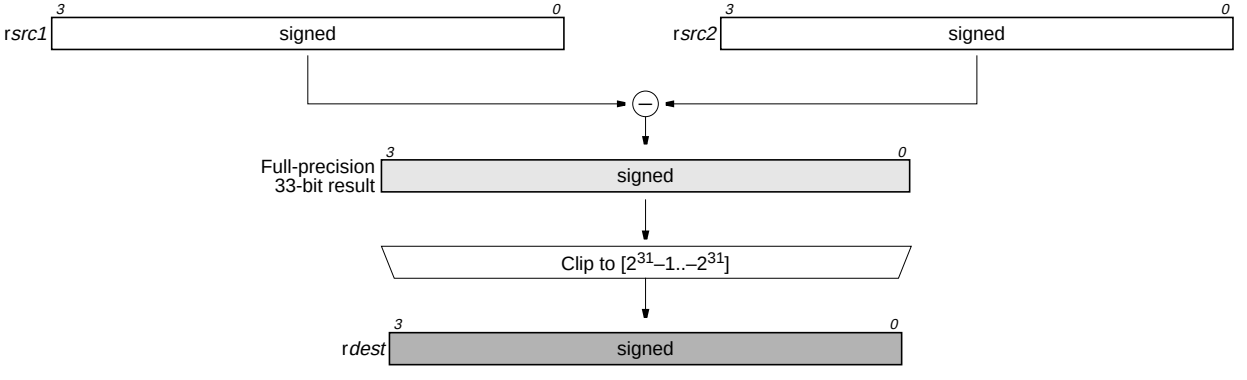
Function unit	dspalu
Operation code	68
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspisub operation computes the difference r*src1*–r*src2*, clips the result into the 32-bit range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into r*dest*. All values are signed integers.



The dspisub operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	dspisub r30 r40 → r60	r60 ← 0x1101
r10 = 0, r30 = 0x1200, r40 = 0xff	IF r10 dspisub r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	IF r20 dspisub r30 r40 → r100	r100 ← 0x1101
r50 = 0x7fffffff, r90 = 0xffffffff	dspisub r50 r90 → r110	r110 ← 0x7fffffff
r70 = 0x80000000, r80 = 1	dspisub r70 r80 → r120	r120 ← 0x80000000

dspuadd

Clipped unsigned add

SYNTAX

```
[ IF rguard ] dspuadd rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← zero_ext32to64(rsrc1) + zero_ext32to64(rsrc2)
    if (unsigned)temp > 0x00000000ffffffff then
        rdest ← 0xffffffff
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

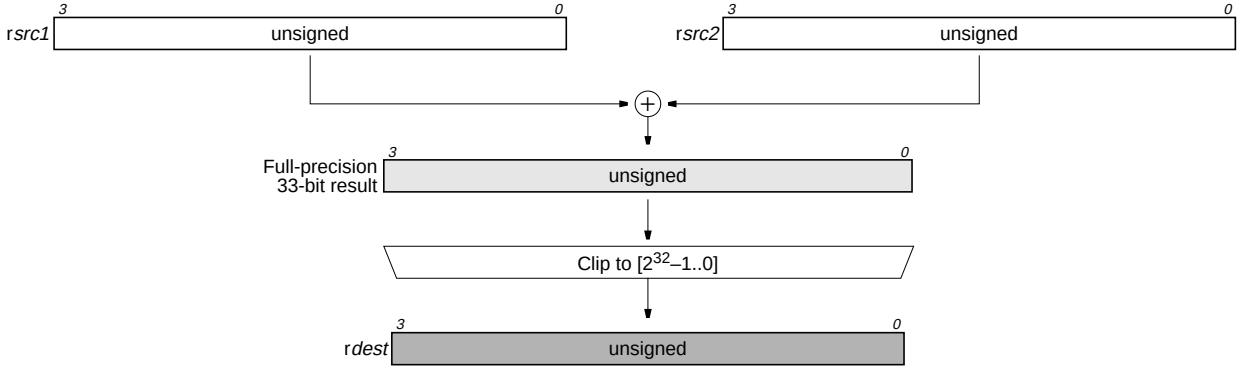
Function unit	dspalu
Operation code	67
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul dspisub dspumul dspusub

DESCRIPTION

As shown below, the dspuadd operation computes unsigned sum r*src1*+r*src2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into r*dest*.



The dspuadd operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	dspuadd r30 r40 → r60	r60 ← 0x12ff
r10 = 0, r30 = 0x1200, r40 = 0xff	IF r10 dspuadd r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	IF r20 dspuadd r30 r40 → r100	r100 ← 0x12ff
r50 = 0xffffffff, r90 = 1	dspuadd r50 r90 → r110	r110 ← 0xffffffff
r70 = 0x80000001, r80 = 0x7fffffff	dspuadd r70 r80 → r120	r120 ← 0xffffffff

Clipped unsigned multiply

dspumul

SYNTAX

```
[ IF rguard ] dspumul rsrc1 rsrc2 → rdest
```

OPERATION

```
if rguard then {
   temp ← zero_ext32to64(rsrc1) × zero_ext32to64(rsrc2)
   if (unsigned)temp > 0x00000000ffffffff then
      rdest ← 0xffffffff
   else
      rdest ← temp<31:0>
}
```

ATTRIBUTES

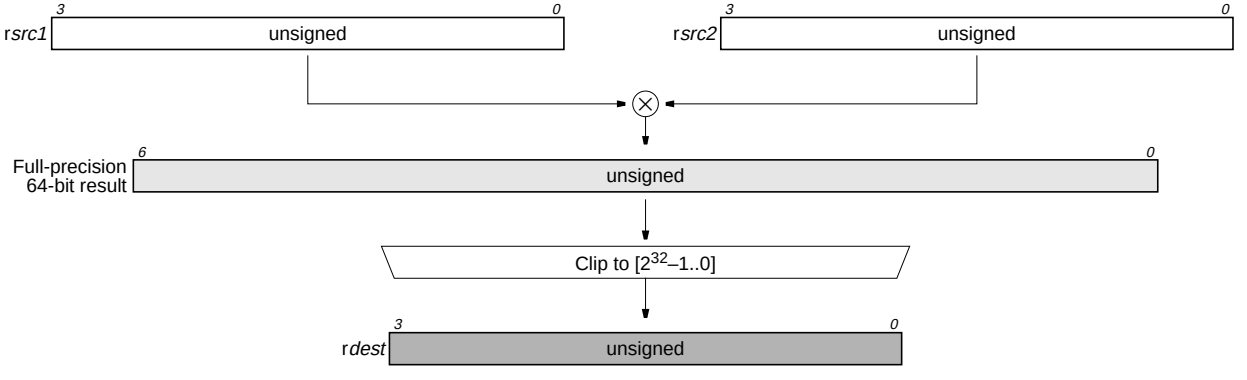
Function unit	ifmul
Operation code	142
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

dspiabs dspiadd dspisub
dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspumul operation computes unsigned product r*src1*×r*src2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into r*dest*.



The dspumul operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x10, r40 = 0x20	dspumul r30 r40 → r60	r60 ← 0x200
r10 = 0, r30 = 0x10, r40 = 0x20	IF r10 dspumul r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x10, r40 = 0x20	IF r20 dspumul r30 r40 → r100	r100 ← 0x200
r50 = 0x40000000, r90 = 2	dspumul r50 r90 → r110	r110 ← 0x80000000
r80 = 0xffffffff	dspumul r80 r80 → r120	r120 ← 0xffffffff
r70 = 0x80000000, r90 = 2	dspumul r70 r90 → r120	r120 ← 0xffffffff

dspuquadaddui — Quad clipped add of unsigned/signed bytes

SYNTAX

```
[ IF rguard ] dspuquadaddui rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   for (i ← 0, m ← 31, n ← 24; i < 4; i ← i + 1, m ← m – 8, n ← n – 8) {
      temp ← zero_ext8to32(rsrc1<m:n>) + sign_ext8to32(rsrc2<m:n>)
      if temp < 0 then
         rdest<m:n> ← 0
      else if temp > 0xff then
         rdest<m:n> ← 0xff
      else rdest<m:n> ← temp<7:0>
   }
}
```

ATTRIBUTES

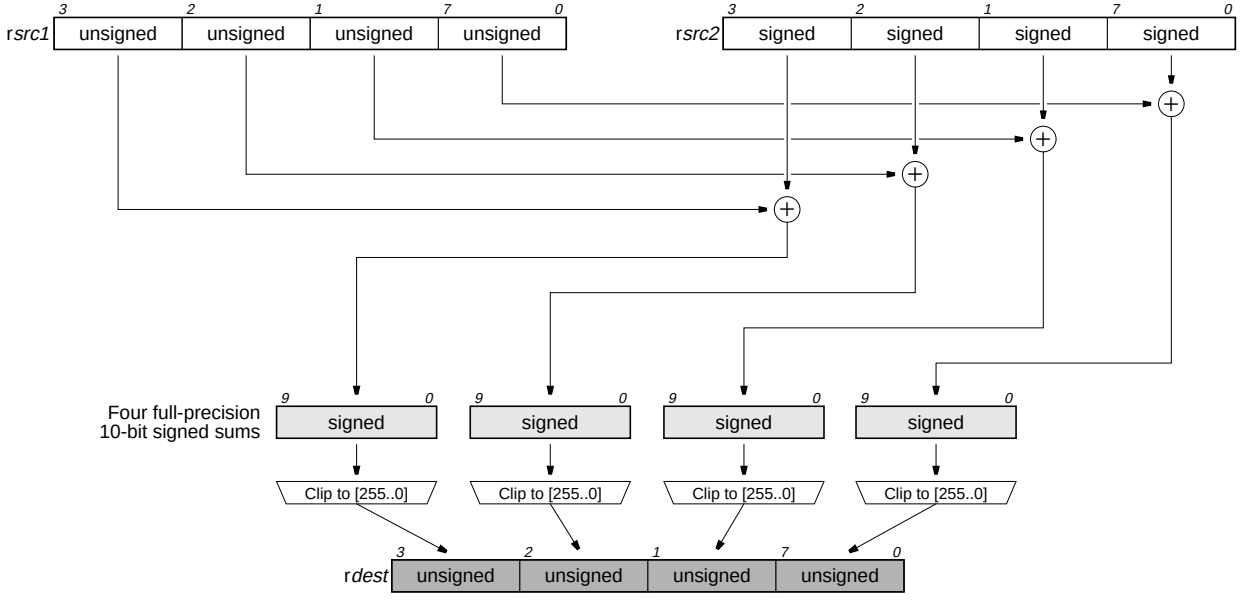
Function unit	dspalu
Operation code	78
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualadd

DESCRIPTION

As shown below, the dspuquadaddui operation computes four separate sums of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. The bytes in r*src1* are considered unsigned values; the bytes in r*src2* are considered signed. The four sums are clipped into the unsigned range [255..0] (or [0xff..0]); thus, the final byte sums are unsigned. All computations are performed without loss of precision.



The dspuquadaddui operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x02010001, r40 = 0xffffff01	dspuquadaddui r30 r40 → r50	r50 ← 0x01000002
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649c649c	IF r10 dspuquadaddui r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649c649c	IF r20 dspuquadaddui r60 r70 → r90	r90 ← 0xff38c800

Clipped unsigned subtract

dspusub

SYNTAX

```
[ IF rguard ] dspusub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   temp ← zero_ext32to64(rsrc1) – zero_ext32to64(rsrc2)
   if (signed)temp < 0 then
      rdest ← 0
   else
      rdest ← temp<31:0>
}
```

ATTRIBUTES

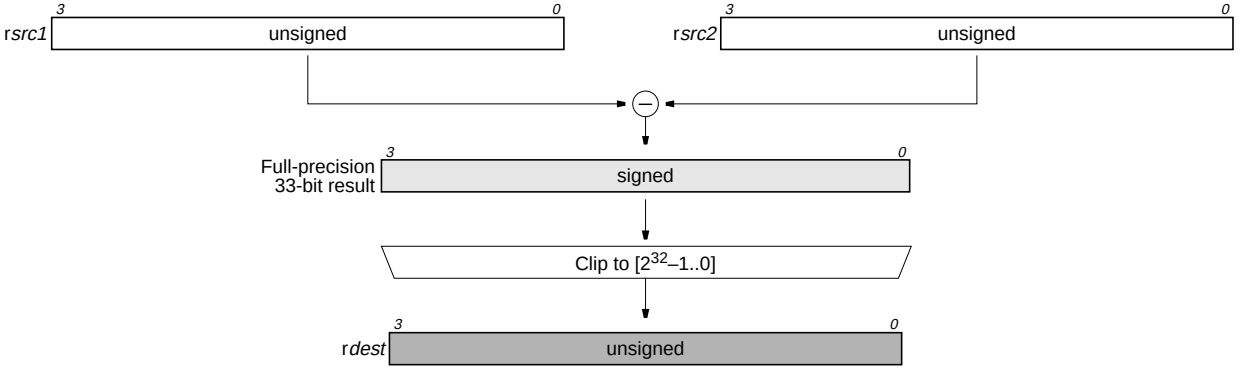
Function unit	dspalu
Operation code	69
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul
dspisub dspuadd dspumul

DESCRIPTION

As shown below, the dspusub operation computes unsigned difference r*src1*–r*src2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into r*dest*.



The dspusub operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	dspusub r30 r40 → r60	r60 ← 0x1101
r10 = 0, r30 = 0x1200, r40 = 0xff	IF r10 dspusub r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	IF r20 dspusub r30 r40 → r100	r100 ← 0x1101
r50 = 0, r90 = 1	dspusub r50 r90 → r110	r110 ← 0
r70 = 0x80000001, r80 = 0xffffffff	dspusub r70 r80 → r120	r120 ← 0

dualasr

Dual-16 arithmetic shift right

SYNTAX

```
[ IF rguard ] dualasr rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	shifter
Operation code	102
Number of operands	2
Modifier	No
Modifier range	-
Latency	1
Issue slots	1,2

FUNCTION

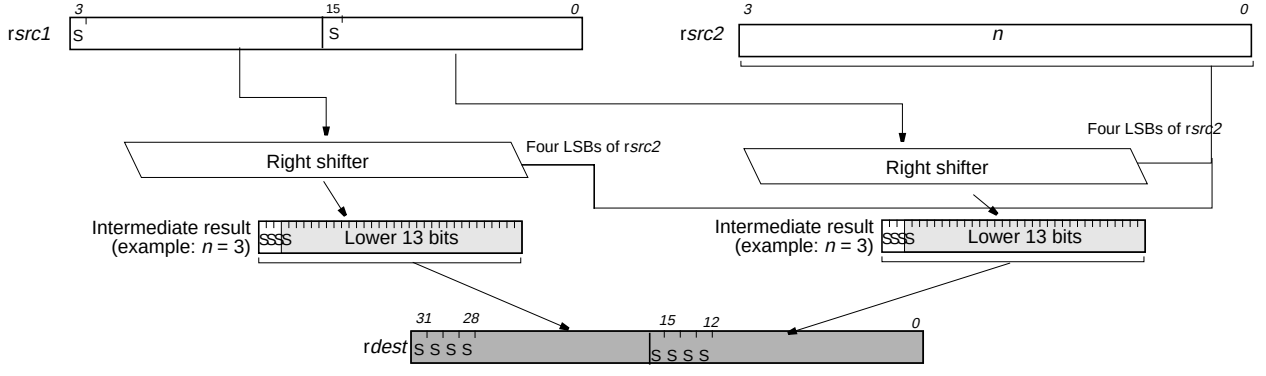
```
if rguard then {
    n <- rsrc2<3:0>
    rdest<31:31-n> <- rsrc1<31>
    rdest<30-n:16> <- rsrc1<30:16+n>
    rdest<15:15-n> <- rsrc1<15>
    rdest<14-n:0> <- rsrc1<14:n>
    if rsrc2<31:4> != 0 {
        rdest<31:16> <- rsrc1<31>
        rdest<15:0> <- rsrc1<15>
    }
}
```

SEE ALSO

asl asli asri lsl lsli lsr lsri rol roli

DESCRIPTION

The argument rsrc1 contains two 16-bit signed integers, rsrc1<31:16> and rsrc1<15:0>. Rsrc2 specifies an unsigned shift amount, and the two 16-bit integers shifted right by this amount. The sign bits rsrc1<31> and rsrc1<15> are replicated as needed within each 16-bit value from the left. If the rsrc2<31:4> value is not zero, then take this as a shift by 16 or more, i.e. extend the sign bit into either result.



The dualasr operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the modification of the destination register. If the LSB of rguard is 1, rdest is written; otherwise, rdest is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x70087008, r40 = 0x1	`dualasr r30 r40 -> r50`	r50 <- 0x38043804
r30 = 0x70087008, r40 = 0x2	`dualasr r30 r40 -> r50`	r50 <- 0x1c021c02
r10 = 0, r30 = 0x70087008, r40 = 0x2	`IF r10 dualasr r30 r40 -> r50`	no change, since guard is false
r10 = 1, r30 = 0x70084008, r40 = 0x4	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0x07000400
r10 = 1, r30 = 0x800c800c, r40 = 0x4	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0xf800f800
r10 = 1, r30 = 0x700c700c, r40 = 0xf	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0x00000000
r10 = 1, r30 = 0x700c800c, r40 = 0xf	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0x0000ffff
r10 = 1, r30 = 0x800c700c, r40 = 0xf	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0xffff0000
r10 = 1, r30 = 0x800c700c, r40 = 0x10000000	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0xffff0000
r10 = 1, r30 = 0x800c700c, r40 = 0x10	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0xffff0000

Dual-16 clip signed to signed

dualiclipi

SYNTAX

```
[ IF rguard ] dualiclipi rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
rdest<31:16> <- min(max(rscrc1<31:16>, -rsrc2<15:0>-1), rsrc2<15:0>)
rdest<15:0> <- min(max(rscrc1<15:0>, -rsrc2<15:0>-1), rsrc2<15:0>)
}

ATTRIBUTES

Function unit	dspalu
Operation code	82
Number of operands	2
Modifier	No
Modifier range	-
Latency	2
Issue slots	1,3

SEE ALSO

iclipi uclipi dualuclipi imin imax quadumax quadumin

DESCRIPTION

The argument rsrc1 contains two signed16-bit integers, rsrc1<31:16> and rsrc1<15:0>. Each integer value is clipped into the signed integer range (-rsrc2 -1) to rsrc2. The value in rsrc2 contains an unsigned integer and must have the value between 0 and 0x7fff inclusive.

The dualiclipi operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the modification of the destination register. If the LSB of rguard is 1, rdest is written; otherwise, rdest is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00800080, r40 = 0x7f	`dualiclipi r30 r40 -> r50`	r50 <- 0x007f007f
r30 = 0x7ffff7ffff, r40 = 0x7ffe	`dualiclipi r30 r40 -> r50`	r50 <- 0x7ffe7ffe
r10 = 0, r30 = 0x7ffff7ffff, r40 = 0x7ffe	`IF r10 dualiclipi r30 r40 -> r50`	no change, since guard is false
r10 = 1, r30 = 0x12345678, r40 = 0xabc	`IF r10 dualiclipi r30 r40 -> r50`	r50 <- 0x0abc0abc
r10 = 1, r30 = 0x80008000, r40 = 0x03ff	`IF r10 dualiclipi r30 r40 -> r50`	r50 <- 0xfc00fc00
r10 = 1, r30 = 0x800003fe, r40 = 0x03ff	`IF r10 dualiclipi r30 r40 -> r50`	r50 <- 0xfc0003fe
r10 = 1, r30 = 0x000f03fe, r40 = 0x03ff	`IF r10 dualiclipi r30 r40 -> r50`	r50 <- 0x000f03fe

dualuclipi

Dual-16 clip signed to unsigned

SYNTAX

```
[ IF rguard ] dualuclipi rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
rdest<31:16> <- min(max(rscrc1<31:16>, 0), rsrc2<15:0>)
rdest<15:0> <- min(max(rscrc1<15:0>, 0), rsrc2<15:0>)
}

ATTRIBUTES

Function unit	dspalu
Operation code	83
Number of operands	2
Modifier	No
Modifier range	-
Latency	2
Issue slots	1,3

SEE ALSO

iclipi uclipi dualiclipi imin imax quadumax quadumin

DESCRIPTION

The argument rsrc1 contains two 16-bit signed integers, rsrc1<31:16> and rsrc1<15:0>. Each integer value is clipped into the unsigned integer range 0 to rsrc2. The value in rsrc2 contains an unsigned integer and must have the value between 0 and 0xffff inclusive.

The dualuclipi operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the modification of the destination register. If the LSB of rguard is 1, rdest is written; otherwise, rdest is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00800080, r40 = 0x7f	`dualuclipi r30 r40 -> r50`	r50 <- 0x007f007f
r30 = 0x7ffff7ffff, r40 = 0x7ffe	`dualuclipi r30 r40 -> r50`	r50 <- 0x7ffe7ffe
r10 = 0, r30 = 0x7ffff7ffff, r40 = 0x7ffe	`IF r10 dualuclipi r30 r40 -> r50`	no change, since guard is false
r10 = 1, r30 = 0x12345678, r40 = 0xabc	`IF r10 dualuclipi r30 r40 -> r50`	r50 <- 0x0abc0abc
r10 = 1, r30 = 0x80008000, r40 = 0x03ff	`IF r10 dualuclipi r30 r40 -> r50`	r50 <- 0x00000000
r10 = 1, r30 = 0x800003fe, r40 = 0x03ff	`IF r10 dualuclipi r30 r40 -> r50`	r50 <- 0x000003fe
r10 = 1, r30 = 0x000f03fe, r40 = 0x03ff	`IF r10 dualuclipi r30 r40 -> r50`	r50 <- 0x000f03fe

Floating-point absolute value

fabsval

SYNTAX

```
[ IF rguard ] fabsval rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 < 0 then
      rdest ← –(float)rsrc1
   else
      rdest ← (float)rsrc1
}
```

ATTRIBUTES

Function unit	falu
Operation code	115
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

iabs dspiabs dspidualabs fabsvalflags readpcsw writepcsw

DESCRIPTION

The `fabsval` operation computes the absolute value of the argument r*src1* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. If an argument is denormalized, zero is substituted for the argument before computing the absolute value, and the IFZ flag in the PCSW is set. If `fabsval` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fabsvalflags` operation computes the exception flags that would result from an individual `fabsval`.

The `fabsval` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fabsval r30 → r90`	r90 ← 0x40400000 (3.0)
r35 = 0xbf800000 (-1.0)	`fabsval r35 → r95`	r95 ← 0x3f800000 (1.0)
r40 = 0x00400000 (5.877471754e-39)	`fabsval r40 → r100`	r100 ← 0x0 (+0.0), IFZ set
r45 = 0xffffffff (QNaN)	`fabsval r45 → r105`	r105 ← 0xffffffff (QNaN)
r50 = 0xffbfffff (SNaN)	`fabsval r50 → r110`	r110 ← 0xffffffff (QNaN), INV set
r10 = 0, r55 = 0xff7fffff (–3.402823466e+38)	`IF r10 fabsval r55 → r115`	no change, since guard is false
r20 = 1, r55 = 0xff7fffff (–3.402823466e+38)	`IF r20 fabsval r55 → r120`	r120 ← 0x7f7fffff (3.402823466e+38)

fabsvalflags

IEEE status flags from floating-point absolute value

SYNTAX

```
[ IF rguard ] fabsvalflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags(abs_val((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	116
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

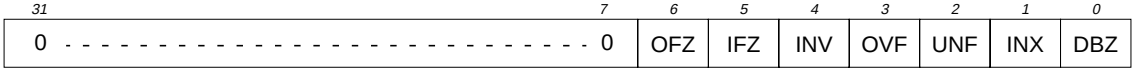
SEE ALSO

`fabsval faddflags readpcsw`

DESCRIPTION

The `fabsvalflags` operation computes the IEEE exceptions that would result from computing the absolute value of r*src1* and writes a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If r*src1* is denormalized, the IFZ bit in the result is set.

The `fabsvalflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fabsvalflags r30 → r90`	r90 ← 0x0
r35 = 0xbf800000 (-1.0)	`fabsvalflags r35 → r95`	r95 ← 0x0
r40 = 0x00400000 (5.877471754e-39)	`fabsvalflags r40 → r100`	r100 ← 0x20 (IFZ)
r45 = 0xffffffff (QNaN)	`fabsvalflags r45 → r105`	r105 ← 0x0
r50 = 0xffbfffff (SNaN)	`fabsvalflags r50 → r110`	r110 ← 0x10 (INV)
r10 = 0, r55 = 0xff7fffff (–3.402823466e+38)	`IF r10 fabsvalflags r55 → r115`	no change, since guard is false
r20 = 1, r55 = 0xff7fffff (–3.402823466e+38)	`IF r20 fabsvalflags r55 → r120`	r120 ← 0x0

Floating-point add

fadd

SYNTAX

```
[ IF rguard ] fadd rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← (float)r*src1* + (float)r*src2*

ATTRIBUTES

Function unit	falu
Operation code	22
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

faddflags iadd dspiadd dspidualadd readpcsw writepcsw

DESCRIPTION

The `fadd` operation computes the sum r*src1*+r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the sum, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fadd` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `faddflags` operation computes the exception flags that would result from an individual `fadd`.

The `fadd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fadd r60 r30 → r90`	r90 ← 0xc0000000 (–2.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fadd r40 r60 → r95`	r95 ← 0x00000000 (0.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r10 fadd r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r20 fadd r40 r80 → r110`	r110 ← 0x40400000 (3.0), INX flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fadd r40 r81 → r111`	r111 ← 0x40400000 (3.0), IFZ flag set
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x80800000 (–1.175494351e-38)	`fadd r82 r83 → r112`	r112 ← 0x00000000 (0.0), OFZ, UNF, INX flags set
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fadd r84 r85 → r113`	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38)	`fadd r70 r70 → r120`	r120 ← 0x7f800000 (+INF), OVF, INX flags set
r80 = 0x00800000 (1.763241526e–38)	`fadd r80 r80 → r125`	r125 ← 0x01000000 (2.350988702e–38)

faddflags

IEEE status flags from floating-point add

SYNTAX

```
[ IF rguard ] faddflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← ieee_flags((float)r*src1* + (float)r*src2*)

ATTRIBUTES

Function unit	falu
Operation code	112
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

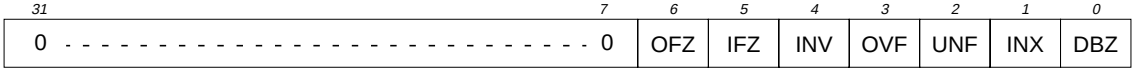
SEE ALSO

fadd fsubflags readpcsw

DESCRIPTION

The `faddflags` operation computes the IEEE exceptions that would result from computing the sum r*src1*+r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the sum, and the IFZ bit in the result is set. If the sum would be denormalized, the OFZ bit in the result is set.

The `faddflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r10 = 0x7f7fffff (3.402823466e+38), r20 = 0x3f800000 (1.0)	`faddflags r10 r20 → r60`	r60 ← 0x2 (INX)
r30 = 0, r10 = 0x7f7fffff (3.402823466e+38)	`IF r30 faddflags r10 r10 → r50`	no change, since guard is false
r40 = 1, r10 = 0x7f7fffff (3.402823466e+38)	`IF r40 faddflags r10 r10 → r70`	r70 ← 0xa (OVF INX)
r80 = 0x00a00000 (1.469367939e–38), r81 = 0x80800000 (–1.17549435e–38)	`faddflags r80 r81 → r100`	r100 ← 0x46 (OFZ UNF INX)
r95 = 0x7f800000 (+INF), r96 = 0xff800000 (–INF)	`faddflags r95 r96 → r105`	r105 ← 0x10 (INV)
r98 = 0x40400000 (3.0), r99 = 0x00400000 (5.877471754e–39)	`faddflags r98 r99 → r111`	r111 ← 0x20 (IFZ)

Floating-point divide

fdiv

SYNTAX

```
[ IF rguard ] fdiv rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← (float)r*src1* / (float)r*src2*

ATTRIBUTES

Function unit	ftough
Operation code	108
Number of operands	2
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

SEE ALSO

`fdivflags` `readpcsw` `writepcsw`

DESCRIPTION

The `fdiv` operation computes the quotient r*src1*÷r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the quotient, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fdiv` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fdivflags` operation computes the exception flags that would result from an individual `fdiv`.

The `fdiv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fdiv r60 r30 → r90`	r90 ← 0xc0400000 (–3.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fdiv r40 r60 → r95`	r95 ← 0xbf800000 (–1.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fdiv r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fdiv r40 r80 → r110`	r110 ← 0x7f400000 (2.552117754e38)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fdiv r40 r81 → r111`	r111 ← 0x7f800000 (+INF), IFZ, DBZ flags set
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x80800000 (–1.175494351e–38)	`fdiv r82 r83 → r112`	r112 ← 0xbfc00000 (-1.5)
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fdiv r84 r85 → r113`	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38)	`fdiv r70 r70 → r120`	r120 ← 0x3f800000 (1.0)
r80 = 0x00800000 (1.763241526e–38)	`fdiv r80 r80 → r125`	r125 ← 0x3f800000 (1.0)
r75 = 0x40400000 (3.0), r76 = 0x0 (0.0)	`fdiv r75 r76 → r126`	r126 ← 0x7f800000 (+INF), DBZ flag set

fdivflags

IEEE status flags from floating-point divide

SYNTAX

```
[ IF rguard ] fdivflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← ieee_flags((float)r*src1* / (float)r*src2*)

ATTRIBUTES

Function unit	ftough
Operation code	109
Number of operands	2
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

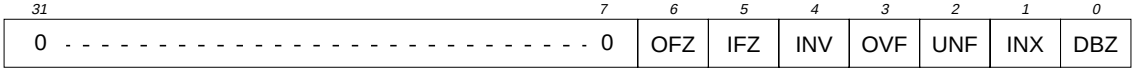
SEE ALSO

fdiv faddflags readpcsw

DESCRIPTION

The `fdivflags` operation computes the IEEE exceptions that would result from computing the quotient r*src1*÷r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the quotient, and the IFZ bit in the result is set. If the quotient would be denormalized, the OFZ bit in the result is set.

The `fdivflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x7f7fffff (3.402823466e+38), r40 = 0x3f800000 (1.0)	`fdivflags r30 r40 → r100`	r100 ← 0
r10 = 0, r50 = 0x7f7fffff (3.402823466e+38) r60 = 0x3e000000 (0.125)	`IF r10 fdivflags r50 r60 → r110`	no change, since guard is false
r20 = 1, r50 = 0x7f7fffff (3.402823466e+38) r60 = 0x3e000000 (0.125)	`IF r20 fdivflags r50 r60 → r111`	r111 ← 0xa (OVF INX)
r70 = 0x40400000 (3.0), r80 = 0x00400000 (5.877471754e–39)	`fdivflags r70 r80 → r112`	r112 ← 0x21 (IFZ DBZ)
r85 = 0x7f800000 (+INF), r86 = 0xff800000 (–INF)	`fdivflags r85 r86 → r113`	r113 ← 0x10 (INV)

Floating-point compare equal

feql

SYNTAX

```
[ IF rguard ] feql rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 = (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	148
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

ieql feqlflags fneq
readpcsw writepcsw

DESCRIPTION

The `feql` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `feql` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `feqlflags` operation computes the exception flags that would result from an individual `feql`.

The `feql` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`feql r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`feql r30 r30 → r90`	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 feql r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 feql r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`feql r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`feql r30 r61 → r121`	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`feql r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`feql r60 r65 → r126`	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	`feql r50 r50 → r127`	r127 ← 1

feqlflags

IEEE status flags from floating-point compare equal

SYNTAX

```
[ IF rguard ] feqlflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((float)r*src1* = (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	149
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

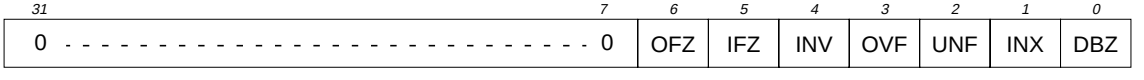
SEE ALSO

feql ieql fgtrflags readpcsw

DESCRIPTION

The `feqlflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `feqlflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`feqlflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`feqlflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 feqlflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 feqlflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`feqlflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`feqlflags r30 r61 → r121`	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`feqlflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`feqlflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`feqlflags r50 r50 → r127`	r127 ← 0

Floating-point compare greater or equal

fgeq

SYNTAX

```
[ IF rguard ] fgeq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 >= (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	146
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

igeq fgeqflags fgtr readpcsw writepcsw

DESCRIPTION

The `fgeq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fgeq` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fgeqflags` operation computes the exception flags that would result from an individual `fgeq`.

The `fgeq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgeq r30 r40 → r80`	r80 ← 1
r30 = 0x40400000 (3.0)	`fgeq r30 r30 → r90`	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgeq r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgeq r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgeq r30 r60 → r120`	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgeq r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgeq r50 r55 → r125`	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgeq r60 r65 → r126`	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	`fgeq r50 r50 → r127`	r127 ← 1

fgeqflags

IEEE status flags from floating-point compare greater or equal

SYNTAX

```
[ IF rguard ] fgeqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* >= (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	147
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

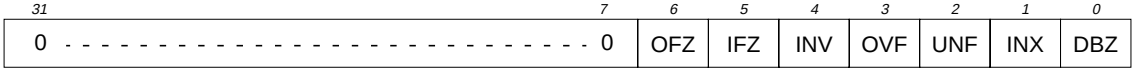
SEE ALSO

fgeq igeq fgtrflags readpcsw

DESCRIPTION

The `fgeqflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*>=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fgeqflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgeqflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fgeqflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgeqflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgeqflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgeqflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgeqflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgeqflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgeqflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fgeqflags r50 r50 → r127`	r127 ← 0

Floating-point compare greater

fgtr

SYNTAX

```
[ IF rguard ] fgtr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 > (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	144
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

igtr fgtrflags fgeq
readpcsw writepcsw

DESCRIPTION

The `fgtr` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fgtr` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fgtrflags` operation computes the exception flags that would result from an individual `fgtr`.

The `fgtr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgtr r30 r40 → r80`	r80 ← 1
r30 = 0x40400000 (3.0)	`fgtr r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgtr r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgtr r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgtr r30 r60 → r120`	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgtr r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgtr r50 r55 → r125`	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgtr r60 r65 → r126`	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	`fgtr r50 r50 → r127`	r127 ← 0

fgtrflags

IEEE status flags from floating-point compare greater

SYNTAX

```
[ IF rguard ] fgtrflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((float)r*src1* > (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	145
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

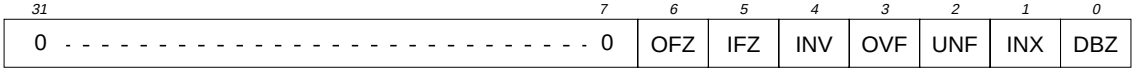
SEE ALSO

fgtr igtr fgeqflags readpcsw

DESCRIPTION

The `fgtrflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*>r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fgtrflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgtrflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fgtrflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgtrflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgtrflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgtrflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgtrflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgtrflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgtrflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fgtrflags r50 r50 → r127`	r127 ← 0

Floating-point compare less-than or equal

pseudo-op for fgeq

fleq

SYNTAX

```
[ IF rguard ] fleq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 <= (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	146
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

ileq fgeq fleqflags readpcsw writepcsw

DESCRIPTION

The `fleq` operation is a pseudo operation transformed by the scheduler into an `fgeq` with the arguments exchanged (`fleq`'s r*src1* is `fgeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fleq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fleq` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fleqflags` operation computes the exception flags that would result from an individual `fleq`.

The `fleq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fleq r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fleq r30 r30 → r90`	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fleq r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fleq r60 r30 → r110`	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fleq r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fleq r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fleq r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fleq r60 r65 → r126`	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	`fleq r50 r50 → r127`	r127 ← 1

fleqflags

IEEE status flags from floating-point compare less-than or equal

pseudo-op for fgeqflags

SYNTAX

```
[ IF rguard ] fleqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((float)r*src1* <= (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	147
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

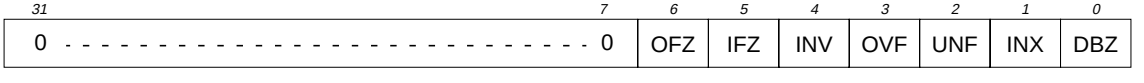
`fleq ileq fgeqflags readpcsw`

DESCRIPTION

The `fleqflags` operation is a pseudo operation transformed by the scheduler into an `fgeqflags` with the arguments exchanged (`fleqflags`'s r*src1* is `fgeqflags`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fleqflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*<=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fleqflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fleqflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fleqflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fleqflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fleqflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fleqflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fleqflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fleqflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fleqflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fleqflags r50 r50 → r127`	r127 ← 0

Floating-point compare less-than

pseudo-op for fgtr

fles

SYNTAX

```
[ IF rguard ] fles rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 < (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	144
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

iles fgtr flesflags readpcsw writepcsw

DESCRIPTION

The `fles` operation is a pseudo operation transformed by the scheduler into an `fgtr` with the arguments exchanged (`fles`'s r*src1* is `fgtr`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fles` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fles` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `flesflags` operation computes the exception flags that would result from an individual `fles`.

