Chapter 3

DSP16/DSP16A

Instruction Se

CHAPTER 3. DSP16/DSP16A INSTRUCTION SET

CONTENTS

3. DSP16/DSP16A INSTRUCTION SET

All DSP16/DSP16A instructions are 16 bits wide and have a C-like syntax. Pipelining of the instructions is necessary to achieve the real-time performance required by many signal-processing applications. To facilitate programming, the degree of pipelining in the DSP16/DSP16A device has been reduced and the latency effects present in previous generation DSPs have been eliminated. The instructions fall into one of five possible categories:

- **Multiply/ALU** instructions are the primary instructions used to implement signal-processing programs. These instructions perform multiply/accumulate, logical, and other ALU functions and also transfer data between memory and registers in the data arithmetic unit.
- **Special Function** instructions are used to perform such operations as rounding, negation, and logical left shifts and arithmetic right shifts of accumulators. Special function instructions may be conditionally executed on the basis of the state of internal flags.
- **Control** instructions are used to control program flow. The **call**, **goto**, and **return** instructions are provided and may be conditionally executed on the basis of the state of internal flags.
- **Data Move** instructions are used to transfer data between registers, memory, and accumulators. Immediate loads of certain registers are also possible.
- **Cache** instructions allow the implementation of low overhead loops by loading a set of multiply/ALU and special function instructions into a cache memory and repetitively executing them (up to 127 times).

The following sections describe in detail the notation used in the instruction set, the addressing modes supported, the internal flags used by conditional instructions, and the five groups of instructions.

Note: Only multiply/ALU and special function instructions set DAU flags.

3.1 NOTATION

The following operators are used to describe the instruction set:

Operator	Meaning
*	16 × 16 → 32-bit multiplication (Denotes register-indirect addressing when used as a prefix to an address register)
+	36-bit addition
–	36-bit subtraction
++	Register postincrement
– –	Register postdecrement
>>	Arithmetic right shift
<<	Logical left shift

&	32-bit bitwise AND
|	32-bit bitwise OR
^	32-bit bitwise EXCLUSIVE OR
:	Compound addressing

For all instructions listed in this chapter, the following are true:

- Brackets, [ ], are not part of the instruction syntax, but indicate that the enclosed item is optional.
- Parentheses, ( ), and braces, { }, are part of the instruction syntax and must appear where shown in the instruction.

The valid instruction groups for the DSP16/DSP16A device are represented in Tables 3-3 to 3-12. The items in Tables 3-3 to 3-12 and 3-14 to 3-23 that are written in lower-case letters are proper statements and must appear where shown in the instruction. The items with capital letters are not proper statements and are replaced with immediate data, a register name, or a condition.

3.2 ADDRESSING MODES

The DSP16/DSP16A Digital Signal Processor allows immediate, indirect, and compound addressing modes. Instructions using indirect and compound addressing are typically used to encode real-time, signal-processing algorithms and, hence, require less program memory and execute faster than immediate addressing.

3.2.1 Immediate Addressing

In immediate addressing, the operand is supplied in the instruction. This situation is useful when initializing registers and is provided at the expense of one additional ROM location and one instruction cycle of execution time. A short immediate addressing mode is supplied to set the YAAU registers, r0—r3, j, k, rb, and re which are 9 bits wide on the DSP16. The DSP16A YAAU registers are 16 bits wide, so short immediate addressing may only be used when loading values that are 9 bits long or less. Short immediate instructions execute in one cycle, use one ROM location, and are cacheable.

3.2.2 Indirect Addressing

Indirect addressing allows a register to be used as a pointer to another location. The terms X and Y specify the source of data from memory to registers or the destination of data from registers to memory:

X = *pt++ or *pt++i

Y = one of: *rM, *rM++, *rM– –, *rM++j

Note: M = one of: 0, 1, 2, 3

The term X represents the ROM data to be copied into the x register. The term Y represents the RAM data to be copied into the specified register or the data written to RAM from a register. The

www.DataSheet4U.com

mnemonics for X and Y indicate register indirect addressing with a postmodification of the address pointer. The asterisk preceding the RAM or ROM address register stands for "the data pointed to by the address in the register." The mnemonics have the following meaning:

- ***r*M**. This example means "the data pointed to by the address in the register." The contents of the register are not altered by the operation.
- ***r*M++, pt++**. The "++" following the address register indicates a postincrement of the address register. This example means "the data pointed to by the address in the register; add 1 to the contents of the register after the operation is complete."
- ***r*M—**. The "—" following the address register indicates a postdecrement of the address register. This example means "the data pointed to by the address of the register: subtract 1 from the contents of the register after the operation is complete."
- ***r*M++j**. The "++j" following the address register indicates a postincrement of the address register. This example means "the data pointed to by the address in the register; add the value of register j to the contents of the address register after the operation is complete." Negative values of j yield a postdecrement.
- ***pt++i**. The "++i" following the address register indicates a postincrement of the address register. This example means "the data pointed to by the address in the register; add the value of register i to the contents of the address register after the operation is complete." Negative values of i yield a postdecrement.

Modulo (virtual shift) addressing uses indirect addressing to form the equivalent of a cyclic shift register within the RAM. Addresses loaded into registers rb and re define the first and last physical addresses of the modulo, respectively. When a register is used as a memory pointer, its value is compared with re. If its value is equal to the contents of re and the postincrement is +1, then the value in rb is copied into the register after the memory access is complete. See Section 4.2.3.

3.2.3 Compound Addressing

Compound addressing is a memory read/write operation using only one pointer register. The term Z specifies a source and a destination for a compound RAM read followed by a write sequence. The mnemonics for Z are a shorthand notation for the compound addressing functions explained below and shown in Table 3-1. The term *temp* used in the descriptions is a hypothetical register used for illustration only.

Table 3-1. Compound Addressing Instructions

Instruction	Operations		
Z: R	Step 1	Step 2	Step 3
*rMzp:R	temp=R;	R=*rM;	*rM++=temp;
*rMpz:R	temp=R;	R=*rM++;	*rM=temp;
*rMm2:R	temp=R;	R=*rM—;	*rM++2=temp;
*rMjk:R	temp=R;	R=*rM++j;	*rM++k=temp;

Notes:
M can be 0, 1, 2, 3.
R can be one of x, y, yl, r0, r1, r2, r3, pt, pr, i, j, k, c0, c1, c2, rb, re, psw, auc, sioc, srta, sdx, tdms, pioc, a0, a0l, a1, a1l.
R and rM must not be the same register (i.e., r1pz:r1).

As with other instructions that use the y, a0, and a1 registers, the following rules apply when using the compound addressing mode:

- If clearing of the low half of the register is enabled (according to the CLR field of the auc register), the low half of the register is cleared when the high half is loaded.
- If saturation on overflow is enabled (according to the SAT field of the auc register), the value in the accumulator is limited. See Section 2.5.1.

Virtual shift addressing may be used with compound addressing. The contents of the address register are compared with the contents of register re during both the read and write cycles. If the contents of the address register are equal to the contents of re during the read cycle and the "*rMpz" mode is specified, r*M* is loaded with the contents of rb. If the contents of the address register are equal to the contents of re during the write cycle and the "*rMzp" mode is specified, r*M* is loaded with the contents of rb. See Section 4.2.3.

