Chapter 4
DSP16/DSP16A
Device Programming

www.DataSheet4U.com

CHAPTER 4. DSP16/DSP16A DEVICE PROGRAMMING

CONTENTS

www.DataSheet4U.com

4. DSP16/DSP16A DEVICE PROGRAMMING

This chapter discusses various aspects of programming the DSP16 and DSP16A devices. Many of the topics are illustrated in the complete sample programs presented in Appendix B. Techniques for programming the serial and parallel I/O sections of the device may be found in Chapters 5 and 6, respectively.

Chapter 3 described the instruction set specific to the DSP16/DSP16A device. Programming examples in this manual follow the assembler syntax of the *WE* DSP16/DSP16A Support Software Library for DSP16/DSP16A source files. An overview of the DSP16/DSP16A assembly language is provided in the following section.

4.1 DSP16/DSP16A ASSEMBLY-LANGUAGE NOTATION

A DSP16/DSP16A source file exists as a text file and contains DSP16/DSP16A instructions, directives to allow the assembler to interpret the instructions and data, and comments to clarify the use of the program. The syntax of the assembler directives is described in this chapter; also described are conventions and nomenclature used throughout the remainder of this manual. Appropriate formats for DSP16/DSP16A source files are also discussed.

4.1.1 Integer Notation

Decimal, hexadecimal, or octal expressions may be freely mixed when specifying numerical data in a source file. The syntax is identical to C-language programming.

- **Decimal.** Any string of normal digits (0—9) is interpreted as a decimal number, provided it does not have a leading zero.
- **Hexadecimal.** A numerical string beginning with 0x is interpreted as a hexadecimal number and may contain the digits 0—9, a—f, or A—F. For example, 0x0 is the same as 0. 0x010 is the same as 0x10, which is the decimal number 16. And 0xFF or 0xff is the decimal number 127.
- **Octal.** A numerical string beginning with the digit zero is interpreted as an octal number and may contain the digits 0—7. For example, 07 is the decimal number 7 and 010 is the decimal number 8.
- **Fixed-Point.** Numbers with a decimal point are interpreted as binary fixed-point numbers by the DSP16/DSP16A assembler. The number of binary digits to the right of the decimal point is 14 by default, but may be changed (see the *WE® DSP16 and DSP16A Support Software Library User Manual*).

4.1.2 Comments

Comments may be placed in the source file to enhance readability and to provide information for other users. A comment may be placed on a line by itself or may appear at the end of a line containing an instruction. The following lines are examples of valid comments:

www.DataSheet4U.com

```
/* This is a valid comment */
instruction
instruction          /* this is a valid comment */
```

4.1.3 Labels

Labels in a source file serve two purposes: to give a descriptive name to a particular location and to provide a destination for a branch instruction. Labels may consist of upper- and/or lower-case alphanumeric characters and the underscore, although the first character may not be numeric. A label must be terminated with a colon. Labels may be as long as necessary to be descriptive; however, only the first eight characters are significant. The following lines show examples of valid labels:

```
start_1: instruction     /* "start_1" is a valid label */
         instruction
    end: instruction     /* "end" is a valid label */
```

4.1.4 Data Stored in ROM

Data may be stored in a ROM location by using the int directive. The following lines of source code are examples of how to store data in ROM:

```
   table: int     0xFF          /* Initialize one ROM location    */
          2*int   0x10 0xA2     /* Initialize four ROM locations  */
          3*int   23            /* Initialize three ROM locations */
   fixed: int     1.23 -1.634   /* Initialize two ROM locations   */
tab_end:  int     3.721!10      /* Initialize one ROM location    */
```

As shown above, multiple ROM locations may be specified with a single statement. In the second example, two ROM locations are replicated to initialize four ROM locations; however, in the third example, all three locations are initialized to the same value.

Following the label *fixed*, two ROM locations are initialized in a fixed-point notation. By default, the DSP16/DSP16A assembler assumes that fixed-point numbers are to be assembled with 14 bits of precision and 2 bits of magnitude. An environment variable, precision, may be changed to allow other values of precision.