The `fles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fles r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fles r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fles r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fles r60 r30 → r110`	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fles r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fles r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fles r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fles r60 r65 → r126`	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	`fles r50 r50 → r127`	r127 ← 0

flesflags

IEEE status flags from floating-point compare less-than

pseudo-op for fgtrflags

SYNTAX

```
[ IF rguard ] flesflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* < (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	145
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

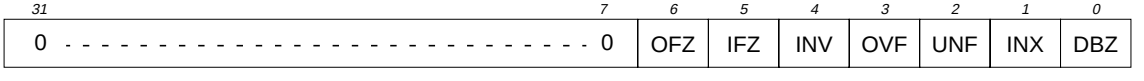
fles iles fleqflags
readpcsw

DESCRIPTION

The `flesflags` operation is a pseudo operation transformed by the scheduler into an `fgtrflags` with the arguments exchanged (`flesflags`'s r*src1* is `fgtrflags`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `flesflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*<r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `flesflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`flesflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`flesflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 flesflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 flesflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`flesflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`flesflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`flesflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`flesflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`flesflags r50 r50 → r127`	r127 ← 0

Floating-point multiply

fmul

SYNTAX

```
[ IF rguard ] fmul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← (float)r*src1* × (float)r*src2*

ATTRIBUTES

Function unit	ifmul
Operation code	28
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

imul umul dspimul dspidualmul fmulflags readpcsw writepcsw

DESCRIPTION

The `fmul` operation computes the product r*src1*×r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the product, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fmul` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fmulflags` operation computes the exception flags that would result from an individual `fmul`.

The `fmul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fmul r60 r30 → r90`	r90 ← 0xc0400000 (-3.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fmul r40 r60 → r95`	r95 ← 0xc1100000 (-9.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fmul r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fmul r40 r80 → r105`	r105 ← 0x1400000 (3.52648305e-38)
r41 = 0x3f000000 (0.5), r80 = 0x00800000 (1.17549435e–38)	`fmul r41 r80 → r110`	r110 ← 0x0, OFZ, UNF, INX flags set
r42 = 0x7f800000 (+INF), r43 = 0x0 (0.0)	`fmul r42 r43 → r106`	r106 ← 0xffffffff (QNaN), INV flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fmul r40 r81 → r111`	r111 ← 0, IFZ flag set
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x8080000 (–1.175494351e–38)	`fmul r82 r83 → r112`	r112 ← 0, UNF, INX flag set
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fmul r84 r85 → r113`	r113 ← 0xff800000 (-INF)
r70 = 0x7f7fffff (3.402823466e+38)	`fmul r70 r70 → r120`	r120 ← 0x7f800000, OVF, INX flags set
r80 = 0x00800000 (1.763241526e–38)	`fmul r80 r80 → r125`	r125 ← 0, UNF, INX flag set

fmulflags

IEEE status flags from floating-point multiply

SYNTAX

```
[ IF rguard ] fmulflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← ieee_flags((float)r*src1* × (float)r*src2*)

ATTRIBUTES

Function unit	ifmul
Operation code	143
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

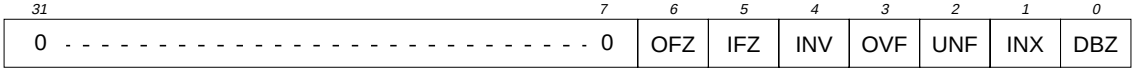
SEE ALSO

`fmul faddflags readpcsw`

DESCRIPTION

The `fmulflags` operation computes the IEEE exceptions that would result from computing the product r*src1*×r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the product, and the IFZ bit in the result is set. If the product would be denormalized, the OFZ bit in the result is set.

The `fmulflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fmulflags r60 r30 → r90`	r90 ← 0
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fmulflags r40 r60 → r95`	r95 ← 0
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fmulflags r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fmulflags r40 r80 → r105`	r105 ← 0
r41 = 0x3f000000 (0.5), r80 = 0x00800000 (1.17549435e–38)	`fmulflags r41 r80 → r110`	r110 ← 0x46 (OFZ UNF INX)
r42 = 0x7f800000 (+INF), r43 = 0x0 (0.0)	`fmulflags r42 r43 → r106`	r106 ← 0x10 (INV)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fmulflags r40 r81 → r111`	r111 ← 0x20 (IFZ)
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x8080000 (–1.175494351e–38)	`fmulflags r82 r83 → r112`	r112 ← 0x06 (UNF INX)
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fmulflags r84 r85 → r113`	r113 ← 0
r70 = 0x7f7fffff (3.402823466e+38)	`fmulflags r70 r70 → r120`	r120 ← 0x0a (OVF INX)
r80 = 0x00800000 (1.763241526e–38)	`fmulflags r80 r80 → r125`	r125 ← 0x06 (UNF INX)

Floating-point compare not equal

fneq

SYNTAX

```
[ IF rguard ] fneq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 != (float)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	150
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

ineq feql fneqflags readpcsw writepcsw

DESCRIPTION

The fneq operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If fneq causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit writepcsw operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The fneqflags operation computes the exception flags that would result from an individual fneq.

The fneq operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	fneq r30 r40 → r80	r80 ← 1
r30 = 0x40400000 (3.0)	fneq r30 r30 → r90	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r10 fneq r60 r30 → r100	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r20 fneq r60 r30 → r110	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	fneq r30 r60 → r120	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	fneq r30 r61 → r121	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	fneq r50 r55 → r125	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	fneq r60 r65 → r126	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	fneq r50 r50 → r127	r127 ← 0

fneqflags

IEEE status flags from floating-point compare not equal

SYNTAX

```
[ IF rguard ] fneqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((float)r*src1* != (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	151
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

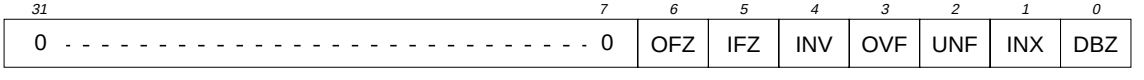
SEE ALSO

fneq ineq fleqflags readpcsw

DESCRIPTION

The `fneqflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*!=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fneqflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fneqflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fneqflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fneqflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fneqflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fneqflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fneqflags r30 r61 → r121`	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fneqflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fneqflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fneqflags r50 r50 → r127`	r127 ← 0

Sign of floating-point value

fsign

SYNTAX

```
[ IF rguard ] fsign rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (float)rsrc1 = 0.0 then
      rdest ← 0
   else if (float)rsrc1 < 0.0 then
      rdest ← 0xffffffff
   else
      rdest ← 1
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	152
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

fsignflags readpcsw writepcsw

DESCRIPTION

The fsign operation sets the destination register, r*dest*, to either 0, 1, or –1 depending on the sign of the argument in r*src1*. rdest is set to 0 if r*src1* is equal to zero, to 1 if r*src1* is positive, or to –1 if r*src1* is negative. The argument is treated as an IEEE single-precision floating-point value; the result is an integer. If the argument is denormalized, zero is substituted before computing the comparison, and the IFZ flag in the PCSW is set; thus, the result of fsign for a denormalized argument is 0. If fsign causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit writepcsw operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The fsignflags operation computes the exception flags that would result from an individual fsign.

The fsign operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	fsign r30 → r100	r100 ← 1
r40 = 0xbf800000 (-1.0)	fsign r40 → r105	r105 ← 0xffffffff (-1)
r50 = 0x80800000 (-1.175494351e-38)	fsign r50 → r110	r110 ← 0xffffffff (-1)
r60 = 0x80400000 (-5.877471754e-39)	fsign r60 → r115	r115 ← 0, IFZ flag set
r10 = 0, r70 = 0xffffffff (QNaN)	IF r10 fsign r70 → r116	no change, since guard is false
r20 = 1, r70 = 0xffffffff (QNaN)	IF r20 fsign r70 → r117	r117 ← 0, INV flag set
r80 = 0xff800000 (-INF)	fsign r80 → r120	r120 ← 0xffffffff (-1)

fsignflags

IEEE status flags from floating-point sign

SYNTAX

```
[ IF rguard ] fsignflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest* ← ieee_flags(sign((float)r*src1*))

ATTRIBUTES

Function unit	fcomp
Operation code	153
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

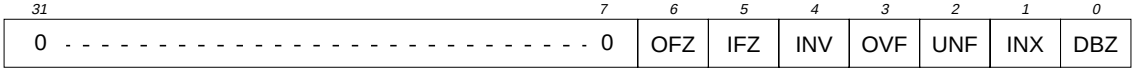
SEE ALSO

fsign readpcsw

DESCRIPTION

The `fsignflags` operation computes the IEEE exceptions that would result from computing the sign of r*src1* and stores a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If the argument is denormalized, zero is substituted before computing the sign, and the IFZ bit in the result is set.

The `fsignflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fsignflags r30 → r100`	r100 ← 0
r40 = 0xbf800000 (-1.0)	`fsignflags r40 → r105`	r105 ← 0
r50 = 0x80800000 (-1.175494351e-38)	`fsignflags r50 → r110`	r110 ← 0
r60 = 0x80400000 (-5.877471754e-39)	`fsignflags r60 → r115`	r115 ← 0x20 (IFZ)
r10 = 0, r70 = 0xffffffff (QNaN)	`IF r10 fsignflags r70 → r116`	no change, since guard is false
r20 = 1, r70 = 0xffffffff (QNaN)	`IF r20 fsignflags r70 → r117`	r117 ← 0x10 (INV)
r80 = 0xff800000 (-INF)	`fsignflags r80 → r120`	r120 ← 0

Floating-point square root

fsqrt

SYNTAX

```
[ IF rguard ] fsqrt rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest* ← square_root(r*src1*)

ATTRIBUTES

Function unit	ftough
Operation code	110
Number of operands	1
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

SEE ALSO

`fsqrtflags` `readpcsw` `writepcsw`

DESCRIPTION

The `fsqrt` operation computes the squareroot of r*src1* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the squareroot, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fsqrt` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fsqrtflags` operation computes the exception flags that would result from an individual `fsqrt`.

The `fsqrt` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0)	`fsqrt r60 → r90`	r90 ← 0xffffffff (QNaN), INV flag set
r40 = 0x40400000 (3.0)	`fsqrt r40 → r95`	r95 ← 0x3fddb3d7 (1.732051), INX flag set
r10 = 0, r40 = 0x40400000 (3.0)	`IF r10 fsqrt r40 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0)	`IF r20 fsqrt r40 → r110`	r110 ← 0x3fddb3d7 (1.732051), INX flag set
r82 = 0x00c00000 (1.763241526e–38)	`fsqrt r82 → r112`	r112 ← 0x201cc471 (1.32787105e-19), INX flag set
r84 = 0x7f800000 (+INF)	`fsqrt r84 → r113`	r113 ← 0x7f800000 (+INF)
r70 = 0x7f7fffff (3.402823466e+38)	`fsqrt r70 → r120`	r120 ← 0x5f7fffff (1.8446743e19), INX flag set
r80 = 0x00400000 (5.877471754e-39)	`fsqrt r80 → r125`	r125 ← 0, IFZ flag set

fsqrtflags IEEE status flags from floating-point square root

SYNTAX

```
[ IF rguard ] fsqrtflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← ieee_flags(square_root((float)r*src1*))

ATTRIBUTES

Function unit	ftough
Operation code	111
Number of operands	1
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

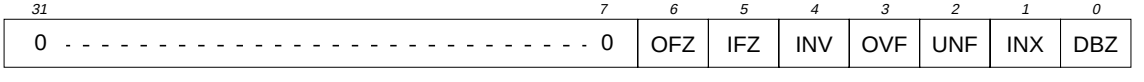
SEE ALSO

fsqrt readpcsw

DESCRIPTION

The `fsqrtflags` operation computes the IEEE exceptions that would result from computing the squareroot of r*src1* and stores a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If the argument is denormalized, zero is substituted before computing the squareroot, and the IFZ bit in the result is set. If the result is denormalized, and the OFZ flag in the PCSW is set.

The `fsqrtflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0)	`fsqrtflags r60 → r90`	r90 ← 0x10 (INV)
r40 = 0x40400000 (3.0)	`fsqrtflags r40 → r95`	r95 ← 0x2 (INX)
r10 = 0, r40 = 0x40400000 (3.0)	`IF r10 fsqrtflags r40 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0)	`IF r20 fsqrtflags r40 → r110`	r110 ← 0x2 (INX)
r82 = 0x00c00000 (1.763241526e–38)	`fsqrtflags r82 → r112`	r112 ← 0x2 (INX)
r84 = 0x7f800000 (+INF)	`fsqrtflags r84 → r113`	r113 ← 0
r70 = 0x7f7fffff (3.402823466e+38)	`fsqrtflags r70 → r120`	r120 ← 0x2 (INX)
r80 = 0x00400000 (5.877471754e-39)	`fsqrtflags r80 → r125`	r125 ← 0x20 (IFZ)

Floating-point subtract

fsub

SYNTAX

```
[ IF rguard ] fsub rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← (float)r*src1* – (float)r*src2*

ATTRIBUTES

Function unit	falu
Operation code	113
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

fsubflags isub dspisub dspidualsub readpcsw writepcsw

DESCRIPTION

The `fsub` operation computes the difference r*src1*–r*src2* and writes the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the difference, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fsub` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fsubflags` operation computes the exception flags that would result from an individual `fsub`.

The `fsub` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fsub r60 r30 → r90`	r90 ← 0xc0800000 (-4.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fsub r40 r60 → r95`	r95 ← 0x40c00000 (6.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r10 fsub r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r20 fsub r40 r80 → r110`	r110 ← 0x40400000 (3.0), INX flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fsub r40 r81 → r111`	r111 ← 0x40400000 (3.0), IFZ flag set
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x0080000 (1.175494351e-38)	`fsub r82 r83 → r112`	r112 ← 0x0, OFZ flag set
r84 = 0x7f800000 (+INF), r85 = 0x7f800000 (+INF)	`fsub r84 r85 → r113`	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38) r86 = 0xff7fffff (-3.402823466e+38)	`fsub r70 r86 → r120`	r120 ← 0x7f800000 (+INF), OVF, INX flag set
r87 = 0xffffffff (QNaN)) r30 = 0x3f800000 (1.0	`fsub r87 r30 → r125`	r125 ← 0xffffffff (QNaN)
r87 = 0xffbfffff (SNaN)) r30 = 0x3f800000 (1.0	`fsub r87 r30 → r125`	r125 ← 0xffffffff (QNaN), INV flag set
r83 = 0x0080001 (1.175494421e-38), r89 = 0x0080000 (1.175494351e-38)	`fsub r83 r89 → r126`	r126 ← 0x0, UNF flag set

fsubflags

IEEE status flags from floating-point subtract

SYNTAX

```
[ IF rguard ] fsubflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← ieee_flags((float)r*src1* – (float)r*src2*)

ATTRIBUTES

Function unit	falu
Operation code	114
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

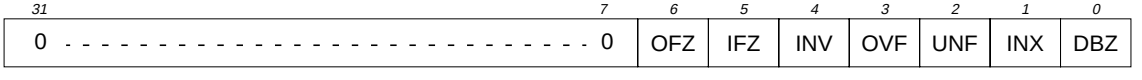
SEE ALSO

`fsub faddflags readpcsw`

DESCRIPTION

The `fsubflags` operation computes the IEEE exceptions that would result from computing the difference r*src1*–r*src2* and writes a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the difference, and the IFZ bit in the result is set. If the difference would be denormalized, the OFZ bit in the result is set.

The `fsubflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fsubflags r60 r30 → r90`	r90 ← 0
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fsubflags r40 r60 → r95`	r95 ← 0
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r10 fsubflags r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r20 fsubflags r40 r80 → r110`	r110 ← 0x2 (INX)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fsubflags r40 r81 → r111`	r111 ← 0x20 (IFZ)
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x0080000 (1.175494351e-38)	`fsubflags r82 r83 → r112`	r112 ← 0x40 (OFZ)
r84 = 0x7f800000 (+INF), r85 = 0x7f800000 (+INF)	`fsubflags r84 r85 → r113`	r113 ← 0x10 (INV)
r70 = 0x7f7fffff (3.402823466e+38) r86 = 0xff7fffff (-3.402823466e+38)	`fsubflags r70 r86 → r120`	r120 ← 0xA (OVF,INX)
r87 = 0xffffffff (QNaN)) r30 = 0x3f800000 (1.0	`fsubflags r87 r30 → r125`	r125 ← 0x0
r87 = 0xffbfffff (SNaN)) r30 = 0x3f800000 (1.0	`fsubflags r87 r30 → r125`	r125 ← 0x10 (INV)
r83 = 0x0080001 (1.175494421e-38), r89 = 0x0080000 (1.175494351e-38)	`fsubflags r83 r89 → r126`	r126 ← 0x4 (UNF)

Funnel-shift 1byte

funshift1

SYNTAX

```
[ IF rguard ] funshift1 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest*<31:8> ← r*src1*<23:0>
    r*dest*<7:0> ← r*src2*<31:24>

ATTRIBUTES

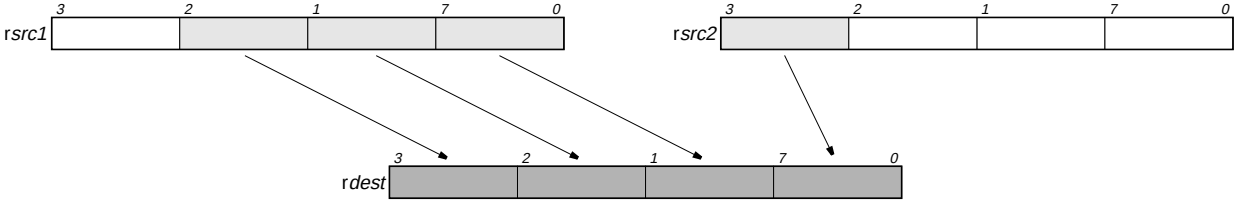
Function unit	shifter
Operation code	99
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

`funshift2 funshift3 rol`

DESCRIPTION

As shown below, the `funshift1` operation effectively shifts left by one byte the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift1` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift1 r30 r40 → r50`	r50 ← 0xbbccdd11
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift1 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift1 r40 r30 → r70`	r70 ← 0x223344aa

funshift2

Funnel-shift 2 bytes

SYNTAX

```
[ IF rguard ] funshift2 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest*<31:16> ← r*src1*<15:0>
r*dest*<15:0> ← r*src2*<31:16>

ATTRIBUTES

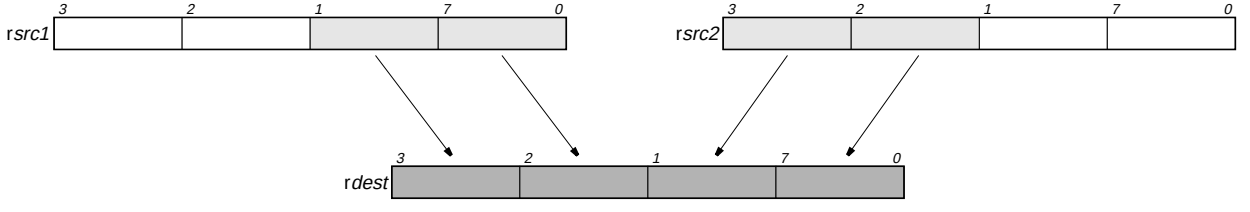
Function unit	shifter
Operation code	100
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift1 funshift3 rol

DESCRIPTION

As shown below, the `funshift2` operation effectively shifts left by two bytes the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift2` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift2 r30 r40 → r50`	r50 ← 0xccdd1122
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift2 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift2 r40 r30 → r70`	r70 ← 0x3344aabb

Funnel-shift 3 bytes

funshift3

SYNTAX

```
[ IF rguard ] funshift3 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest*<31:24> ← r*src1*<7:0>

r*dest*<23:0> ← r*src2*<31:8>

ATTRIBUTES

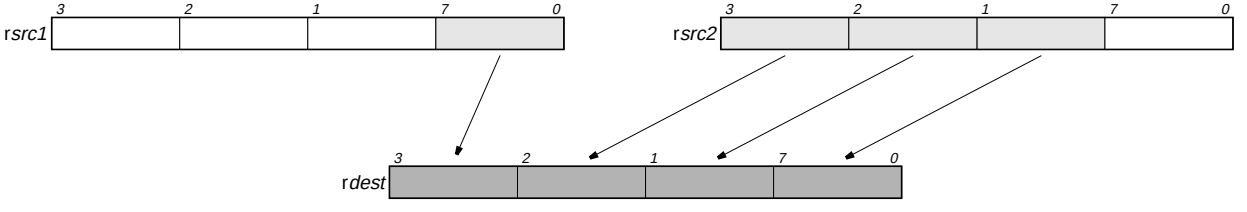
Function unit	shifter
Operation code	101
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

`funshift1 funshift2 rol`

DESCRIPTION

As shown below, the `funshift3` operation effectively shifts left by three bytes the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift3` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift3 r30 r40 → r50`	r50 ← 0xdd112233
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift3 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift3 r40 r30 → r70`	r70 ← 0x44aabbcc

h_dspiabs

Clipped signed absolute value

SYNTAX

```
[ IF rguard ] h_dspiabs r0 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc2 >= 0 then
      rdest ← rsrc2
   else if rsrc2 = 0x80000000 then
      rdest ← 0x7fffffff
   else
      rdest ← –rsrc2
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	65
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

h_dspiabs dspidualabs dspiadd dspimul dspisub dspuadd dspumul dspusub

DESCRIPTION

The h_dspiabs operation computes the absolute value of rsrc2, clips the result into the range [0x0..0x7fffffff], and stores the clipped value into r*dest*. All values are signed integers. This operation requires a zero as first argument. The programmer is advised to use the unary pseudo operation dspiabs instead.

The h_dspiabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	h_dspiabs r0 r30 → r60	r60 ← 0x00000001
r10 = 0, r40 = 0x80000001	IF r10 h_dspiabs r0 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0x80000001	IF r20 h_dspiabs r0 r40 → r100	r100 ← 0x7fffffff
r50 = 0x80000000	h_dspiabs r0 r50 → r80	r80 ← 0x7fffffff
r90 = 0x7fffffff	h_dspiabs r0 r90 → r110	r110 ← 0x7fffffff

Dual clipped absolute value of signed 16-bit halfwords

h_dspidualabs

SYNTAX

```
[ IF rguard ] h_dspidualabs r0 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   temp1 ← sign_ext16to32(rsrc2<15:0>)
   temp2 ← sign_ext16to32(rsrc2<31:16>)
   if temp1 = 0xffff8000 then temp1 ← 0x7fff
   if temp2 = 0xffff8000 then temp2 ← 0x7fff
   if temp1 < 0 then temp1 ← –temp1
   if temp2 < 0 then temp2 ← –temp2
   rdest<31:16> ← temp2<15:0>
   rdest<15:0> ← temp1<15:0>
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	72
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualabs dspiabs dspidualadd dspidualmul dspidualsub dspiabs

DESCRIPTION

The h_dspidualabs operation performs two 16-bit clipped, signed absolute value computations separately on the high and low 16-bit halfwords of rsrc2. Both absolute values are clipped into the range [0x0..0x7fff] and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers. This operation requires a zero as first argument. The programmer is advised to use the dspidualabs pseudo operation instead.

The h_dspidualabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0032	h_dspidualabs r0 r30 → r60	r60 ← 0x00010032
r10 = 0, r40 = 0x80008001	IF r10 h_dspidualabs r0 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0x80008001	IF r20 h_dspidualabs r0 r40 → r100	r100 ← 0x7fff7fff
r50 = 0x0032ffff	h_dspidualabs r0 r50 → r80	r80 ← 0x00320001
r90 = 0x7fffffff	h_dspidualabs r0 r90 → r110	r110 ← 0x7fff0001

h_iabs

Hardware absolute value

SYNTAX

```
[ IF rguard ] h_iabs r0 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc2 < 0 then
        rdest ← –rsrc2
    else
        rdest ← rsrc2
}
```

ATTRIBUTES

Function unit	alu
Operation code	44
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iabs fabsval

DESCRIPTION

The `h_iabs` operation computes the absolute value of rsrc2 and stores the result into r*dest*. The argument is a signed integer; the result is an unsigned integer. This operation requires a zero as first argument. The programmer is advised to use the `iabs` pseudo operation instead.

The `h_iabs` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`h_iabs r0 r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	`IF r10 h_iabs r0 r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	`IF r20 h_iabs r0 r40 → r90`	r90 ← 0xc
r50 = 0x80000001	`h_iabs r0 r50 → r100`	r100 ← 0x7fffffff
r60 = 0x80000000	`h_iabs r0 r60 → r110`	r110 ← 0x80000000
r20 = 1	`h_iabs r0 r20 → r120`	r120 ← 1

Hardware 16-bit store with displacement

h_st16d

SYNTAX

```
[ IF rguard ] h_st16d(d) rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 1
  else
    bs ← 0
  mem[r*src2* + *d* + (1 ⊕ bs)] ← r*src1*<7:0>
  mem[r*src2* + *d* + (0 ⊕ bs)] ← r*src1*<15:8>
}

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	n/a
Issue slots	4, 5

SEE ALSO

st16 st16d st8 st8d st32 st32d readpcsw ijmpf

DESCRIPTION

The h_st16d operation stores the least-significant 16-bit halfword of r*src1* into the memory locations pointed to by the address in r*src2* + *d*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If h_st16d is misaligned (the memory address computed by r*src2* + *d* is not a multiple of 2), the result of h_st16d is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The h_st16d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st16d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfe, r80 = 0x44332211	`h_st16d(2) r80 r10`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd05, r70 = 0xaabbccdd	`IF r50 h_st16d(-4) r70 r20`	no change, since guard is false
r60 = 1, r30 = 0xd06, r70 = 0xaabbccdd	`IF r60 h_st16d(-4) r70 r30`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

h_st32d

Hardware 32-bit store with displacement

SYNTAX

```
[ IF rguard ] h_st32d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   mem[rsrc2 + d + (3 ⊕ bs)] ← rsrc1<7:0>
   mem[rsrc2 + d + (2 ⊕ bs)] ← rsrc1<15:8>
   mem[rsrc2 + d + (1 ⊕ bs)] ← rsrc1<24:16>
   mem[rsrc2 + d + (0 ⊕ bs)] ← rsrc1<31:24>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	31
Number of operands	2
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	n/a
Issue slots	4, 5

SEE ALSO

st32 st32d st16 st16d st8 st8d readpcsw ijmpf

DESCRIPTION

The h_st32d operation stores all 32 bits of r*src1* into the memory locations pointed to by the address in r*src2* + *d*. The *d* value is an opcode modifier, must be in the range –256 and 252 inclusive, and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If h_st32d is misaligned (the memory address computed by r*src2* + *d* is not a multiple of 4), the result of h_st32d is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The h_st32d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st32d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r80 = 0x44332211	`h_st32d(4) r80 r10`	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd0b, r70 = 0xaabbccdd	`IF r50 h_st32d(-8) r70 r20`	no change, since guard is false
r60 = 1, r30 = 0xd0c, r70 = 0xaabbccdd	`IF r60 h_st32d(-8) r70 r30`	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

Hardware 8-bit store with displacement

h_st8d

SYNTAX

```
[ IF rguard ] h_st8d(d) rsrc1 rsrc2
```

FUNCTION

if r*guard* **then**

mem[r*src2* + *d*] ← r*src1*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	7 bits
Modifier range	–64..63
Latency	n/a
Issue slots	4, 5

SEE ALSO

st8 st8d st16 st16d st32 st32d

DESCRIPTION

The h_st8d operation stores the least-significant 8-bit byte of r*src1* into the memory location pointed to by the address formed from the sum r*src2* + *d*. The value of the opcode modifier *d* must be in the range -64 and 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The h_st8d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st8d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`h_st8d(3) r80 r10`	[0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 h_st8d(-4) r70 r20`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 h_st8d(-4) r70 r30`	[0xcfe] ← 0xdd

hicycles

Read clock cycle counter, most-significant word

SYNTAX

```
[ IF rguard ] hicycles → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← CCCOUNT<63:32>

ATTRIBUTES

Function unit	fcomp
Operation code	155
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`cycles` `curcycles` `writepcsw`

DESCRIPTION

Refer to Section 3.1.6, “CCCOUNT—Clock Cycle Counter” for a description of the CCCOUNT operation. The `hicycles` operation copies the high 32 bits of the slave register Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*. The contents of the master counter are transferred to the slave CCCOUNT register only on a successful interruptible jump and on processor reset. Thus, if `cycles` and `hicycles` are executed without intervening interruptible jumps, the operation pair is guaranteed to be a coherent sample of the master clock-cycle counter. The master counter increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The h`icycles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	`hicycles → r60`	r60 ← 0xabcdefff
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 hicycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 hicycles → r100`	r100 ← 0xabcdefff

Absolute value

pseudo-op for h_iabs

iabs

SYNTAX

```
[ IF rguard ] iabs rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 < 0 then
      rdest ← –rsrc1
   else
      rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	alu
Operation code	44
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

h_iabs dspiabs dspidualabs fabsval

DESCRIPTION

The `iabs` operation is a pseudo operation transformed by the scheduler into an `h_iabs` with zero as the first argument and a second argument equal to the `iabs` argument. (Note: pseudo operations cannot be used in assembly source files.)

The `iabs` operation computes the absolute value of r*src1* and stores the result into r*dest*. The argument is a signed integer; the result is an unsigned integer.

The `iabs` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`iabs r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	`IF r10 iabs r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	`IF r20 iabs r40 → r90`	r90 ← 0xc
r50 = 0x80000001	`iabs r50 → r100`	r100 ← 0x7fffffff
r60 = 0x80000000	`iabs r60 → r110`	r110 ← 0x80000000
r20 = 1	`iabs r20 → r120`	r120 ← 1

iadd

Signed add

SYNTAX

```
[ IF rguard ] iadd rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* + r*src2*

ATTRIBUTES

Function unit	alu
Operation code	12
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`iaddi carry dspiadd dspidualadd fadd`

DESCRIPTION

The `iadd` operation computes the sum r*src1*+r*src2* and stores the result into r*dest*. The operands can be either both signed or unsigned integers. No overflow or underflow detection is performed.

The `iadd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	`iadd r60 r60 → r80`	r80 ← 0x200
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 iadd r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	`IF r20 iadd r60 r30 → r90`	r90 ← 0x1011
r70 = 0xffffff00, r40 = 0xffffff9c	`iadd r70 r40 → r100`	r100 ← 0xfffffe9c

Add with immediate

iaddi

SYNTAX

```
[ IF rguard ] iaddi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest* ← r*src1* + *n*

ATTRIBUTES

Function unit	alu
Operation code	5
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iadd carry

DESCRIPTION

The `iaddi` operation sums a single argument in r*src1* and an immediate modifier *n* and stores the result in r*dest*. The value of *n* must be between 0 and 127, inclusive.

The `iaddi` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf11	`iaddi(127) r30 → r70`	r70 ← 0xf90
r10 = 0, r40 = 0xffffff9c	`IF r10 iaddi(1) r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xffffff9c	`IF r20 iaddi(1) r40 → r90`	r90 ← 0xffffff9d
r50 = 0x1000	`iaddi(15) r50 → r120`	r120 ← 0x100f
r60 = 0xfffffff0	`iaddi(2) r60 → r110`	r110 ← 0xfffffff2
r60 = 0xfffffff0	`iaddi(17) r60 → r120`	r120 ← 1

iavgonep

Signed average

SYNTAX

```
[ IF rguard ] iavgonep rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← (sign_ext32to64(r*src1*) + sign_ext32to64(r*src2*) + 1) >> 1;

ATTRIBUTES

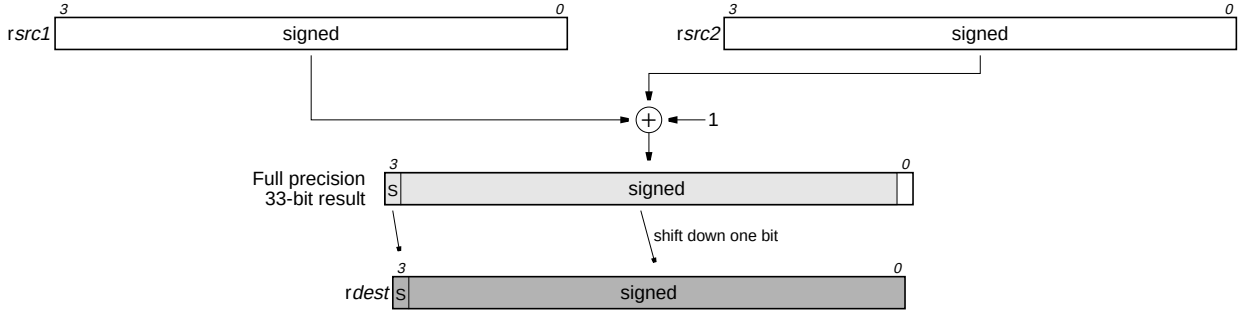
Function unit	dspalu
Operation code	25
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

quadavg iadd

DESCRIPTION

As shown below, the `iavgonep` operation returns the average of the two arguments. This operation computes the sum r*src1*+r*src2*+1, shifts the sum right by 1 bit, and stores the result into r*dest*. The operands are signed integers.



The `iavgonep` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10, r70 = 0x20	`iavgonep r60 r70 → r80`	r80 ← 0x18
r10 = 0, r60 = 0x10, r30 = 0x20	`IF r10 iavgonep r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x9, r30 = 0x20	`IF r20 iavgonep r60 r30 → r90`	r90 ← 0x15
r70 = 0xfffffff7, r40 = 0x2	`iavgonep r70 r40 → r100`	r100 ← 0xfffffffd
r70 = 0xfffffff7, r40 = 0x3	`iavgonep r70 r40 → r100`	r100 ← 0xfffffffd

Signed select byte

ibytesel

SYNTAX

```
[ IF rguard ] ibytesel rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	alu
Operation code	56
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

FUNCTION

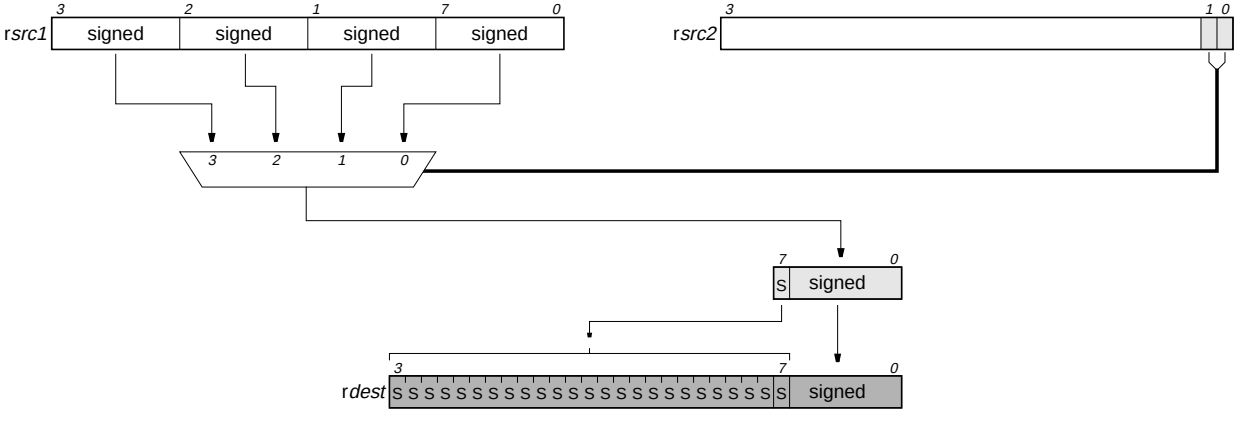
```
if rguard then {
   if rsrc2 = 0 then
      rdest ← sign_ext8to32(rsrc1<7:0>)
   else if rsrc2 = 1 then
      rdest ← sign_ext8to32(rsrc1<15:8>)
   else if rsrc2 = 2 then
      rdest ← sign_ext8to32(rsrc1<23:16>)
   else if rsrc2 = 3 then
      rdest ← sign_ext8to32(rsrc1<31:24>)
}
```

SEE ALSO

ubytesel sex8 packbytes

DESCRIPTION

As shown below, the `ibytesel` operation selects one byte from the argument, r*src1*, sign-extends the byte to 32 bits, and stores the result in r*dest*. The value of r*src2* determines which byte is selected, with r*src2*=0 selecting the LSB of r*src1* and r*src2*=3 selecting the MSB of r*src1*. If rsrc2 is not between 0 and 3 inclusive, the result of `ibytesel` is undefined.