3.3 PROCESSOR FLAGS

Control and special function instructions may be conditionally executed on the basis of internal flags set by the previous ALU operation, the condition of one of the counters, or the value of a randomly set bit in the device. Multiply/ALU function statements and special function instructions affect the flags; loading an accumulator with a multiply/ALU transfer statement or a data move instruction does not affect the flags. The processor flags and their meanings are:

LMI **Logical Minus** – A logical minus is determined by the state of bit 35 of the last DAU operation result. If bit 35=1, the result is a negative number and LMI is true.

LEQ **Logical Equal** – A logical equal is determined by the sum of bits 35—0 of the last DAU operation result. If the sum of the bits equals zero, the result is zero and LEQ is true.

LLV **Logical Overflow (36-Bit Overflow)** – LLV is true if the sign of the result of an operation cannot be represented in a 36-bit accumulator.

LMV **Mathematical Overflow (32-Bit Overflow)** – LMV is true if the overflow bits (35—31) of the accumulator used in the last DAU operation are not identical. This indicates a number not representable in 32 bits.

Table 3-2 shows the mnemonics that are used in conditional instructions and their meanings. The state of the internal flags that causes the condition to be true is enclosed in parentheses after the description. For example, when testing the condition le, the result is true if either the logical minus (LMI) or logical equal (LEQ) flags are true.

Table 3-2. Conditional Mnemonics

Test	Meaning	Test	Meaning
pl	Result is nonnegative (not LMI).	mi	Result is negative (LMI).
eq	Result is equal to zero (LEQ).	ne	Result is not equal to zero (not LEQ).
gt	Result is greater than zero (not LMI and not LEQ).	le	Result is less than or equal to zero (LMI or LEQ).
lvs	Logical overflow set (LLV).	lvc	Logical overflow clear (not LLV).
mvs	Mathematical overflow set (LMV).	mvc	Mathematical overflow clear (not LMV).
c0ge*	Counter 0 greater than or equal to zero.	c0lt*	Counter 0 less than zero.
c1ge*	Counter 1 greater than or equal to zero.	c1lt*	Counter 1 less than zero.
heads†	Pseudorandom sequence bit set.	tails†	Pseudorandom sequence bit clear.
true	The condition is always satisfied in an if instruction.	false	The condition is never satisfied in an if instruction.

* Testing each of these conditions increments the respective counter being tested.

† The heads or tails condition is determined by a randomly set or cleared bit, respectively. The bit is randomly set with probability of 0.5. The random bit is generated by a 10-state pseudorandom sequence generator that is updated after either a heads or tails test. The pseudorandom sequence may be reset by writing any value to the pi register. Writing to the pi register does not affect the contents of the pi register except while in an interrupt service routine. A random rounding function can be implemented by using either of these two conditions.

3.4 MULTIPLY/ALU GROUP

The multiply/ALU instructions are the primary instructions used to implement signal-processing programs. Statements from this group can be combined to generate multiply/accumulate, logical, and other ALU functions and to transfer data between memory and registers in the data arithmetic unit. In the examples presented, the statements should be read from right to left, top to bottom. Statements within a multiply/ALU instruction are executed essentially in parallel. The multiply/ALU instructions usually consist of more than one part. Each part of an instruction is called a statement. The general rule is that valid instructions can be formed by choosing one statement from each statement column in Table 3-3. If either statement is not required, then a single statement from either column also constitutes a valid instruction. Conversely, valid instructions can be decomposed into separate statements, with each coming from a different column in the Table 3-3.

The multiply/ALU instructions consist of two parts: a function and a transfer (see Table 3-3). The statements in the function column can be separated into two types: those involving the multiplier and those involving only the ALU in the data arithmetic unit. The multiply/accumulate instructions typically used in signal-processing applications are assembled from statements from the function column that include the multiplication of the data in x and y[31—16]. In a multiply/accumulate instruction, the x and y registers are loaded with the operands, the product of the previous operands is generated, and the previous product is accumulated in a0 or a1.

The following example shows how a typical multiply/accumulate sequence is implemented.

Example:

```
Instruction #
   (1)                              y=Y    x=X
   (2)                    p=x*y
   (3)      aD=aS+p
```

In the example presented, the data in the X source is copied into the x register and the data in the Y source into bits 31—16 of the y register in line 1. In line 2, the product of the data in x and y[31—16] is generated and stored in p. In line 3 the data in the source accumulator, aS, and the data in p are added and the result loaded into the destination accumulator. Note that lines 2 and 3 could also have specified memory transfer operations for later instructions.

www.DataSheet4U.com

The ALU instructions perform one of the following:

- The logical operations of AND, OR, or XOR between an accumulator and the data in the y register.
- The addition or subtraction of the data in the y register from an accumulator.
- The load of an accumulator with the data in the y register.

The y register must be loaded prior to the ALU operation.

The following example shows how a typical logical operation is implemented.

```
(1)              y=Y
(2)  aD=aS&y
```

In this example, the data in the Y source is copied into the y register in line 1. In line 2, the logical AND of the data in the source accumulator, aS, and the data in y as a result of line 1 are calculated and the result is loaded into the destination accumulator.

All multiply/ALU instructions require 1 word of memory. The number of instruction cycles required to execute an instruction in the multiply/ALU group is a function of the statement selected from the transfer column in Table 3-3. Instructions with statements in the transfer column involving a write to RAM are executed in two instruction cycles whether the instruction is in or out of the cache. Instructions with statements in the transfer column involving a read from the RAM and the ROM simultaneously are executed in two instruction cycles if not in the cache and one instruction cycle if in the cache. An instruction with no transfer statement executes in one instruction cycle either in or out of the cache. The remaining instructions are executed in one instruction cycle either in or out of the cache. Table 3-3 gives the number of instruction cycles for each case. The multiply/ALU instructions use one ROM location.

The no operation (**nop**) instruction is a special-case encoding of a multiply/ALU instruction and is executed in one instruction cycle. The assembly-language notation representation of a no operation instruction is either **nop** or a single semicolon (;).

Note that the function statements and transfer statements in Table 3-3 are chosen independently. Any function statement may be combined with any transfer statement to form a valid multiply/ALU instruction.

Table 3-3. Multiply/ALU Instructions

Function Statements		Transfer Statements		Cycles Out/In Cache
	p=x*y	y=Y	x=X	2/1
aD=p	p=x*y	y=aT	x=X	2/1
aD=aS+p	p=x*y	y[l]=Y		1/1
aD=aS−p	p=x*y	aT[l]=Y		1/1
aD=p		x=Y		1/1
aD=aS+p		Y		1/1
aD=aS−p		Y=y[l]		2/2
aD=y		Y=aT[l]		2/2
aD=aS+y		Z:y	x=X	2/2
aD=aS−y		Z: y[l]		2/2
aD=aS&y		Z: aT[l]		2/2
aD=aSly				
aD=aS^y				
aS−y				
aS&y				

Table 3-4. Replacement Table for Multiply/ALU Instructions

Replace	Value	Meaning
aD, aS, aT	a0, a1	One of two DAU accumulators.
X	*pt++,*pt++i	ROM location pointed to by pt. pt is postmodified by +1 and i, respectively.
Y	*rM, *rM++, *rM—, *rM++j	RAM location pointed to by rM. (M= 0, 1, 2, 3). rM is postmodified by 0,+1,−1, and j, respectively.
Z	*rMzp, *rMpz, *rMm2, *rMjk	Read/write compound addressing. rM (M = 0, 1, 2, 3) is used twice. First, postmodified by 0, +1, −1, and j respectively and second, postmodified by +1, 0,+ 2, and k, respectively.

www.DataSheet4U.com

On the basis of the information given in Table 3-4, apply the following information to the function and transfer statements in Table 3-3:

- Loads of a0, a1, and y clear the lower half of the selected register when the appropriate CLR field bits in the auc register are zeroed.
- Loads of a0l, a1l, and yl do not change the data in the high half of the selected register.
- The y and p operands are sign-extended to match the width of the accumulators.