The last example demonstrates another method to specify the precision of a fixed-point format. The suffix !N (where N is the desired precision) can be used to force different precision encodings "on the fly." In this case, 3.721 is encoded with 6 magnitude bits and 10 precision bits.

4.1.5 RAM Variables

RAM variables can be allocated similarly to data stored in ROM by surrounding the int directives with the .ram and .endram directives. Note that RAM locations are allocated without being initialized. The following sequence allocates six RAM variables:

www.DataSheet4U.com

```
.ram
data1:  int       /* allocate 1 RAM variable  */
data2:  2*int     /* allocate 2 RAM variables */
data4:  3*int     /* allocate 3 RAM variables */
.endram
```

4.1.6 DSP16/DSP16A Source-File Format

A DSP16/DSP16A source file is prepared as a text file by using a text editor with the *UNIX* Operating System or *MS-DOS* Operating System. (See Appendix B for complete DSP16/DSP16A program listings.) When creating a source file, the following conventions should be observed:

- The source file name must end with ".s".
- Directives beginning with "." (such as .ram and .endram) must begin in the first column.
- White space is used to separate the fields of instructions. Either a space or a tab character constitutes white space. Using tabs to separate and align the fields improves the readability of source files.
- Labels normally begin in the first column to enhance readability, but may be indented if desired.
- It is customary, but not required, to place the title and a brief description of the program at the top of the file for reference.

4.2 PROGRAMMING TECHNIQUES

The following sections describe problems commonly encountered when programming the DSP16/DSP16A device and their possible solutions. In general, many of the problems encountered when programming other digital signal processors (such as latency and pipeline effects) have been eliminated by the design of the DSP16 and DSP16A devices.

4.2.1 Instruction Set Ambiguities

Several instructions, which normally would be written identically, can be interpreted as various types of instructions. This interpretation of the instructions determines the number of ROM locations used to store the instruction, the number of instruction cycles used to execute the instruction, and whether or not the instruction affects the flags. Hence, the interpretation can be critical. For example, the instruction

```
a0 = y
```

could be a multiply/ALU, special function, or data move instruction. When the instruction is interpreted as a multiply/ALU or special function instruction, the instruction requires one ROM location and executes in one instruction cycle. When the instruction is interpreted as a data move instruction, the instruction requires one ROM location and executes in two instruction cycles. The interpretation of the instruction is critical if conditional testing based on the results of the instruction execution is performed. The DSP16/DSP16A flags are affected by the multiply/ALU and special function instructions, but not by the data move instructions.

www.DataSheet4U.com

www.DataSheet4U.com

The *WE* DSP16/DSP16A Support Software Library provides optional mnemonics that may be used with an instruction to specify its type. Table 4-1 shows the mnemonics that can be used to specify the type of instruction. For example, the instruction

```
au  a0 = y
```

is interpreted as a multiply/ALU instruction.

Table 4-1. Optional Mnemonics

Use	To Specify
au	Multiply/ALU instruction
if true	Special function instruction
set	Short immediate instruction
move	Data move instruction

If an instruction may be encoded several ways, the assembler chooses the encoding based on the following priority:

1. Special function
2. Multiply/ALU
3. Short immediate
4. Data move

4.2.2 Polling for I/O

When not using interrupt driven I/O, polling for input and output conditions is the simplest means of handling I/O timing. The following segment of code continuously polls the pioc register to determine if the condition IBF is true, meaning that there is data in the serial input register waiting to be processed. When data is loaded into the serial input buffer from an external device, program execution continues below the wait loop.

```
        y = 0x010            /* place mask into y register  */
wait:   a0 = pioc            /* check pioc register for IBF */
        a0 & y               /* - look only at bit 4        */
        if eq goto wait      /* - if no input, wait.        */
          .
          .
        *r0 = sdx            /* move data into RAM */
```

This same code fragment can be used to poll any I/O condition by changing the value in register y, which is used to mask the unwanted bits of the pioc register. For example, use 0x04 to check only the condition PIDS, which indicates that a parallel input was performed.