The `ibytesel` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x44332211, r40 = 1	`ibytesel r30 r40 → r50`	r50 ← 0x00000022
r10 = 0, r60 = 0xddccbbaa, r70 = 2	`IF r10 ibytesel r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xddccbbaa, r70 = 2	`IF r20 ibytesel r60 r70 → r90`	r90 ← 0xffffffcc
r100 = 0xffffff7f, r110 = 0	`ibytesel r100 r110 → r120`	r120 ← 0x0000007f

iclipi

Clip signed to signed

SYNTAX

```
[ IF rguard ] iclipi rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← min(max(r*src1*, –r*src2*–1), r*src2*)

ATTRIBUTES

Function unit	dspalu
Operation code	74
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

uclipi uclipu imin imax

DESCRIPTION

The `iclipi` operation returns the value of r*src1* clipped into the unsigned integer range (–r*src2*–1) to r*src2*, inclusive. The argument r*src1* is considered a signed integer; rsrc2 is considered an unsigned integer and must have a value between 0 and 0x7fffffff inclusive.

The `iclipi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`iclipi r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 iclipi r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 iclipi r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`iclipi r100 r110 → r120`	r120 ← 0xffc00000

Invalidate all instruction cache blocks

iclr

SYNTAX

```
[ IF rguard ] iclr
```

FUNCTION

```
if rguard then {
    block ← 0
    for all blocks in instruction cache {
        icache_reset_valid_block(block)
        block ← block + 1
    }
}
```

ATTRIBUTES

Function unit	branch
Operation code	184
Number of operands	0
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	2, 3, 4

SEE ALSO

dcb dinvalid

DESCRIPTION

The `iclr` operation resets the valid bits of all blocks in the instruction cache.

`iclr` does clear the valid bits of locked blocks. `iclr` does not change the replacement status of instruction-cache blocks.

`iclr` ensures coherency between caches and main memory by discarding all pending prefetch operations.

The side effect time behavior of `iclr` is such that if instruction *i* performs an iclr, instructions *i, i+1, i+2* will be included in the discard from the instruction cache, but *i+3* will be retained.

The `iclr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`iclr`	
r10 = 0	`IF r10 iclr`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 iclr`	

ident

Identity

pseudo-op for iadd

SYNTAX

```
[ IF rguard ] ident rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← r*src1*

ATTRIBUTES

Function unit	alu
Operation code	12
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iadd

DESCRIPTION

The `ident` operation is a pseudo operation transformed by the scheduler into an `iadd` with r0 (always contains 0) as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

The `ident` operation copies the argument r*src1* to r*dest*. It is used by the instruction scheduler to implement register to register copying.

The `ident` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x100	`ident r30 → r40`	r40 ← 0x100
r10 = 0, r50 = 0x12345678	`IF r10 ident r50 → r60`	no change, since guard is false
r20 = 1, r50 = 0x12345678	`IF r20 ident r50 → r70`	r70 ← 0x12345678

Signed compare equal

ieql

SYNTAX

```
[ IF rguard ] ieql rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 = rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	37
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq ueql ieqli ineq

DESCRIPTION

The `ieql` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ieql` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ieql r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ieql r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 ieql r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ieql r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`ieql r70 r70 → r110`	r110 ← 1

ieqli

Signed compare equal with immediate

SYNTAX

```
[ IF rguard ] ieqli(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 = n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	4
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieql igeqi ueqli ineqi

DESCRIPTION

The `ieqli` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ieqli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ieqli(2) r30 → r80`	r80 ← 0
r30 = 3	`ieqli(3) r30 → r90`	r90 ← 1
r30 = 3	`ieqli(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ieqli(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ieqli(63) r40 → r100`	r100 ← 0
r60 = 0xffffffc0	`ieqli(-64) r60 → r120`	r120 ← 1

Sum of products of signed 16-bit halfwords

ifir16

SYNTAX

```
[ IF rguard ] ifir16 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← sign_ext16to32(r*src1*<31:16>) × sign_ext16to32(r*src2*<31:16>) +
sign_ext16to32(r*src1*<15:0>) × sign_ext16to32(r*src2*<15:0>)

ATTRIBUTES

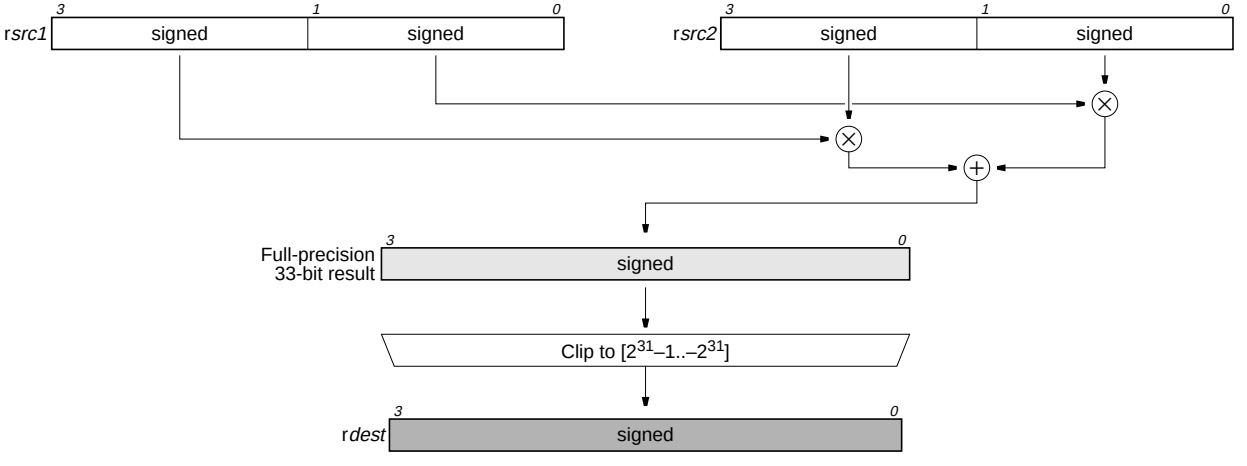
Function unit	dspmul
Operation code	93
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ii ifir8ui ufir8uu
ifir16

DESCRIPTION

As shown below, the `ifir16` operation computes two separate products of the two pairs of corresponding 16-bit halfwords of r*src1* and r*src2*; the two products are summed, and the result is written to r*dest*. All values are considered signed; thus, the intermediate products and the final sum of products are signed. All intermediate computations are performed without loss of precision; the final sum of products is clipped into the range [0x80000000..0x7fffffff] before being written into r*dest*.



The `ifir16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00020003, r40 = 0x00010002	`ifir16 r30 r40 → r50`	r50 ← 0x8
r10 = 0, r60 = 0xff9c0064, r70 = 0x0064ff9c	`IF r10 ifir16 r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xff9c0064, r70 = 0x0064ff9c	`IF r20 ifir16 r60 r70 → r90`	r90 ← 0xffffb1e0
r30 = 0x00020003, r70 = 0x0064ff9c	`ifir16 r30 r70 → r100`	r100 ← 0xffffff9c

ifir8ii — Signed sum of products of signed bytes

SYNTAX

```
[ IF rguard ] ifir8ii rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← sign_ext8to32(r*src1*<31:24>) × sign_ext8to32(r*src2*<31:24>) +
sign_ext8to32(r*src1*<23:16>) × sign_ext8to32(r*src2*<23:16>) +
sign_ext8to32(r*src1*<15:8>) × sign_ext8to32(r*src2*<15:8>) +
sign_ext8to32(r*src1*<7:0>) × sign_ext8to32(r*src2*<7:0>)

ATTRIBUTES

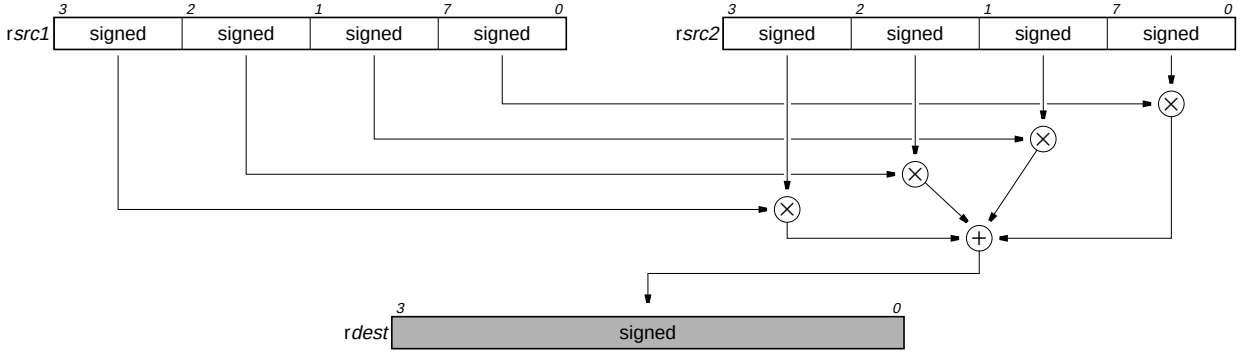
Function unit	dspmul
Operation code	92
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ui ufir8uu ifir16 ufir16

DESCRIPTION

As shown below, the `ifir8ii` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. All values are considered signed; thus, the intermediate products and the final sum of products are signed. All computations are performed without loss of precision.



The `ifir8ii` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ifir8ii r70 r30 → r90`	r90 ← 0xfa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ifir8ii r70 r30 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ifir8ii r80 r40 → r110`	r110 ← 0xffff63c0
r50 = 0x80808080, r60 = 0xffffffff	`ifir8ii r50 r60 → r120`	r120 ← 0x200

Signed sum of products of unsigned/signed bytes

ifir8ui

SYNTAX

```
[ IF rguard ] ifir8ui rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspmul
Operation code	91
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

FUNCTION

if r*guard* **then**

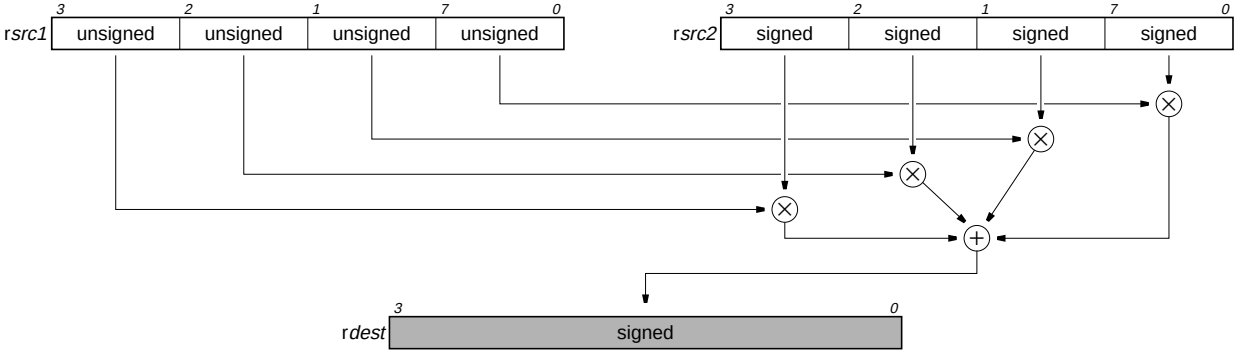
r*dest* ← zero_ext8to32(r*src1*<31:24>) × sign_ext8to32(r*src2*<31:24>) +
zero_ext8to32(r*src1*<23:16>) × sign_ext8to32(r*src2*<23:16>) +
zero_ext8to32(r*src1*<15:8>) × sign_ext8to32(r*src2*<15:8>) +
zero_ext8to32(r*src1*<7:0>) × sign_ext8to32(r*src2*<7:0>)

SEE ALSO

ifir8ii ufir8uu ifir16 ufir16

DESCRIPTION

As shown below, the `ifir8ui` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. The bytes from r*src1* are considered unsigned, but the bytes from r*src2* are considered signed; thus, the intermediate products and the final sum of products are signed. All computations are performed without loss of precision.



The `ifir8ui` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ifir8ui r30 r70 → r90`	r90 ← 0xfa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ifir8ui r30 r70 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ifir8ui r40 r80 → r110`	r110 ← 0x2bc0
r50 = 0x80808080, r60 = 0xffffffff	`ifir8ui r60 r50 → r120`	r120 ← 0xfffe0200

ifixieee

Convert floating-point to integer using PCSW rounding mode

SYNTAX

```
[ IF rguard ] ifixieee rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← (long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	121
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ufixieee ifixrz ufixrz

DESCRIPTION

The `ifixieee` operation converts the single-precision IEEE floating-point value in r*src1* to a signed integer and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ifixieee` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifixieeeflags` operation computes the exception flags that would result from an individual `ifixieee`.

The `ifixieee` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixieee r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ifixieee r35 → r102`	r102 ← 3, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixieee r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixieee r40 → r110`	r110 ← 0x80000000 (-2^{31}), INV flag set
r45 = 0x7f800000 (+INF))	`ifixieee r45 → r112`	r112 ← 0x7fffffff (2^{31}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ifixieee r50 → r115`	r115 ← -2, INX flag set
r60 = 0x00400000 (5.877471754e-39)	`ifixieee r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ifixieee r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ifixieee r80 → r122`	r122 ← 0, INV flag set

IEEE status flags from convert floating-point to integer using PCSW rounding mode

ifixieeeflags

SYNTAX

```
[ IF rguard ] ifixieeeflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	122
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

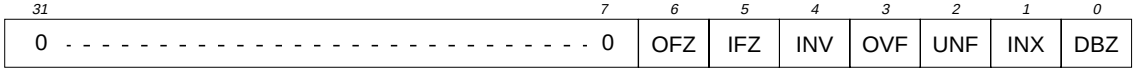
SEE ALSO

ifixieee ufixieeeflags
ifixrzflags ufixrzflags

DESCRIPTION

The `ifixieeeflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to a signed integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ifixieeeflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixieeeflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ifixieeeflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixieeeflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixieeeflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ifixieeeflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ifixieeeflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x00400000 (5.877471754e-39)	`ifixieeeflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ifixieeeflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ifixieeeflags r80 → r122`	r122 ← 0x10 (INV)

ifixrz

Convert floating-point to integer with round toward zero

SYNTAX

```
[ IF rguard ] ifixrz rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← (long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	21
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifixieee ufixieee ufixrz

DESCRIPTION

The `ifixrz` operation converts the single-precision IEEE floating-point value in r*src1* to a signed integer and writes the result into r*dest*. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding for ANSI C. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ifixrz` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifixrzflags` operation computes the exception flags that would result from an individual `ifixrz`.

The `ifixrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixrz r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ifixrz r35 → r102`	r102 ← 2, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixrz r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixrz r40 → r110`	r110 ← 0x80000000 (-2^{31}), INV flag set
r45 = 0x7f800000 (+INF))	`ifixrz r45 → r112`	r112 ← 0x7fffffff (2^{31}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ifixrz r50 → r115`	r115 ← -1, INX flag set
r60 = 0x00400000 (5.877471754e-39)	`ifixrz r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ifixrz r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ifixrz r80 → r122`	r122 ← 0, INV flag set

IEEE status flags from convert floating-point to integer with round toward zero

ifixrzflags

SYNTAX

```
[ IF rguard ] ifixrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	129
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

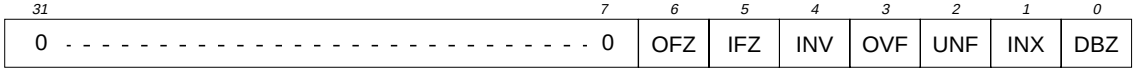
SEE ALSO

ifixrz ufixrzflags
ifixieeeflags
ufixieeeflags

DESCRIPTION

The `ifixrzflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to a signed integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. If r*src1* is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ifixrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixrzflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ifixrzflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixrzflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixrzflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ifixrzflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ifixrzflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x00400000 (5.877471754e-39)	`ifixrzflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ifixrzflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ifixrzflags r80 → r122`	r122 ← 0x10 (INV)

iflip

If non-zero negate

SYNTAX

```
[ IF rguard ] iflip rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 = 0 then
        rdest ← rsrc2
    else
        rdest ← –rsrc2
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	77
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

inonzero izero

DESCRIPTION

The `iflip` operation copies r*src2* to r*dest* if r*src1* = 0; otherwise (if r*src1* != 0), r*dest* is set to the two's-complement of r*src2*. All values are signed integers.

The `iflip` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0, r40 = 1	`iflip r30 r40 → r50`	r50 ← 0x1
r10 = 0, r60 = 0xffff0000, r70 = 0xabc	`IF r10 iflip r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xffff0000, r70 = 0xabc	`IF r20 iflip r60 r70 → r90`	r90 ← 0xffff544
r30 = 0, r100 = 0xffffff9c	`iflip r30 r100 → r110`	r110 ← 0xffffff9c
r40 = 1, r110 = 0xffffffff	`iflip r40 r110 → r120`	r120 ← 0x1

Convert signed integer to floating-point

ifloat

SYNTAX

```
[ IF rguard ] ifloat rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← (float) ((long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	20
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ufloat ifloatrz ufloatrz ifixieee ifloatflags

DESCRIPTION

The `ifloat` operation converts the signed integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If `ifloat` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifloatflags` operation computes the exception flags that would result from an individual `ifloat`.

The `ifloat` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloat r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (-1)	`ifloat r40 → r105`	r105 ← 0xbf800000 (-1.0)
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloat r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloat r50 → r115`	r115 ← 0xc0400000 (–3.0)
r60 = 0x7fffffff (2147483647)	`ifloat r60 → r117`	r117 ← 0x4f000000 (2.147483648e+9), INX flag set
r70 = 0x80000000 (-2147483648)	`ifloat r70 → r120`	r120 ← 0xcf000000 (-2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ifloat r80 → r122`	r122 ← 0x4f000000 (2.147483648e+9), INX flag set

ifloatflags

IEEE status flags from convert signed integer to floating-point

SYNTAX

```
[ IF rguard ] ifloatflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((float) ((long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	130
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

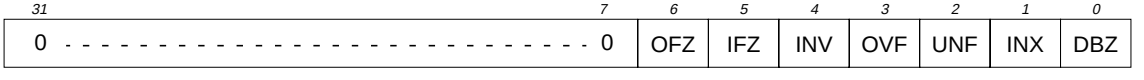
SEE ALSO

ifloat ifloatrzflags
ufloatflags ufloatrzflags

DESCRIPTION

The `ifloatflags` operation computes the IEEE exceptions that would result from converting the signed integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW.

The `ifloatflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (-1)	`ifloatflags r40 → r105`	r105 ← 0
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatflags r50 → r115`	r115 ← 0
r60 = 0x7fffffff (2147483647)	`ifloatflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (-2147483648)	`ifloatflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ifloatflags r80 → r122`	r122 ← 0x02 (INX)

Convert signed integer to floating-point with rounding toward zero

ifloatrz

SYNTAX

```
[ IF rguard ] ifloatrz rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    rdest ← (float) ((long)rsrc1)
}
```

ATTRIBUTES

Function unit	falu
Operation code	117
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifloat ufloatrz ifixieee ifloatflags

DESCRIPTION

The `ifloatrz` operation converts the signed integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If `ifloatrz` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifloatrzflags` operation computes the exception flags that would result from an individual `ifloatrz`.

The `ifloatrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatrz r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (-1)	`ifloatrz r40 → r105`	r105 ← 0xbf800000 (-1.0)
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatrz r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatrz r50 → r115`	r115 ← 0xc0400000 (–3.0)
r60 = 0x7fffffff (2147483647)	`ifloatrz r60 → r117`	r117 ← 0x4effffff (2.147483520e+9), INX flag set
r70 = 0x80000000 (-2147483648)	`ifloatrz r70 → r120`	r120 ← 0xcf000000 (-2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ifloatrz r80 → r122`	r122 ← 0x4effffff (2.147483520e+9), INX flag set

ifloatrzflags

IEEE status flags from convert signed integer to floating-point with rounding toward zero

SYNTAX

```
[ IF rguard ] ifloatrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((float) ((long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	118
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

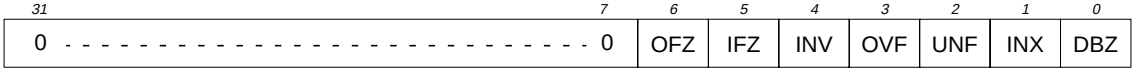
SEE ALSO

ifloatrz ifloatflags ufloatflags ufloatrzflags

DESCRIPTION

The `ifloatrzflags` operation computes the IEEE exceptions that would result from converting the signed integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored.

The `ifloatrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatrzflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (-1)	`ifloatrzflags r40 → r105`	r105 ← 0
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatrzflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatrzflags r50 → r115`	r115 ← 0
r60 = 0x7fffffff (2147483647)	`ifloatrzflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (-2147483648)	`ifloatrzflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ifloatrzflags r80 → r122`	r122 ← 0x02 (INX)

Signed compare greater or equal

igeq

SYNTAX

```
[ IF rguard ] igeq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 >= rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	14
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ileq igeqi

DESCRIPTION

The `igeq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igeq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`igeq r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 igeq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 igeq r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`igeq r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`igeq r70 r70 → r110`	r110 ← 1

igeqi

Signed compare greater or equal with immediate

SYNTAX

```
[ IF rguard ] igeqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 >= n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	1
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq iles ieqli

DESCRIPTION

The `igeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`igeqi(2) r30 → r80`	r80 ← 1
r30 = 3	`igeqi(3) r30 → r90`	r90 ← 1
r30 = 3	`igeqi(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 igeqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 igeqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`igeqi(-64) r60 → r120`	r120 ← 0

Signed compare greater

igtr

SYNTAX

```
[ IF rguard ] igtr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 > rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	15
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iles igtri

DESCRIPTION

The `igtr` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igtr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`igtr r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 igtr r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 igtr r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`igtr r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`igtr r70 r70 → r110`	r110 ← 0

igtri

Signed compare greater with immediate

SYNTAX

```
[ IF rguard ] igtri(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 > n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	0
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr igeqi

DESCRIPTION

The `igtri` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igtri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`igtri(2) r30 → r80`	r80 ← 1
r30 = 3	`igtri(3) r30 → r90`	r90 ← 0
r30 = 3	`igtri(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 igtri(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 igtri(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`igtri(-64) r60 → r120`	r120 ← 0

Signed immediate

iimm

SYNTAX

`iimm(`*n*`) → r`*dest*

FUNCTION

r*dest* ← *n*

ATTRIBUTES

Function unit	const
Operation code	191
Number of operands	0
Modifier	32 bits
Modifier range	0x80000000 ..0x7fffffff
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

uimm

DESCRIPTION

The `iimm` operation stores the signed 32-bit opcode modifier *n* into r*dest*. Note: this operation is not guarded.

EXAMPLES

Initial Values	Operation	Result
	`iimm(2) → r10`	r10 ← 2
	`iimm(0x100) → r20`	r20 ← 0x100
	`iimm(0xfffc0000) → r30`	r30 ← 0xfffc0000

ijmpf

Interruptible indirect jump on false

SYNTAX

```
[ IF rguard ] ijmpf rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
  if (rsrc1 & 1) = 0 then {
    DPC ← rsrc2
    if exception is pending then
      service exception
    elseif interrupt is pending then
      service interrupts
    else
      PC, SPC ← rsrc2
  }
}
```

ATTRIBUTES

Function unit	branch
Operation code	181
Number of operands	2
Modifier	no
Modifier range	—
Delay	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt jmpi ijmpt ijmpi

DESCRIPTION

The `ijmpf` operation conditionally changes the program flow and allows pending interrupts or exceptions to be serviced. If neither interrupts or exceptions are pending and the LSB of r*src1* is 0, the DPC, PC, and SPC registers are set equal to r*src2*. If an interrupt or exception is pending and the LSB of r*src1* is 0, DPC is set equal to r*src2* and the service routine is invoked, where exceptions have priorities over interrupts. If the LSB of r*src1* is 1, program execution continues with the next sequential instruction.

The `ijmpf` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 0, r70 = 0x330	`ijmpf r50 r70`	program execution continues at 0x330 after first servicing pending interrupts
r20 = 1, r70 = 0x330	`ijmpf r20 r70`	since r20 is true, program execution continues with next sequential instruction
r30 = 0, r50 = 0, r60 = 0x8000	`IF r30 ijmpf r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 0, r60 = 0x8000	`IF r40 ijmpf r50 r60`	program execution continues at 0x8000 after first servicing pending interrupts

Interruptible jump immediate

ijmpi

SYNTAX

```
[ IF rguard ] ijmpi(address)
```

FUNCTION

if r*guard* **then {**
  DPC ← *address*
  if exception is pending **then**
    service exception
  else if interrupt is pending **then**
    service interrupts
  else
    PC, SPC ← address
}

ATTRIBUTES

Function unit	branch
Operation code	179
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Delay	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt jmpi ijmpf ijmpt

DESCRIPTION

The `ijmpi` operation changes the program flow and allows pending interrupts or exceptions to be serviced. If no interrupts or exceptions are pending, the DPC, PC, and SPC registers are set equal to *address*. If an exception or interrupts is pending, DPC is set equal to *address* and a service routine is invoked, where exceptions have priorities over interrupts. *address* is an immediate opcode modifier.

The `ijmpi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds a condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified.

EXAMPLES

Initial Values	Operation	Result
	`ijmpi(0x330)`	program execution continues at 0x330
r30 = 0	`IF r30 ijmpi(0x8000)`	since guard is false, program execution continues with next sequential instruction
r40 = 1	`IF r40 ijmpi(0x8000)`	program execution continues at 0x8000

ijmpt

Interruptible indirect jump on true

SYNTAX

```
[ IF rguard ] ijmpt rsrc1 rsrc2
```

ATTRIBUTES

Function unit	branch
Operation code	177
Number of operands	2
Modifier	no
Modifier range	—
Delay	3
Issue slots	2, 3, 4

FUNCTION

```
if rguard then {
  if (rsrc1 & 1) = 1 then {
    DPC ← rsrc2
    if exception is pending then
      service exception
    elseif interrupt is pending then
      service interrupts
    else
      PC, SPC ← rsrc2
  }
}
```

SEE ALSO

jmpf jmpt jmpi ijmpf ijmpi

DESCRIPTION

The `ijmpt` operation conditionally changes the program flow and allows pending interrupts or exceptions to be serviced. If no interrupts or exceptions are pending and the LSB of r*src1* is 1, the DPC, PC, and SPC registers are set equal to r*src2*. If an exception or interrupt is pending and the LSB of r*src1* is 1, DPC is set equal to r*src2* and a service routine is invoked, where exceptions have priority over interrupts. If the LSB of r*src1* is 0, program execution continues with the next sequential instruction.

The `ijmpt` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 1, r70 = 0x330	`ijmpt r50 r70`	program execution continues at 0x330 after first servicing pending interrupts
r20 = 0, r70 = 0x330	`ijmpt r20 r70`	since r20 is false, program execution continues with next sequential instruction
r30 = 0, r50 = 1, r60 = 0x8000	`IF r30 ijmpt r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 1, r60 = 0x8000	`IF r40 ijmpt r50 r60`	program execution continues at 0x8000 after first servicing pending interrupts

Signed 16-bit load

pseudo-op for ild16d(0)

ild16

SYNTAX

```
[ IF rguard ] ild16 rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[(rsrc1 +(1 ⊕ bs)]
   temp<15:8> ← mem[(rsrc1 + (0 ⊕ bs)]
   rdest ← sign_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	6
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild16d ild16r ild16x

DESCRIPTION

The `ild16` operation is a pseudo operation transformed by the scheduler into an `ild16d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ild16` operation loads the 16-bit memory value from the address contained in r*src1*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 2, the result of `ild16` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22, [0xd01] = 0x11	`ild16 r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r30 ild16 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r40 ild16 r20 → r80`	r80 ← 0xffff8433
r50 = 0xd01	`ild16 r50 → r90`	r90 undefined, since 0xd01 is not a multiple of 2

ild16d

Signed 16-bit load with displacement

SYNTAX

```
[ IF rguard ] ild16d(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[(rsrc1 + d + (1 ⊕ bs)]
    temp<15:8> ← mem[(rsrc1 + d + (0 ⊕ bs)]
    rdest ← sign_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	6
Number of operands	1
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	3
Issue slots	4, 5

SEE ALSO

ild16 uld16 uld16d ild16r uld16r ild16x uld16x

DESCRIPTION

The `ild16d` operation loads the 16-bit memory value from the address computed by r*src1* + *d*, sign extends it to 32 bits, and stores the result in r*dest*. The *d* value is an opcode modifier, must be in the range –128 to 126 inclusive, and must be a multiple of 2. If the memory address computed by r*src1* + *d* is not a multiple of 2, the result of `ild16d` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22, [0xd03] = 0x11	`ild16d(2) r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r30 ild16d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r40 ild16d(-4) r20 → r80`	r80 ← 0xffff8433
r50 = 0xd01	`ild16d(-4) r50 → r90`	r90 undefined, since 0xd01 +(–4) is not a multiple of 2

Signed 16-bit load with index

ild16r

SYNTAX

```
[ IF rguard ] ild16r rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[(rsrc1 + rsrc2 +(1 ⊕ bs)]
   temp<15:8> ← mem[(rsrc1 + rsrc2 + (0 ⊕ bs)]
   rdest ← sign_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	195
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild16 uld16 ild16d uld16d uld16r ild16x uld16x

DESCRIPTION

The `ild16r` operation loads the 16-bit memory value from the address computed by r*src1* + r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 2, the result of `ild16r` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22, [0xd03] = 0x11	`ild16r r10 r20 → r80`	r80 ← 0x00002211
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 ild16r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 ild16r r40 r30 → r100`	r100 ← 0xffff8433
r70 = 0xd01, r30 = 0xfffffffc	`ild16r r70 r30 → r110`	r110 undefined, since 0xd01 +(–4) is not a multiple of 2

ild16x

Signed 16-bit load with scaled index

SYNTAX

```
[ IF rguard ] ild16x rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[(rsrc1 + (2 × rsrc2) + (1 ⊕ bs)]
   temp<15:8> ← mem[(rsrc1 + (2 × rsrc2) + (0 ⊕ bs)]
   rdest ← sign_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	196
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild16 uld16 ild16d uld16d ild16r uld16r uld16x

DESCRIPTION

The ild16x operation loads the 16-bit memory value from the address computed by r*src1* + 2×r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address computed by r*src1* + 2×r*src2* is not a multiple of 2, the result of ild16x is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by ild16x to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The ild16x operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ild16x has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r30 = 1, [0xd02] = 0x22, [0xd03] = 0x11	`ild16x r10 r30 → r100`	r100 ← 0x00002211
r50 = 0, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 ild16x r40 r20 → r80`	no change, since guard is false
r60 = 1, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 ild16x r40 r20 → r90`	r90 ← 0xffff8433
r70 = 0xd01, r30 = 1	`ild16x r70 r30 → r110`	r110 undefined, since 0xd01 + 2×1 is not a multiple of 2

Signed 8-bit load

pseudo-op for ild8d(0)

ild8

SYNTAX

```
[ IF rguard ] ild8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← sign_ext8to32(mem[r*src1*])

ATTRIBUTES

Function unit	dmem
Operation code	192
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld8 ild8d uld8d ild8r uld8r

DESCRIPTION

The `ild8` operation is a pseudo operation transformed by the scheduler into an `ild8d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ild8` operation loads the 8-bit memory value from the address contained in r*src1*, sign extends it to 32 bits, and stores the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22	`ild8 r10 → r60`	r60 ← 0x00000022
r30 = 0, r20 = 0xd04, [0xd04] = 0x84	`IF r30 ild8 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84	`IF r40 ild8 r20 → r80`	r80 ← 0xffffff84
r50 = 0xd01, [0xd01] = 0x33	`ild8 r50 → r90`	r90 ← 0x00000033

ild8d

Signed 8-bit load with displacement

SYNTAX

```
[ IF rguard ] ild8d(d) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← sign_ext8to32(mem[r*src1* + *d*])

ATTRIBUTES

Function unit	dmem
Operation code	192
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	3
Issue slots	4, 5

SEE ALSO

ild8 uld8 uld8d ild8r uld8r

DESCRIPTION

The `ild8d` operation loads the 8-bit memory value from the address computed by r*src1* + *d*, sign extends it to 32 bits, and stores the result in r*dest*. The *d* value is an opcode modifier in the range -64 to 63, inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22	`ild8d(2) r10 → r60`	r60 ← 0x000022
r30 = 0, r20 = 0xd04, [0xd00] = 0x84	`IF r30 ild8d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84	`IF r40 ild8d(-4) r20 → r80`	r80 ← 0xffffff84
r50 = 0xd05, [0xd01] = 0x33	`ild8d(-4) r50 → r90`	r90 ← 0x00000033

Signed 8-bit load with index

ild8r

SYNTAX

```
[ IF rguard ] ild8r rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← sign_ext8to32(mem[r*src1* + r*src2*])

ATTRIBUTES

Function unit	dmem
Operation code	193
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild8 uld8 ild8d uld8d uld8r

DESCRIPTION

The `ild8r` operation loads the 8-bit memory value from the address computed by r*src1* + r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22	`ild8r r10 r20 → r80`	r80 ← 0x00000022
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r50 ild8r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r60 ild8r r40 r30 → r100`	r100 ← 0xffffff84
r70 = 0xd05, r30 = 0xfffffffc, [0xd01] = 0x33	`ild8r r70 r30 → r110`	r110 ← 0x00000033

ileq

Signed compare less or equal

pseudo-op for igeq

SYNTAX

```
[ IF rguard ] ileq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 <= rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	14
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq ileqi

DESCRIPTION

The `ileq` operation is a pseudo operation transformed by the scheduler into an `igeq` with the arguments exchanged (`ileq`'s r*src1* is `igeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `ileq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ileq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ileq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ileq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, 0x100	`IF r20 ileq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ileq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ileq r70 r70 → r110`	r110 ← 1

Signed compare less or equal with immediate

ileqi

SYNTAX

```
[ IF rguard ] ileqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 <= n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	42
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ileq igeqi

DESCRIPTION

The `ileqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ileqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ileqi(2) r30 → r80`	r80 ← 0
r30 = 3	`ileqi(3) r30 → r90`	r90 ← 1
r30 = 3	`ileqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ileqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ileqi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`ileqi(-64) r60 → r120`	r120 ← 1

iles

Signed compare less

pseudo-op for igtr

SYNTAX

```
[ IF rguard ] iles rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 < rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	15
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr ilesi

DESCRIPTION

The `iles` operation is a pseudo operation transformed by the scheduler into an `igtr` with the arguments exchanged (`iles`'s r*src1* is `igtr`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `iles` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `iles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`iles r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 iles r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, 0x100	`IF r20 iles r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`iles r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`iles r70 r70 → r110`	r110 ← 0

Signed compare less with immediate

ilesi

SYNTAX

```
[ IF rguard ] ilesi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 < n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	2
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iles ileqi

DESCRIPTION

The `ilesi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ilesi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ilesi(2) r30 → r80`	r80 ← 0
r30 = 3	`ilesi(3) r30 → r90`	r90 ← 0
r30 = 3	`ilesi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ilesi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ilesi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`ilesi(-64) r60 → r120`	r120 ← 1

imax

Signed maximum

SYNTAX

```
[ IF rguard ] imax rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > rsrc2 then
        rdest ← rsrc1
    else
        rdest ← rsrc2
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	24
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

imin

DESCRIPTION

The `imax` operation sets the destination register, r*dest*, to the contents of r*src1* if r*src1*>r*src2*; otherwise, r*dest* is set to the contents of r*src2*. The arguments are treated as signed integers.

The `imax` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`imax r30 r20 → r80`	r80 ← 2
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 imax r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 imax r60 r40 → r90`	r90 ← 0x100
r70 = 0xffffff00, r40 = 0xffffff9c	`imax r70 r40 → r100`	r100 ← 0xffffff9c

Signed minimum

imin

SYNTAX

```
[ IF rguard ] imin rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 > rsrc2 then
      rdest ← rsrc2
   else
      rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	23
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

imax

DESCRIPTION

The `imin` operation sets the destination register, r*dest*, to the contents of r*src2* if r*src1*>r*src2*; otherwise, r*dest* is set to the contents of r*src1*. The arguments are treated as signed integers.

The `imin` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`imin r30 r20 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 imin r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 imin r60 r40 → r90`	r90 ← 0xffffff9c
r70 = 0xffffff00, r40 = 0xffffff9c	`imin r70 r40 → r100`	r100 ← 0xffffff00

imul

Signed multiply

SYNTAX

```
[ IF rguard ] imul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  temp ← (sign_ext32to64(r*src1*) × sign_ext32to64(r*src2*))
  r*dest* ← temp<31:0>

ATTRIBUTES

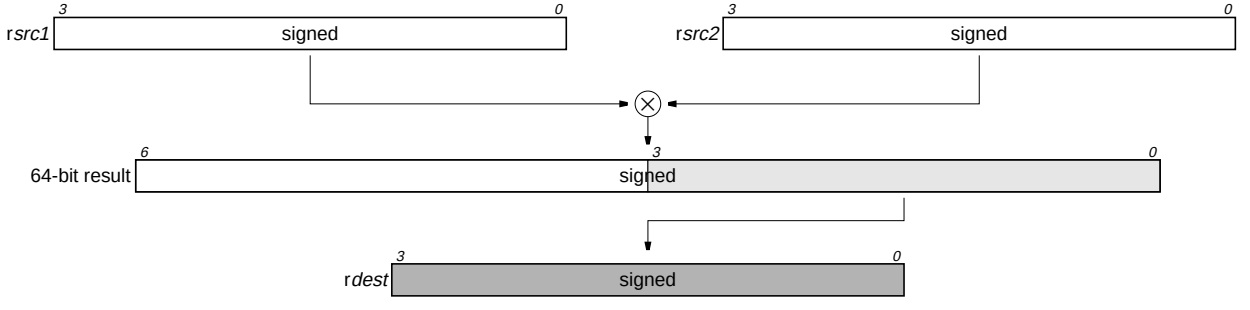
Function unit	ifmul
Operation code	27
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

umul imulm umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the `imul` operation computes the product r*src1*×r*src2* and writes the least-significant 32 bits of the full 64-bit product into r*dest*. The operands are considered signed integers. No overflow or underflow detection is performed.



The `imul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	`imul r60 r60 → r80`	r80 ← 0x10000
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 imul r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	`IF r20 imul r60 r30 → r90`	r90 ← 0xf1100
r70 = 0xffffff00, r40 = 0xffffff9c	`imul r70 r40 → r100`	r100 ← 0x6400

Signed multiply, return most-significant 32 bits

imulm

SYNTAX

```
[ IF rguard ] imulm rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	ifmul
Operation code	139
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

FUNCTION

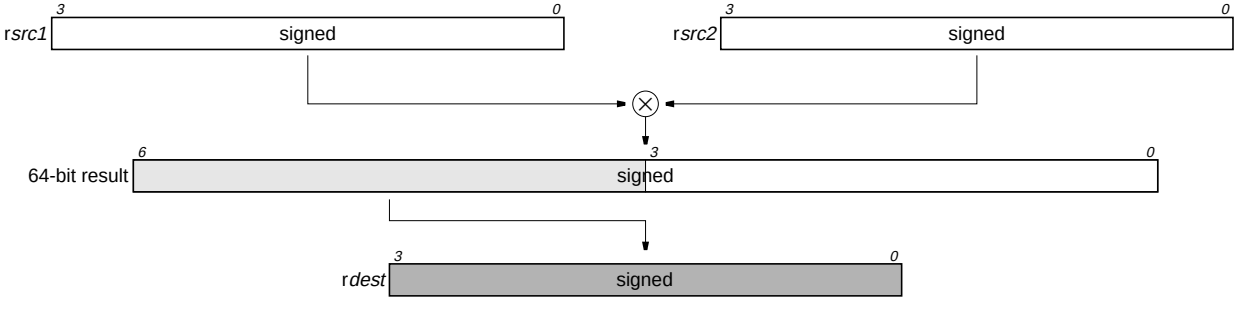
if r*guard* **then**
  temp ← (sign_ext32to64(r*src1*) × sign_ext32to64(r*src2*))
  r*dest* ← temp<63:32>

SEE ALSO

umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the `imulm` operation computes the product r*src1*×r*src2* and writes the most-significant 32 bits of the full 64-bit product into r*dest*. The operands are considered signed integers.