3.4.1 Function Statements

In the execution of these statements, the width of the number is extended to 36 bits, which is the size of the accumulators. This extension is accomplished by extending the sign bit in the p register to retain the correct 2's complement value. The multiplier performs a 2's complement multiply, using x and the high half of y (bits 31—16).

The statements must be written in the exact format shown. If the statements are written in any other way, for example, aD=p+aS instead of aD=aS+p, the assembler produces an error message.

- **p=x*y.** The contents of the x and the y (bits 31—16) registers are multiplied and the result is placed in the p register.
- **aD=p p=x*y.** The contents of the p register are copied into the destination accumulator, aD. The contents of the x and the y (bits 31—16) registers are multiplied and the result is placed in the p register. The bit alignment of the p register is a function of the ALIGN field of the auc register.
- **aD=aS+p p=x*y.** The contents of the source accumulator, aS, are added to the contents of the p register and the result is placed in the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register. The contents of the x and the y (bits 31—16) registers are multiplied and the result is placed in the p register.
- **aD=aS−p p=x*y.** The contents of the p register are subtracted from the contents of the source accumulator, aS, and the result is placed in the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register. The contents of the x and the y (bits 31—16) registers are multiplied and the result is placed in the p register.
- **aD=p.** The contents of the p register are copied into the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register.
- **aD=aS+p.** The contents of the source accumulator, aS, are added to the contents of the p register, and the result is placed in the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register.
- **aD=aS−p.** The contents of the p register are subtracted from the contents of the source accumulator, aS, and the result is placed in the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register.
- **aD=y.** The contents of the y register are copied into the destination accumulator, aD.
- **aD=aS+y.** The contents of the source accumulator, aS, are added to the contents of the y register and the result is placed in the destination accumulator, aD.
- **aD=aS−y.** The contents of the y register are subtracted from the contents of the source

accumulator, aS, and the result is placed in the destination accumulator, aD.

- **aD=aS&y.** The contents of the source accumulator, aS, are ANDed with the contents of the y register, and the result is placed in the destination accumulator, aD.
- **aD=aS|y.** The contents of the source accumulator, aS, are ORed with the contents of the y register, and the result is placed in the destination accumulator, aD.
- **aD=aS^y.** The contents of the source accumulator, aS, are XORed with the contents of the y register, and the result is placed in the destination accumulator, aD.
- **aS−y.** The contents of the y register are subtracted from the contents of the source accumulator, aS. The result is not placed in the destination accumulator, aD; however, the ALU flags are affected by the results of the subtraction.
- **aS&y.** The contents of the source accumulator, aS, are ANDed to the contents of the y register. The result is not placed in the destination accumulator, aD; however, the ALU flags are affected by the results of the AND function.

3.4.2 Transfer Statements

The transfer statements allow the user to transfer data from memory to the x and y registers and the accumulators, or from the y register and the accumulators to memory.

- **y=Y x=X.** The data from the specified Y source is loaded into the high half (bits 31—16) of the y register. The data from the specified X source is loaded into the x register. If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **y=aT x=X.** The data in the high half (bits 31—16) of the specified accumulator is loaded into the high half (bits 31—16) of the y register. The data from the specified X source is loaded into the x register. If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **y=Y.** The data from the specified Y source is loaded into the high half of the y register (bits 31—16). If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **yl=Y.** The data from the specified Y source is loaded into the low half of the y register (bits 15—0). The data in the high half of y is not altered.
- **aT=Y.** The data from the specified Y source is loaded into the high half (bits 31—16) of the specified accumulator. The guard bits (35—32) are loaded with the value of bit 31. If clearing of aTl is enabled (according to the CLR field of the auc register), the low half of the accumulator is cleared (0) when the high half is loaded.

www.DataSheet4U.com

- **aTl=Y**. The data from the specified Y source is loaded into the low half (bits 15—0) of the specified accumulator. The data in the high half of the accumulator is not altered.
- **x=Y**. The data from the specified Y source is loaded into the x register.
- **Y**. No data is transferred. This transfer statement is used to modify the address register specified.
- **Y=y**. The data in the high half of the y register (bits 31—16) is loaded into the specified Y destination.
- **Y=yl**. The data in the low half of the y register (bits 15—0) is loaded into the specified Y destination.
- **Y=aT**. The data in the high half (bits 31—16) of the specified accumulator is written into the specified Y destination. If saturation on overflow is selected (according to the SAT field of the auc register), the accumulator value is limited. See Section 2.5.1.
- **Y=aTl**. The data in the low half (bits 15—0) of the specified accumulator is written into the specified Y destination. If saturation on overflow is selected (according to the SAT field of the auc register), the accumulator value is limited. See Section 2.5.1.
- **Z:y x=X**. The data from the specified X source is loaded into the x register. The data from the specified Z source is loaded into the high half (bits 31—16) of the y register, and the old data from the high half of the y register is loaded into the Z destination. If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **Z:y**. The data from the specified Z source is loaded into the high half (bits 31—16) of the y register and the old data from the high half of the y register is loaded into the Z destination. If clearing of yl is enabled (according to the CLR field of the auc register), then yl is cleared (0) when the high half is loaded.
- **Z:yl**. The data from the specified Z source is loaded into the low half (bits 15—0) of the y register and the old data of the low half of the y register is loaded into the Z destination. Data in the high half of the y register is not altered.
- **Z:aT**. The data from the specified Z source is loaded into the high half (bits 31—16) of the specified accumulator. If clearing of aTl is enabled (according to the CLR field of the auc register), the low half of the accumulator is cleared (0) when the high half is loaded. The guard bits (35—32) are loaded with the value of bit 31. The old data from the high half of the accumulator is loaded into the Z destination. If saturation on overflow is enabled (according to the SAT field of the auc register), the accumulator value is limited. See Section 2.5.1.
- **Z:aTl**. The data from the specified Z source is loaded into the low half (bits 15—0) of the specified accumulator and the old data from the high half of the accumulator is loaded into the Z destination. The data in the high half of the accumulator is not altered. If saturation on overflow is enabled (according to the SAT field of the auc register), the accumulator value is limited. See Section 2.5.1.

3.4.3 No Operation

- **nop**. Single cycle no operation. **N * nop** (i.e., **4 * nop**) may be used to perform N no operation instructions.
- **;**. The semicolon is an optional no operation mnemonic. **N * ;** may also be used to perform N no operation instructions.

3.5 SPECIAL FUNCTION GROUP

Instructions from the special function group are **always** executed in one instruction cycle. They require one word of program memory. Using the special function instructions, the DSP16/DSP16A device can be used to implement a number of algorithms, which include the following nonlinear functions: absolute value, signum, minimum and maximum value finder, A-law and μ-law conversions, division, half-wave and full-wave rectification, and rounding. Special function instructions are executed either conditionally or unconditionally. Both the condition and its complement are available for use in special function instructions. A special function instruction uses one ROM location. Instructions from this group can be used in the cache.