4.2.3 Modulo Addressing

Modulo addressing is provided to allow efficient implementation of cyclical memory accesses. To use modulo addressing, the first RAM address of the modulo must be loaded into register rb and the last RAM address into re. The register being used as the memory pointer must be postincremented by +1. Each time the pointer is used, its value is compared with the contents of register re (before the postincrement is performed). If the two values are equal, the value of register rb is loaded into the register being used to address the RAM and the cycle repeats.

It is important to note that whenever register re contains a value not equal to zero, modulo addressing is active. On reset, the value of re is zero. Whenever modulo addressing is not used, this register should contain zero and should not be used to store any number other than the address of the end of a modulo.

4.2.4 Random Number Generation

The DAU includes a 10-state pseudorandom binary sequence (PRBS) generator, which is used to toggle a bit in the DAU. The status of this bit may be determined by testing for the "heads" or "tails" condition. The following segment of code generates a 16-bit random number in the high half of accumulator a0 by randomly setting each of the 16 bits:

```
do 16 {
   if heads a0h = a0h + 1      /* if heads, set bit to 1 */
   a0 = a0 << 1                /* shift left 1 position  */
}
```

The pseudorandom sequence is incremented each time it is tested and may be reset by writing any value to the pi register (writing to the pi register does not affect its contents except when in an interrupt service routine). (See Section 4.2.5.)

4.2.5 Programming Tips

The following section describes several practical programming tips that may not be obvious to a new user of the DSP16/DSP16A.

1) When loading count values into c0 and c1, the count value is 1 − *count*, where *count* is the desired number of times the loop is to be executed. An easy way to assemble the loop counter load is to let the assembler compute the 1 − *count* value. For example, if a loop is to be repeated 10 times, the following code could be used:

```
         c0 = 1 - 10
   loop: ...
         if c0lt goto loop
```

The assembler correctly computes 1 − count, and the code is easier to read.

www.DataSheet4U.com

2) If extra 16-bit registers are needed, there are several possible ways to "create" them.

a) If not using interrupts or development system breakpoints, an **icall** instruction may be placed at location 0. This causes a branch to location 2 (where program execution begins) and makes the DSP16/DSP16A "think" that it is in an interrupt service routine (ISR). While in an ISR, the DSP16/DSP16A no longer updates the pi register each time the pc register changes, and the pi register may be written to (writes to pi do not affect its contents when not in an ISR, but writing the pi register resets the pseudorandom sequence generator). When in an ISR, the pi register is not used by the DSP16/DSP16A and is free for use as a general-purpose 16-bit register.

b) While not in a subroutine, the pr register is available as a general-purpose 16-bit register.

c) While not doing ROM table lookups, the pt register is available as a general-purpose 16-bit register. It can easily be incremented or modified using:

```
{y = Y, Y = aT, Z : y}  x = *pt++    /* load of y necessary when */
                                     /* loading x from ROM       */
```

or

```
i = N                                /* or -N */
{y = Y, Y = aT, Z : y}  x = *pt++i
```

Note: The XAAU adder is only 12 bits wide, therefore, modifying as above is modulo 4K, i.e.,

```
pt = 4095
{y = Y, Y = aT, Z : y}  x = *pt++
                                   /* pt is now 0,      */
                                   /* not 4096 (2**12) */
```

However,

```
a0 = pt
a0h = a0h + 1
pt = a0
```

is no problem, except above 32767 unless saturation logic is disabled on a0 (since a value above 32767 appears to be an overflowed 2's complement value).

3) While not using modulo addressing (re = 0), the rb register is available as a general-purpose register. The re register is not available since a non-zero value enables modulo addressing. Note that all YAAU register are 9 bits wide in the DSP16 and 16 bits wide in the DSP16A.