The `imulm` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10000	`imulm r60 r60 → r80`	r80 ← 0x00000001
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 imulm r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x10001000, r30 = 0xf1100000	`IF r20 imulm r60 r30 → r90`	r90 ← 0xff10ff11
r70 = 0xffffff00, r40 = 0x64	`imulm r70 r40 → r100`	r100 ← 0xffffffff

ineg

Signed negate

pseudo-op for isub

SYNTAX

```
[ IF rguard ] ineg rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest* ← –rsrc1

ATTRIBUTES

Function unit	alu
Operation code	13
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

isub

DESCRIPTION

The `ineg` operation is a pseudo operation transformed by the scheduler into an `isub` with r0 (always contains 0) as the first argument and r*src1* as the second argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ineg` operation computes the negative of r*src1* and writes the result into r*dest*. The argument is a signed integer; the result is an unsigned integer. If rsrc1 = 0x80000000, then `ineg` returns 0x80000000 since the positive value is not representable.

The `ineg` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`ineg r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	`IF r10 ineg r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	`IF r20 ineg r40 → r90`	r90 ← 0xc
r50 = 0x80000001	`ineg r50 → r100`	r100 ← 0x7fffffff
r60 = 0x80000000	`ineg r60 → r110`	r110 ← 0x80000000
r20 = 1	`ineg r20 → r120`	r120 ← 0xffffffff

Signed compare not equal

ineq

SYNTAX

```
[ IF rguard ] ineq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 != rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	39
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieql igtr ineqi

DESCRIPTION

The `ineq` operation sets the destination register, r*dest*, to 1 if the two arguments, r*src1* and r*src2*, are not equal; otherwise, r*dest* is set to 0.

The `ineq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ineq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x1000, r30 = 3	`IF r10 ineq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 ineq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ineq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ineq r70 r70 → r110`	r110 ← 0

ineqi

Signed compare not equal with immediate

SYNTAX

```
[ IF rguard ] ineqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 != n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	3
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ineq igeqi ieqli

DESCRIPTION

The `ineqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ineqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ineqi(2) r30 → r80`	r80 ← 1
r30 = 3	`ineqi(3) r30 → r90`	r90 ← 0
r30 = 3	`ineqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ineqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ineqi(63) r40 → r100`	r100 ← 1
r60 = 0xffffffc0	`ineqi(-64) r60 → r120`	r120 ← 0

If nonzero select zero

inonzero

SYNTAX

```
[ IF rguard ] inonzero rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 != 0 then
      rdest ← 0
   else
      rdest ← rsrc2
}
```

ATTRIBUTES

Function unit	alu
Operation code	47
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

izero iflip

DESCRIPTION

The `inonzero` operation writes 0 into r*dest* if the value of r*src1* is not zero; otherwise, r*src2* is copied to r*dest*. The operands are considered signed integers.

The `inonzero` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`inonzero r30 r20 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 inonzero r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 inonzero r60 r40 → r90`	r90 ← 0
r10 = 0, r40 = 0xffffff9c	`inonzero r10 r40 → r100`	r100 ← 0xffffff9c
r20 = 1, r60 = 0x100	`inonzero r20 r60 → r110`	r110 ← 0
r10 = 0, r70 = 0x456789	`inonzero r10 r70 → r120`	r120 ← 0x456789

isub

Subtract

SYNTAX

```
[ IF rguard ] isub rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* – r*src2*

ATTRIBUTES

Function unit	alu
Operation code	13
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

isubi borrow dspisub dspidualsub fsub

DESCRIPTION

The `isub` operation computes the difference r*src1*–r*src2* and writes the result into r*dest*. The operands can be either both signed or unsigned integers. No overflow or underflow detection is performed.

The `isub` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`isub r30 r40 → r80`	r80 ← 0xffffffff
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 isub r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 isub r50 r60 → r90`	r90 ← 0xf00
r70 = 0x80000000, r40 = 4	`isub r70 r40 → r100`	r100 ← 0x7ffffffc

Subtract with immediate

isubi

SYNTAX

```
[ IF rguard ] isubi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest* ← r*src1* – *n*

ATTRIBUTES

Function unit	alu
Operation code	32
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

isub borrow

DESCRIPTION

The `isubi` operation computes the difference of a single argument in r*src1* and an immediate modifier *n* and stores the result in r*dest*. The value of *n* must be between 0 and 127, inclusive.

The `isubi` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf11	`isubi(127) r30 → r70`	r70 ← 0xe92
r10 = 0, r40 = 0xffffff9c	`IF r10 isubi(1) r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xffffff9c	`IF r20 isubi(1) r40 → r90`	r90 ← 0xffffff9b
r50 = 0x1000	`isubi(15) r50 → r120`	r120 ← 0x0ff1
r60 = 0xfffffff0	`isubi(2) r60 → r110`	r110 ← 0xffffffee
r20 = 1	`isubi(17) r20 → r120`	r120 ← 0xfffffff0

izero

If zero select zero

SYNTAX

```
[ IF rguard ] izero rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 = 0 then
        rdest ← 0
    else
        rdest ← rsrc2
}
```

ATTRIBUTES

Function unit	alu
Operation code	46
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

inonzero iflip

DESCRIPTION

The izero operation writes 0 into r*dest* if the value of r*src1* is equal to zero; otherwise, r*src2* is copied to r*dest*. The operands are considered signed integers.

The izero operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	izero r30 r20 → r80	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 2	IF r10 izero r60 r30 → r50	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	IF r20 izero r60 r40 → r90	r90 ← 0xffffff9c
r10 = 0, r40 = 0xffffff9c	izero r10 r40 → r100	r100 ← 0
r20 = 1, r60 = 0x100	izero r20 r60 → r110	r110 ← 0x100
r20 = 1, r70 = 0x456789	izero r20 r70 → r120	r120 ← 0x456789

Indirect jump on false

jmpf

SYNTAX

```
[ IF rguard ] jmpf rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if (r*src1* & 1) = 0 **then**
    PC ← r*src2*
}

ATTRIBUTES

Function unit	branch
Operation code	180
Number of operands	2
Modifier	No
Modifier range	—
Delay	3
Issue slots	2, 3, 4

SEE ALSO

`jmpt jmpi ijmpf ijmpt ijmpi`

DESCRIPTION

The `jmpf` operation conditionally changes the program flow. If the LSB of r*src1* is 0, the PC register is set equal to r*src2*; otherwise, program execution continues with the next sequential instruction.

The `jmpf` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 0, r70 = 0x330	`jmpf r50 r70`	program execution continues at 0x330
r20 = 1, r70 = 0x330	`jmpf r20 r70`	since r20 is true, program execution continues with next sequential instruction
r30 = 0, r50 = 0, r60 = 0x8000	`IF r30 jmpf r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 0, r60 = 0x8000	`IF r40 jmpf r50 r60`	program execution continues at 0x8000

jmpi

Jump immediate

SYNTAX

```
[ IF rguard ] jmpi(address)
```

FUNCTION

if r*guard* **then**
  PC ← *address*

ATTRIBUTES

Function unit	branch
Operation code	178
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Delay	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt ijmpf ijmpt ijmpi

DESCRIPTION

The `jmpi` operation changes the program flow by setting the PC register equal to the immediate opcode modifier *address*.

The `jmpi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds a condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken.

EXAMPLES

Initial Values	Operation	Result
	`jmpi(0x330)`	program execution continues at 0x330
r30 = 0	`IF r30 jmpi(0x8000)`	since guard is false, program execution continues with next sequential instruction
r40 = 1	`IF r40 jmpi(0x8000)`	program execution continues at 0x8000

Indirect jump on true

jmpt

SYNTAX

```
[ IF rguard ] jmpt rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if (r*src1* & 1) = 1 **then**
    PC ← r*src2*
}

ATTRIBUTES

Function unit	branch
Operation code	176
Number of operands	2
Modifier	no
Modifier range	—
Delay	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpi ijmpf ijmpt ijmpi

DESCRIPTION

The `jmpt` operation conditionally changes the program flow. If the LSB of r*src1* is 1, the PC register is set equal to r*src2*; otherwise, program execution continues with the next sequential instruction.

The `jmpt` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 1, r70 = 0x330	`jmpt r50 r70`	program execution continues at 0x330
r20 = 0, r70 = 0x330	`jmpt r20 r70`	since r20 is false, program execution continues with next sequential instruction
r30 = 0, r50 = 1, r60 = 0x8000	`IF r30 jmpt r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 1, r60 = 0x8000	`IF r40 jmpt r50 r60`	program execution continues at 0x8000

ld32

32-bit load

pseudo-op for ld32d(0)

SYNTAX

```
[ IF rguard ] ld32 rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   rdest<7:0> ← mem[rsrc1 + (3 ⊕ bs)]
   rdest<15:8> ← mem[rsrc1 + (2 ⊕ bs)]
   rdest<23:16> ← mem[rsrc1 + (1 ⊕ bs)]
   rdest<31:24> ← mem[rsrc1 + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	7
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ld32d ld32r ld32x st32 st32d h_st32d

DESCRIPTION

The ld32 operation is a pseudo operation transformed by the scheduler into an ld32d(0) with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The ld32 operation loads the 32-bit memory value from the address contained in r*src1* and stores the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 4, the result of ld32 is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The ld32 operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by ld32.

The ld32 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ld32 has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	ld32 r10 → r60	r60 ← 0x84332211
r30 = 0, r20 = 0xd04, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r30 ld32 r20 → r70	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r40 ld32 r20 → r80	r80 ← 0x48665544
r50 = 0xd01	ld32 r50 → r90	r90 undefined, since 0xd01 is not a multiple of 4

32-bit load with displacement

ld32d

SYNTAX

```
[ IF rguard ] ld32d(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   rdest<7:0> ← mem[rsrc1 + d + (3 ⊕ bs)]
   rdest<15:8> ← mem[rsrc1 + d + (2 ⊕ bs)]
   rdest<23:16> ← mem[rsrc1 + d + (1 ⊕ bs)]
   rdest<31:24> ← mem[rsrc1 + d + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	7
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	4, 5

SEE ALSO

ld32 ld32r ld32x st32 st32d h_st32d

DESCRIPTION

The ld32d operation loads the 32-bit memory value from the address computed by r*src1* + *d* and stores the result in r*dest*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4. If the memory address computed by r*src1* + *d* is not a multiple of 4, the result of ld32d is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The ld32d operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by ld32d.

The ld32d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ld32d has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	ld32d(4) r10 → r60	r60 ← 0x84332211
r30 = 0, r20 = 0xd0c, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r30 ld32d(-8) r20 → r70	no change, since guard is false
r40 = 1, r20 = 0xd0c, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r40 ld32d(-8) r20 → r80	r80 ← 0x48665544
r50 = 0xd01	ld32d(-8) r50 → r90	r90 undefined, since 0xd01 +(–8) is not a multiple of 4

ld32r

32-bit load with index

SYNTAX

```
[ IF rguard ] ld32r rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   rdest<7:0> ← mem[rsrc1 + rsrc2 + (3 ⊕ bs)]
   rdest<15:8> ← mem[rsrc1 + rsrc2 + (2 ⊕ bs)]
   rdest<23:16> ← mem[rsrc1 + rsrc2 + (1 ⊕ bs)]
   rdest<31:24> ← mem[rsrc1 + rsrc2 + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	200
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ld32 ld32d ld32x st32 st32d h_st32d

DESCRIPTION

The ld32r operation loads the 32-bit memory value from the address computed by r*src1* + r*src2* and stores the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 4, the result of ld32r is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The ld32r operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by ld32r.

The ld32r operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ld32r has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r20 = 0x4, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	ld32r r10 r20 → r80	r80 ← 0x84332211
r50 = 0, r40 = 0xd0c, r30 = 0xfffffff8, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r50 ld32r r40 r30 → r90	no change, since guard is false
r60 = 1, r40 = 0xd0c, r30 = 0xfffffff8, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r60 ld32r r40 r30 → r100	r100 ← 0x48665544
r50 = 0xd01, r30 = 0xfffffff8	ld32r r70 r30 → r110	r110 undefined, since 0xd01 +(–8) is not a multiple of 2

32-bit load with scaled index

ld32x

SYNTAX

```
[ IF rguard ] ld32x rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   rdest<7:0> ← mem[rsrc1 + (4 × rsrc2) +(3 ⊕ bs)]
   rdest<15:8> ← mem[rsrc1 + (4 × rsrc2) + (2 ⊕ bs)]
   rdest<23:16> ← mem[rsrc1 + (4 × rsrc2) + (1 ⊕ bs)]
   rdest<31:24> ← mem[rsrc1 + (4 × rsrc2) + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	201
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ld32 ld32d ld32r st32 st32d h_st32d

DESCRIPTION

The ld32x operation loads the 32-bit memory value from the address computed by r*src1* + 4×r*src2* and stores the result in r*dest*. If the memory address computed by r*src1* + 4×r*src2* is not a multiple of 4, the result of ld32x is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The ld32x operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by ld32x.

The ld32x operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ld32x has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r30 = 0x1, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	ld32x r10 r30 → r100	r100 ← 0x84332211
r50 = 0, r40 = 0xd0c, r20 = 0xfffffffe, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r50 ld32x r40 r20 → r80	no change, since guard is false
r60 = 1, r40 = 0xd0c, r20 = 0xfffffffe, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	IF r60 ld32x r40 r20 → r90	r90 ← 0x48665544
r70 = 0xd01, r30 = 0x1	ld32x r70 r30 → r110	r110 undefined, since 0xd01 + 4×1 is not a multiple of 4

lsl

Logical shift left

pseudo-op for asl

SYNTAX

```
[ IF rguard ] lsl rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   n ← rsrc2<4:0>
   rdest<31:n> ← rsrc1<31–n:0>
   rdest<n–1:0> ← 0
   if rsrc2<31:5> != 0 {
      rdest <- 0
   }
}
```

ATTRIBUTES

Function unit	shifter
Operation code	19
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

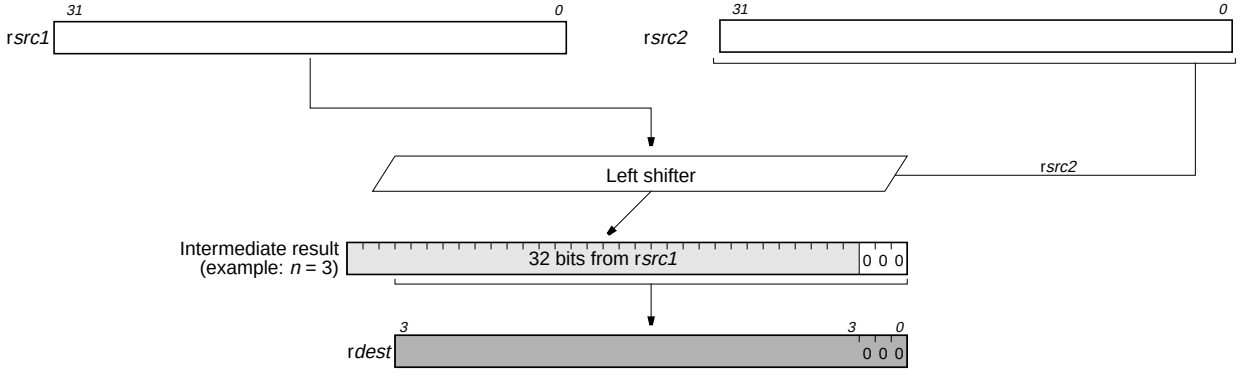
SEE ALSO

asl asli asr asri lsli lsr lsri rol roli

DESCRIPTION

The lsl operation is a pseudo operation that is transformed by the scheduler into an asl with the same arguments. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the lsl operation takes two arguments, r*src1* and r*src2*. R*src2* specify an unsigned shift amount, and rdest is set to r*src1* logically shifted left by this amount. If the rsrc2<31:5> value is not zero, then take this as a shift by 32 or more bits. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The lsl operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	lsl r60 r30 → r90	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	IF r10 lsl r60 r30 → r100	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	IF r20 lsl r60 r30 → r110	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	lsl r70 r40 → r120	r120 ← 0xfffffff0
r80 = 0xe, r50 = 0xfffffffe	lsl r80 r50 → r125	r125 ← 0x00000000 (shift by more than 32))
r30 = 0x7008000f, r45 = 0x20	lsl r30 r45 → r100	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x80000000	lsl r30 r45 → r100	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x23	lsl r30 r45 → r100	r100 ← 0x00000000

Logical shift left immediate

pseudo-op for asli

lsli

SYNTAX

```
[ IF rguard ] lsli(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:*n*> ← r*src1*<31–*n*:0>
    r*dest*<*n*–1:0> ← 0
}

ATTRIBUTES

Function unit	shifter
Operation code	11
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

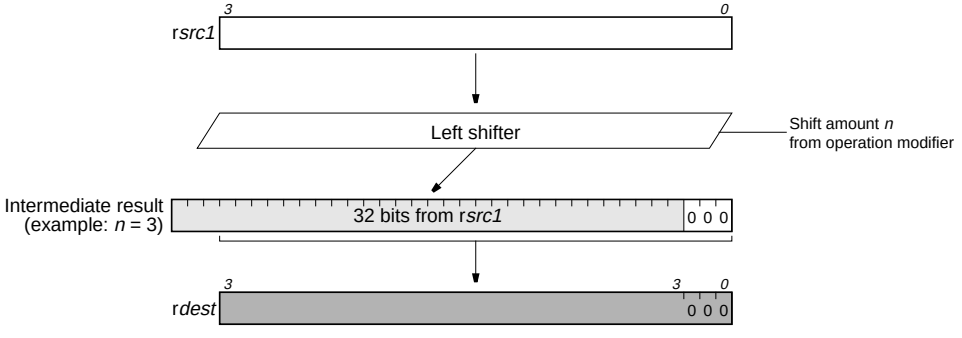
SEE ALSO

asl asli asr asri lsl lsr lsri rol roli

DESCRIPTION

The `lsli` operation is a pseudo operation that is transformed by the scheduler into an `asli` with the same argument and opcode modifier. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `lsli` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* logically shifted left by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `lsli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	`lsli(3) r60 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20	`IF r10 lsli(3) r60 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20	`IF r20 lsli(3) r60 → r110`	r110 ← 0x100
r70 = 0xfffffffc	`lsli(2) r70 → r120`	r120 ← 0xfffffff0
r80 = 0xe	`lsli(30) r80 → r125`	r125 ← 0x80000000

lsr

Logical shift right

SYNTAX

```
[ IF rguard ] lsr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   n ← rsrc2<4:0>
   rdest<31:32–n> ← 0
   rdest<31–n:0> ← rsrc1<31:n>
   if rsrc2<31:5> != 0 {
      rdest <- 0
   }
}
```

ATTRIBUTES

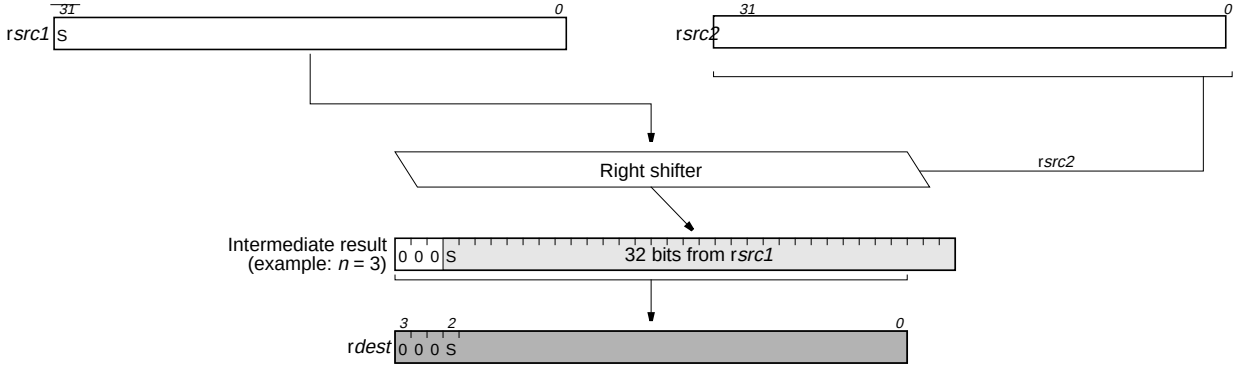
Function unit	shifter
Operation code	96
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr asri lsl lsli lsri rol roli

DESCRIPTION

As shown below, the `lsr` operation takes two arguments, r*src1* and r*src2*. R*src2* specifies an unsigned shift amount, and r*src1* is logically shifted right by this amount. If the rsrc2<31:5> value is not zero, then take this as a shift by 32 or more bits. Zeros fill vacated bits from the left.



The `lsr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f, r20 = 1	`lsr r30 r20 → r50`	r50 ← 0x38040007
r30 = 0x7008000f, r42 = 2	`lsr r30 r42 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f, r44 = 4	`IF r10 lsr r30 r44 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f, r44 = 4	`IF r20 lsr r30 r44 → r80`	r80 ← 0x07008000
r40 = 0x80030007, r44 = 4	`lsr r40 r44 → r90`	r90 ← 0x08003000
r30 = 0x7008000f, r45 = 0x1f	`lsr r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x1f	`lsr r30 r45 → r100`	r100 ← 0x00000001
r30 = 0x7008000f, r45 = 0x20	`lsr r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x80000000	`lsr r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x23	`lsr r30 r45 → r100`	r100 ← 0x00000000

Logical shift right immediate

lsri

SYNTAX

```
[ IF rguard ] lsri(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:32–*n*> ← 0
    r*dest*<31–*n*:0> ← r*src1*<31:*n*>
}

ATTRIBUTES

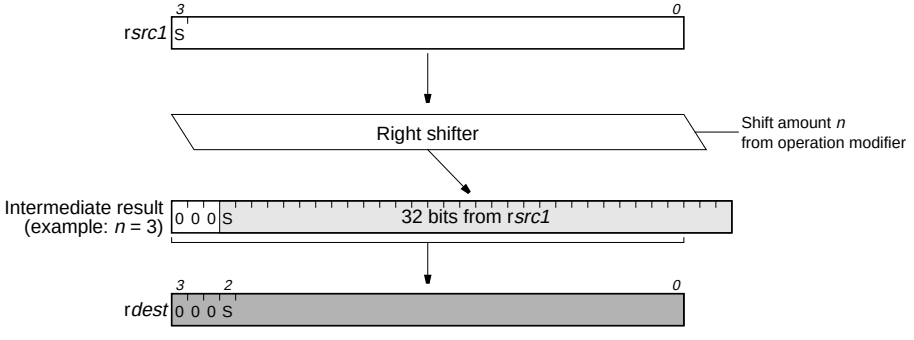
Function unit	shifter
Operation code	9
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr asri lsl lsli lsr rol roli

DESCRIPTION

As shown below, the `lsri` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* that is equal to r*src1* logically shifted right by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros fill vacated bits from the left.



The `lsri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f	`lsri(1) r30 → r50`	r50 ← 0x38040007
r30 = 0x7008000f	`lsri(2) r30 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f	`IF r10 lsri(4) r30 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f	`IF r20 lsri(4) r30 → r80`	r80 ← 0x07008000
r40 = 0x80030007	`lsri(4) r40 → r90`	r90 ← 0x08003000
r30 = 0x7008000f	`lsri(31) r30 → r100`	r100 ← 0x00000000
r40 = 0x80030007	`lsri(31) r40 → r110`	r110 ← 0x00000001

mergedual16lsb

Merge dual 16-bit lsb bytes

SYNTAX

```
[ IF rguard ] mergedual16lsb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
rdest<31:24> <- rsrc1<23:16>
rdest<23:16> <- rsrc1<7:0>
rdest<15:8> <- rsrc2<23:16>
rdest<7:0> <- rsrc2<7:0>
}

ATTRIBUTES

Function unit	shifter
Operation code	103
Number of operands	2
Modifier	No
Modifier range	-
Latency	1
Issue slots	1,2

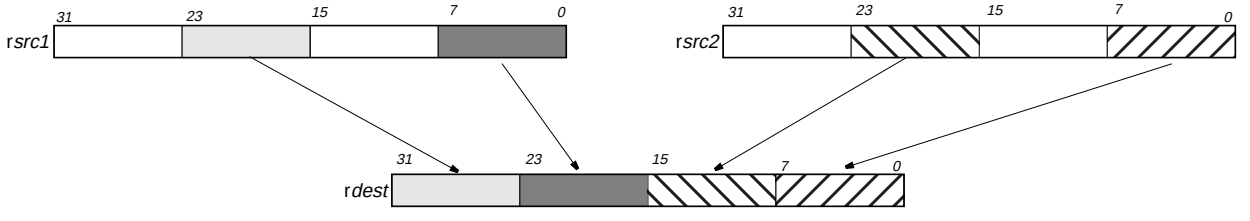
SEE ALSO

mergelsb mergemsb
pack16lsb pack16msb

DESCRIPTION

The arguments rsrc1 and rsrc2 are vectors of two 16-bit data. The mergedual16lsb operation merges the least significant bytes from each 16-bit data rsrc1 and rsrc2 into one 32-bit data in dest register, to convert to quad 8-bit.

The mergedual16lsb operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the modification of the destination register. If the LSB of rguard is 1, rdest is written; otherwise, rdest is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergedual16lsb r30 r40 -> r50`	r50 <- 0x3478bbdd
r10 = 0, r30 = 0x12345678, r40 = 0xaabbccdd	`IF r10 mergedual16lsb r30 r40 -> r50`	no change, since guard is false
r10 = 1, r30 = 0x01020304, r40 = 0x0a0b0c0d	`IF r10 mergedual16lsb r30 r40 -> r50`	r50 <- 0x02040b0d

Merge least-significant byte

mergelsb

SYNTAX

```
[ IF rguard ] mergelsb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<7:0> ← r*src2*<7:0>
    r*dest*<15:8> ← r*src1*<7:0>
    r*dest*<23:16> ← r*src2*<15:8>
    r*dest*<31:24> ← r*src1*<15:8>
}

ATTRIBUTES

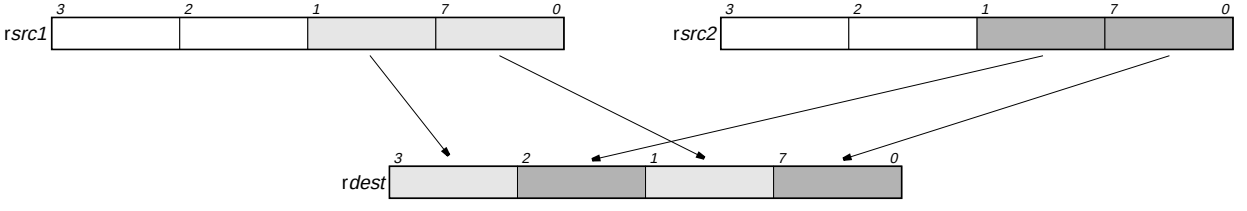
Function unit	alu
Operation code	57
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb
packbytes mergemsb

DESCRIPTION

As shown below, the `mergelsb` operation interleaves the two pairs of least-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The least-significant byte from r*src2* is packed into the least-significant byte of r*dest*; the least-significant byte from r*src1* is packed into the second-least-significant byte of r*dest*; the second-least-significant byte from r*src2* is packed into the second-most-significant byte of r*dest*; and the second-least-significant byte from r*src1* is packed into the most-significant byte of r*dest*.



The `mergelsb` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergelsb r30 r40 → r50`	r50 ← 0x56cc78dd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 mergelsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 mergelsb r40 r30 → r70`	r70 ← 0xcc56dd78

mergemsb

Merge most-significant byte

SYNTAX

```
[ IF rguard ] mergemsb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<7:0> ← r*src2*<23:15>
    r*dest*<15:8> ← r*src1*<23:15>
    r*dest*<23:16> ← r*src2*<31:24>
    r*dest*<31:24> ← r*src1*<31:24>
}

ATTRIBUTES

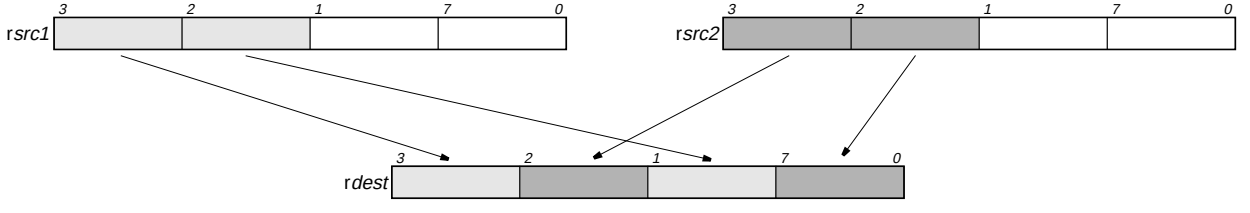
Function unit	alu
Operation code	58
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb
packbytes mergelsb

DESCRIPTION

As shown below, the mergemsb operation interleaves the two pairs of most-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The second-most-significant byte from r*src2* is packed into the least-significant byte of r*dest*; the second-most-significant byte from r*src1* is packed into the second-least-significant byte of r*dest*; the most-significant byte from r*src2* is packed into the second-most-significant byte of r*dest*; and the most-significant byte from r*src1* is packed into the most-significant byte of r*dest*.



The mergemsb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergemsb r30 r40 → r50`	r50 ← 0x12aa34bb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 mergemsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 mergemsb r40 r30 → r70`	r70 ← 0xaa12bb34

No operation

nop

SYNTAX

nop

FUNCTION

No operation

ATTRIBUTES

Function unit	-
Operation code	-
Number of operands	-
Modifier	-
Modifier range	-
Latency	1
Issue slots	1-5

SEE ALSO

DESCRIPTION

The NOP operation does not change any DSPCPU state. It is mainly used to fill-up the empty issue slots. Only two bits are used to code the NOP operation.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	nop	No change in any regsiters

pack16lsb

Pack least-significant 16-bit halfwords

SYNTAX

```
[ IF rguard ] pack16lsb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<15:0> ← r*src2*<15:0>
    r*dest*<31:16> ← r*src1*<15:0>
}

ATTRIBUTES

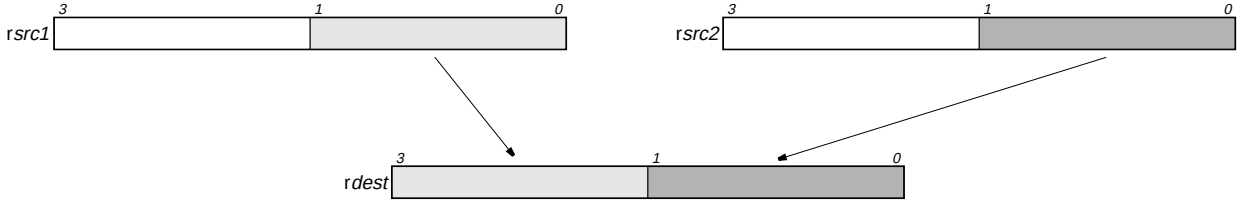
Function unit	alu
Operation code	53
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16msb packbytes
mergelsb mergemsb

DESCRIPTION

As shown below, the pack16lsb operation packs the two least-significant halfwords from the arguments r*src1* and r*src2* into r*dest*. The halfword from r*src1* is packed into the most-significant halfword of r*dest*; the halfword from r*src2* is packed into the least-significant halfword of r*dest*.



The pack16lsb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`pack16lsb r30 r40 → r50`	r50 ← 0x5678ccdd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 pack16lsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 pack16lsb r40 r30 → r70`	r70 ← 0xccdd5678

Pack most-significant 16 bits

pack16msb

SYNTAX

```
[ IF rguard ] pack16msb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<15:0> ← r*src2*<31:16>
    r*dest*<31:16> ← r*src1*<31:16>
}

ATTRIBUTES

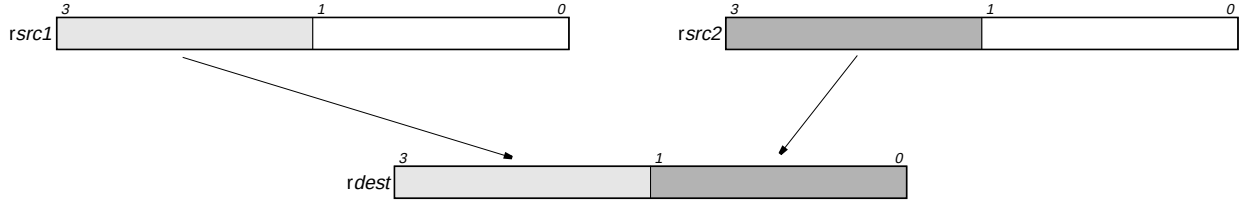
Function unit	alu
Operation code	54
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb packbytes
mergelsb mergemsb

DESCRIPTION

As shown below, the pack16msb operation packs the two most-significant halfwords from the arguments r*src1* and r*src2* into r*dest*. The halfword from r*src1* is packed into the most-significant halfword of r*dest*; the halfword from r*src2* is packed into the least-significant halfword of r*dest*.