The special function instructions can be conditioned on the basis of the results of previous multiply/ALU and special function instructions, the value of one of the counters (c0, c1), or the value of a randomly set bit in the DSP16 device. The result of the most recent accumulator operation prior to the special function instruction establishes the state of the flags for the conditions associated with logical or mathematical functions.

The special functions given in Table 3-5 can be conditionally executed as **if** *CON instruction* and with an event counter as **ifc** *CON instruction*, meaning that:

```
if CON is true then
    c1 = c1 + 1
    instruction
    c2 = c1
else
    c1 = c1 + 1
```

Note: When using the event counter (**ifc** instruction), if *CON* is c0lt or c0gt, then c0 is not incremented; if *CON* is c1lt or c1gt, then c1 is incremented once.

Table 3-5. Special Function Instructions	
Instruction	**Description**
aD=aS>>1 aD=aS>>4 aD=aS>>8 aD=aS>>16	Arithmetic right shift (sign preserved) of 36-bit accumulators.
aD=aS aD=−aS	—
aD=rnd(aS)	Round upper 20 bits of accumulator.
aDh=aSh+1	Increment high half of accumulator (lower half cleared).
aD=aS+1	Increment accumulator.
aD=y aD=p	—
aD=aS<<1 aD=aS<<4 aD=aS<<8 aD=aS<<16	Logical left shift (sign-extended from bit 31) of the least significant 32 bits of the 36-bit accumulators.

Table 3-6. Replacement Table for Special Function Instructions		
Replace	**Value**	**Meaning**
aD,aS	a0,a1	One of two DAU accumulators.
CON	mi, pl, eq, ne, gt, le, lvs, mvs, mvc, c0ge, c0lt, c1ge, c1lt, heads, tails, true, false	See Table 3-2 for definitions of processor flags.

3.5.1 Special Function Statements

The statements must be written in the exact format shown. If the statements are written in any other way, for example, aD=1+aS instead of aD=aS+1, the assembler produces an error message.

- **aD=aS>>1**. The contents of the source accumulator, aS, are divided by 2 and the result is placed in the destination accumulator, aD. The sign bit is preserved.
- **aD=aS>>4**. The contents of the source accumulator, aS, are divided by 2^4 and the result is placed in the destination accumulator, aD. The sign bit is preserved.
- **aD=aS>>8**. The contents of the source accumulator, aS, are divided by 2^8 and the result is placed in the destination accumulator, aD. The sign bit is preserved.
- **aD=aS>>16**. The contents of the source accumulator, aS, are divided by 2^{16} and the result is placed in the destination accumulator, aD. The sign bit is preserved.
- **aD=aS<<1**. The contents of the source accumulator, aS, are logically shifted one bit left and the result is placed in the destination accumulator, aD. The sign bit is extended from bit 31.

- **aD=aS<<4**. The contents of the source accumulator, aS, are logically shifted four bits left and the result is placed in the destination accumulator, aD. The sign bit is extended from bit 31.
- **aD=aS<<8**. The contents of the source accumulator, aS, are logically shifted eight bits left and the result is placed in the destination accumulator, aD. The sign bit is extended from bit 31.
- **aD=aS<<16**. The contents of the source accumulator, aS, are logically shifted sixteen bits left and the result is placed in the destination accumulator, aD. The sign bit is extended from bit 31.
- **aD=aS**. The contents of the source accumulator, aS, are placed in the destination accumulator, aD.
- **aD=−aS**. The 2's complement of the contents of the source accumulator, aS, is placed in the destination accumulator, aD.
- **aD=rnd(aS)**. The contents of the source accumulator, aS, are rounded to 16 bits, and the sign-extended result is placed in aD[35 — 16] with zeros in aD[15 — 0].
- **aDh=aSh+1**. The value 0x000010000 is added to the contents of the source accumulator, aS, and the result is placed in the destination accumulator, aD. This statement increments by one the data in the high half of the source accumulator. The low half of aD is cleared.
- **aD=aS+1**. The value 0x000000001 is added to the contents of the source accumulator, aS, and the result is placed in the destination accumulator, aD. This statement increments by one the data in the source accumulator.
- **aD=y**. The contents of the y register are written to the destination accumulator, aD.
- **aD=p**. The contents of the p register are written to the destination accumulator, aD. The bit alignment of the p register is a function of the ALIGN field of the auc register.

3.6 CONTROL GROUP

The control instructions allow the user to implement goto, call, and return commands. There is no latency when branching, i.e., the instruction executed following the control instruction has the address specified in the pc register after execution of the control instruction. Control instructions are executed either conditionally or unconditionally. Both the condition and its complement are available for use in control instructions. A control instruction uses one ROM location; conditional control instructions require two ROM locations. The execution time for an unconditional control instruction is two instruction cycles, and the execution time for conditional control instructions is three instruction cycles. The **icall** instruction executes in three cycles. Control instructions may not be executed in the cache.

The control instructions can be conditioned on the basis of the results of previous multiply/ALU and special function instructions, the value of one of the counters (c0, c1), or the value of a randomly set bit in the DSP16/DSP16A device. The result of the most recent accumulator operation prior to the control instruction establishes the state of the flags for the conditions associated with logical or mathematical functions.

www.DataSheet4U.com

An example of a control instruction conditionally executed is **if** *CON* **goto** *JA*.

Control Instructions*	
goto *JA*	icall†
goto pt	return (goto pr)
call *JA*	ireturn† (goto pi)
call pt	

* Control instructions cannot be used in the cache.
† **icall** and **ireturn** can not be conditionally executed.

Table 3-7. Replacement Table for Control Function Instructions

Replace	Value	Meaning
CON	mi, pl, eq, ne, gt, le, lvs, mvs, mvc, c0ge, c0lt, c1ge, c1lt, heads, tails, true, false	See Table 3-2 for definitions of processor flags.
JA	12-bit value	Least significant 12 bits of an absolute address within the same 4 Kword memory section.

3.6.1 Control Statements

- **goto JA.** The goto JA instruction moves the immediate value JA into the lower 12 bits of the program counter (pc) register, when goto JA is executed. The upper 4 bits of pc remain unchanged. The instruction with address JA is the next instruction executed. The goto JA instruction does not affect the program return (pr) register, and can be used in a subroutine without losing the return address of the subroutine.
- **call JA.** The call JA instruction moves the contents of the program counter (pc) register into the program return (pr) register and the immediate data JA into the lower 12 bits of the pc register. The upper 4 bits of pc remain unchanged. The pr register holds the return address of the subroutine (the address of the instruction following call JA); i.e., if call JA is located at address i, then the pr register is loaded with address i+1. The instruction with address N is the next instruction executed.
- **goto pt.** The goto pt instruction moves the contents of pt into the program counter (pc) register, when goto pt is executed. The instruction with address equal to the contents of pt is the next instruction executed. Since pt is a 16-bit register, goto pt allows branches to any location in the 64 Kword program space. The goto pt instruction does not affect the program return register.