4) If a write of 0 to a RAM location is required, and modulo addressing is not being used, the re register can be used (re is zero by definition).

```
*rN [++, --, ++j] = re
```

clears the RAM location pointed to by rN with no setup required.

www.DataSheet4U.com

5) If adding and subtracting accumulators without using y is desired, the following instructions could be used to perform an add (assuming that only the high half of an accumulator is being used or the high half is a whole number and the low half is a fraction):

```
a0 = a0 >> 4
a0h = a0h + 1
a0 = a0 << 4          /* adds 16 (2**4) to a0        */
                      /* (similarly, << 8 adds 256) */

a0 = -a0
a0h = a0h + 1
a0 = -a0              /* subtracts 1 from a0 */
```

Shifting left and adding can be used to add fractions.

6) The following two-cycle data move instructions can be coded as a single-cycle multiply/ALU instruction (when executing in the cache) by doing a dummy load to x.

```
do 40 {
      y = aN          /* 2-cycle data move */
      }
```

This takes 81 machine cycles, while:

```
do 40 {
      y = aN   x = *pt++     /* single-cycle when in cache */
      }
```

takes only 43 machine cycles (2 when it is loaded the first time and 2 the last time it is executed). In both cases, the **do** instruction requires 1 cycle. Note that this is a trivial example to make the cycle counts more obvious. This "trick" is most useful when the kernel of an operation is in the cache and a result needs to be multiplied by a coefficient or operated on by the ALU.

7) The above assumes that pt has already been set and that postincrementing pt does not affect anything. If this is not true (postincrementing pt is not desired), the following can be done:

```
i = 0
do 40 {
      y = aN   x = *pt++i        /* postincrement by 0 */
      }
```

This does not alter the value of pt.

4.2.6 Concurrent Interrupts

Consideration must be given to situations in which multiple interrupting conditions occur. The DSP16/DSP16A device does not allow nesting of interrupts; however, there are other ways to guarantee that all interrupts can be recognized and serviced.

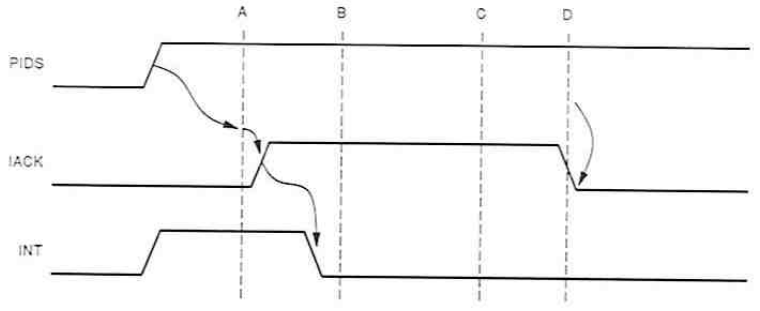
www.DataSheet4U.com

Case 1

If an internal and external interrupt request occur at nearly the same time and before the execution of the branch-to-one (start of interrupt service routine), the status field in the pioc register can be examined. In this case, the status will indicate that both interrupts are pending. They can be serviced accordingly.

RULE: An interrupt occurring after an internal interrupt occurs and before IACK is asserted (in response to the internal interrupt) causes the INT bit in the pioc register (bit 0) to be set, providing that INT meets its assertion time requirements.

The INT signal is negated on the rising edge of IACK.



A. Branch-to-one instruction executed. Beginning of interrupt service responding to negation of PIDS.

B. pioc register has PIDS and INT status bits set.

C. ireturn instruction executed. End of interrupt service routine.

D. Next interruptible instruction.

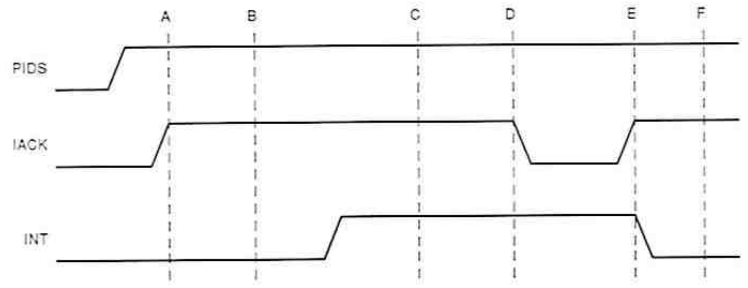
Figure 4-1. Case 1 – Internal Interrupt (PIDS) and INT Occur Before Assertion of IACK