The pack16msb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	pack16msb r30 r40 → r50	r50 ← 0x1234aabb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	IF r10 pack16msb r40 r30 → r60	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	IF r20 pack16msb r40 r30 → r70	r70 ← 0xaabb1234

packbytes

Pack least-significant byte

SYNTAX

```
[ IF rguard ] packbytes rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<7:0> ← r*src2*<7:0>
    r*dest*<15:8> ← r*src1*<7:0>
}

ATTRIBUTES

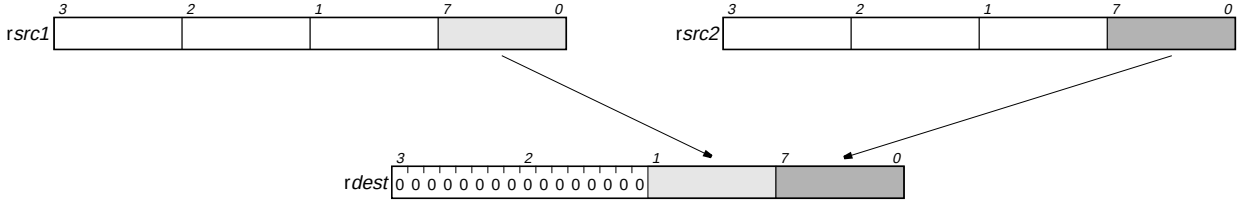
Function unit	alu
Operation code	52
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb
mergelsb mergemsb

DESCRIPTION

As shown below, the packbytes operation packs the two least-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The byte from r*src1* is packed into the second-least-significant byte of r*dest*; the byte from r*src2* is packed into the least-significant byte of r*dest*. The two most-significant bytes of r*dest* are filled with zeros.



The packbytes operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	packbytes r30 r40 → r50	r50 ← 0x000078dd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	IF r10 packbytes r40 r30 → r60	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	IF r20 packbytes r40 r30 → r70	r70 ← 0x0000dd78

prefetch

pseudo-op for prefd(0)

pref

SYNTAX

```
[ IF rguard ] pref rsrc1
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + 0) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	209
Number of operands	1
Modifier	-
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefd prefr allocd allocr allocx

DESCRIPTION

The pref operation is a pseudo operation transformed by the scheduler into an prefd(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The pref operation loads the one full cache block size of memory value from the address computed by ((rsrc1+0) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A pref operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`pref r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 pref r10`	since guard is false, pref operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 pref r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000, TM1100 and TM1300 and it is not guaranteed to be available in future generations of Trimedia products.

pref16x

prefetch with 16-bit scaled index

SYNTAX

```
[ IF rguard ] pref16x rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + (2 x rscr2)) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	211
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref32x prefd prefr allocd allocr allocx

DESCRIPTION

The pref16x operation loads one full cache block from the main memory at the address computed by ((rsrc1+ (2 x rscr2)) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. The data cache has hardware to simultaneously sustain two cache misses or prefetches. A pref16x operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref16x operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xc cache_block_size = 0x40	`pref16x r10 r12`	Loads a cache line for the address space from 0xabc0 to 0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xc, cache_block_size = 0x40	`IF r11 pref16x r10 r12`	since guard is false, pref16x operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 pref16x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000, TM1100 and TM1300 and it is not guaranteed to be available in future generations of Trimedia products.

prefetch with 32-bit scaled index

pref32x

SYNTAX

```
[ IF rguard ] pref32x rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + (4 x rscr2)) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	212
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x prefd prefr allocd allocr allocx

DESCRIPTION

The pref32x operation loads the one full cache block size of memory value from the address computed by ((rsrc1+ (4 x rscr2)) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A pref32x operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref32x operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xd cache_block_size = 0x40	`pref32x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xd, cache_block_size = 0x40	`IF r11 pref32x r10 r12`	since guard is false, pref32x operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 pref32x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000, TM1100 and TM1300 and it is not guaranteed to be available in future generations of this product.

prefd

prefetch with displacement

SYNTAX

```
[ IF rguard ] prefd(d) rsrc1
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + d) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	209
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefr allocd allocr allocx

DESCRIPTION

The prefd operation loads the one full cache block size of memory value from the address computed by ((rsrc1+d) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A prefd operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The prefd operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`prefd(0xd) r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 prefd(0xd) r10`	since guard is false, prefd operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 prefd(ox1) r10`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000, TM1100 and TM1300 and it is not guaranteed to be available in future generations of this product.

prefetch with index

prefr

SYNTAX

```
[ IF rguard ] prefr rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + rscr2) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	210
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefd allocd allocr allocx

DESCRIPTION

The prefr operation loads the one full cache block size of memory value from the address computed by ((rsrc1+rscr2) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A prefr operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The `prefr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of r*guard* is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xd cache_block_size = 0x40	`prefr r10 r12`	Loads a cache line for the address space from 0xabc0 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xd, cache_block_size = 0x40	`IF r11 prefr r10 r12`	since guard is false, prefr operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 prefr r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

NOTE: This operation is supported only in TM1000, TM1100 and TM1300 and it is not guaranteed to be available in future generations of this product.

quadavg

Unsigned byte-wise quad average

SYNTAX

```
[ IF rguard ] quadavg rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspalu
Operation code	73
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

FUNCTION

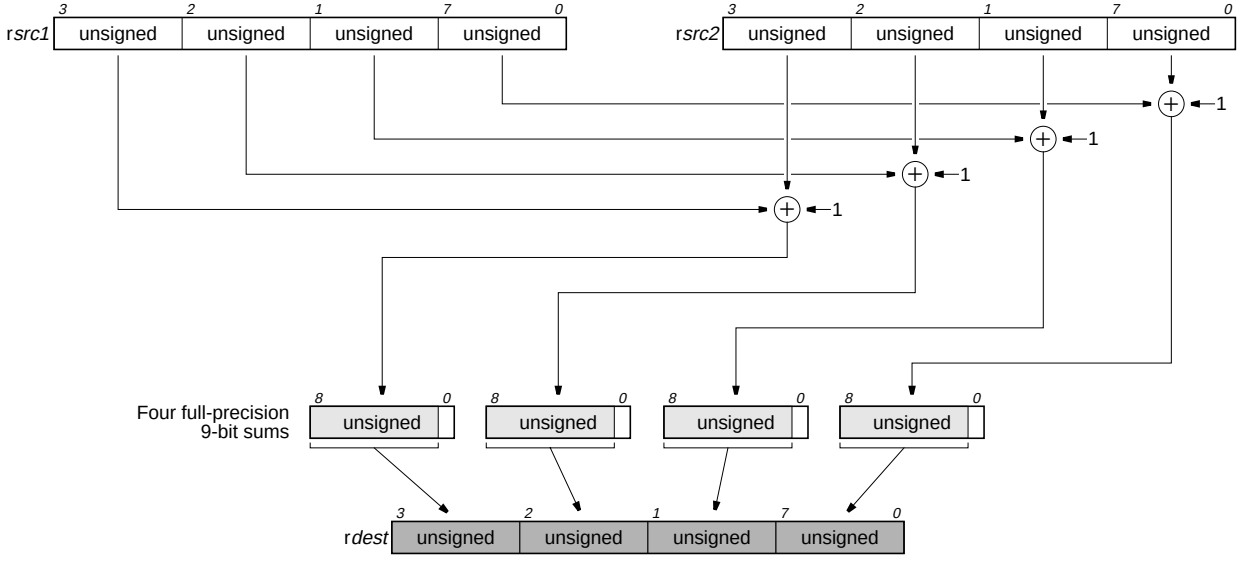
if r*guard* **then {**
  temp ← (zero_ext8to32(r*src1*<7:0>) + zero_ext8to32(r*src2*<7:0>) + 1) / 2
  r*dest*<7:0> ← temp<7:0>
  temp ← (zero_ext8to32(r*src1*<15:8>) + zero_ext8to32(r*src2*<15:8>) + 1) / 2
  r*dest*<15:8> ← temp<7:0>
  temp ← (zero_ext8to32(r*src1*<23:16>) + zero_ext8to32(r*src2*<23:16>) + 1) / 2
  r*dest*<23:16> ← temp<7:0>
  temp ← (zero_ext8to32(r*src1*<31:24>) + zero_ext8to32(r*src2*<31:24>) + 1) / 2
  r*dest*<31:24> ← temp<7:0>
}

SEE ALSO

iavgonep dspuquadaddui ifir8ii

DESCRIPTION

As shown below, the quadavg operation computes four separate averages of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The least-significant 8 bits of each average is written to the corresponding byte in r*dest*. No overflow or underflow detection is performed.



The quadavg operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0201000e, r40 = 0xffffff02	quadavg r30 r40 → r50	r50 ← 0x81808008
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649c649c	IF r10 quadavg r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649c649c	IF r20 quadavg r60 r70 → r90	r90 ← 0x809c6480

Unsigned byte-wise quad maximum

quadumax

SYNTAX

```
[ IF rguard ] quadumax rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<7:0> ← **if** r*src1*<7:0> > r*src2*<7:0> **then** r*src1*<7:0> **else** r*src2*<7:0>
r*dest*<15:8> ← **if** r*src1*<15:8> > r*src2*<15:8> **then** r*src1*<15:8> **else** r*src2*<15:8>
r*dest*<23:16> ← **if** r*src1*<23:16> > r*src2*<23:16> **then** r*src1*<23:16> **else** r*src2*<23:16>
r*dest*<31:24> ← **if** r*src1*<31:24> > r*src2*<31:24> **then** r*src1*<31:24> **else** r*src2*<31:24>
}

ATTRIBUTES

Function unit	dspalu
Operation code	81
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1,3

SEE ALSO

imax imin quadumin

DESCRIPTION

The quadumax operation computes four separate maximum values of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The quadumax operation is particularly suited to implement median computation on packed pixel data structures:

```
MEDIAN_Q(a,b,c) (QUADUMIN( QUADUMAX( QUADUMIN((a),(b)), (c)), QUADUMAX((a),(b))))
```

The quadumax operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0201000e, r40 = 0xff00ff02	`quadumax r30 r40 → r50`	r50 ← 0xff01ff0e
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649d649c	`IF r10 quadumax r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649d649c	`IF r20 quadumax r60 r70 → r90`	r90 ← 0x9c9d649c

quadumin

Unsigned bytewise quad minimum

SYNTAX

```
[ IF rguard ] quadumin rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   rdest<7:0> ← if rsrc1<7:0> < rsrc2<7:0> then rsrc1<7:0> else rsrc2<7:0>
   rdest<15:8> ← if rsrc1<15:8> < rsrc2<15:8> then rsrc1<15:8> else rsrc2<15:8>
   rdest<23:16> ← if rsrc1<23:16> < rsrc2<23:16> then rsrc1<23:16> else rsrc2<23:16>
   rdest<31:24> ← if rsrc1<31:24> < rsrc2<31:24> then rsrc1<31:24> else rsrc2<31:24>
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	80
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1,3

SEE ALSO

imin imax quadumax

DESCRIPTION

The quadumin operation computes four separate minimum values of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The quadumin operation is particularly suited to implement median computation on packed pixel data structures:

```
MEDIAN_Q(a,b,c) (QUADUMIN(QUADUMAX( QUADUMIN((a),(b)), (c)), QUADUMAX((a),(b))))
```

The quadumin operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0201000e, r40 = 0xff00ff02	`quadumin r30 r40 → r50`	r50 ← 0x02000002
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649d649c	`IF r10 quadumin r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649d649c	`IF r20 quadumin r60 r70 → r90`	r90 ← 0x649c6464

Unsigned quad 8-bit multiply most significant

quadumulmsb

SYNTAX

```
[ IF rguard ] quadumulmsb rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspmul
Operation code	89
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

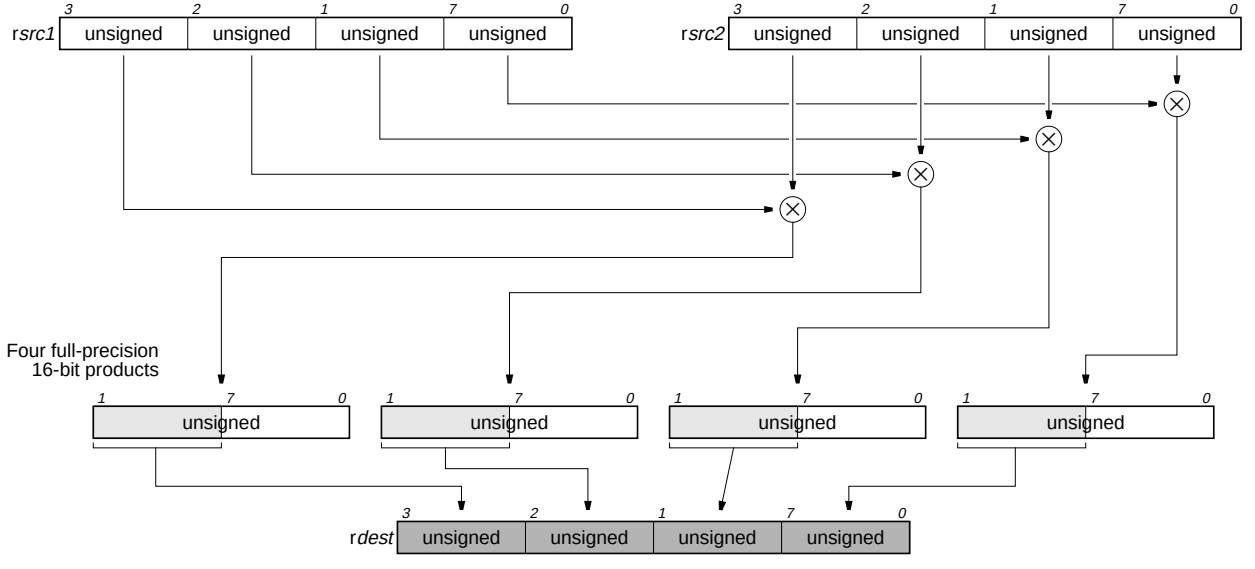
quadavg dspuquadaddui ifir8ii

FUNCTION

if r*guard* **then {**
  temp ← (zero_ext8to32(r*src1*<7:0>) × zero_ext8to32(r*src2*<7:0>))
  r*dest*<7:0> ← temp<15:8>
  temp ← (zero_ext8to32(r*src1*<15:8>) × zero_ext8to32(r*src2*<15:8>))
  r*dest*<15:8> ← temp<15:8>
  temp ← (zero_ext8to32(r*src1*<23:16>) × zero_ext8to32(r*src2*<23:16>))
  r*dest*<23:16> ← temp<15:8>
  temp ← (zero_ext8to32(r*src1*<31:24>) × zero_ext8to32(r*src2*<31:24>))
  r*dest*<31:24> ← temp<15:8>
}

DESCRIPTION

As shown below, the quadumulmsb operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The most-significant 8 bits of each 16-bit product is written to the corresponding byte in r*dest*.



The quadumulmsb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0210800e, r40 = 0xffffff02	quadumulmsb r30 r40 → r50	r50 ← 0x010f7f00
r10 = 0, r60 = 0x80ff1010, r70 = 0x80ff100f	IF r10 quadumulmsb r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x80ff1010, r70 = 0x80ff100f	IF r20 quadumulmsb r60 r70 → r90	r90 ← 0x40fe0100

rdstatus

Read data cache status bits

SYNTAX

```
[ IF rguard ] rdstatus(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   set_addr ← rsrc1 + d

      /* set_addr<10:6> selects set */

   rdest<9:0> ← dcache_LRU_set(set_addr)
   rdest<17:10> ← dcache_dirty_set(set_addr)
   rdest<31:18> ← 0
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	203
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

rdtag

DESCRIPTION

The `rdstatus` operation reads the LRU and dirty bits associated with a set in the data cache and writes these bits into the destination register r*dest*. The target set in the data cache is determined by bits 10..6 of the result of r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

The result of `rdstatus` contains LRU information in bits 9..0 and dirty-bit information in bits 17..10. All other bits of r*dest* are set to zero.

`rdstatus` requires two stall cycles to complete.

The dual-ported data cache uses two separate copies of tag and status information. A `rdstatus` operation returns the LRU and dirty information stored in the cache port that corresponds to the operation slot in which the `rdstatus` operation is issued.

The `rdstatus` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`rdstatus(0) r30 → r60`	
r10 = 0	`IF r10 rdstatus(4) r40 → r70`	no change, since guard is false
r20 = 1	`IF r20 rdstatus(8) r50 → r80`	

Read data cache address tag

rdtag

SYNTAX

```
[ IF rguard ] rdtag(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   block_addr ← rsrc1 + d

      /* block_addr<13:11> selects element, block_addr<10:6> selects set */

   rdest<21:0> ← dcache_tag_block(block_addr)
   rdest<31:22> ← 0
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	202
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

rdstatus

DESCRIPTION

The rdtag operation reads the address tag associated with a block in the data cache and writes these bits into the destination register r*dest*. The target block in the data cache is determined by bits 13..6 of the result of r*src1* + *d*. Bits 10..6 of r*src1* + *d* select the cache set and 13..11 of r*src1* + *d* select the element within that set. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

rdtag writes the address tag for the selected block in bits 21..0 of r*dest*. All other bits of r*dest* are set to zero.

rdtag requires no stall cycles to complete.

The dual-ported data cache uses two separate copies of tag and status information. A rdtag operation returns the address tag information stored in the cache port that corresponds to the operation slot in which the rdtag operation is issued.

The rdtag operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	rdtag(0) r30 → r60	
r10 = 0	IF r10 rdtag(4) r40 → r70	no change, since guard is false
r20 = 1	IF r20 rdtag(8) r50 → r80	

readdpc

Read destination program counter

SYNTAX

```
[ IF rguard ] readdpc → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← DPC
}

ATTRIBUTES

Function unit	fcomp
Operation code	156
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

writedpc readspc ijmpf ijmpi ijmpt

DESCRIPTION

The readdpc writes the current value of the DPC (Destination Program Counter) processor register to r*dest*.

Interruptible jumps write their target address to the DPC. If an interrupt or exception is taken at an interruptible jump, execution of the interrupted program can be resumed by jumping to the value contained in DPC. This operation can be used to save state before idling a task in a multi-tasking environment.

The readdpc operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
DPC = 0xbeebee	`readdpc → r100`	r100 ← 0xbeebee
r20 = 0, DPC = 0xabba	`IF r20 readdpc → r101`	no change, since guard is false
r21 = 1, DPC = 0xabba	`IF r21 readdpc → r102`	r102 ← 0xabba

Read program control and status word

readpcsw

SYNTAX

```
[ IF rguard ] readpcsw → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← PCSW
}

ATTRIBUTES

Function unit	fcomp
Operation code	158
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

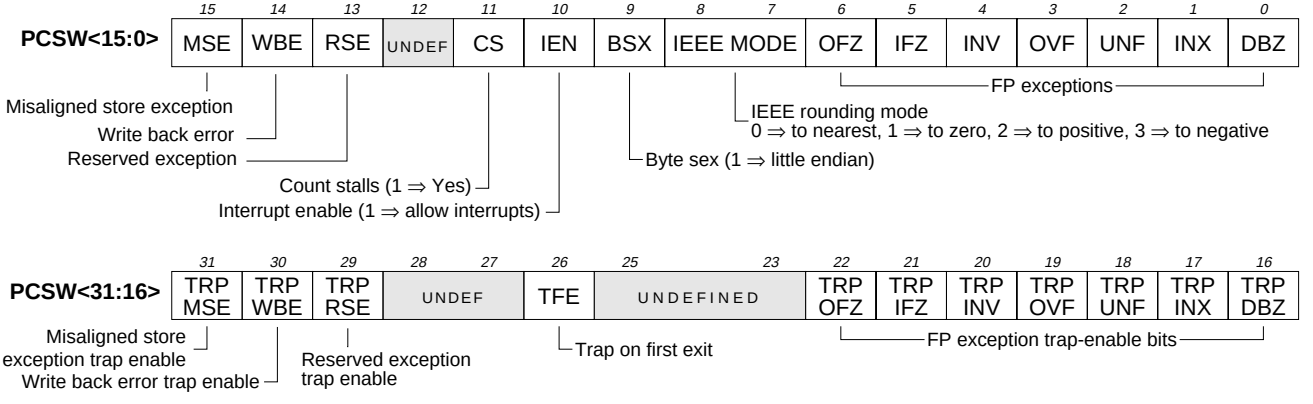
writepcsw

DESCRIPTION

The readpcsw writes the current value of the PCSW (Program Control and Status Word) processor register to r*dest*. The layout of PCSW is shown below.

Fields in the PCSW have two chief purposes: to control aspects of processor operation and to record events that occur during program execution. Thus, readpcsw can be used to determine current processor operating modes and what events have occurred; this operation can also be used to save state before idling a task in a multi-tasking environment.

The readpcsw operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.



EXAMPLES

Initial Values	Operation	Result
PCSW = 0x80110642	readpcsw → r100	r100 ← 0x80110642 (trap on MSE, INV and DBZ enabled, IEN=1 - interrupts enabled, BSX=1 - little endian mode of operation, OFZ=1 - a denormalized result was produced somewhere, INX=1 - an inexact result was produced somewhere)
r20 = 0, PCSW = 0x80000000	IF r20 readpcsw → r101	no change, since guard is false
r21 = 1, PCSW = 0x80000000	IF r21 readpcsw → r102	r102 ← 0x80000000 (trap on MSE enabled)

readspc

Read source program counter

SYNTAX

```
[ IF rguard ] readspc → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← SPC
}

ATTRIBUTES

Function unit	fcomp
Operation code	157
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

writespc readdpc ijmpf ijmpi ijmpt

DESCRIPTION

The `readspc` writes the current value of the SPC (Source Program Counter) processor register to r*dest*.

An interruptible jump that is not interrupted (no NMI, INT, or EXC event was pending when the jump was executed) writes its target address to SPC. The value of SPC allows an exception-handling routine to determine the start address of the block of scheduled code (called a decision tree) that was executing before the exception was taken.This operation can be used to save state before idling a task in a multi-tasking environment.

The `readspc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
SPC = 0xbeebee	`readspc → r100`	r100 ← 0xbeebee
r20 = 0, SPC = 0xabba	`IF r20 readspc → r101`	no change, since guard is false
r21 = 1, SPC = 0xabba	`IF r21 readspc → r102`	r102 ← 0xabba

Rotate left

rol

SYNTAX

```
[ IF rguard ] rol rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   n ← rsrc2<4:0>
   rdest<31:n> ← rsrc1<31–n:0>
   rdest<n–1:0> ← rsrc1<31:32–n>
}
```

ATTRIBUTES

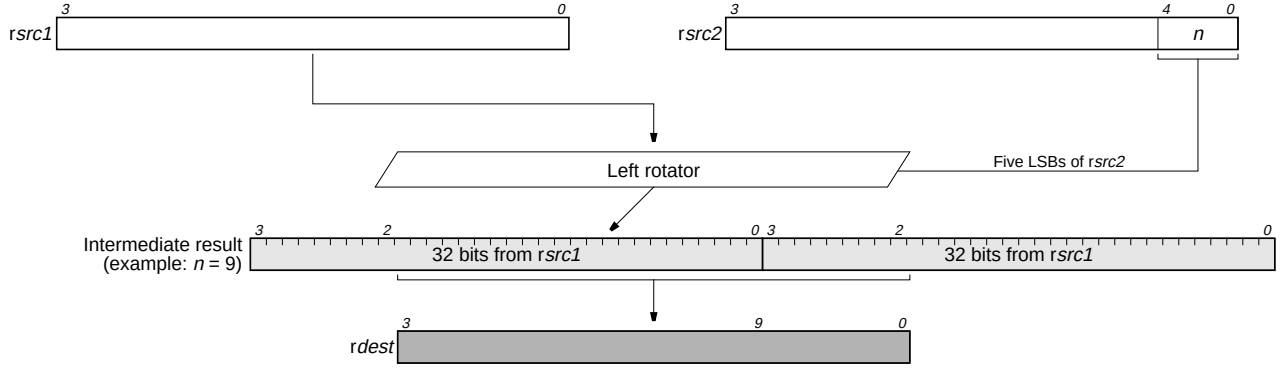
Function unit	shifter
Operation code	97
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

roli asr asri lsl lsli lsr lsri

DESCRIPTION

As shown below, the `rol` operation takes two arguments, r*src1* and r*src2*. The least-significant five bits of r*src2* specify an unsigned rotate amount, and r*dest* is set to r*src1* rotated left by this amount. The most-significant n bits of r*src1*, where n is the rotate amount, appear as the least-significant n bits in r*dest*.



The `rol` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	`rol r60 r30 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	`IF r10 rol r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	`IF r20 rol r60 r30 → r110`	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	`rol r70 r40 → r120`	r120 ← 0xfffffff3
r80 = 0xe, r50 = 0xfffffffe	`rol r80 r50 → r125`	r125 ← 0x80000003 (r50 is effectively equal to 0x1e)

roli

Rotate left by immediate

SYNTAX

```
[ IF rguard ] roli(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<31:*n*> ← r*src1*<31–*n*:0>
    r*dest*<*n*–1:0> ← r*src1*<31:32–*n*>
}

ATTRIBUTES

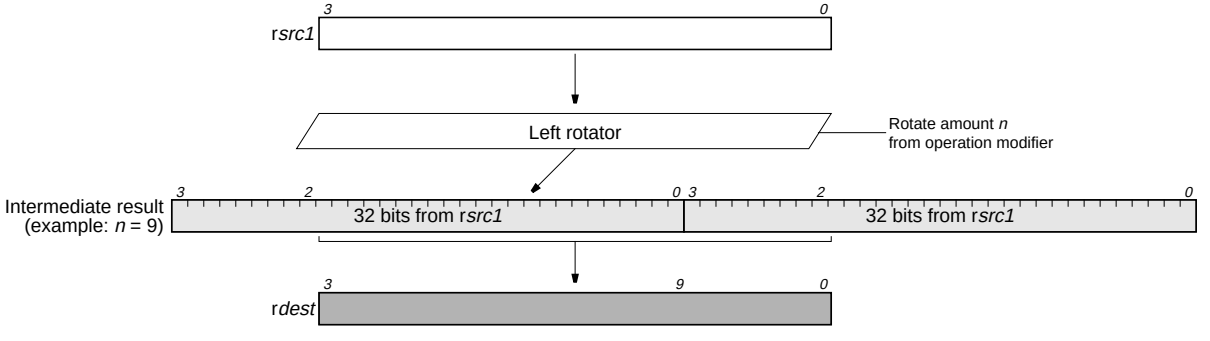
Function unit	shifter
Operation code	98
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

rol asl asli asr asri lsl lsli lsr lsri

DESCRIPTION

As shown below, the `roli` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* rotated left by *n* bits. The value of *n* must be between 0 and 31, inclusive. The most-significant *n* bits of r*src1* appear as the least-significant *n* bits in r*dest*.



The `roli` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	`roli(3) r60 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20	`IF r10 roli(3) r60 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20	`IF r20 roli(3) r60 → r110`	r110 ← 0x100
r70 = 0xfffffffc	`roli(2) r70 → r120`	r120 ← 0xfffffff3
r80 = 0xe	`roli(30) r80 → r125`	r125 ← 0x80000003

Sign extend 16 bits

sex16

SYNTAX

```
[ IF rguard ] sex16 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← sign_ext16to32(r*src1*<15:0>)

ATTRIBUTES

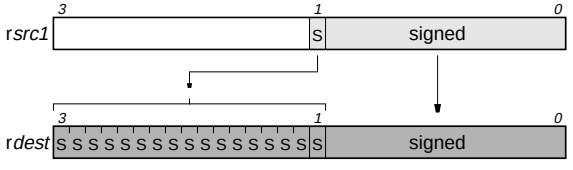
Function unit	alu
Operation code	51
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

zex16 sex8 zex8

DESCRIPTION

As shown below, the sex16 operation sign extends the least-significant 16bit halfword of the argument, r*src1*, to 32 bits and stores the result in r*dest*.



The sex16 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of the guard is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	sex16 r30 → r60	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	IF r10 sex16 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	IF r20 sex16 r40 → r100	r100 ← 0xffffff91
r50 = 0x00000091	sex16 r50 → r110	r110 ← 0x00000091

sex8

Sign extend 8 bits

pseudo-op for ibytesel

SYNTAX

```
[ IF rguard ] sex8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← sign_ext8to32(r*src1*<7:0>)

ATTRIBUTES

Function unit	alu
Operation code	56
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

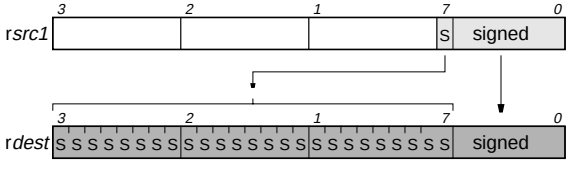
SEE ALSO

ibytesel sex16 zex8 zex16

DESCRIPTION

The sex8 operation is a pseudo operation transformed by the scheduler into a ibytesel with r*src1* as the first argument and r0 (always contains 0) as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the sex8 operation sign extends the least-significant halfword of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The sex8 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	sex8 r30 → r60	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	IF r10 sex8 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	IF r20 sex8 r40 → r100	r100 ← 0xffffff91
r50 = 0x00000091	sex8 r50 → r110	r110 ← 0xffffff91

16-bit store

pseudo-op for h_st16d(0)

st16

SYNTAX

```
[ IF rguard ] st16 rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   mem[rsrc1 + (1 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + (0 ⊕ bs)] ← rsrc2<15:8>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

st16d h_st16d st8 st8d st32 st32d

DESCRIPTION

The st16 operation is a pseudo operation transformed by the scheduler into an h_st16d(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st16 operation stores the least-significant 16-bit halfword of r*src2* into the memory locations pointed to by the address in r*src1*. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If st16 is misaligned (the memory address in r*src1* is not a multiple of 2), the result of st16 is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The result of an access by st16 to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The st16 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st16 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	st16 r10 r80	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	IF r50 st16 r20 r70	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	IF r60 st16 r30 r70	[0xd02] ← 0xcc, [0xd03] ← 0xdd

st16d

16-bit store with displacement

pseudo-op for h_st16d

SYNTAX

```
[ IF rguard ] st16d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   mem[rsrc1 + d + (1 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + d + (0 ⊕ bs)] ← rsrc2<15:8>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	n/a
Issue slots	4, 5

SEE ALSO

st16 h_st16d st8 st8d st32 st32d

DESCRIPTION

The `st16d` operation is a pseudo operation transformed by the scheduler into an `h_st16d` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st16d` operation stores the least-significant 16-bit halfword of r*src2* into the memory locations pointed to by the address in r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If `st16d` is misaligned (the memory address computed by r*src1* + *d* is not a multiple of 2), the result of `st16d` is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The result of an access by `st16d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st16d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st16d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfe, r80 = 0x44332211	`st16d(2) r10 r80`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd05, r70 = 0xaabbccdd	`IF r50 st16d(–4) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd06, r70 = 0xaabbccdd	`IF r60 st16d(–4) r30 r70`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

32-bit store

pseudo-op for h_st32d(0)

st32

SYNTAX

```
[ IF rguard ] st32 rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   mem[rsrc1 + (3 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + (2 ⊕ bs)] ← rsrc2<15:8>
   mem[rsrc1 + (1 ⊕ bs)] ← rsrc2<23:16>
   mem[rsrc1 + (0 ⊕ bs)] ← rsrc2<31:24>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	31
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st32d st32d st16 st16d st8 st8d

DESCRIPTION

The st32 operation is a pseudo operation transformed by the scheduler into an h_st32d(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st32 operation stores all 32 bits of r*src2* into the memory locations pointed to by the address in r*src1*. The *d* value is an opcode modifier and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If st32 is misaligned (the memory address in r*src1* is not a multiple of 4), the result of st32 is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The st32 operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by st32.

The st32 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st32 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	st32 r10 r80	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	IF r50 st32 r20 r70	no change, since guard is false
r60 = 1, r30 = 0xd04, r70 = 0xaabbccdd	IF r60 st32 r30 r70	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

st32d

32-bit store with displacement

pseudo-op for h_st32d

SYNTAX

```
[ IF rguard ] st32d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 3
   else
      bs ← 0
   mem[rsrc1 + d + (3 ⊕ bs)] ← rsrc2<7:0>
   mem[rsrc1 + d + (2 ⊕ bs)] ← rsrc2<15:8>
   mem[rsrc1 + d + (1 ⊕ bs)] ← rsrc2<23:16>
   mem[rsrc1 + d + (0 ⊕ bs)] ← rsrc2<31:24>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	31
Number of operands	2
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st32d st32 st16 st16d st8 st8d

DESCRIPTION

The st32d operation is a pseudo operation transformed by the scheduler into an h_st32d with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st32d operation stores all 32 bits of r*src2* into the memory locations pointed to by the address in r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 and 252 inclusive, and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If st32d is misaligned (the memory address computed by r*src1* + *d* is not a multiple of 4), the result of st32d is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The st32d operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by st32d.

The st32d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st32d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r80 = 0x44332211	`st32d(4) r10 r80`	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd0b, r70 = 0xaabbccdd	`IF r50 st32d(-8) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd0c, r70 = 0xaabbccdd	`IF r60 st32d(-8) r30 r70`	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

8-bit store

pseudo-op for h_st8d(0)

st8

SYNTAX

```
[ IF rguard ] st8 rsrc1 rsrc2
```

FUNCTION

if r*guard* **then**
    mem[r*src1*] ← r*src2*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st8d st8d st16 st16d st32 st32d

DESCRIPTION

The st8 operation is a pseudo operation transformed by the scheduler into an h_st8d(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st8 operation stores the least-significant 8-bit byte of r*src2* into the memory location pointed to by the address in r*src1*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The result of an access by st8 to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The st8 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st8 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	st8 r10 r80	[0xd00] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	IF r50 st8 r20 r70	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	IF r60 st8 r30 r70	[0xd02] ← 0xdd

st8d

8-bit store with displacement

pseudo-op for h_st8d

SYNTAX

```
[ IF rguard ] st8d(d) rsrc1 rsrc2
```

FUNCTION

if r*guard* **then**
    mem[r*src1* + *d*] ← r*src2*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	7 bits
Modifier range	–64..63
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st8d st8 st16 st16d st32 st32d

DESCRIPTION

The st8d operation is a pseudo operation transformed by the scheduler into an h_st8d with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The st8d operation stores the least-significant 8-bit byte of r*src2* into the memory location pointed to by the address formed from the sum r*src1* + *d*. The value of the opcode modifier *d* must be in the range -64 and 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The result of an access by st8d to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The st8d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st8d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st8d(3) r10 r80`	[0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st8d(-4) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st8d(-4) r30 r70`	[0xcfe] ← 0xdd

Select unsigned byte

ubytesel

SYNTAX

```
[ IF rguard ] ubytesel rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	alu
Operation code	55
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

FUNCTION

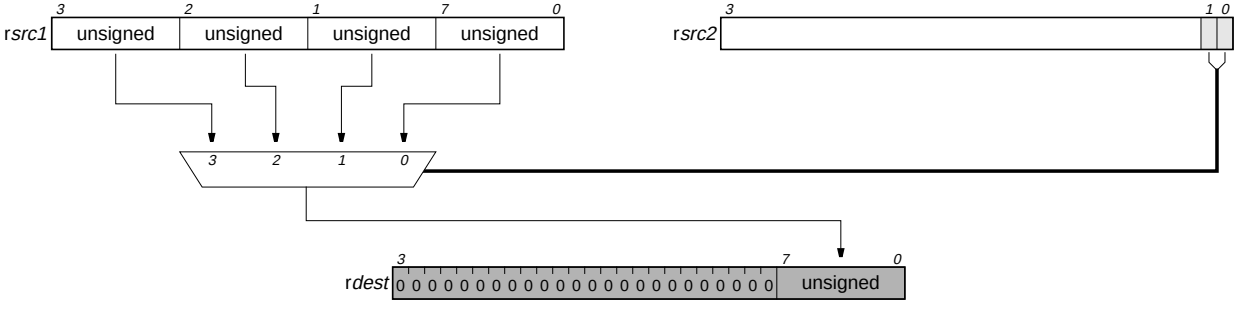
```
if rguard then {
   if rsrc2 = 0 then
      rdest ← zero_ext8to32(rsrc1<7:0>)
   else if rsrc2 = 1 then
      rdest ← zero_ext8to32(rsrc1<15:8>)
   else if rsrc2 = 2 then
      rdest ← zero_ext8to32(rsrc1<23:15>)
   else if rsrc2 = 3 then
      rdest ← zero_ext8to32(rsrc1<31:24>)
}
```

SEE ALSO

ibytesel sex8 packbytes

DESCRIPTION

As shown below, the ubytesel operation selects one byte from the argument, r*src1*, zero-extends the byte to 32 bits, and stores the result in r*dest*. The value of r*src2* determines which byte is selected, with r*src2*=0 selecting the LSB of r*src1* and r*src2*=3 selecting the MSB of r*src1*. If rsrc2 is not between 0 and 3 inclusive, the result of ubytesel is undefined.



The ubytesel operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x44332211, r40 = 1	ubytesel r30 r40 → r50	r50 ← 0x00000022
r10 = 0, r60 = 0xddccbbaa, r70 = 2	IF r10 ubytesel r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0xddccbbaa, r70 = 2	IF r20 ubytesel r60 r70 → r90	r90 ← 0x000000cc
r100 = 0xffffff7f, r110 = 0	ubytesel r100 r110 → r120	r120 ← 0x0000007f

uclipi

Clip signed to unsigned

SYNTAX

```
[ IF rguard ] uclipi rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← min(max(r*src1*, 0), r*src2*)

ATTRIBUTES

Function unit	dspalu
Operation code	75
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iclipi uclipu imin imax

DESCRIPTION

The `uclipi` operation returns the value of r*src1* clipped into the unsigned integer range 0 to r*src2*, inclusive. The argument r*src1* is considered a signed integer; r*src2* is considered an unsigned integer.

The `uclipi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`uclipi r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 uclipi r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 uclipi r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`uclipi r100 r110 → r120`	r120 ← 0

Clip unsigned to unsigned

uclipu

SYNTAX

```
[ IF rguard ] uclipu rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > rsrc2 then
        rdest ← rsrc2
    else
        rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	76
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iclipi uclipi imin imax

DESCRIPTION

The `uclipu` operation returns the value of r*src1* clipped into the unsigned integer range 0 to r*src2*, inclusive. The arguments r*src1* and r*src2* are considered unsigned integers.

The `uclipu` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`uclipu r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 uclipu r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 uclipu r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`uclipu r100 r110 → r120`	r120 ← 0x3fffff

ueql

Unsigned compare equal

pseudo-op for ieql

SYNTAX

```
[ IF rguard ] ueql rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 = rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	37
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieql ueqli igeq uneq

DESCRIPTION

The `ueql` operation is a pseudo operation transformed by the scheduler into an `ieql` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `ueql` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ueql` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ueql r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ueql r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 ueql r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ueql r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`ueql r70 r70 → r110`	r110 ← 1

Unsigned compare equal with immediate

ueqli

SYNTAX

```
[ IF rguard ] ueqli(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 = n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	38
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieqli ueql igeqi uneqi

DESCRIPTION

The `ueqli` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ueqli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ueqli(2) r30 → r80`	r80 ← 0
r30 = 3	`ueqli(3) r30 → r90`	r90 ← 1
r30 = 3	`ueqli(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ueqli(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ueqli(63) r40 → r100`	r100 ← 0
r60 = 0x07f	`ueqli(127) r60 → r120`	r120 ← 1

ufir16 Sum of products of unsigned 16-bit halfwords

SYNTAX

```
[ IF rguard ] ufir16 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← zero_ext16to32(r*src1*<31:16>) × zero_ext16to32(r*src2*<31:16>) +
zero_ext16to32(r*src1*<15:0>) × zero_ext16to32(r*src2*<15:0>)

ATTRIBUTES

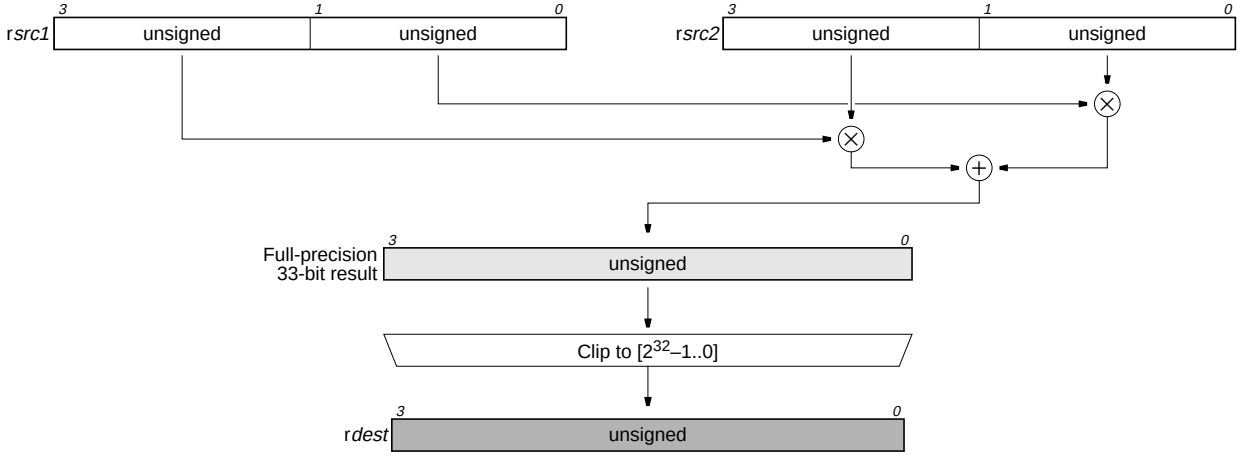
Function unit	dspmul
Operation code	94
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir16 ifir8ii ifir8ui
ufir8uu

DESCRIPTION

As shown below, the ufir16 operation computes two separate products of the two pairs of corresponding 16-bit halfwords of r*src1* and r*src2*; the two products are summed, and the result is written to r*dest*. All halfwords are considered unsigned; thus, the intermediate products and the final sum of products are unsigned. All intermediate computations are performed without loss of precision; the final sum of products is clipped into the range [0xffffffff..0] before being written into r*dest*.