- **call pt.** The call pt instruction moves the contents of the program counter (pc) register into the program return (pr) register and the data in pt into the pc register. The pr register holds the return address of the subroutine (the address of the instruction following call pt); i.e., if the call pt is located at address i, then the pr register is loaded with the value i+1. The instruction with address equal to the contents of pt is the next instruction executed.
- **icall.** The icall instruction moves the contents of the program counter (pc) register into the program interrupt (pi) register and the address 2 into the pc register. The pi register holds the return address of the interrupt routine (the address following the icall instruction); i.e., if the icall instruction is located at address i, then the pi register is loaded with the value i+1. The icall instruction is used by the DSP16/DSP16A Development Systems for breakpointing and is, therefore, reserved for that purpose when development system breakpoints are used.
- **return /goto pr.** The return instruction moves the contents of the program return (pr) register into the program counter (pc) register. The pr register holds the return address of the subroutine. Execution of the instruction with address equal to the contents of pr follows the execution of the return instruction. The goto pr instruction works identically to the return instruction.
- **ireturn /goto pi.** The ireturn instruction moves the contents of the program interrupt (pi) register into the program counter (pc) register. The pi register holds the interrupt return address. When an interrupt occurs, the value of the pc register is written into the pi register. Execution of the instruction with address equal to the contents of pi follows the execution of the ireturn instruction. The goto pi instruction works identically to the ireturn instruction.

3.7 DATA MOVE INSTRUCTIONS

Data move instructions transfer from a RAM location to a register, from a register to a RAM location, from an accumulator to a register, from a register to an accumulator; and load immediate data to a register. Data move instructions involving immediate data loaded into YAAU registers use one ROM location and execute in one instruction cycle if the data can be encoded in the instruction itself (R = M, M ≤ 9 bits) or two ROM locations if the data is not contained in the instruction (R = N). All other data move instructions use one ROM location. Data move instructions are executed in two instruction cycles except for those instructions in which the immediate data is encoded in the instruction which are executed in one instruction cycle as noted above (R = M). All data move instructions, with the exception of two-word immediate moves, may be executed inside the cache.

Data Move Instructions	
R = N	aT = R
R = M	Y = R
R = Y	Z : R
R = aS	

www.DataSheet4U.com

Table 3-8. Replacement Table for Data Move Instructions

Replace	Value	Meaning
R	x	DAU register – signed, 16 bits.
	y	DAU register – signed, 16 bits.[1]
	yl	DAU register – unsigned, 16 bits.
	auc	DAU control register – unsigned, 7 bits.
	c0	DAU counter 0 – signed, 8 bits.
	c1	DAU counter 1 – signed, 8 bits.
	c2	DAU counter 2 – signed, 8 bits.
	r0	YAAU ptr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	r1	YAAU ptr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	r2	YAAU ptr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	r3	YAAU ptr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	rb	YAAU mod. addr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	re	YAAU mod. addr. reg. – unsigned, 9 bits (16 bits in DSP16A).
	j	YAAU inc. reg. – signed, 9 bits (16 bits in DSP16A).
	k	YAAU inc. reg. – signed, 9 bits (16 bits in DSP16A).
	pt	XAAU pointer register – unsigned, 16 bits.
	pr	XAAU program return register – unsigned, 16 bits.
	pi	XAAU program interrupt register – unsigned, 16 bits.[2]
	i	XAAU increment register – signed, 12 bits.
	psw	Processor status word.
	sioc	Serial I/O control register.[3]
	sdx	Serial I/O data register.
	tdms	Serial I/O tdms control register.[3]
	srta	Serial receive/transmit address.[3]
	pioc	Parallel I/O control register.
	pdx0	Parallel I/O data register with PSEL = 0 (pin 72).
	pdx1	Parallel I/O data register with PSEL = 1 (pin 72).
aD, aS	a0, a1	High half of accumulator.[1]
Y	*rM,*rM++, *rM−−,*rM++j	Same as in multiply/ALU instructions.
Z	*rMzp,*rMpz, *rMm2,*rMjk	Same as in multiply/ALU instructions.
N	16-bit value	Immediate data.
M	9-bit value	Immediate data for YAAU registers.

Notes:
When reading signed registers less than 16 bits wide, their contents are sign-extended to 16 bits. When reading unsigned registers less than 16 bits wide, their contents are zero-extended to 16 bits. When short immediate addressing is used to write to YAAU registers in the DSP16A, unsigned registers are zero-extended from 9 to 16 bits. Signed registers (j,k) are sign-extended from 9 to 16 bits.

[1]Data moves to y, a0, or a1 load the high half (bits 31—16) of the register. If clearing of the destination is enabled (according to the CLR field of the auc register), the low half of the destination register is cleared (0) when the high half is loaded.

[2]The pi register acts as a "shadow" of the pc register. Each time the pc changes, its value is also loaded into pi. "Shadowing" is disabled when executing an interrupt service routine, therefore, pi contains the contents of pc prior to the interrupt. Writes to pi do not alter its contents, except during interrupt service routines.

[3]sioc, tdms, and srta registers are not readable.

3.7.1 Data Move Instruction Statements

The data move instruction statements must be written in the exact format shown. If the statements are written in any other way, for example, R: Z instead of Z:R, the assembler generates incorrect code and produces an error message. Data move instructions execute in two instruction cycles and require 1 word of program memory (immediate loads, R = N, require two words of program memory). Short immediate data move instructions require one word of program memory and execute in one cycle.

- **R=N** loads the immediate data value, N, into the specified destination register, R. This form of the data move instructions may not be executed in the cache.
- **R=M** loads a 9-bit immediate data value, M, into one of the YAAU registers (rb, re, r0, r1, r2, or r3). This special case immediate instruction is often referred to as a "short immediate" or "register set" instruction. Short immediate instructions require one word of program memory, execute in one cycle, and may be executed inside the cache.
- **R=Y** loads the data contained in the specified Y source into the specified destination register, R.
- **R=aS** loads the data contained in bits 31—16 of the specified transfer accumulator, aS, into the specified destination register, R. If saturation on overflow is enabled (according to the SAT field of the auc register), then the accumulator is limited. (See Section 2.5.1.)
- **Y=R** loads the data contained in the specified source register, R, into the specified Y destination.
- **aT=R** loads the data contained in the specified source register, R, into bits 31—16 of the specified accumulator. If clearing of aTl is enabled (according to the CLR field of the auc register), then aTl is cleared (0) when the high half is loaded. The guard bits are loaded with the value of bit 31.
- **Z: R** writes data from the specified Z source to the specified R destination register and writes the old data in the source register, R, to the Z destination (see Section 3.2.4 for an explanation of this data transfer mode).

3.8 CACHE INSTRUCTIONS

The cache instructions allow the implementation of low overhead loops to conserve program memory. When used, the cache instruction treats the specified NI instructions as a loop to be executed K times. Both cache instructions use one ROM location. The **do** instruction executes in one instruction cycle, while the **redo** instruction executes in two instruction cycles.

Cache Instructions	

do *K* {
instruction1
instruction2
.
.
instructionNI
}
	redo *K*

Table 3-9. Replacement Table for Cache Instructions

Replace	Value	Meaning
K	$2 \le K \le 127$	Number of times the instructions are to be executed.
NI	$1 \le NI \le 15$	1 to 15 instructions may be included.

3.8.1 Cache Statements

When the cache is used to repeat a block of NI instructions, the cycle timings of the instructions are as follows:

1. The "first pass" does not affect cycle timings except for the last instruction in the block of NI instructions. This instruction executes in two cycles.
2. During pass 2 through pass K+1, each instruction is executed "in the cache" (see Table 3-3).
3. During the last (Kth) pass, the block of instructions executes "inside the cache," except for the last instruction, which executes outside the cache.

The instructions remain in the cache memory and may be re-executed using the **redo** command without the need to reload the cache.

- **redo k**. When the redo k instruction is used, the DSP executes the NI instructions currently in the cache's memory k times. On the last iteration, the last instruction is executed outside the cache.