www.DataSheet4U.com

Case 2

If INT is asserted (high) when IACK is already asserted (i.e., when the DSP16 device is servicing another interrupt), then INT must remain asserted until the next rising edge of IACK. This is because the internal interrupt request is cleared on the falling edge of IACK. This guarantees that the interrupt request (assertion of INT) will be serviced at the next interruptible instruction after the currently executing interrupt service routine has finished.

RULE: To guarantee recognition of INT when it is asserted during an interrupt service routine (IACK high), INT should not be negated until the next rising edge of IACK, providing that INT assertion time is met.

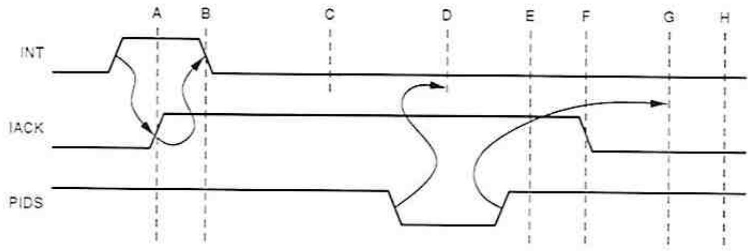


A. Branch-to-one instruction executed in response to internal interrupt (PIDS).
B. pioc register has PIDS status bit set.
C. ireturn instruction executed.
D. Next interruptible instruction.
E. Branch-to-one instruction executed in response to INT.
F. pioc register has INT status bit set.

Figure 4-2. Case 2 – INT Asserted During Service of Internal Interrupt After pioc Status is Checked

Case 3

Internal interrupt requests remain pending until the respective pdx or sdx registers are serviced. Hence, if an external interrupt is being serviced and another internal interrupt request is generated, the internal interrupt request remains pending and causes a second interrupt to be taken at the next interruptible instruction. In this way, the internal interrupt is not missed if it occurs during the servicing of another external interrupt.



A. Branch-to-one instruction.
B. pioc register has INT status bit set.
C. Read of pioc register status.
D. pioc register has PIDS status bit set.
E. iretum instruction executed.
F. Next interruptible instruction.
G. Branch-to-one instruction. Begin to service internal interrupt.
H. Service internal interrupt.

Figure 4-3. Case 3 – Internal Interrupt Asserted While Servicing an External Interrupt

www.DataSheet4U.com
www.DataSheet4U.com

www.DataSheet4U.com

Case 4

If it is possible for two or more interrupt requests to be pending, the easiest method for servicing these interrupts is to service the external interrupt first and then the internal interrupt requests individually (by taking a new interrupt for each internal request) until no more interrupts are pending. The drawback of this procedure is that if external interrupts are frequent, there may be a large latency when servicing internal interrupts.

4.2.7 Interrupt Latency

Two classes of DSP16/DSP16A instructions are not interruptible. The first class contains all branch instructions. The second class contains instructions that are executing in the cache (i.e., any instruction when executing from the on-chip instruction cache cannot be interrupted).

Interrupt latency is bounded by the longest in-cache operation. In situations where interrupt latency is critical, in-cache operations should be split into smaller cache operations whose execution time is less than any critical latency requirements. In this situation, an interruptible instruction must be placed between successive cache instructions.

For example:

```
do 93 {
       instr
       instr
         .
         .
         .
       instr
      }
nop                      /* interruptible instruction */
redo 50
nop                      /* interruptible instruction */
redo 50
```

www.DataSheet4U.com