The ufir16 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00020003, r40 = 0x00010002	ufir16 r30 r40 → r50	r50 ← 8
r10 = 0, r60 = 0x80000064, r70 = 0x00648000	IF r10 ufir16 r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x80000064, r70 = 0x00648000	IF r20 ufir16 r60 r70 → r90	r90 ← 0x00640000
r30 = 0x00020003, r70 = 0x00648000	ufir16 r30 r70 → r100	r100 ← 0x000180c8

Unsigned sum of products of unsigned bytes

ufir8uu

SYNTAX

```
[ IF rguard ] ufir8uu rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspmul
Operation code	90
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

FUNCTION

if r*guard* **then**

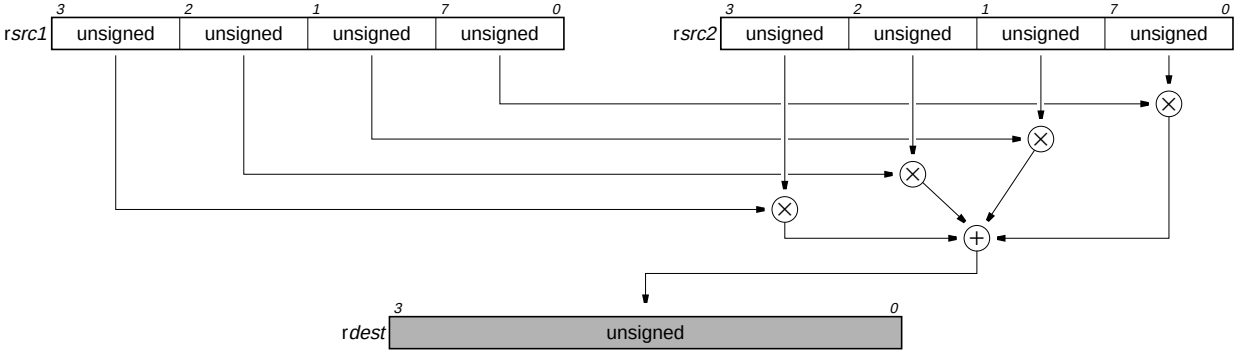
r*dest* ← zero_ext8to32(r*src1*<31:24>) × zero_ext8to32(r*src2*<31:24>) +
zero_ext8to32(r*src1*<23:16>) × zero_ext8to32(r*src2*<23:16>) +
zero_ext8to32(r*src1*<15:8>) × zero_ext8to32(r*src2*<15:8>) +
zero_ext8to32(r*src1*<7:0>) × zero_ext8to32(r*src2*<7:0>)

SEE ALSO

ifir8ui ifir8ii ifir16
ufir16

DESCRIPTION

As shown below, the `ufir8uu` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. All values are considered unsigned. All computations are performed without loss of precision.



The `ufir8uu` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ufir8uu r70 r30 → r90`	r90 ← 0x1efa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ufir8uu r70 r30 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ufir8uu r80 r40 → r110`	r110 ← 0xf3c0
r50 = 0x80808080, r60 = 0xffffffff	`ufir8uu r50 r60 → r120`	r120 ← 0x1fe00

ufixieee

Convert floating-point to unsigned integer using PCSW rounding mode

SYNTAX

```
[ IF rguard ] ufixieee rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← (unsigned long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	123
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifixieee ifixrz ufixrz

DESCRIPTION

The `ufixieee` operation converts the single-precision IEEE floating-point value in r*src1* to an unsigned integer and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ufixieee` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufixieeeflags` operation computes the exception flags that would result from an individual `ufixieee`.

The `ufixieee` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixieee r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ufixieee r35 → r102`	r102 ← 3, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixieee r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixieee r40 → r110`	r110 ← 0x0, INV flag set
r45 = 0x7f800000 (+INF))	`ufixieee r45 → r112`	r112 ← 0xffffffff (2^{32}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ufixieee r50 → r115`	r115 ← 0, INV flag set
r60 = 0x00400000 (5.877471754e-39)	`ufixieee r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ufixieee r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ufixieee r80 → r122`	r122 ← 0, INV flag set

IEEE status flags from convert floating-point to unsigned integer using PCSW rounding mode

ufixieeeflags

SYNTAX

```
[ IF rguard ] ufixieeeflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((unsigned long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	124
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

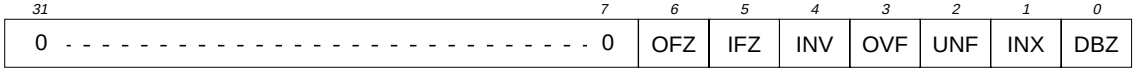
SEE ALSO

ufixieee ifixieeeflags ifixrzflags ufixrzflags

DESCRIPTION

The `ufixieeeflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to an unsigned integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ufixieeeflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixieeeflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ufixieeeflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixieeeflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixieeeflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ufixieeeflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ufixieeeflags r50 → r115`	r115 ← 0x10 (INV)
r60 = 0x00400000 (5.877471754e-39)	`ufixieeeflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ufixieeeflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ufixieeeflags r80 → r122`	r122 ← 0x10 (INV)

ufixrz

Convert floating-point to unsigned integer with round toward zero

SYNTAX

```
[ IF rguard ] ufixrz rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← (unsigned long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	125
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifixieee ufixieee ifixrz

DESCRIPTION

The `ufixrz` operation converts the single-precision IEEE floating-point value in r*src1* to an unsigned integer and writes the result into r*dest*. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ufixrz` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufixrzflags` operation computes the exception flags that would result from an individual `ufixrz`.

The `ufixrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixrz r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ufixrz r35 → r102`	r102 ← 2, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixrz r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixrz r40 → r110`	r110 ← 0x0, INV flag set
r45 = 0x7f800000 (+INF))	`ufixrz r45 → r112`	r112 ← 0xffffffff (2^{32}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ufixrz r50 → r115`	r115 ← 0, INV flag set
r60 = 0x00400000 (5.877471754e-39)	`ufixrz r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ufixrz r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ufixrz r80 → r122`	r122 ← 0, INV flag set

IEEE status flags from convert floating-point to unsigned integer with round toward zero

ufixrzflags

SYNTAX

```
[ IF rguard ] ufixrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((unsigned long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	126
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

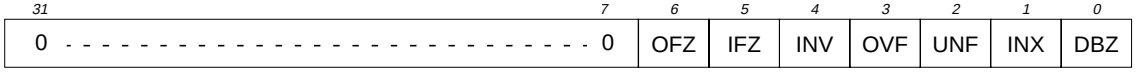
SEE ALSO

ufixrz ifixrzflags
ifixieeeflags
ufixieeeflags

DESCRIPTION

The `ufixrzflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to an unsigned integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. If an argument is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ufixrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixrzflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ufixrzflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixrzflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixrzflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ufixrzflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ufixrzflags r50 → r115`	r115 ← 0x10 (INV)
r60 = 0x00400000 (5.877471754e-39)	`ufixrzflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ufixrzflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ufixrzflags r80 → r122`	r122 ← 0x10 (INV)

ufloat

Convert unsigned integer to floating-point

SYNTAX

```
[ IF rguard ] ufloat rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← (float) ((unsigned long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	127
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifloat ifloatrz ufloatrz ifixieee ufloatflags

DESCRIPTION

The `ufloat` operation converts the unsigned integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If `ufloat` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufloatflags` operation computes the exception flags that would result from an individual `ufloat`.

The `ufloat` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloat r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (4294967295)	`ufloat r40 → r105`	r105 ← 0x4f800000 (4.294967296e+9), INX flag set
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloat r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloat r50 → r115`	r115 ← 0x4f800000 (4.294967296e+9), INX flag set
r60 = 0x7fffffff (2147483647)	`ufloat r60 → r117`	r117 ← 0x4f000000 (2.147483648e+9), INX flag set
r70 = 0x80000000 (2147483648)	`ufloat r70 → r120`	r120 ← 0x4f000000 (2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ufloat r80 → r122`	r122 ← 0x4f000000 (2.147483648e+9), INX flag set

IEEE status flags from convert unsigned integer to floating-point

ufloatflags

SYNTAX

```
[ IF rguard ] ufloatflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((float) ((unsigned long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	128
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

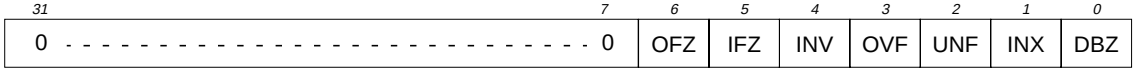
SEE ALSO

ufloat ifloatflags
ifloatrzflags
ufloatrzflags

DESCRIPTION

The `ufloatflags` operation computes the IEEE exceptions that would result from converting the unsigned integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW.

The `ufloatflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (4294967295)	`ufloatflags r40 → r105`	r105 ← 0x02 (INX)
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x7fffffff (2147483647)	`ufloatflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (2147483648)	`ufloatflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ufloatflags r80 → r122`	r122 ← 0x02 (INX)

ufloatrz

Convert unsigned integer to floating-point with rounding toward zero

SYNTAX

```
[ IF rguard ] ufloatrz rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest* ← (float) ((unsigned long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	119
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifloatrz ifloat ufloat ifixieee ufloatflags

DESCRIPTION

The `ufloatrz` operation converts the unsigned integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If `ufloatrz` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufloatrzflags` operation computes the exception flags that would result from an individual `ufloatrz`.

The `ufloatrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatrz r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (4294967295)	`ufloatrz r40 → r105`	r105 ← 0x4f7fffff (4.294967040e+9), INX flag set
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatrz r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatrz r50 → r115`	r115 ← 0x4f7fffff (4.294967040e+9), INX flag set
r60 = 0x7fffffff (2147483647)	`ufloatrz r60 → r117`	r117 ← 0x4effffff (2.147483520e+9), INX flag set
r70 = 0x80000000 (2147483648)	`ufloatrz r70 → r120`	r120 ← 0x4f000000 (2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ufloatrz r80 → r122`	r122 ← 0x4effffff (2.147483520e+9), INX flag set

IEEE status flags from convert unsigned integer to floating-point with rounding toward zero

ufloatrzflags

SYNTAX

```
[ IF rguard ] ufloatrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((float) ((unsigned long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	120
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

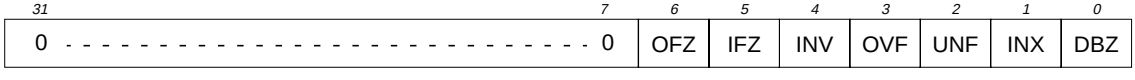
SEE ALSO

ufloatrz ifloatflags
ufloatflags ifloatrzflags

DESCRIPTION

The `ufloatrzflags` operation computes the IEEE exceptions that would result from converting the unsigned integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored.

The `ufloatrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatrzflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (4294967295)	`ufloatrzflags r40 → r105`	r105 ← 0x02 (INX)
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatrzflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatrzflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x7fffffff (2147483647)	`ufloatrzflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (2147483648)	`ufloatrzflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ufloatrzflags r80 → r122`	r122 ← 0x02 (INX)

ugeq

Unsigned compare greater or equal

SYNTAX

```
[ IF rguard ] ugeq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 >= (unsigned)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	35
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq ugeqi

DESCRIPTION

The ugeq operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The ugeq operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ugeq r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ugeq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 ugeq r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ugeq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ugeq r70 r70 → r110`	r110 ← 1

Unsigned compare greater or equal with immediate

ugeqi

SYNTAX

```
[ IF rguard ] ugeqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 >= (unsigned)n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	36
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ugeq igeqi

DESCRIPTION

The ugeqi operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The ugeqi operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ugeqi(2) r30 → r80`	r80 ← 1
r30 = 3	`ugeqi(3) r30 → r90`	r90 ← 1
r30 = 3	`ugeqi(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ugeqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ugeqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`ugeqi(127) r60 → r120`	r120 ← 1

ugtr

Unsigned compare greater

SYNTAX

```
[ IF rguard ] ugtr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 > (unsigned)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr ugtri

DESCRIPTION

The ugtr operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The ugtr operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ugtr r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ugtr r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 ugtr r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ugtr r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ugtr r70 r70 → r110`	r110 ← 0

Unsigned compare greater with immediate

ugtri

SYNTAX

```
[ IF rguard ] ugtri(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 > (unsigned)n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	34
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtri ugtr

DESCRIPTION

The `ugeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ugeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ugtri(2) r30 → r80`	r80 ← 1
r30 = 3	`ugtri(3) r30 → r90`	r90 ← 0
r30 = 3	`ugtri(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ugtri(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ugtri(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`ugtri(127) r60 → r120`	r120 ← 1

uimm

Unsigned immediate

SYNTAX

`uimm(`*n*`)` → r*dest*

FUNCTION

r*dest* ← *n*

ATTRIBUTES

Function unit	const
Operation code	191
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iimm

DESCRIPTION

The `uimm` operation writes the unsigned 32-bit opcode modifier *n* into r*dest*. Note: this operation is not guarded.

EXAMPLES

Initial Values	Operation	Result
	`uimm(2) → r10`	r10 ← 2
	`uimm(0x100) → r20`	r20 ← 0x100
	`uimm(0xfffc0000) → r30`	r30 ← 0xfffc0000

Unsigned 16-bit load

pseudo-op for uld16d(0)

uld16

SYNTAX

```
[ IF rguard ] uld16 rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[rsrc1 + (1 ⊕ bs)]
   temp<15:8> ← mem[rsrc1 + (0 ⊕ bs)]
   rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	197
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld16d ild16 ild16d uld16r ild16r uld16x ild16x

DESCRIPTION

The `uld16` operation is a pseudo operation transformed by the scheduler into an `uld16d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `uld16` operation loads the 16-bit memory value from the address contained in r*src1*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 2, the result of `uld16` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22, [0xd01] = 0x11	`uld16 r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r30 uld16 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r40 uld16 r20 → r80`	r80 ← 0x00008433
r50 = 0xd01	`uld16 r50 → r90`	r90 undefined (0xd01 is not a multiple of 2)

uld16d

Unsigned 16-bit load with displacement

SYNTAX

```
[ IF rguard ] uld16d(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[rsrc1 + d + (1 ⊕ bs)]
   temp<15:8> ← mem[rsrc1 + d + (0 ⊕ bs)]
   rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	197
Number of operands	1
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	3
Issue slots	4, 5

SEE ALSO

uld16 ild16 ild16d uld16r ild16r uld16x ild16x

DESCRIPTION

The `uld16d` operation loads the 16-bit memory value from the address computed by r*src1* + *d*, zero extends it to 32 bits, and writes the result in r*dest*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. If the memory address computed by r*src1* + *d* is not a multiple of 2, the result of `uld16d` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22, [0xd03] = 0x11	`uld16d(2) r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r30 uld16d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r40 uld16d(-4) r20 → r80`	r80 ← 0x00008433
r50 = 0xd01	`uld16d(-4) r50 → r90`	r90 undefined (0xd01 +(–4) is not a multiple of 2)

Unsigned 16-bit load with index

uld16r

SYNTAX

```
[ IF rguard ] uld16r rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[rsrc1 + rsrc2 + (1 ⊕ bs)]
   temp<15:8> ← mem[rsrc1 + rsrc2 + (0 ⊕ bs)]
   rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	198
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld16 ild16 uld16d ild16d ild16r uld16x ild16x

DESCRIPTION

The `uld16r` operation loads the 16-bit memory value from the address computed by r*src1* + r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 2, the result of `uld16r` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22, [0xd03] = 0x11	`uld16r r10 r20 → r80`	r80 ← 0x00002211
r50 = 0, r40 = 0xd04, r30 = 0xffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 uld16r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 uld16r r40 r30 → r100`	r100 ← 0x00008433
r70 = 0xd01, r30 = 0xffffffc	`uld16r r70 r30 → r110`	r110 undefined (0xd01 +(–4) is not a multiple of 2)

uld16x

Unsigned 16-bit load with scaled index

SYNTAX

```
[ IF rguard ] uld16x rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if PCSW.bytesex = LITTLE_ENDIAN then
      bs ← 1
   else
      bs ← 0
   temp<7:0> ← mem[rsrc1 + (2 × rsrc2) + (1 ⊕ bs)]
   temp<15:8> ← mem[rsrc1 + (2 × rsrc2) + (0 ⊕ bs)]
   rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	199
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld16 ild16 uld16d ild16d uld16r ild16r ild16x

DESCRIPTION

The `uld16x` operation loads the 16-bit memory value from the address computed by r*src1* + 2×r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address computed by r*src1* + 2×r*src2* is not a multiple of 2, the result of `uld16x` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16x` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16x` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16x` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r30 = 1, [0xd02] = 0x22, [0xd03] = 0x11	`uld16x r10 r30 → r100`	r100 ← 0x00002211
r50 = 0, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 uld16x r40 r20 → r80`	no change, since guard is false
r60 = 1, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 uld16x r40 r20 → r90`	r90 ← 0x00008433
r70 = 0xd01, r30 = 1	`uld16x r70 r30 → r110`	r110 undefined (0xd01 + 2×1 is not a multiple of 2)

Unsigned 8-bit load

pseudo-op for uld8d(0)

uld8

SYNTAX

```
[ IF rguard ] uld8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext8to32(mem[r*src1*])

ATTRIBUTES

Function unit	dmem
Operation code	8
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild8 uld8d ild8d uld8r ild8r

DESCRIPTION

The `uld8` operation is a pseudo operation transformed by the scheduler into an `uld8d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `uld8` operation loads the 8-bit memory value from the address contained in r*src1*, zero extends it to 32 bits, and writes the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `uld8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld8` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22	`uld8 r10 → r60`	r60 ← 0x00000022
r30 = 0, r20 = 0xd04, [0xd04] = 0x84	`IF r30 uld8 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84	`IF r40 uld8 r20 → r80`	r80 ← 0x00000084
r50 = 0xd01, [0xd01] = 0x33	`uld8 r50 → r90`	r90 ← 0x00000033

uld8d

Unsigned 8-bit load with displacement

SYNTAX

```
[ IF rguard ] uld8d(d) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext8to32(mem[r*src1* + *d*])

ATTRIBUTES

Function unit	dmem
Operation code	8
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	3
Issue slots	4, 5

SEE ALSO

uld8 ild8 ild8d uld8r ild8r

DESCRIPTION

The uld8d operation loads the 8-bit memory value from the address computed by r*src1* + *d*, zero extends it to 32 bits, and writes the result in r*dest*. The *d* value is an opcode modifier in the range –64 to 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by uld8d to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The uld8d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and uld8d has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22	`uld8d(2) r10 → r60`	r60 ← 0x000022
r30 = 0, r20 = 0xd04, [0xd00] = 0x84	`IF r30 uld8d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84	`IF r40 uld8d(-4) r20 → r80`	r80 ← 0x00000084
r50 = 0xd05, [0xd01] = 0x33	`uld8d(-4) r50 → r90`	r90 ← 0x00000033

Unsigned 8-bit load with index

uld8r

SYNTAX

```
[ IF rguard ] uld8r rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext8to32(mem[r*src1* + r*src2*])

ATTRIBUTES

Function unit	dmem
Operation code	194
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld8 ild8 uld8d ild8d ild8r

DESCRIPTION

The `uld8r` operation loads the 8-bit memory value from the address computed by r*src1* + r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `uld8r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld8r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld8r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22	`uld8r r10 r20 → r80`	r80 ← 0x00000022
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r50 uld8r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r60 uld8r r40 r30 → r100`	r100 ← 0x00000084
r70 = 0xd05, r30 = 0xfffffffc, [0xd01] = 0x33	`uld8r r70 r30 → r110`	r110 ← 0x00000033

uleq

Unsigned compare less or equal

pseudo-op for ugeq

SYNTAX

```
[ IF rguard ] uleq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 <= (unsigned)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	35
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ileq uleqi

DESCRIPTION

The uleq operation is a pseudo operation transformed by the scheduler into an ugeq with the arguments exchanged (uleq's r*src1* is ugeq's r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The uleq operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The uleq operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`uleq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 uleq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 uleq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`uleq r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`uleq r70 r70 → r110`	r110 ← 1

Unsigned compare less or equal with immediate

uleqi

SYNTAX

```
[ IF rguard ] uleqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 <= (unsigned)n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	43
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

uleq ileqi

DESCRIPTION

The `uleqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `uleqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`uleqi(2) r30 → r80`	r80 ← 0
r30 = 3	`uleqi(3) r30 → r90`	r90 ← 1
r30 = 3	`uleqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 uleqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 uleqi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`uleqi(127) r60 → r120`	r120 ← 0

ules

Unsigned compare less

pseudo-op for ugtr

SYNTAX

```
[ IF rguard ] ules rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 < (unsigned)rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iles ugtr

DESCRIPTION

The `ules` operation is a pseudo operation transformed by the scheduler into an `ugtr` with the arguments exchanged (`ules`'s r*src1* is `ugtr`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `ules` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ules` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ules r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ules r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 ules r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ules r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`ules r70 r70 → r110`	r110 ← 0

Unsigned compare less with immediate

ulesi

SYNTAX

```
[ IF rguard ] ulesi(n) rsrc1 → rdest
```

ATTRIBUTES

Function unit	alu
Operation code	41
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 < (unsigned)n then
      rdest ← 1
   else
      rdest ← 0
}
```

SEE ALSO

ules ilesi

DESCRIPTION

The `ulesi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ulesi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ulesi(2) r30 → r80`	r80 ← 0
r30 = 3	`ulesi(3) r30 → r90`	r90 ← 0
r30 = 3	`ulesi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ulesi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ulesi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`ulesi(127) r60 → r120`	r120 ← 0

ume8ii

Unsigned sum of absolute values of signed 8-bit differences

SYNTAX

```
[ IF rguard ] ume8ii rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**

r*dest* ← abs_val(sign_ext8to32(r*src1*<31:24>) – sign_ext8to32(r*src2*<31:24>)) +
abs_val(sign_ext8to32(r*src1*<23:16>) – sign_ext8to32(r*src2*<23:16>)) +
abs_val(sign_ext8to32(r*src1*<15:8>) – sign_ext8to32(r*src2*<15:8>)) +
abs_val(sign_ext8to32(r*src1*<7:0>) – sign_ext8to32(r*src2*<7:0>))

ATTRIBUTES

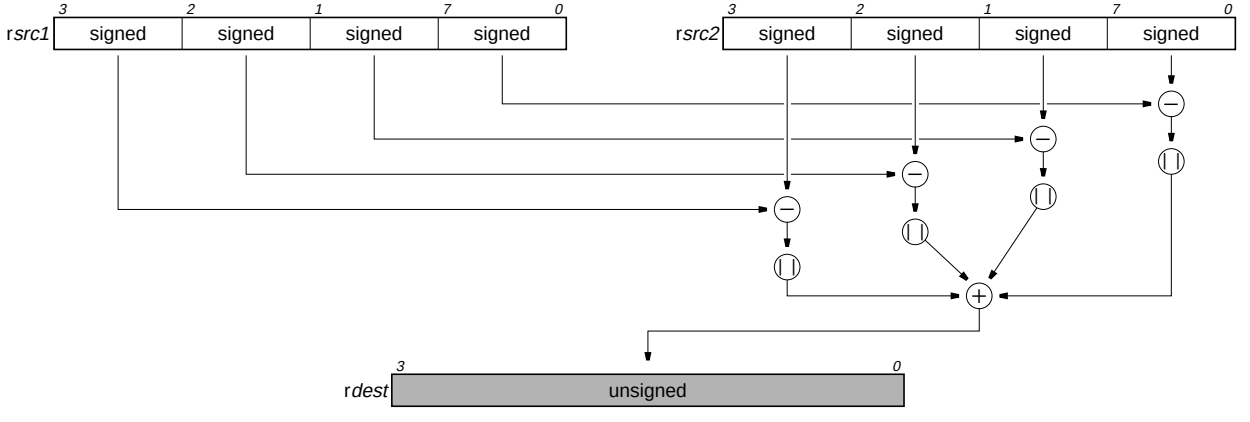
Function unit	dspalu
Operation code	64
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

ume8uu

DESCRIPTION

As shown below, the ume8ii operation computes four separate differences of the four pairs of corresponding signed 8-bit bytes of r*src1* and r*src2*; the absolute values of the four differences are summed, and the sum is written to r*dest*. All computations are performed without loss of precision.



The ume8ii operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`ume8ii r80 r30 → r100`	r100 ← 0x14
r10 = 0, r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`IF r10 ume8ii r80 r30 → r70`	no change, since guard is false
r20 = 1, r90 = 0x64649c9c, r40 = 0x649c649c	`IF r20 ume8ii r90 r40 → r110`	r110 ← 0x190
r40 = 0x649c649c, r90 = 0x64649c9c	`ume8ii r40 r90 → r120`	r120 ← 0x190
r50 = 0x80808080, r60 = 0x7f7f7f7f	`ume8ii r50 r60 → r125`	r125 ← 0x3fc

Sum of absolute values of unsigned 8-bit differences

ume8uu

SYNTAX

```
[ IF rguard ] ume8uu rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspalu
Operation code	26
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

FUNCTION

if r*guard* **then**

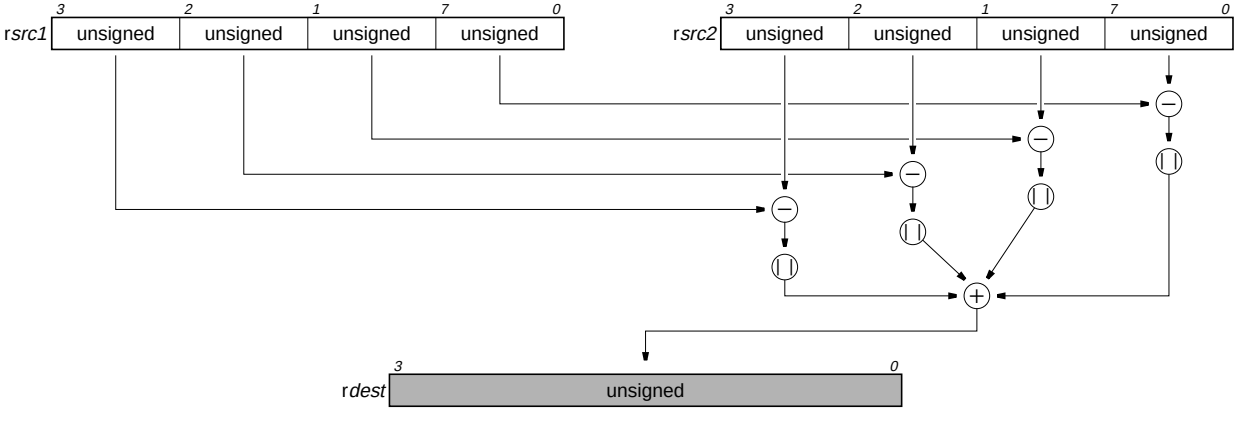
r*dest* ← abs_val(zero_ext8to32(r*src1*<31:24>) – zero_ext8to32(r*src2*<31:24>)) +
abs_val(zero_ext8to32(r*src1*<23:16>) – zero_ext8to32(r*src2*<23:16>)) +
abs_val(zero_ext8to32(r*src1*<15:8>) – zero_ext8to32(r*src2*<15:8>)) +
abs_val(zero_ext8to32(r*src1*<7:0>) – zero_ext8to32(r*src2*<7:0>))

SEE ALSO

ume8ii

DESCRIPTION

As shown below, the ume8uu operation computes four separate differences of the four pairs of corresponding unsigned 8-bit bytes of r*src1* and r*src2*. The absolute values of the four differences are summed and the result is written to r*dest*. All computations are performed without loss of precision.



The ume8uu operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`ume8uu r80 r30 → r100`	r100 ← 0x14
r10 = 0, r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`IF r10 ume8uu r80 r30 → r70`	no change, since guard is false
r20 = 1, r90 = 0x64649c9c, r40 = 0x649c649c	`IF r20 ume8uu r90 r40 → r110`	r110 ← 0x70
r40 = 0x649c649c, r90 = 0x64649c9c	`ume8uu r40 r90 → r120`	r120 ← 0x70
r50 = 0x80808080, r60 = 0x7f7f7f7f	`ume8uu r50 r60 → r125`	r125 ← 0x4

umin

Minimum of unsigned values

pseudo-op for uclipu

SYNTAX

```
[ IF rguard ] umin rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > rsrc2 then
        rdest ← rsrc2
    else
        rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	76
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iclipi uclipi imin imax

DESCRIPTION

The `umin` operation returns the minimum value of r*src1* and r*src2*. The arguments r*src1* and r*src2* are considered unsigned integers.

The `umin` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`umin r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 umin r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 umin r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`umin r100 r110 → r120`	r120 ← 0x3fffff

Unsigned multiply

umul

SYNTAX

```
[ IF rguard ] umul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  temp ← zero_ext32to64(r*src1*) × zero_ext32to64(r*src2*)
  r*dest* ← temp<31:0>

ATTRIBUTES

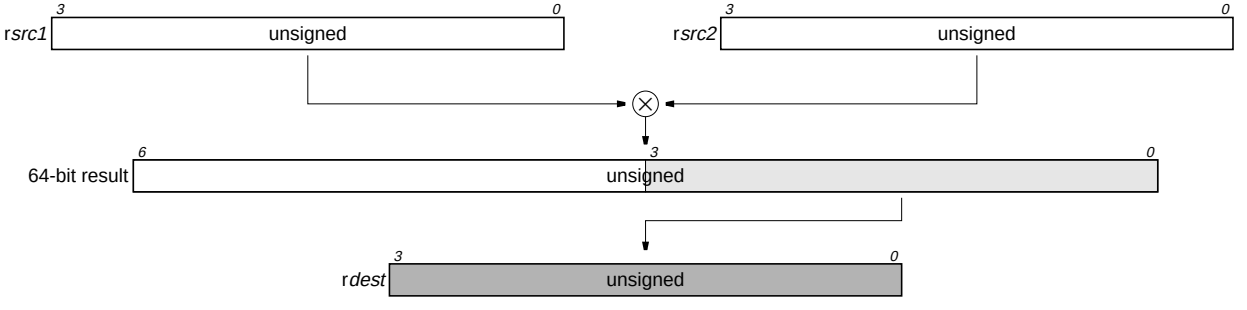
Function unit	ifmul
Operation code	138
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

imul imulm umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the umul operation computes the product r*src1*×r*src2* and writes the least-significant 32 bits of the full 64-bit product into r*dest*. The operands are considered unsigned integers. No overflow or underflow detection is performed.



The umul operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	umul r60 r60 → r80	r80 ← 0x10000
r10 = 0, r60 = 0x100, r30 = 0xf11	IF r10 umul r60 r30 → r50	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	IF r20 umul r60 r30 → r90	r90 ← 0xf1100
r70 = 0x100, r40 = 0xfffff9c	umul r70 r40 → r100	r100 ← 0xffff9c00

umulm

Unsigned multiply, return most-significant 32 bits

SYNTAX

```
[ IF rguard ] umulm rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  temp ← zero_ext32to64(r*src1*) × zero_ext32to64(r*src2*)
  r*dest* ← temp<63:32>

ATTRIBUTES

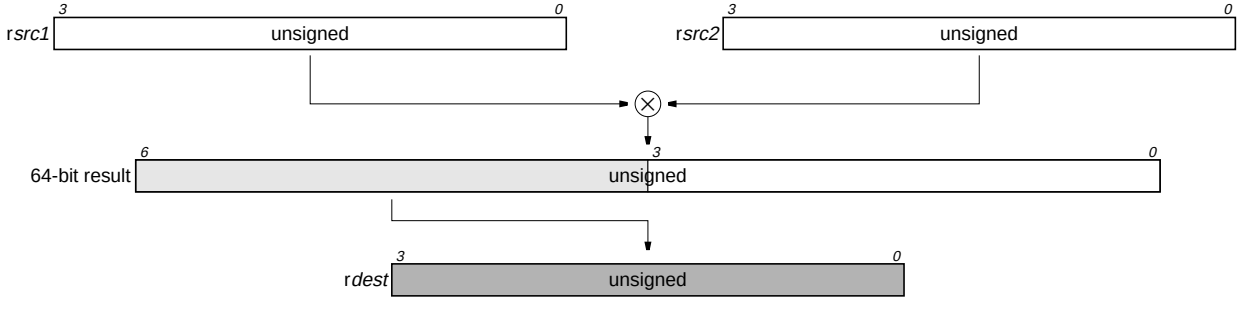
Function unit	ifmul
Operation code	140
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the umulm operation computes the product r*src1*×r*src2* and writes the most-significant 32 bits of the 64-bit product into r*dest*. The operands are considered unsigned integers.



The umulm operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10000	umulm r60 r60 → r80	r80 ← 0x00000001
r10 = 0, r60 = 0x100, r30 = 0xf11	IF r10 umulm r60 r30 → r50	no change, since guard is false
r20 = 1, r60 = 0x10001000, r30 = 0xf1100000	IF r20 umulm r60 r30 → r90	r90 ← 0xf110f11
r70 = 0xffffff00, r40 = 0x100	umulm r70 r40 → r100	r100 ← 0xff

Unsigned compare not equal

pseudo-op for ineq

uneq

SYNTAX

```
[ IF rguard ] uneq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
   if rsrc1 != rsrc2 then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	39
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ineq igtr uneqi

DESCRIPTION

The uneq operation is a pseudo operation transformed by the scheduler into an ineq. (Note: pseudo operations cannot be used in assembly source files.)

The uneq operation sets the destination register, r*dest*, to 1 if the two arguments, r*src1* and r*src2*, are not equal; otherwise, r*dest* is set to 0.

The uneq operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`uneq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x1000, r30 = 3	`IF r10 uneq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 uneq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`uneq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`uneq r70 r70 → r110`	r110 ← 0

uneqi

Unsigned compare not equal with immediate

SYNTAX

```
[ IF rguard ] uneqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
   if (unsigned)rsrc1 != (unsigned)n then
      rdest ← 1
   else
      rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	40
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

uneq ineqi ueqli

DESCRIPTION

The `uneqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `uneqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`uneqi(2) r30 → r80`	r80 ← 1
r30 = 3	`uneqi(3) r30 → r90`	r90 ← 0
r30 = 3	`uneqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 uneqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 uneqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`uneqi(127) r60 → r120`	r120 ← 1

Write destination program counter

writedpc

SYNTAX

```
[ IF rguard ] writedpc rsrc1
```

FUNCTION

if r*guard* **then {**
  DPC ← r*src1*
}

ATTRIBUTES

Function unit	fcomp
Operation code	160
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`readdpc writespc ijmpf ijmpi ijmpt`

DESCRIPTION

The `writedpc` copies the value of r*src1* to the DPC (Destination Program Counter) processor register. Whenever a hardware update (during an interruptible jump) and a software update (through a `writedpc`) coincide, the software update takes precedence.

Interruptible jumps write their target address to the DPC. The value of DPC is intended to be used by an exception-handling routine as a jump address to resume execution of the program that was running before the exception was taken.

The `writedpc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of DPC. If the LSB of r*guard* is 1, DPC is written; otherwise, DPC is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xbeebee	`writedpc r30`	DPC ← 0xbeebee
r20 = 0, r31 = 0xabba	`IF r20 writedpc r31`	no change, since guard is false
r21 = 1, r31 = 0xabba	`IF r21 writedpc r31`	DPC ← 0xabba

writepcsw

Write program control and status word

SYNTAX

```
[ IF rguard ] writepcsw rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
PCSW ← (PCSW & ~r*src2*) | (r*src1* & r*src2*)
}

ATTRIBUTES

Function unit	fcomp
Operation code	161
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

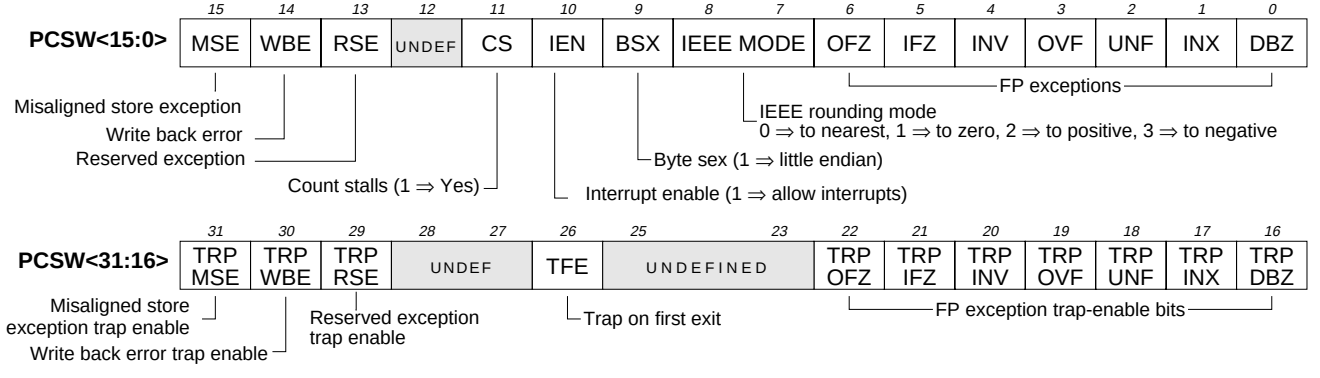
`readpcsw fadd faddflags ijmpf cycles hicycles`

DESCRIPTION

The `writepcsw` copies the value of r*src1* to the PCSW (Program Control and Status Word) processor register using rsrc2 as a mask. A bit in PCSW is affected by `writepcsw` only if the corresponding bit in rsrc2 is set to 1; the value of any bit in PCSW with a corresponding 0-bit in rsrc2 will not be changed by `writepcsw`. Whenever a hardware update (e.g., when a floating-point exception is raised) and a software update (through a `writepcsw`) coincide, the PCSW bits currently being updated by hardware will reflect the hardware-determined value while the bits not being affected by hardware will reflect the value in the `writepcsw` operand. The layout of PCSW is shown below. The programmer should take care not to alter UNDEF fields in the PCSW.