Note: Control group instructions and two-word data move instructions may not be executed from the cache.

3.9 INSTRUCTION SET SUMMARY

This section explains, in detail, the instruction set for the DSP16/DSP16A. Refer to Appendix A for instruction set formats and field encodings.

goto JA (branch direct)

(PC) ← (PC bits 15—12)(JA)

Program control jumps to location JA (within the same 4 Kword page). The lower 12 bits of the PC are written with the 12-bit immediate value of JA. The upper 4 bits of the PC remain unchanged (the **goto pt** instruction is used for branches outside the current 4 Kword page).

Bit	15			12	11	0
Field	0	0	0	0	JA	

Words: 1
Cycles: 2
Group: Control
Addressing: Immediate
Flags affected: None
Interruptible: No
Cacheable: No
Format: 4

www.DataSheet4U.com

goto B (branch direct)

(pc) ← (B)

Program control jumps to the location pointed to by the register encoded in the B field. The pc is written with the 16-bit value of the register. The following branch destinations are specified in the B field:

B Field	Action
000	return (same as goto pr)
001	ireturn (same as goto pi)
010	goto pt
011	call pt*
1xx	Reserved

* For this instruction, note that the current pc is also saved in the pr register before the jump.

Bit	15				11	10		8	7							0
Field	1	1	0	0	0	B			0	0	0	0	0	0	0	0

Words: 1
Cycles: 2
Group: Control
Addressing: Register
Flags affected: None
Interruptible: No
Cacheable: No
Format: 5

if CON goto/call/return (conditional branch qualifier)

test CONdition;
if true, execute the following control statement

The condition CON is tested (encoded in the CON field). If the condition is true, the next instruction (which must be a control instruction) is executed. If false, the control instruction is not executed. The CON field is encoded as:

CON	Flag	CON	Flag
00000	mi (negative result)	01001	tails (random bit clear)†
00001	pl (positive result)	01010	c0ge (counter0 ≥ 0)*
00010	eq (result = 0)	01011	c0lt (counter0 < 0)*
00011	ne (result ≠ 0)	01100	c1ge (counter1 ≥ 0)*
00100	lvs (logical overflow set)	01101	c1lt (counter1 < 0)*
00101	lvc (logical overflow clear)	01110	true (always)
00110	mvs (math. overflow set)	01111	false (never)
00111	mvc (math. overflow clear)	10000	gt (result > 0)
01000	heads (random bit set)†	10001	le (result ≤ 0)

* Using the c0ge or c0lt conditions also causes the value of the c0 counter to be postincremented.
Using the c1ge or c1lt conditions also causes the value of the c1 counter to be postincremented.
† The random bit is updated after each test of heads or tails.

The ensuing control opcode can be any of the following:

goto JA goto pt call JA call pt return (goto pr)

Note that **ireturn** and **icall** are the only control instructions that cannot be conditionally executed.

Bit		15										5	4		0
Field	word 1	1	1	0	1	0	0	0	0	0	0	0	CON		
	word 2	CONTROL OPCODE													

Words: 1
Cycles: 3 (including the branch/call/return)
Group: Control
Addressing: None
Flags affected: None
Interruptible: No
Cacheable: No
Format: 6

call JA (call subroutine direct)

(pr) ← (pc + 1)
(pc) ← (pc bits 15—12)(JA)

The subroutine at address JA (within the same 4 Kword page) is called. First the return address (the address of the first instruction following the call) is placed into the pr register. Then the lower 12 bits of the pc are written with the 12-bit immediate value of JA. The upper 4 bits of pc remain unchanged (the **call pt** instruction is used for calling subroutines out of the current 4 Kword page).

Bit	15			12	11	0
Field	1	0	0	0	JA	

Words: 1
Cycles: 2
Group: Control
Addressing: Immediate
Flags affected: None
Interruptible: No
Cacheable: No
Format: 4

icall (software interrupt)

(pi) ← (pc + 1)
(pc) ← 2
IACK

The interrupt handler is called, just as it would be by an external interrupt. The interrupt return register is set to next pc + 1, and the pc is set to 2, to start execution at the interrupt handler. Note that external interrupts vector to location 1, and icall vectors to location 2. The interrupt acknowledge pin (IACK) is set just as it would be by an external interrupt.

Bit	15															0
Field	1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	0

Words: 1
Cycles: 3
Group: Control
Addressing: None
Flags affected: None
Interruptible: No
Cacheable: No
Format: 6

www.DataSheet4U.com

```
do K {
    instr1
      .
      .          (loop in cache; cache loaded with new contents)
      .
    instrNI
}
```

execute the next NI instructions K times

The next NI instructions are loaded into the cache concurrent with their execution. They are then executed within the cache K–1 more times, at (potentially) higher speed.

The iteration count K can be between 2 and 127, inclusive, and the number of instructions NI must be between 1 and 15, inclusive.

Notes on cache performance:

The do instruction executes in one cycle. When the cache is used to repeat a block of NI instructions, the cycle timings of the instructions are as follows:

1. The "first pass" does not affect cycle timings except for the last instruction in the block of NI instructions. This instruction executes in two cycles.
2. During pass 2 through pass K+1, each instruction is executed "in the cache" (see Table 3-3).
3. During the last (Kth) pass, the block of instructions executes "inside the cache" except for the last instruction, which executes outside the cache.

The instructions remain in the cache memory and may be re-executed using the **redo** command without the need to reload the cache.

Bit	15				11	10	7	6	0
Field	0	1	1	1	0	NI		K	

Words: 1
Cycles: 1
Group: Cache
Addressing: Immediate
Flags affected: None
Interruptible: No
Cacheable: No
Format: 10

redo K

(loop in cache; cache contents unaffected)

execute the current contents of the cache K times

The current contents of the cache (loaded with a previous **do** instruction) are executed within the cache K additional times. The iteration count K can be between 2 and 127, inclusive.

Notes on cache performance:

The **redo** instruction executes in two cycles. All instructions require the in-cache time to execute, except the last instruction of the last iteration, which requires the out-of-cache time to execute. Thereafter, instructions (fetched from ROM) require their normal out-of-cache time to execute.

Bit	15				11	10			7	6	0
Field	0	1	1	1	0	0	0	0	0	K	

Words: 1
Cycles: 2
Group: Cache
Addressing: Immediate
Flags affected: None
Interruptible: No
Cacheable: No
Format: 10

www.DataSheet4U.com

R = M (short immediate load)

(R) ← (M)

The contents of register R are replaced with the 9-bit immediate value of M. The value of R can be any of the following:

Register	R	Register	R
j	000	r0	100
k	001	r1	101
rb	010	r2	110
re	011	r3	111

For the DSP16, these registers are all 9 bits wide. For the DSP16A, these registers are 16 bits wide and the j and k registers are sign-extended (2's complement). The others are zero-extended.

Bit	15 12	11 9	8 0
Field	0 0 0 1	R	M

Words: 1
Cycles: 1
Group: Data Move
Addressing: Immediate
Flags affected: None
Interruptible: Yes
Cacheable: Yes
Format: 9

Notes:

1) In Appendix A, this instruction is encoded using a 2-bit I field that corresponds to the two LSBs of the R field shown above. The most significant bit of R is the least significant bit of the T field used in the instruction set encodings in Appendix A.