Fields in the PCSW have two chief purposes: to control aspects of processor operation and to record events that occur during program execution. Thus, `writepcsw` can be used to effect changes in some aspects of processor operation and to clear fields that record events; this operation can also be used to restore state before resuming an idled task in a multi-tasking environment. Note: The latency of writepcsw is 1, i.e. the PCSW reflects the new value in the next cycle. But it takes additional 3 cycles for updates to the exception flags and exception enable bits to take effect in the hardware. Therefore 3 delay slots / nops shall be inserted between writepcsw and the next interruptible jump, if exception flags or enable bits are changed. This guarantees that the new state is recognized in the interrupt logic during execution of the ijump.

The `writepcsw` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of PCSW. If the LSB of r*guard* is 1, PCSW is written; otherwise, PCSW is unchanged.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x100, r40 = 0x180	`writepcsw r30 r40`	PCSW.IEEE MODE = to positive infinity
r20 = 0, r50 = 0x0, r60 = 0x400	`IF r20 writepcsw r50 r60`	no change, since guard is false
r21 = 1, r50 = 0x0, r60 = 0x400	`IF r21 writepcsw r50 r60`	PCSW.IEN = 0 (disable interrupts)
r70 = 0x80110000, r80 = 0xffff0000	`writepcsw r70 r80`	enable trap on MSE, INV and DBZ exclusively

Write source program counter

writespc

SYNTAX

```
[ IF rguard ] writespc rsrc1
```

FUNCTION

if r*guard* **then**
  SPC ← r*src1*

ATTRIBUTES

Function unit	fcomp
Operation code	159
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`readspc writedpc ijmpf ijmpi ijmpt`

DESCRIPTION

The `writespc` copies the value of r*src1* to the SPC (Source Program Counter) processor register. Whenever a hardware update (during an interruptible jump) and a software update (through a `writespc`) coincide, the software update takes precedence.

An interruptible jump that is not interrupted (no NMI, INT, or EXC event was pending when the jump was executed) writes its target address to SPC. The value of SPC is intended to allow an exception-handling routine to determine the start address of the block of scheduled code (called a decision tree) that was executing before the exception was taken.

The `writespc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of SPC. If the LSB of r*guard* is 1, SPC is written; otherwise, SPC is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xbeebee	`writespc r30`	SPC ← 0xbeebee
r20 = 0, r31 = 0xabba	`IF r20 writespc r31`	no change, since guard is false
r21 = 1, r31 = 0xabba	`IF r21 writespc r31`	SPC ← 0xabba

zex16

Zero extend 16 bits

pseudo-op for pack16lsb

SYNTAX

```
[ IF rguard ] zex16 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext16to32(r*src1*<15:0>)

ATTRIBUTES

Function unit	alu
Operation code	53
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

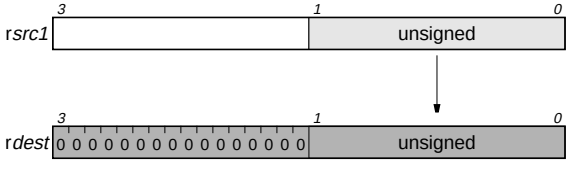
SEE ALSO

sex16 sex8 zex8

DESCRIPTION

The `zex16` operation is a pseudo operation transformed by the scheduler into a `pack16lsb` with 0 as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `zex16` operation zero extends the least-significant 16-bit halfword of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The `zex16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	`zex16 r30 → r60`	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	`IF r10 zex16 r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	`IF r20 zex16 r40 → r100`	r100 ← 0x0000ff91
r50 = 0x00000091	`zex16 r50 → r110`	r110 ← 0x00000091

Zero extend 8 bits

pseudo-op for ubytesel

zex8

SYNTAX

```
[ IF rguard ] zex8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext8to32(r*src1*<7:0>)

ATTRIBUTES

Function unit	alu
Operation code	55
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

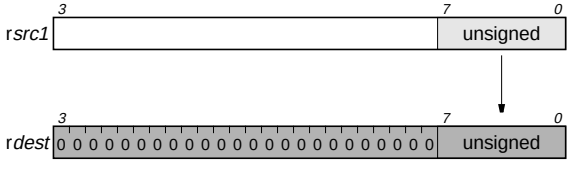
SEE ALSO

ubytesel sex16 sex8 zex16

DESCRIPTION

The zex8 operation is a pseudo operation transformed by the scheduler into a ubytesel with r0 (always contains 0) as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the zex8 operation zero extends the least-significant byte of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The zex8 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	zex8 r30 → r60	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	IF r10 zex8 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	IF r20 zex8 r40 → r100	r100 ← 0x00000091
r50 = 0x00000091	zex8 r50 → r110	r110 ← 0x00000091

MMIO Register Summary Chapter B

by Gert Slavenburg, and Selliah Rathnam

B.1 MMIO REGISTERS

The following table lists all the MMIO registers implemented in TM1300. The registers are grouped according to the unit to which they belong. For compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as zeroes.

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
DSPCPU Registers				
DRAM_BASE	10 0000	R/W	R/W	Start of DRAM address aperture
DRAM_LIMIT	10 0004	R/W	R/W	End of DRAM address aperture
MMIO_BASE	10 0400	R/W	R/W	Start of 2-MB MMIO-register address aperture
EXCVEC	10 0800	R/W	R/W	Interrupt vector (handler start address) for exceptions
ISETTING0	10 0810	R/W	R/W	Interrupt mode & priority settings for sources 0-7
ISETTING1	10 0814	R/W	R/W	Interrupt mode & priority settings for sources 8-15
ISETTING2	10 0818	R/W	R/W	Interrupt mode & priority settings for sources 16-23
ISETTING3	10 081c	R/W	R/W	Interrupt mode & priority settings for sources 24-31
IPENDING	10 0820	R/W	R/W	Interrupt-pending status bit for all 32 sources
ICLEAR	10 0824	R/W	R/W	Interrupt-clear bit for all 32 sources
IMASK	10 0828	R/W	R/W	Interrupt-mask bit for all 32 sources
INTVEC0	10 0880	R/W	R/W	Interrupt vector (handler start address) for source 0
INTVEC1	10 0884	R/W	R/W	Interrupt vector (handler start address) for source 1
INTVEC2	10 0888	R/W	R/W	Interrupt vector (handler start address) for source 2
INTVEC3	10 088c	R/W	R/W	Interrupt vector (handler start address) for source 3
INTVEC4	10 0890	R/W	R/W	Interrupt vector (handler start address) for source 4
INTVEC5	10 0894	R/W	R/W	Interrupt vector (handler start address) for source 5
INTVEC6	10 0898	R/W	R/W	Interrupt vector (handler start address) for source 6
INTVEC7	10 089c	R/W	R/W	Interrupt vector (handler start address) for source 7
INTVEC8	10 08a0	R/W	R/W	Interrupt vector (handler start address) for source 8
INTVEC9	10 08a4	R/W	R/W	Interrupt vector (handler start address) for source 9
INTVEC10	10 08a8	R/W	R/W	Interrupt vector (handler start address) for source 10
INTVEC11	10 08ac	R/W	R/W	Interrupt vector (handler start address) for source 11
INTVEC12	10 08b0	R/W	R/W	Interrupt vector (handler start address) for source 12
INTVEC13	10 08b4	R/W	R/W	Interrupt vector (handler start address) for source 13
INTVEC14	10 08b8	R/W	R/W	Interrupt vector (handler start address) for source 14
INTVEC15	10 08bc	R/W	R/W	Interrupt vector (handler start address) for source 15
INTVEC16	10 08c0	R/W	R/W	Interrupt vector (handler start address) for source 16
INTVEC17	10 08c4	R/W	R/W	Interrupt vector (handler start address) for source 17
INTVEC18	10 08c8	R/W	R/W	Interrupt vector (handler start address) for source 18
INTVEC19	10 08cc	R/W	R/W	Interrupt vector (handler start address) for source 19

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
INTVEC20	10 08d0	R/W	R/W	Interrupt vector (handler start address) for source 20
INTVEC21	10 08d4	R/W	R/W	Interrupt vector (handler start address) for source 21
INTVEC22	10 08d8	R/W	R/W	Interrupt vector (handler start address) for source 22
INTVEC23	10 08dc	R/W	R/W	Interrupt vector (handler start address) for source 23
INTVEC24	10 08e0	R/W	R/W	Interrupt vector (handler start address) for source 24
INTVEC25	10 08e4	R/W	R/W	Interrupt vector (handler start address) for source 25
INTVEC26	10 08e8	R/W	R/W	Interrupt vector (handler start address) for source 26
INTVEC27	10 08ec	R/W	R/W	Interrupt vector (handler start address) for source 27
INTVEC28	10 08f0	R/W	R/W	Interrupt vector (handler start address) for source 28
INTVEC29	10 08f4	R/W	R/W	Interrupt vector (handler start address) for source 29
INTVEC30	10 08f8	R/W	R/W	Interrupt vector (handler start address) for source 30
INTVEC31	10 08fc	R/W	R/W	Interrupt vector (handler start address) for source 31
TIMER1_TMODULUS	10 0c00	R/W	R/W	Contains: (maximum count value for timer 1) + 1
TIMER1_TVALUE	10 0c04	R/W	R/W	Current value of timer 1 counter
TIMER1_TCTL	10 0c08	R/W	R/W	Timer 1 control (prescale value, source select, run bit)
TIMER2_TMODULUS	10 0c20	R/W	R/W	Contains: (maximum count value for timer 2) + 1
TIMER2_TVALUE	10 0c24	R/W	R/W	Current value of timer 2 counter
TIMER2_TCTL	10 0c28	R/W	R/W	Timer 2 control (prescale value, source select, run bit)
TIMER3_TMODULUS	10 0c40	R/W	R/W	Contains: (maximum count value for timer 3) + 1
TIMER3_TVALUE	10 0c44	R/W	R/W	Current value of timer 3 counter
TIMER3_TCTL	10 0c48	R/W	R/W	Timer 3 control (prescale value, source select, run bit)
SYSTIMER_TMODULUS	10 0c60	R/W	R/W	Contains: (maximum count value for system timer) + 1
SYSTIMER_TVALUE	10 0c64	R/W	R/W	Current value of system timer/counter
SYSTIMER_TCTL	10 0c68	R/W	R/W	System timer control (prescale value, source select, run bit)
BICTL	10 1000	R/W	R/W	Instruction breakpoint control
BINSTLOW	10 1004	R/W	R/W	Start of address range that causes instruction breakpoints
BINSTHIGH	10 1008	R/W	R/W	End of address range that causes instruction breakpoints
BDCTL	10 1020	R/W	R/W	Data breakpoint control
BDATAALOW	10 1030	R/W	R/W	Start of address range that causes data breakpoints
BDATAAHIGH	10 1034	R/W	R/W	End of address range that causes data breakpoints
BDATAVAL	10 1038	R/W	R/W	Compare value for data breakpoints
BDATAMASK	10 103c	R/W	R/W	Compare mask for compare value for data breakpoints
Cache And Memory System				
DRAM_CACHEABLE_LIMIT	10 0008	R/W	R/W	Start of non-cacheable region in DRAM
MEM_EVENTS	10 000c	R/W	R/W	Selects two cache-related events for counting
DC_LOCK_CTL	10 0010	R/W	R/W	Enable bit for data-cache locking, also PCI hole disable
DC_LOCK_ADDR	10 0014	R/W	R/W	Start of address range that will be locked into the data cache
DC_LOCK_SIZE	10 0018	R/W	R/W	Size of address range that will be locked into the data cache
DC_PARAMS	10 001c	R/—	R/—	Data-cache geometry (blocksize, associativity, # of sets)
IC_PARAMS	10 0020	R/—	R/—	Instruction-cache geometry (blocksize, assoc., # of sets)
MM_CONFIG	10 0100	R/—	R/—	DRAM settings (rank size, bus width, refresh interval)
ARB_BW_CTL	10 0104	R/W	R/W	Internal bus arbitration control (bandwidth/latency allocation)
ARB_RAISE	10 010C	R/W	R/W	Arbiter Priority Raising timer
POWER_DOWN	10 0108	R/W	R/W	Write to this register to initiate power down
IC_LOCK_CTL	10 0210	R/W	R/W	Enable bit for instruction-cache locking
IC_LOCK_ADDR	10 0214	R/W	R/W	Start of address range that will be locked into the instruction cache

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
IC_LOCK_SIZE	10 0218	R/W	R/W	Size of address range that will be locked into the instruction cache
PLL_RATIOS	10 0300	R/—	R/—	Sets ratios of external and internal clock frequencies
BLOCK_POWER_DOWN	10 3428	R/W	R/W	Powers up and down individual blocks
Video In				
VI_STATUS	10 1400	R/—	R/—	Status of video-in unit
VI_CTL	10 1404	R/W	R/W	Sets operation and interrupt modes for video in
VI_CLOCK	10 1408	R/W	R/W	Sets clock source (internal/external), frequency
VI_CAP_START	10 140c	R/W	R/W	Sets capture start x and y offsets
VI_CAP_SIZE	10 1410	R/W	R/W	Sets capture size width and height
VI_BASE1 VI_Y_BASE_ADR	10 1414	R/W	R/W	Capture modes: sets base address of Y-value array Message/raw modes: sets base address of buffer 1
VI_BASE2 VI_U_BASE_ADR	10 1418	R/W	R/W	Capture modes: sets base address of U-value array Message/raw modes: sets base address of buffer 2
VI_SIZE VI_V_BASE_ADR	10 141c	R/W	R/W	Capture modes: sets base address of V-value array Message/raw modes: sets size of buffers
VI_UV_DELTA	10 1420	R/W	R/W	Capture modes: address delta for adjacent U, V lines
VI_Y_DELTA	10 1424	R/W	R/W	Capture modes: address delta for adjacent Y lines
Video Out				
VO_STATUS	10 1800	R/—	R/—	Status of video-out unit
VO_CTL	10 1804	R/W	R/W	Sets operation and interrupt modes for video out
VO_CLOCK	10 1808	R/W	R/W	Sets video-out clock frequency
VO_FRAME	10 180c	R/W	R/W	Sets frame parameters (preset, start, length)
VO_FIELD	10 1810	R/W	R/W	Sets field parameters (overlap, field-1 line, field-2 line)
VO_LINE	10 1814	R/W	R/W	Sets field parameters (starting pixel, frame width)
VO_IMAGE	10 1818	R/W	R/W	Sets image parameters (height, width)
VO_YTHR	10 181c	R/W	R/W	Sets threshold for YTR interrupt, image v/h offsets
VO_OLSTART	10 1820	R/W	R/W	Sets overlay image parameters (start line/pixel, alpha)
VO_OLHW	10 1824	R/W	R/W	Sets overlay image parameters (height, width)
VO_YADD	10 1828	R/W	R/W	Sets Y-component/buffer-1 starting address
VO_UADD	10 182c	R/W	R/W	Sets U-component/buffer-2 starting address
VO_VADD	10 1830	R/W	R/W	Sets V-component address/buffer-1 length
VO_OLADD	10 1834	R/W	R/W	Sets overlay image address/buffer-2 length
VO_VUF	10 1838	R/W	R/W	Sets start-of-line-to-start-of-line address offsets (U, V)
VO_YOLF	10 183c	R/W	R/W	Sets start-of-line-to-start-of-line addr. offsets (Y, overlay)
EVO_CTL	10 1840	R/W	R/W	Sets operations for enhance video out
EVO_MASK	10 1844	R/W	R/W	Sets YUV mask values foe the chroma-key process
EVO_CLIP	10 1848	R/W	R/W	Sets output clip values
EVO_KEY	10 184c	R/W	R/W	Sets YUV chroma-key values
EVO_SLVDLY	10 1850	R/W	R/W	Sets delay cycles for genlock mode
Audio In				
AI_STATUS	10 1c00	R/—	R/—	Status of audio-in unit
AI_CTL	10 1c04	R/W	R/W	Sets operation and interrupt modes for audio in
AI_SERIAL	10 1c08	R/W	R/W	Sets clock ratios and internal/external clock generation
AI_FRAMING	10 1c0c	R/W	R/W	Sets format of serial data stream

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
AI_FREQ	10 1c10	R/W	R/W	Sets AI_OSCLK frequency
AI_BASE1	10 1c14	R/W	R/W	Sets base address of buffer 1
AI_BASE2	10 1c18	R/W	R/W	Sets base address of buffer 2
AI_SIZE	10 1c1c	R/W	R/W	Sets number of samples in buffers
Audio Out				
AO_STATUS	10 2000	R/—	R/—	Status of audio-out unit
AO_CTL	10 2004	R/W	R/W	Sets operation and interrupt modes for audio out
AO_SERIAL	10 2008	R/W	R/W	Sets clock ratios and internal/external clock generation
AO_FRAMING	10 200c	R/W	R/W	Sets format of serial data stream
AO_FREQ	10 2010	R/W	R/W	Set AO_OSCLK frequency
AO_BASE1	10 2014	R/W	R/W	Sets base address of buffer 1
AO_BASE2	10 2018	R/W	R/W	Sets base address of buffer 2
AO_SIZE	10 201c	R/W	R/W	Sets number of samples in buffers
AO_CC	10 2020	R/W	R/W	Codec control field values
AO_CFC	10 2024	R/W	R/W	Codec Frame Control
AO_TSTAMP	10 2028	R/—	R/W	Timestamp of the last buffer
SPDIF Out				
SDO_STATUS	10 4C00	R/—	R/—	Status register
SDO_CTL	10 4C04	R/W	R/W	Control register
SDO_FREQ	10 4C08	R/W	R/W	Frequency register
SDO_BASE1	10 4C0C	R/W	R/W	Base address of buffer 1
SDO_BASE2	10 4C10	R/W	R/W	Base address of buffer 2
SDO_SIZE	10 4C14	R/W	R/W	Number of samples in buffers
SDO_TSTAMP	10 4C18	R/—	R/—	Timestamp of the last buffer
PCI Interface				
BIU_STATUS	10 3004	R/—	R/—	Status of PCI interface (done/busy bits, error bits)
BIU_CTL	10 3008	R/W	R/W	Sets operation and interrupt modes for PCI
PCI_ADR	10 300c	R/W	—/—	Holds address for DSPCPU PCI access
PCI_DATA	10 3010	R/W	—/—	Holds data for DSPCPU PCI access
CONFIG_ADR	10 3014	R/W	R/W	Holds address for configuration access
CONFIG_DATA	10 3018	R/W	R/W	Holds data for configuration access
CONFIG_CTL	10 301c	R/W	R/W	Sets read/write, bus number for configuration access
IO_ADR	10 3020	R/W	R/W	Holds address for I/O access
IO_DATA	10 3024	R/W	R/W	Holds data for I/O access
IO_CTL	10 3028	R/W	R/W	Sets read/write, byte-enable for I/O access
SRC_ADR	10 302c	R/W	R/W	Holds source address for DMA operation
DEST_ADR	10 3030	R/W	R/W	Holds destination address for DMA operation
DMA_CTL	10 3034	R/W	R/W	Sets read/write, transfer length for DMA operation
INT_CTL	10 3038	R/W	R/W	Controls interrupt system
XIO_CTL	10 3060	R/W	R/W	XIO control register
JTAG				
JTAG_DATA_IN	10 3800	R/W	R/W	JTAG data input buffer
JTAG_DATA_OUT	10 3804	R/W	R/W	JTAG data output buffer
JTAG_CTL	10 3808	R/W	R/W	JTAG control
Image Co-Processor				

MMIO Register Name	Offset (in hex)	Accessibility		Description
		DSPCPU	External PCI Initiators	
ICP_MPC	10 2400	R/W	R/W	MicroProgram Counter
ICP_MIR	10 2404	R/W	R/W	Micro Instruction Register
ICP_DP	10 2408	R/W	R/W	Data Pointer
ICP_DR	10 2410	R/W	R/W	Data Register
ICP_SR	10 2414	R/W	R/W	Status Register
VLD Co-Processor				
VLD_COMMAND	10 2800	R/W	R/W	Next action to be taken by VLD
VLD_SR	10 2804	R/—	R/—	Bitstream shift register
VLD_QS	10 2808	R/W	R/W	Quantization Scale Code
VLD_PI	10 280C	R/W	R/W	Picture layer Information
VLD_STATUS	10 2810	R/W	R/W	Status Register
VLD_IMASK	10 2814	R/W	R/W	Controls which status bits causes VLD interrupts
VLD_CTL	10 2818	R/W	R/W	Control Register
VLD_BIT_ADR	10 281C	R/W	R/W	Current Bitstream Read Address
VLD_BIT_CNT	10 2820	R/W	R/W	Bitstream remaining byte count
VLD_MBH_ADR	10 2824	R/W	R/W	Macro Block Header output address
VLD_MBH_CNT	10 2828	R/W	R/W	Macro Block Header output remaining count
VLD_RL_ADR	10 282C	R/W	R/W	Run/Length output address
VLD_RL_CNT	10 2830	R/W	R/W	Run/Length output remaining count
I^2C Interface				
IIC_AR	10 3400	R/W	R/W	Address, Byte count and Direction
IIC_DR	10 3404	R/W	R/W	Data Register
IIC_STATUS	10 3408	R/—	R/—	Status Register
IIC_CTL	10 340C	R/W	R/W	Control Register
Synchronous Serial Interface				
SSI_CTL	10 2C00	R/W	R/W	Control Register
SSI_CSR	10 2C04	R/W	R/W	Additional Control and Status register
SSI_TXDR	10 2C10	—/W	—/W	Transmit Data Register
SSI_RXDR	10 2C20	R/—	R/—	Receive Data Register
SSI_RXACK	10 2C24	—/W	—/W	Write a ‘1’ here to ACK read of Receive Data Register
SEM Device				
SEM	10 0500	R/W	R/W	Simple multi-processor semaphore

Endian-ness Appendix C

by Selliah Rathnam, Luis Lucas

C.1 PURPOSE

TM1300 was designed to support both Little and Big Endian systems. The PCI system bus (controlled by the PCI Interface Unit (BIU)) operates in Little Endian mode in both systems. This document describes how the dual endian-ness feature is handled in TM1300.

C.2 LITTLE AND BIG ENDIAN ADDRESSING CONVENTIONS

In Big Endian mode, a given word address (32-bit) base corresponds to the most significant byte (MSB) of the word. Increasing the byte address generally means decreasing the significance of the byte being accessed. In Little Endian mode, the same word address base refers to the least significant byte (LSB) of that word. Increasing the byte address generally means increasing the significance of the byte being accessed. This addressing convention is shown in Figure C-1.

In Figure C-1, there is a two-line 'C' code which defines a 32-bit constant in hex format assigned to the variable 'w' (assumes 'int' is 32-bit) and its address is copied into the byte (character) pointer variable 'cp'. The value of address referenced by the 'cp' has a value of '0x04' in Big Endian machine and a value of '0x07' in Little Endian machine.

It is possible to transfer from one endian-ness to another just by swapping the bytes within a word as shown in Figure C-2.

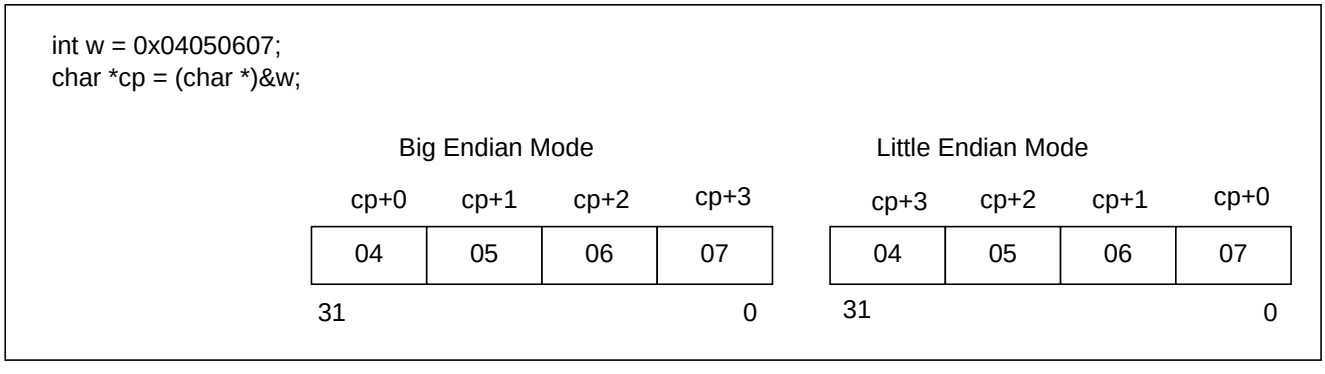


Figure C-1. Big and Little Endian address references

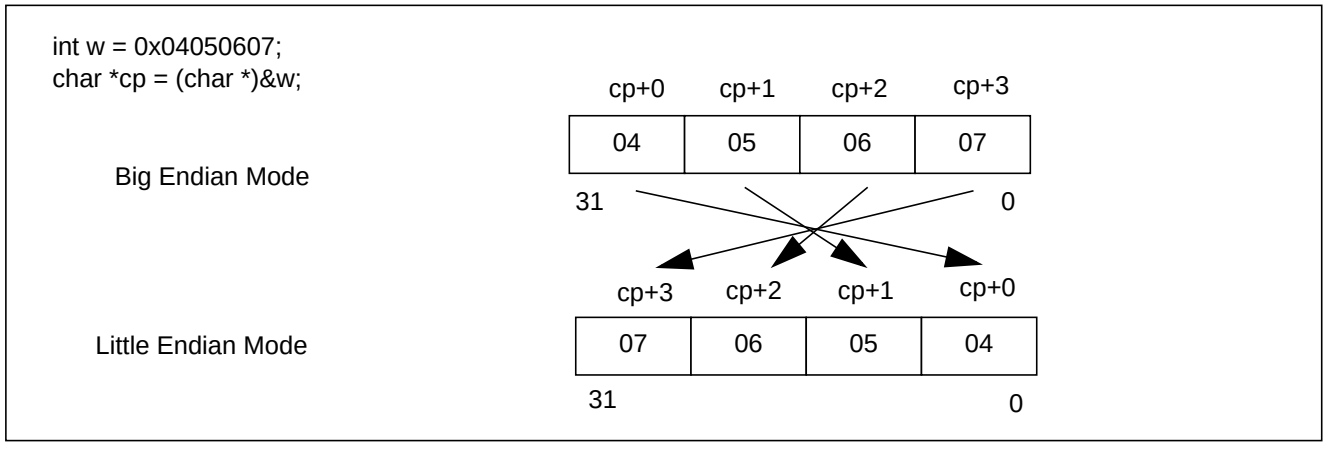


Figure C-2. Data conversion from Big Endian to Little Endian (BSW)

Table C-1. Little Endian data format in TM1300 DSPCPU register, highway, SDRAM memory, PCI bus, host memory, host CPU register

PCSW-BSX value	Endian Mode	Data Transaction type	Address	Data in DSPCPU register msb lsb	Data in highway/ Dcache/SDRAM/ PCI-bus byte3 byte0 [31:24] [7:0]	Data in host CPU register msb lsb	Data in host memory byte3 byte0 [31:24] [7:0]
1	Little	Word r/w	00001000	01020304	01020304	01020304	01020304
1	Little	Half-Word r/w	00001000	xxxx0304	xxxx0304	xxxx0304	xxxx0304
1	Little	Half-Word r/w	00001002	xxxx0304	0304xxxx	xxxx0304	0304xxxx
1	Little	Byte read/write	00001000	xxxxxx04	xxxxxx04	xxxxxx04	xxxxxx04
1	Little	Byte read/write	00001001	xxxxxx04	xxxx04xx	xxxxxx04	xxxx04xx
1	Little	Byte read/write	00001002	xxxxxx04	xx04xxxx	xxxxxx04	xx04xxxx
1	Little	Byte read/write	00001003	xxxxxx04	04xxxxxx	xxxxxx04	04xxxxxx

C.3 TEST TO VERIFY THE CORRECT OPERATION OF TM1300 IN BIG AND LITTLE ENDIAN SYSTEMS

The following test can be used to verify the correct operation of TM1300 in Little Endian and Big Endian systems.

1. Store a 32-bit constant ‘0x04050607’ from the host CPU to the TM1300 SDRAM through the PCI interface. Load the word from the same address to one of the TM1300’s global register and check for the same value.
2. Store a 32-bit constant ‘0x04050607’ from the host CPU to the TM1300 SDRAM through PCI interface. Load a byte from the same address to one of the TM1300 global registers. Check for the value of ‘0x04’ in Big Endian systems, and check for the value ‘0x07’ in Little Endian systems.

C.4 REQUIREMENT FOR THE TM1300 TO OPERATE IN EITHER LITTLE ENDIAN OR BIG ENDIAN MODE

The endian-ness handling in each TM1300 unit is described in the following sections. Most units use the highway/PCI bus to transfer data. The highway/PCI bus has four byte lanes. The bit assignment of the highway/PCI bus lanes is shown in Table C-2.

Table C-2. Bit assignment of the highway/PCI bus lanes

	byte 3	byte 2	byte 1	byte 0
Bits	31:24	23:16	15:8	7:0

The PCI bus and TM1300 highway buses are address-invariant buses, i.e the data corresponding to address offset ‘0’ uses the byte-0 lane of the highway/PCI bus, the data corresponds to address offset ‘1’ uses the byte-1 lane of the highway/PCI bus etc.

C.4.1 Data Cache

The TM1300 PCSW register has a byte-sex (BSX) bit to configure the TM1300 in Big Endian or Little Endian mode. This bit must be set to ‘1’ for the Little Endian mode as defined in Chapter 3, “DSPCPU Architecture.” This BSX bit is used by the TM1300 data cache unit for the store/load operation. Data cache performs three categories of data transactions:

- Read/write data from/to DSPCPU registers to/from data cache or SDRAM
- Read/write of MMIO data from/to DSPCPU registers to/from MMIO registers
- Read/write data from/to DSPCPU registers to/from PCI address space through special registers in the BIU unit.

The DSPCPU endian-ness is determined by the value of the BSX bit in the PCSW register. Table C-1 and Table C-3 describe the data translation format being used by the data cache to transfer the data to/from DSPCPU register to/from data cache or SDRAM. Table C-1 and Table C-3 are restricted to addresses that fall in the DRAM_BASE and DRAM_LIMIT range.

There is no byte-swap required for the MMIO data transaction from/to DSPCPU register to the MMIO registers. However, one of the special registers, PCI_DATA, does not follow the normal MMIO transactions. The data cache byte-swaps the data to/from the PCI_DATA register using the data translation format as defined in Table C-1 and Table C-3 for the memory cycle.

For the PCI configuration cycle and I/O cycle transactions from the DSPCPU, a programmer can byte-swap the data in the DSPCPU registers and write to the PCI_DATA register using MMIO write operations. There is no byte-swap from the PCI_DATA register in BIU unit to the PCI bus. Software uses the Table C-1 or Table C-3 data to byte-swap the data within the CPU register before writing the data to the PCI_DATA register for the configuration and I/O cycle transactions.

Table C-3. Big Endian data format in the TM1300 DSPCPU register, highway, SDRAM memory, PCI bus, host memory, and host CPU register

PCSW-BSX value	Endian Mode	Data transaction type	Address	Data in DSPCPU register msb lsb	Data in highway/ Dcache/SDRAM/ PCI-bus byte3 [31:24] byte0 [7:0]	Data in Host CPU register msb lsb	Data in host memory byte0 [31:24] byte3 [7:0]
0	Big	Word r/w	00001000	01020304	04030201	01020304	01020304
0	Big	Half-word r/w	00001000	xxxx0304	xxxx0403	xxxx0304	0304xxxx
0	Big	Half-word r/w	00001002	xxxx0304	0403xxxx	xxxx0304	xxxx0304
0	Big	Byte read/write	00001000	xxxxxx04	xxxxxx04	xxxxxx04	04xxxxxx
0	Big	Byte read/write	00001001	xxxxxx04	xxxx04xx	xxxxxx04	xx04xxxx
0	Big	Byte read/write	00001002	xxxxxx04	xx04xxxx	xxxxxx04	xxxx04xx
0	Big	Byte read/write	00001003	xxxxxx04	04xxxxxx	xxxxxx04	xxxxxx04

C.4.2 Instruction Cache

It is assumed that the instruction cache always operates in Little Endian regardless of the host and TM1300 endian-ness. Instruction cache does not use the PCSW's byte sex bit (BSX). The compiler supports the loading of instructions in memory differently for Big Endian and Little Endian modes.

C.4.3 TM1300 PCI Interface Unit

The TM1300 highway bus and the PCI bus are address invariant buses, i.e. a data corresponding to address zero is always transferred through the byte-zero line regardless of the endian-ness. The address-invariant nature of the PCI and the highway buses allows data to be transferred from/to PCI bus directly to/from SDRAM without byte swapping in either Big or Little Endian mode The byte swapping of data for Big Endian mode is performed by the data cache unit. However, MMIO data does not go through the byte swapper in the Data cache. This results in using a byte-swapper in the BIU to byte-swap the MMIO data in Big Endian mode.

The TM1300 BIU has a separate byte sex (SE, Swap Enabled) flag defined in its control register (BIU_CTL). This byte-sex flag must be set by the software, i.e. MMIO write operation from the host CPU. This byte-sex flag is used only for MMIO data accesses and none of the MMIO data accesses is affected by this SE flag. Table C-4 shows the byte-swap logic that handles the MMIO accesses from the DSPCPU and host CPU and the non MMIO data accesses from any source.

Table C-4. BIU.SE bit usage in processing data in BIU unit

BIU.SE value	Endian Mode	MMIO access from DSPCPU	MMIO access from PCI side	Non MMIO data
0	Big	No byte-swap	byte-swap	No byte-swap
1	Little	No byte-swap	No byte-swap	No byte-swap

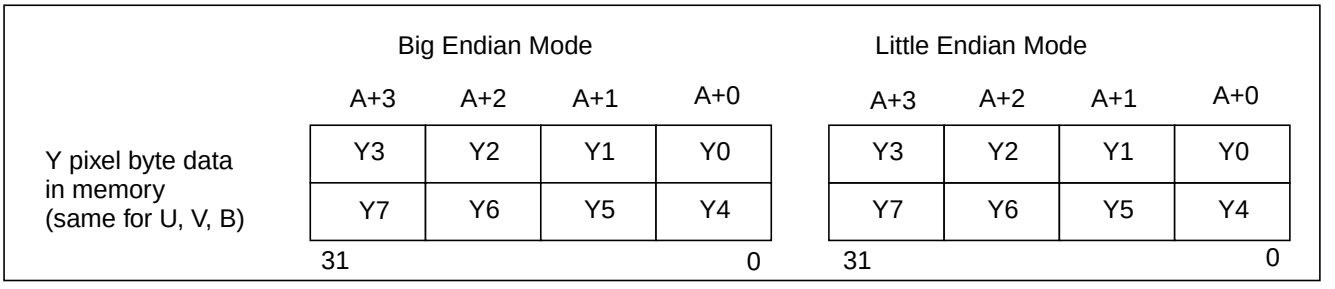
The BIU has several special registers to handle memory, PCI configuration, I/O and DMA accesses. It does not byte-swap the I/O data from the special registers. The data cache and software performs the necessary byte swapping for this data.

When using TM1300 in Little Endian-based systems, the first transaction to the TM1300 is to set the SE bit in the BIU configuration register to avoid unnecessary software byte-swapping in the host CPU for the subsequent MMIO read/write accesses. The SE bit in the BIU_CTL register controls the byte swapping of outgoing and incoming data from PCI bus. The default value of SE is '0', i.e the BIU byte-swaps the MMIO data including the write operation to the BIU_CTL register. Software is required to byte swap the BIU_CTL register value within the host CPU before storing the value in BIU_CTL register. Once, the BIU.SE bit has been set, no additional software byte-swapping is required for further read/write operations to any MMIO registers.