2) When a DSP16A program is encoded, if the immediate value M is greater than 9 bits or if a label is used for M, the assembler defaults to a two-word, two-cycle data move encoding. The short immediate encoding can be forced by using the optional mnemonic *set* (if the value of M is greater than 9 bits, it is truncated to 9 bits). For example:

   set r3 = var1

   forces a short immediate encoding.

R = N (16-bit immediate load)

(R) ← (N)

The contents of register R are replaced with the 16-bit immediate value of N. The value of R can be any of the following:

Register	R Field	Register	R Field
r0 (u)	000000	yl	010010
r1 (u)	000001	auc (u)	010011
r2 (u)	000010	psw	010100
r3 (u)	000011	c0 (s)	010101
j (s)	000100	c1 (s)	010110
k (s)	000101	c2 (s)	010111
rb (u)	000110	sioc	011000
re (u)	000111	srta	011001
pt	001000	sdx	011010
pr	001001	tdms	011011
pi	001010	pioc	011100
i (s)	001011	pdx0	011101
x	010000	pdx1	011110
y	010001		

Register sources j, k, i, c0, c1, and c2 are less than 16 bits and are sign-extended (s). Register sources r0, r1, r2, r3, rb, re, and auc are less than 16 bits and are zero-extended (u). For the DSP16A, registers r0, r1, r2, r3, j, k, rb, and re are 16 bits wide and need no sign- or zero-extension.

Note: writing the psw also writes the a0 and a1 guard bits.

Bit		15 10	9 4	3 0
Field	word 1	0 1 0 1 0 0	R	0 0 0 0
	word 2	Immediate Value (N)		

Words: 2
Cycles: 2
Group: Data Move
Addressing: Immediate
Flags affected: None
Interruptible: Yes
Cacheable: No
Format: 8

www.DataSheet4U.com

R = aS (load register from accumulator)

(R) ← (aS)

The contents of register R are replaced with the current contents of bits 31—16 of accumulator aS. Registers which are less than 16 bits load from the low-order bits of aS[31—16].

The value of S can be 0 to select accumulator a0 or 1 to select accumulator a1. See Appendix A for the possible values of R.
Note: Writing the psw also writes the the a0 and a1 guard bits.

Bit	15					10	11	4	3			0
Field	0	1	0	S	1	0	R		0	0	0	0

Words: 1
Cycles: 2
Group: Data Move
Addressing: Register
Flags affected: None
Interruptible: Yes
Cacheable: Yes
Format: 7

aT = R (load accumulator from register)

(aT) ← (R)

The contents of bits 31—16 of accumulator aT are replaced with the current contents of register R, zero- or sign-extended to 16 bits (if necessary). If clearing aTl is enabled (with the CLR field of the auc register), bits 15—0 of accumulator aT will be cleared. Bits 35—32 (the guard bits) will be loaded with copies of bit 31.

The value of $\overline{aT}$ can be 0 to select a1, or 1 to select a0. (aT is encoded as $\overline{aT}$ in the instruction encodings in Appendix A.) The value of R can be any of the following:

Register	R Field	Register	R Field
r0 (u)	000000	yl	010010
r1 (u)	000001	auc (u)	010011
r2 (u)	000010	psw	010100
r3 (u)	000011	c0 (s)	010101
j (s)	000100	c1 (s)	010110
k (s)	000101	c2 (s)	010111
rb (u)	000110	sioc	011000
re (u)	000111	srta	011001
pt	001000	sdx	011010
pr	001001	tdms	011011
pi	001010	pioc	011100
i (s)	001011	pdx0	011101
x	010000	pdx1	011110
y	010001		

Register sources j, k, i, c0, c1, and c2 are less than 16 bits and are sign-extended (s). Register sources r0, r1, r2, r3, rb, re, and auc are less than 16 bits and are zero-extended (u). For the DSP16A, registers r0, r1, r2, r3, j, k, rb, and re are 16 bits wide and need no sign- or zero-extension.

Bit	15				11	10	9	4	3			0
Field	0	1	0	0	0	$\overline{aT}$	R		0	0	0	0

Words: 1
Cycles: 2
Group: Data Move
Addressing: Register
Flags affected: None
Interruptible: Yes
Cacheable: Yes
Format: 7a

Note: If y is used as the register R, the assembler forces a special function encoding. The resulting instruction moves all 32 bits (sign extended to 36 bits) of y into aT. All DAU flags are affected, and the execution requires only one cycle. If a two-cycle data move is desired, the optional mnemonic *move* may be used. Only the upper 16 bits of y are transferred and no flags are affected. Example:

```
move a0 = y
```


R = Y (load register from internal RAM)

perform (R) ← (*rN); then
modify rN

The contents of register R are replaced with the current contents of the internal RAM location pointed to by rN, where rN is specified by the two most significant bits of the Y field.

00 - r0 01 - r1 10 - r2 11 - r3

The value of rN is then postmodified, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN--
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

See Appendix A for the possible values of destination register R. Registers which are less than 16 bits load from the low-order bits of the memory location. Note: writing the psw also writes the a0 and a1 guard bits.

Bit	15					10	9	4	3	0
Field	0	1	1	1	1	0	R		Y	

Words: 1
Cycles: 2
Group: Data Move
Addressing: Register, Register Indirect
Flags affected: None
Interruptible: Yes
Cacheable: Yes
Format: 7

Note: If y, yl, or x is the destination register, R, the assembler assembles this instruction as a single-cycle multiply/ALU instruction. If a two-cycle move encoding is necessary, the optional mnemonic *move* may be used. For example:

```
move y = *r1
```

forces a move encoding.

www.DataSheet4U.com

Y = R (store register to RAM memory)

(*rN) ← (R); then
modify rN

The contents of the RAM memory location pointed to by rN are replaced with the current contents of register R, zero- or sign-extended to 16 bits (if necessary). rN is specified the two most significant bits of the Y field:

00 - r0 01 - r1 10 - r2 11 - r3

The value of rN is then postmodified, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN−−
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

See Appendix A for possible values of R. Register sources j, k, i, c0, c1, and c2 are less than 16 bits and are sign-extended. Register sources r0, r1, r2, r3, rb, re, and auc are less than 16 bits and are zero-extended. For the DSP16A, registers r0, r1, r2, r3, j, k, rb, and re are 16 bits and need no sign- or zero-extending.

Bit	15				11	10	9	4	3	0
Field	0	1	1	0	0	x	R		Y	

Words: 1
Cycles: 2
Group: Data Move
Addressing: Register, Register Indirect
Flags affected: None
Interruptible: Yes
Cacheable: Yes
Format: 7

Z : R (exchange register with RAM memory)

temp ← (R); then
(R) ← (*rN); then
modify rN (first action); then
(*rN) ← temp; then
modify rN (second action)

The contents of the RAM memory location(s) pointed to by rN are exchanged with the current contents of register R, which is sign- or zero-extended to 16 bits (if necessary). The pointer rN is modified after each of the two memory accesses according to the M field. rN is specified by the two most significant bits of the Z field:

00 - r0 01 - r1 10 - r2 11 - r3

The available options for the postmodification are specified by the two least significant bits of the Z field as follows:

Symbol	2 LSBs of Z	First Action	Second Action
*rNzp	00	no action	postincrement
*rNpz	01	postincrement	no action
*rNm2	10	postdecrement	postincrement by 2
*rNjk	11	postincrement by (j)	postincrement by (k)

Code 11, in this case, means add the current value of the j register to rN after reading *rN, then add the current value of the k register to rN after writing *rN.

www.DataSheet4U.com

See Appendix A for possible values of R. Register sources j, k, i, c0, c1, and c2 are less than 16 bits and are sign-extended. Register sources r0, r1, r2, r3, rb, re, and auc are less than 16 bits and are zero-extended. For the DSP16A, registers r0, r1, r2, r3, j, k, rb, and re are 16 bits and need no sign- or zero-extension. Note: writing the psw also writes the a0 and a1 guard bits.