C.4.4 Image Coprocessor (ICP)

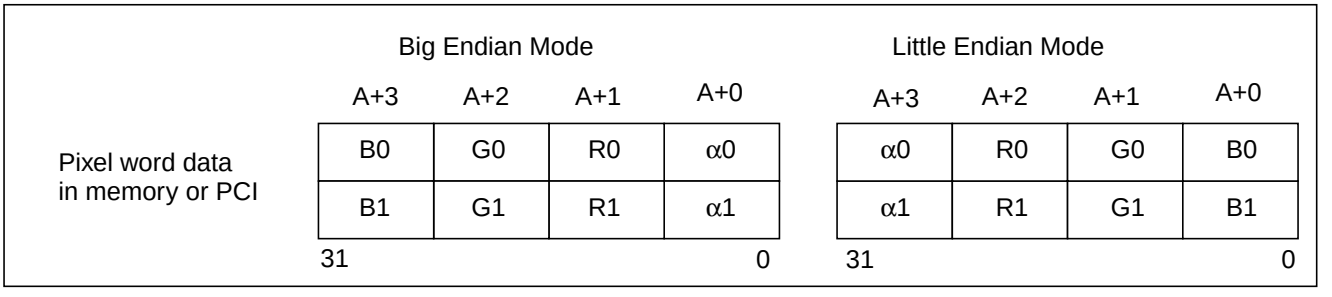
The input source data for the ICP unit might come from different units such as Video In, the DSPCPU, PCI bus, etc. via SDRAM. Data consistency needs to be maintained when the TM1300 operates in Little or Big Endian systems/mode. The ICP needs the capability to operate on the SDRAM as source data and SDRAM or PCI as destination data in either Little or Big Endian mode. Figure C-3, Figure C-4, Figure C-5 and Figure C-6 illustrate the Big and Little Endian memory image format for the image input format (Figure C-3) and the three supported image overlay formats.

The ICP can output the data to either the SDRAM or PCI bus. RGB 8R and RGB 8A pixel formats are byte streams and therefore do not require any byte swapping. Figure C-9 pictures the data format. RGB-24+α, RGB-15+α, RGB-16 and YUV-4:2:2 pixel formats can be used to output the pixels to PCI or SDRAM in both Endian modes. Output formats are shown, respectively, in Figure C-4, Figure C-5, Figure C-8, and Figure C-7. Packed RGB-24 cannot be used in Big Endian mode. Little Endian data format is shown in Figure C-11.



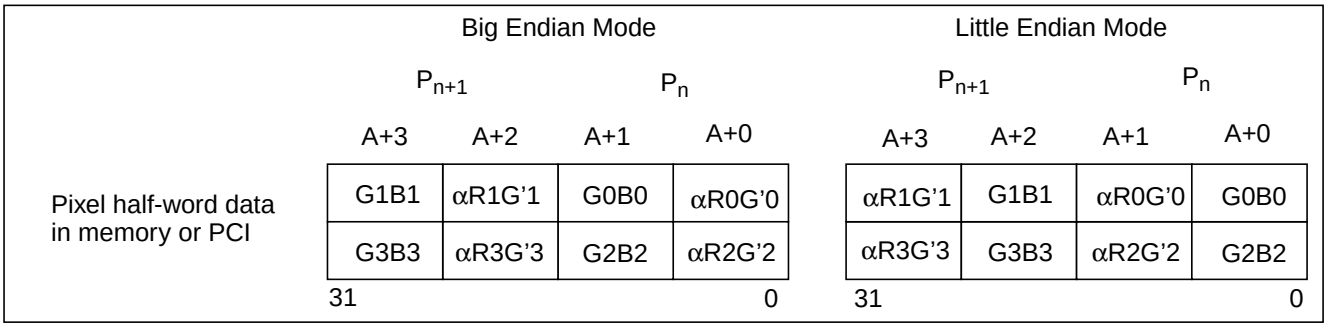
Note: A+0 corresponds to byte-0 lane of SDRAM/Hwy and A+3 corresponds to byte-3 lane of SDRAM/Hwy

Figure C-3. Byte mask, planar YUV 4:2:0 and YUV 4:2:2 for ICP, VO or VI memory data in Little and Big Endian modes



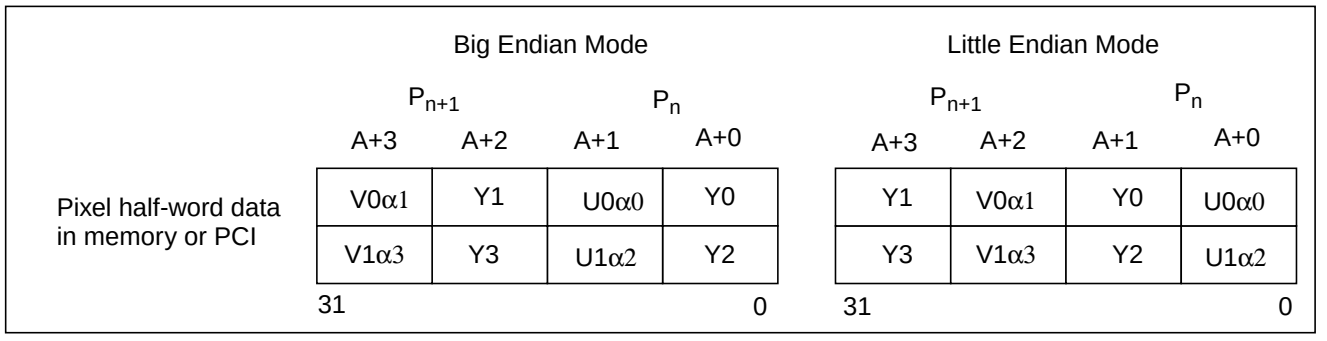
Note: A+0 corresponds to byte-0 lane of SDRAM/Hwy/PCI and A+3 corresponds to byte-3 lane of SDRAM/Hwy/PCI

Figure C-4. RBG-24+α data format for ICP in Little and Big Endian modes



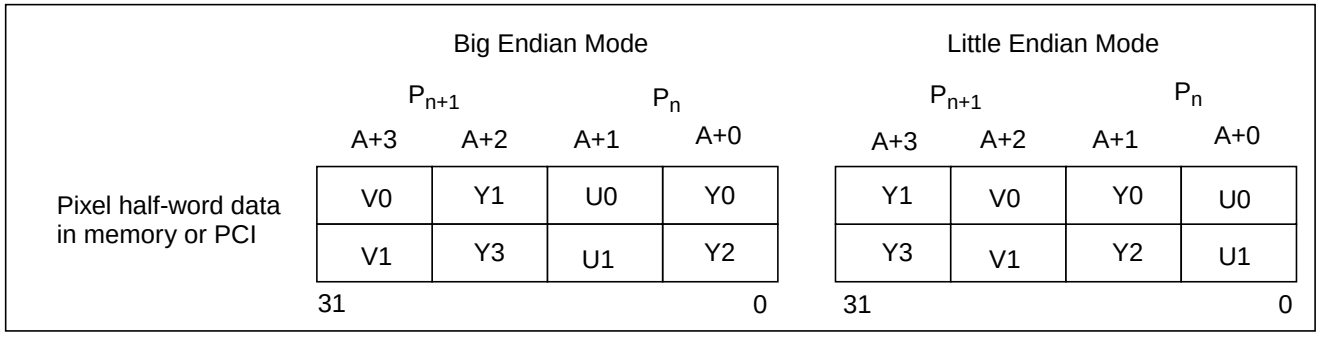
Note: A+0 corresponds to byte-0 lane of SDRAM/Hwy/PCI and A+3 corresponds to byte-3 lane of SDRAM/Hwy/PCI

Figure C-5. RBG-15+α data format for ICP in Little and Big Endian modes



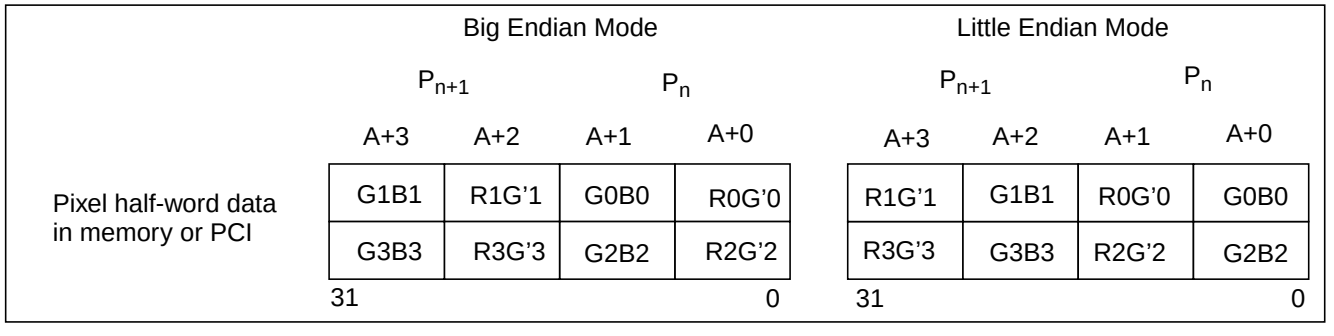
Note: A+0 corresponds to byte-0 lane of SDRAM/Hwy/PCI and A+3 corresponds to byte-3 lane of SDRAM/Hwy/PCI

Figure C-6. Packed YUV 4:2:2+α data format for the ICP or VO in Little and Big Endian modes



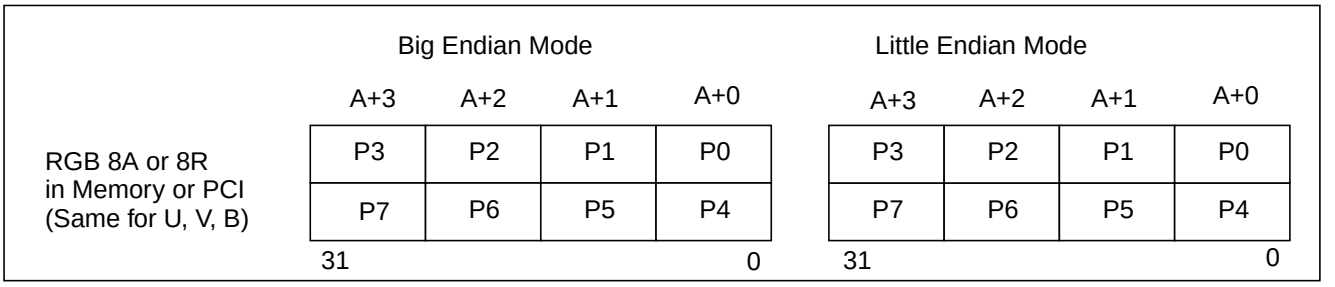
Note: A+0 corresponds to byte-0 lane of SDRAM/Hwy/PCI and A+3 corresponds to byte-3 lane of SDRAM/Hwy/PCI

Figure C-7. Packed YUV 4:2:2 data format for ICP in Little and Big Endian modes



Note: A+0 corresponds to byte-0 lane of SDRAM/Hwy/PCI and A+3 corresponds to byte-3 lane of SDRAM/Hwy/PCI

Figure C-8. RBG-16 data format for ICP in Little and Big Endian modes



Note: A+0 corresponds to byte-zero lane of SDRAM/Hwy/PCI and A+3 corresponds to byte-three lane of SDRAM/Hwy/PCI

Figure C-9. RGB8A and RGB8R data format for ICP in Little and Big Endian modes

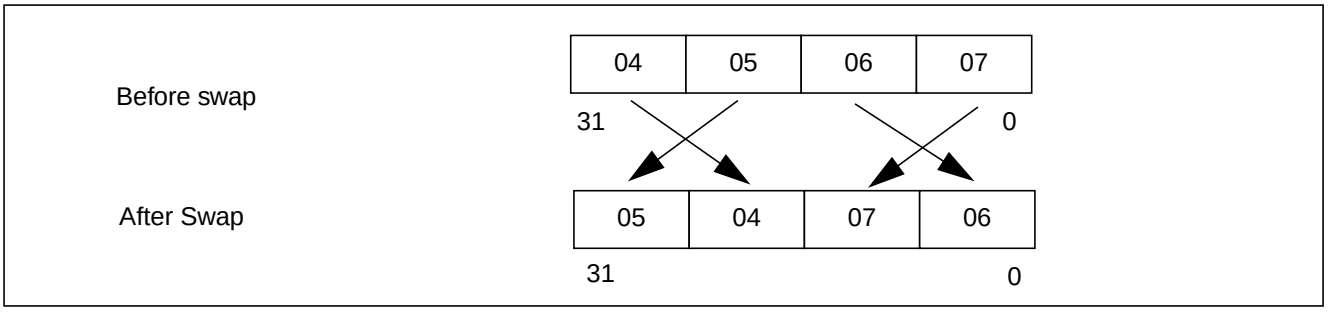
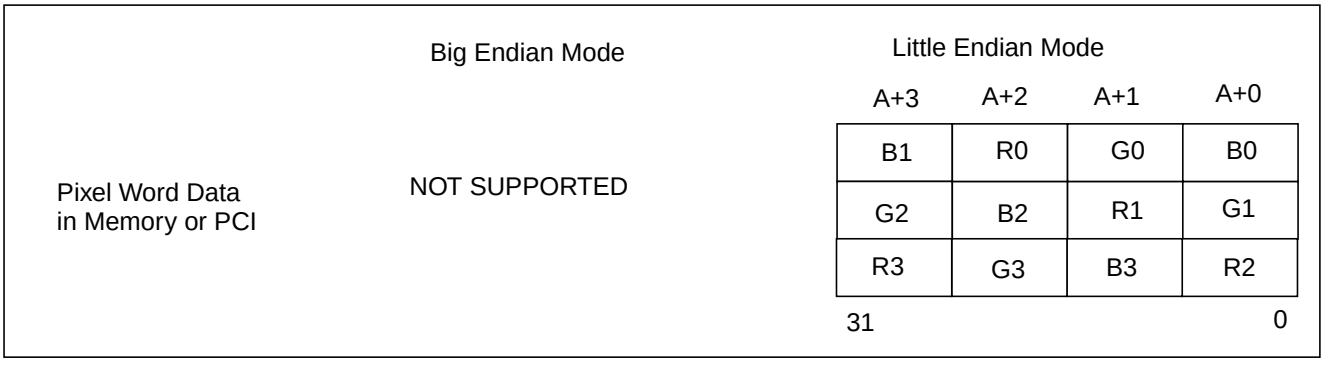


Figure C-10. Half-word swap within a half-word (BSH)



Note: A+0 corresponds to byte-zero lane of SDRAM/Hwy/PCI and A+3 corresponds to byte-three lane of SDRAM/Hwy/PCI

Figure C-11. Packed RBG-24 data format for ICP in Little Endian mode only

The Table C-5 shows the byte-swap implementation of various pixel formats used in the ICP unit. Refer to Figure C-2 and Figure C-10 for the byte-swap code used in Table C-4 and Table C-5. Byte-swapping is performed only in Big Endian mode. No swapping is done in the Little Endian mode.

Table C-5. ICP byte swapping type for input data

Endian-ness	L bit	Pixel Type	Swap Type (see Figure C-2 & Figure C-10)
Big Endian	0	Y,U,V planar	No swap
Big Endian	0	RGB 24+α	BSW
Big Endian	0	YUV-4:2:2+α	BSH
Big Endian	0	RGB 15+α	BSH

Table C-6. ICP byte swapping type for output data

Endian-ness	L bit	Pixel Type	Swap Type (see Figure C-2 & Figure C-10)
Big Endian	0	RGB 8A: 233	No swap
Big Endian	0	RGB 8R: 332	No swap
Big Endian	0	RGB 15+α	BSH
Big Endian	0	RGB 16	BSH
Big Endian	0	RGB 24+α	BSW
Big Endian	0	RGB24 packed	No support for Big Endian
Big Endian	0	YUV- 4:2:2 packed	BSH

The ICP has a byte sex bit (L) defined in its MMIO-based configuration register. The setting of this bit and the BSX bit in the PCSW register should be the same. The L bit must be set by the software.

C.4.5 Video In (VI) and Video Out (VO) Units

The VI unit stores the YUV pixels in planar 4:2:2 or 4:2:0 image format as shown in Figure C-3 and stores the raw 8- and 10-bit data as shown in Figure C-12.

The VO unit uses YUV-4:2:2 planar, YUV-4:2:0 planar, and YUV-4:2:2+α packed as input pixel formats. The planar memory image format of the YUV-4:2:2 and YUV-4:2:0 are shown in Figure C-3. The YUV-4:2:2+α memory image format for overlay is pictured in Figure C-6.

The VI and VO units have a byte-sex bit (Little Endian and LTL_END) defined in the control MMIO registers, VI_CONTROL and VO_CONTROL. The definition of these byte-sex bits and the BSX bit in the PCSW register should be treated as same. Little Endian and LTL_END bits must be set by software.

C.4.6 Audio In (AI), Audio-Out (AO), and SPDIF Out (SDO) Units

The AI unit uses 8-bit mono, 8-bit stereo, 16-bit mono and 16-bit stereo data. The AO unit uses 16-bit mono, 16-bit stereo, 32-bit mono and 32-bit stereo data. The SPDO unit uses 32-bit word data. The memory image format of these data is presented in Figure C-13.

Swapping takes place at the byte level and the bits within a byte are never disturbed. Both the AI and AO units have a byte sex bit (LITTLE_ENDIAN) defined in each units MMIO-based configuration register. The definition of the these bits and the BSX bit in the PCSW register should be treated as same. This byte sex bit must be set by the software.

C.4.7 Variable Length Encoder (VLD) Unit

The VLD inputs data from SDRAM in the form of a bitstream with a byte-aligned starting address and outputs a header stream and a 'run-level' data stream. The VLD unit has a byte sex bit (LITTLE_ENDIAN) defined in its MMIO-based configuration register. The definition of this

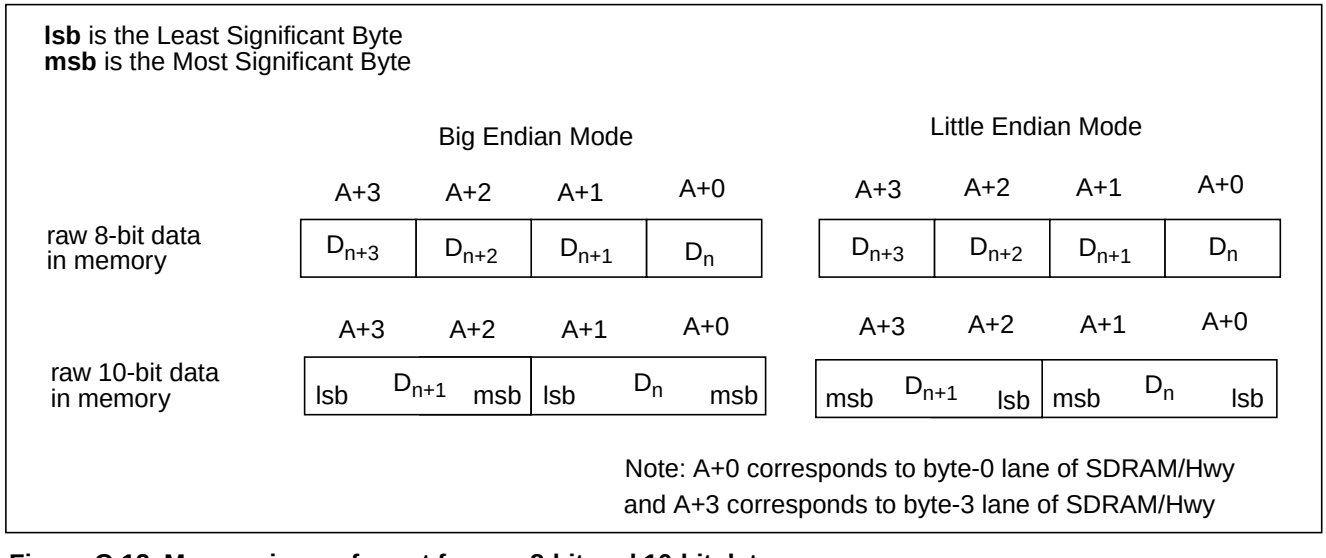


Figure C-12. Memory image format for raw 8-bit and 10-bit data

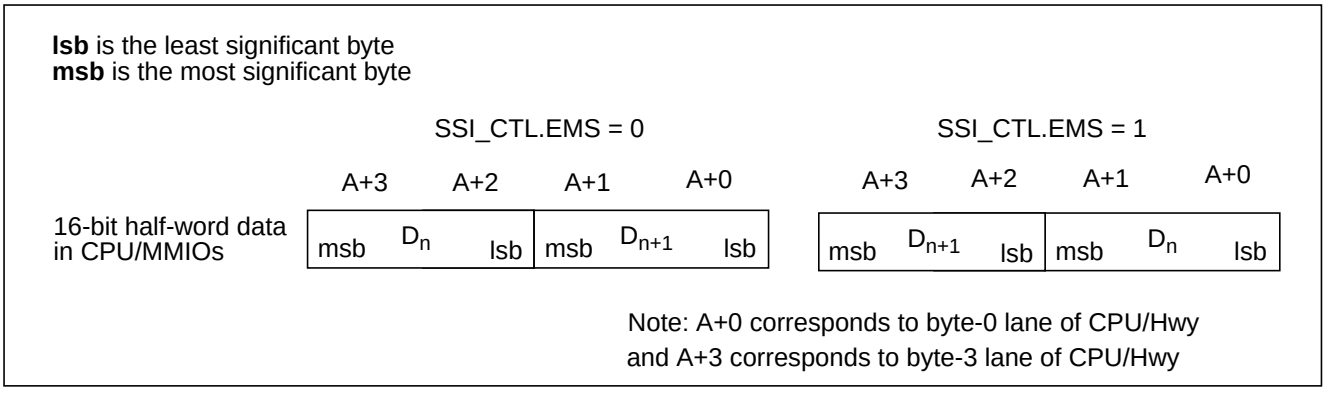


Figure C-15. SSI data format as seen in highway

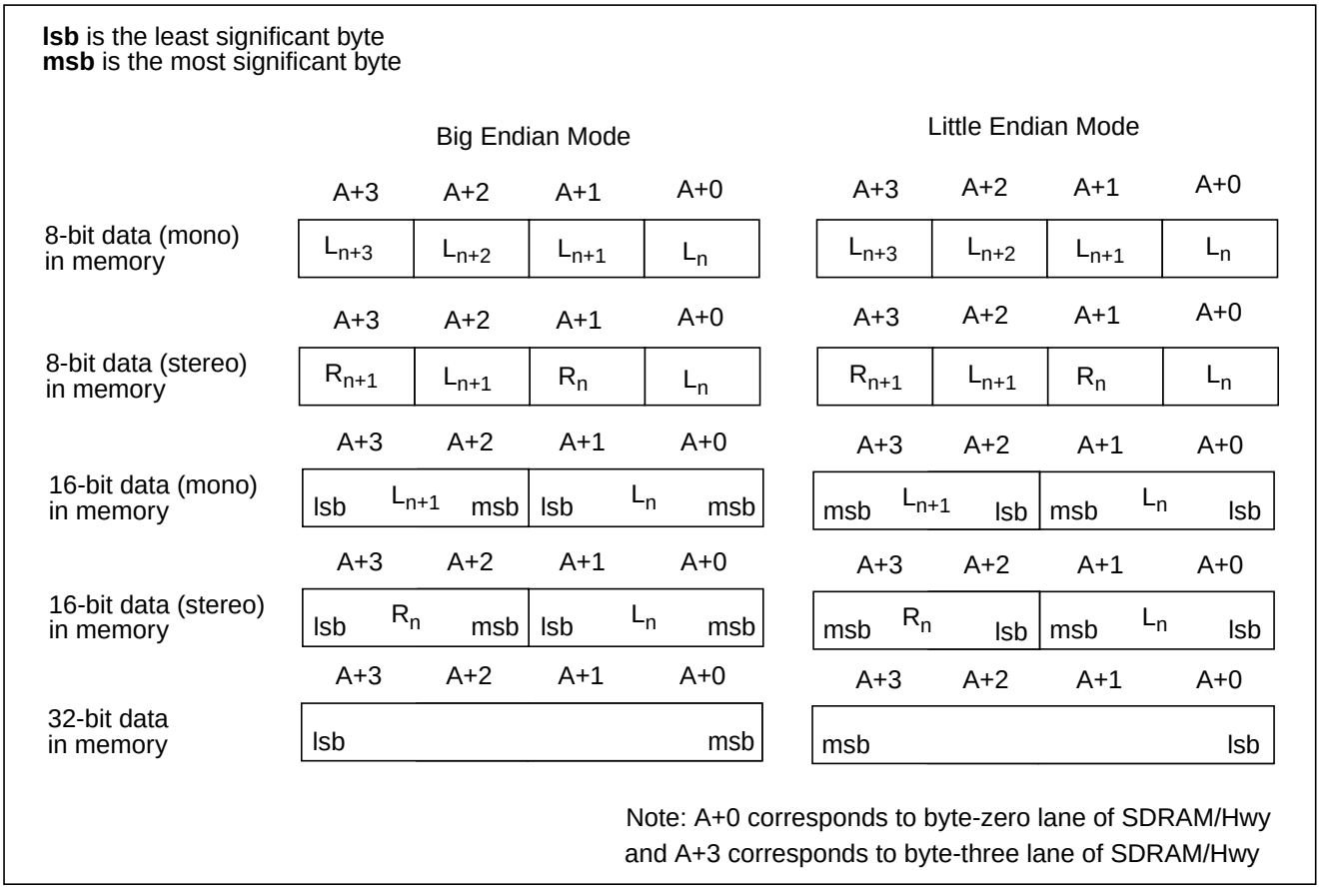


Figure C-13. Memory image format for audio data

bit and the BSX bit in the PCSW register should the same. This byte sex bit must be set by the software.

Figure C-14 describes the VLD input and output data format as seen in the SDRAM and highway bus. The input data is byte oriented and no swapping is required in the VLD unit. However, the output data is read by the DSPCPU in words, thus the VLD needs to swap the output bytes within a word (shown in Figure C-14) to compensate for the CPU swap.

C.4.8 Synchronous Serial Interface (SSI)

The SSI unit has I/O connections through the external serial pins and also to the internal 32-bit data highway via MMIO transactions. The minimum quantity of data to be analyzed by the CPU is 16-bits (i.e. one half word). The SSI uses a 16-bit or 1-bit endian-ness; it is detailed in Section 17.8 on page 17-7. The 32-bit quantity contained in the CPU register is written or read ‘as is’ into/from the SSI MMIO register. The EMS bit in SSI_CTL determines which half-word (16-bit) is sent first as pictured in Figure C-15.

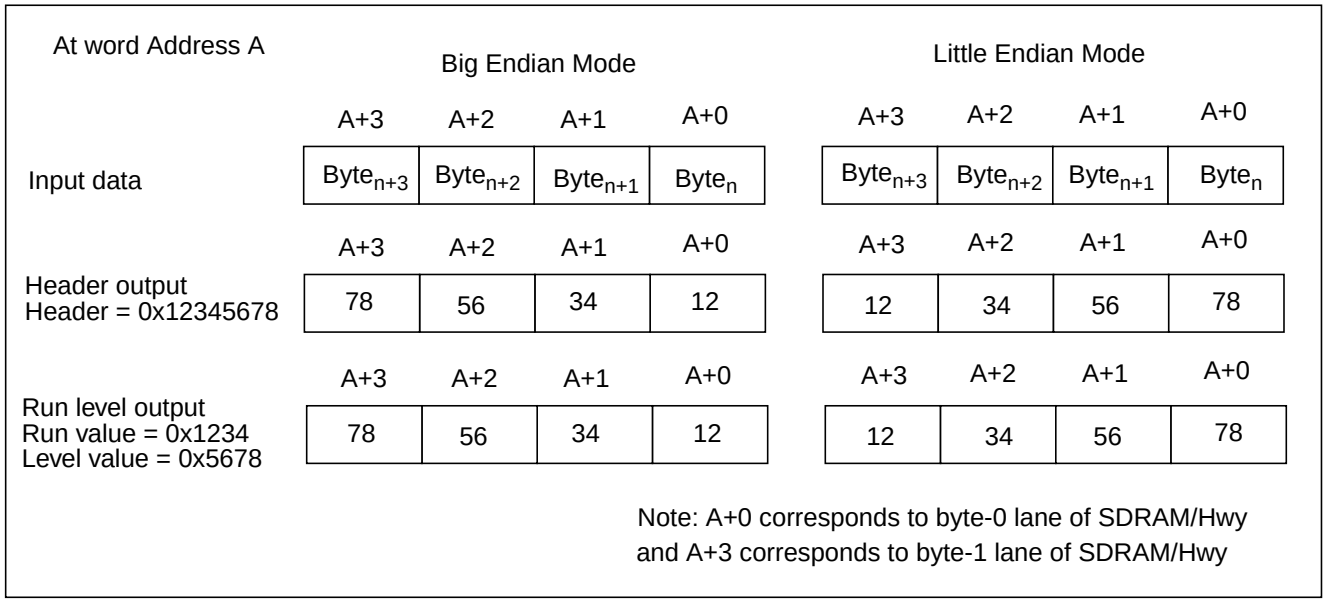


Figure C-14. VLD input and output data format

C.4.9 Compiler

The TCS compiler supports the loading of instruction in memory differently for Big Endian and Little Endian modes.

C.5 SUMMARY

TM1300 is required to operate in the same endian-ness as the host CPU. At reset, TM1300 operates in Big Endian mode; no special steps are required to set the Endian bits. When using TM1300 in Little Endian systems, the first transaction is to set the SE bit in the BIU_CTL register as described in the second paragraph of Section 11.7.5 on page 11-11.

C.6 REFERENCES

1. *PCI Multimedia Design Guide*, revision 1.0 - dated March 29,1994
2. *Designing PCI Cards and Drivers for Power Macintosh Computers*, By Apple Computer, Inc.; Reference: R0650LL/A; Phone: 1-800-282-2732

Index

Numerics

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Philips Semiconductors – a worldwide company

Argentina: see South America

Australia: 3 Figtree Drive, HOMEBUSH, NSW 2140,
Tel. +61 2 9704 8141, Fax. +61 2 9704 8139

Austria: Computerstr. 6, A-1101 WIEN, P.O. Box 213,
Tel. +43 1 60 101 1248, Fax. +43 1 60 101 1210

Belarus: Hotel Minsk Business Center, Bld. 3, r. 1211, Volodarski Str. 6,
220050 MINSK, Tel. +375 172 20 0733, Fax. +375 172 20 0773

Belgium: see The Netherlands

Brazil: see South America

Bulgaria: Philips Bulgaria Ltd., Energoproject, 15th floor,
51 James Bourchier Blvd., 1407 SOFIA,
Tel. +359 2 68 9211, Fax. +359 2 68 9102

Canada: PHILIPS SEMICONDUCTORS/COMPONENTS,
Tel. +1 800 234 7381, Fax. +1 800 943 0087

China/Hong Kong: 501 Hong Kong Industrial Technology Centre,
72 Tat Chee Avenue, Kowloon Tong, HONG KONG,
Tel. +852 2319 7888, Fax. +852 2319 7700

Colombia: see South America

Czech Republic: see Austria

Denmark: Sydhavnsgade 23, 1780 COPENHAGEN V,
Tel. +45 33 29 3333, Fax. +45 33 29 3905

Finland: Sinikalliontie 3, FIN-02630 ESPOO,
Tel. +358 9 615 800, Fax. +358 9 6158 0920

France: 51 Rue Carnot, BP317, 92156 SURESNES Cedex,
Tel. +33 1 4099 6161, Fax. +33 1 4099 6427

Germany: Hammerbrookstraße 69, D-20097 HAMBURG,
Tel. +49 40 2353 60, Fax. +49 40 2353 6300

Hungary: see Austria

India: Philips INDIA Ltd, Band Box Building, 2nd floor,
254-D, Dr. Annie Besant Road, Worli, MUMBAI 400 025,
Tel. +91 22 493 8541, Fax. +91 22 493 0966

Indonesia: PT Philips Development Corporation, Semiconductors Division,
Gedung Philips, Jl. Buncit Raya Kav.99-100, JAKARTA 12510,
Tel. +62 21 794 0040 ext. 2501, Fax. +62 21 794 0080

Ireland: Newstead, Clonskeagh, DUBLIN 14,
Tel. +353 1 7640 000, Fax. +353 1 7640 200

Israel: RAPAC Electronics, 7 Kehilat Saloniki St, PO Box 18053,
TEL AVIV 61180, Tel. +972 3 645 0444, Fax. +972 3 649 1007

Italy: PHILIPS SEMICONDUCTORS, Via Casati, 23 - 20052 MONZA (MI),
Tel. +39 039 203 6838, Fax +39 039 203 6800

Japan: Philips Bldg 13-37, Kohnan 2-chome, Minato-ku, TOKYO 108-
8507, Tel. +81 3 3740 5130, Fax. +81 3 3740 5057

Korea: Philips House, 260-199 Itaewon-dong, Yongsan-ku, SEOUL,
Tel. +82 2 709 1412, Fax. +82 2 709 1415

Malaysia: No. 76 Jalan Universiti, 46200 PETALING JAYA, SELANGOR,
Tel. +60 3 750 5214, Fax. +60 3 757 4880

Mexico: 5900 Gateway East, Suite 200, EL PASO, TEXAS 79905, Tel. +9-
5 800 234 7381, Fax +9-5 800 943 0087

Middle East: see Italy

Netherlands: Postbus 90050, 5600 PB EINDHOVEN, Bldg. VB,
Tel. +31 40 27 82785, Fax. +31 40 27 88399

New Zealand: 2 Wagener Place, C.P.O. Box 1041, AUCKLAND,
Tel. +64 9 849 4160, Fax. +64 9 849 7811

Norway: Box 1, Manglerud 0612, OSLO,
Tel. +47 22 74 8000, Fax. +47 22 74 8341

Pakistan: see Singapore

Philippines: Philips Semiconductors Philippines Inc.,
106 Valero St. Salcedo Village, P.O. Box 2108 MCC, MAKATI,
Metro MANILA, Tel. +63 2 816 6380, Fax. +63 2 817 3474

Poland: Al.Jerozolimskie 195 B, 02-222 WARSAW,
Tel. +48 22 5710 000, Fax. +48 22 5710 001

Portugal: see Spain

Romania: see Italy

Russia: Philips Russia, Ul. Usatcheva 35A, 119048 MOSCOW,
Tel. +7 095 755 6918, Fax. +7 095 755 6919

Singapore: Lorong 1, Toa Payoh, SINGAPORE 319762,
Tel. +65 350 2538, Fax. +65 251 6500

Slovakia: see Austria

Slovenia: see Italy

South Africa: S.A. PHILIPS Pty Ltd., 195-215 Main Road Martindale,
2092 JOHANNESBURG, P.O. Box 58088 Newville 2114,
Tel. +27 11 471 5401, Fax. +27 11 471 5398

South America: Al. Vicente Pinzon, 173, 6th floor, 04547-
130 SÃO PAULO, SP, Brazil, Tel. +55 11 821 2333, Fax. +55 11 821 2382

Spain: Balmes 22, 08007 BARCELONA,
Tel. +34 93 301 6312, Fax. +34 93 301 4107

Sweden: Kottbygatan 7, Akalla, S-16485 STOCKHOLM,
Tel. +46 8 5985 2000, Fax. +46 8 5985 2745

Switzerland: Allmendstrasse 140, CH-8027 ZÜRICH,
Tel. +41 1 488 2741 Fax. +41 1 488 3263

Taiwan: Philips Semiconductors, 6F, No. 96, Chien Kuo N. Rd., Sec. 1,
TAIPEI, Taiwan Tel. +886 2 2134 2886, Fax. +886 2 2134 2874

Thailand: PHILIPS ELECTRONICS (THAILAND) Ltd., 209/2 Sanpavuth-
Bangna Road Prakanong, BANGKOK 10260,
Tel. +66 2 745 4090, Fax. +66 2 398 0793

Turkey: Yukari Dudullu, Org. San. Blg., 2.Cad. Nr. 28 81260 Umraniye,
ISTANBUL, Tel. +90 216 522 1500, Fax. +90 216 522 1813

Ukraine: PHILIPS UKRAINE, 4 Patrice Lumumba str., Building B, Floor 7,
252042 KIEV, Tel. +380 44 264 2776, Fax. +380 44 268 0461

United Kingdom: Philips Semiconductors Ltd., 276 Bath Road, Hayes,
MIDDLESEX UB3 5BX, Tel. +44 208 730 5000, Fax. +44 208 754 8421

United States: 811 East Arques Avenue, SUNNYVALE, CA 94088-3409,
Tel. +1 800 234 7381, Fax. +1 800 943 0087

Uruguay: see South America

Vietnam: see Singapore

Yugoslavia: PHILIPS, Trg N. Pasica 5/v, 11000 BEOGRAD,
Tel. +381 11 3341 299, Fax.+381 11 3342 553

For all other countries apply to: Philips Semiconductors,
International Marketing & Sales Communications, Building BE-p, P.O. Box 218,
5600 MD EINDHOVEN, The Netherlands, Fax. +31 40 27 24825

Internet: http://www.semiconductors.philips.com

Printed in the United States of America Date of release: 2000 May 30 Document order number: 9397 750 07159

Let's make things better.

Philips Semiconductors

PHILIPS