Bit	15				11	10	9	4	3	0
Field	0	1	1	0	1	x	R		Z	

Words: 1
Cycles: 2
Group: Data Move
Addressing: Register, Register Indirect
Flags affected: None
Interruptible: Yes
Cacheable: Yes
Format: 7

Note: R and rM must not be the same register (i.e., r2pz:r2). The two logical PIO registers,pdx0 and pdx1, cannot be used in compound data moves.

if CON F2

(If CONdition is true, then perform special function instruction)

test CONdition;
if true, then perform F2

The specified condition is tested. If it is true, the special function operation F2 is performed. See Appendix A for the conditions that can be tested (encoded in the CON field).

The F2 functions (special function group) that can be conditionally performed (encoded in the F2 field) are as follows:

F2	Operation
0000	aD = aS >> 1
0001	aD = aS << 1
0010	aD = aS >> 4
0011	aD = aS << 4
0100	aD = aS >> 8
0101	aD = aS << 8
0110	aD = aS >> 16
0111	aD = aS << 16
1000	aD = p
1001	aDh = aSh + 1
1010	Reserved
1011	aD = rnd(aS)
1100	aD = y
1101	aD = aS + 1
1110	aD = aS
1111	aD = –aS

Bit	15				11	10	9	8	5	4	0
Field	1	0	0	1	1	D	S	F2		CON	

Words: 1
Cycles: 1
Group: Special Function
Addressing: Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 3

www.DataSheet4U.com

ifc CON F2

(if CONdition is true, then perform special function instruction)
(modify counter1,2 accordingly)

counter c1 = c1 + 1;
test CONdition; if true then {perform F2; c2 = c1}

First, counter c1 is incremented. Next, the specified condition is tested. If the condition is true, the special function operation F2 is performed and counter c2 is set to the value of c1. The conditions that can be tested are encoded in the CON field (see Appendix A).

The possible F2 special functions that can be conditionally performed are:

F2	Operation
0000	aD = aS >> 1
0001	aD = aS << 1
0010	aD = aS >> 4
0011	aD = aS << 4
0100	aD = aS >> 8
0101	aD = aS << 8
0110	aD = aS >> 16
0111	aD = aS << 16
1000	aD = p
1001	aDh = aSh + 1
1010	Reserved
1011	aD = rnd(aS)
1100	aD = y
1101	aD = aS + 1
1110	aD = aS
1111	aD = –aS

The D and S fields are used to specify aD and aS.

Bit	15				11	10	9	8	5	4	0
Field	1	0	0	1	0	D	S	F2		CON	

Words: 1
Cycles: 1
Group: Special Function
Addressing: Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 3

F1 Y

(multiply/ALU operation with postmodification of pointer register)

perform operation F1; then
access *rN; then
postmodify rN (the contents of *rN are not written to a destination)

This instruction performs the following three operations (effectively in sequence):

1. The operation F1 is performed. The possible F1 operations are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Access the internal RAM location pointed to by rN, where rN is specified by the two most significant bits of the Y field as follows (the accessed location is not written to a destination):

   00 - r0 01 - r1 10 - r2 11 - r3

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN--
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8	5	4	3	0
Field	0	0	1	1	0	D	S	F1		0	Y	

Words: 1
Cycles: 1
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 1

F1 Y = a0[l] (multiply/ALU operation with parallel accumulator store)
F1 Y = a1[l]

write the value of aT[l] to *rN; then
modify rN; then
perform operation F1

This instruction performs the following three operations (effectively in sequence):

1. Write the (old) value of a0, a1, a0l, or a1l to the internal RAM location pointed to by rN, where rN is specified by the two most significant bits of the Y field.

   00 - r0 01 - r1 10 - r2 11 - r3

   The X field selects y or yl:

   X = 0 → yl X = 1 → y

2. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN--
11	postincrement by (j)	*rN++j

Code 11 in this case means add the current value of the j register to rN (after accessing *rN).

3. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

Bit		15				11	10	9	8	5	4	3	0
Field	a0	1	1	1	0	0	D	S	F1		X	Y	
	a1	0	0	1	0	0	D	S	F1		X	Y	

Words: 1
Cycles: 2
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 1

F1 x = Y (multiply/ALU operation with parallel load of x register)

perform operation F1; then
copy *rN to x; then
modify rN

This instruction performs the following three operations (effectively in sequence):

1. The multiply/ALU operation F1 is performed. The possible operations for F1 are as follows:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

www.DataSheet4U.com

2. Access the internal RAM location pointed to by rN, and write this value into the x register. rN is specified by the most significant bits of the Y field:

   00 - r0 01 - r1 10 - r2 11 - r3

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field.

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN– –
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8	5	4	3	0
Field	1	0	1	1	0	D	S	F1		0	Y	

Words: 1
Cycles: 1
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 1

F1 y[l] = Y (multiply/ALU operation with parallel load of y register)

perform operation F1; then
copy *rN to y (or yl); then
modify rN

This instruction performs the following three operations (effectively in sequence):

1. The multiply/ALU operation F1 is performed. The possible F1 operations are as follows:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

www.DataSheet4U.com

2. Access the internal RAM location pointed to by rN, and write this value into the y (or yl) register. rN is specified by the two most significant bits of the Y field:

   00 - r0 01 - r1 10 - r2 11 - r3

   The X field selects y or yl:

   X = 0 → yl X = 1 → y

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field:

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN– –
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8	5	4	3	0
Field	1	0	1	1	1	D	S	F1		X	Y	

Words: 1
Cycles: 1
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 1

F1 y = Y x = *pt++[i]

(multiply/ALU operation with parallel load of x and y registers)

perform operation F1; then
(y) ← (*rN); then
modify rN; then
(x) ← (*pt); then
(pt)= (pt)+ [1 or i]

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

www.DataSheet4U.com

2. Access the internal RAM location pointed to by rN, and write this value into the y register. rN is specified by the two most significant bits of the Y field:

   00 - r0 01 - r1 10 - r2 11 - r3

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field:

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN--
11	postincrement by (j)	*rN++j

   Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

4. Access the ROM location pointed to by pt, and write this value into the x register. Either internal or external ROM may be accessed, depending on the state of the EXM pin (and the address, in the case of the DSP16A).

5. Postmodify the value of the pt register by either 1 or i, selected by the X field:

   X = 0 → *pt++ X = 1 → *pt++i

Bit	15				11	10	9	8	5	4	3	0
Field	1	1	1	1	1	D	S	F1		X	Y	

Words: 1
Cycles: 2 (1 cycle if in cache)
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 1

F1 y = a0 x = *pt++[i] (multiply/ALU operation with parallel load of x and y registers)
F1 y = a1 x = *pt++[i]

perform operation F1; then
(y) ← (a0) or (a1); then
(x) ← (*pt); then
(pt)= (pt)+ [1 or i]

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

www.DataSheet4U.com

2. Copy the value in a0 or a1 to the y register. Note that the value copied from a0 or a1 is the value before executing the F1 operation, due to pipelining.
3. Access the ROM location pointed to by pt, and write this value into the x register. Either internal or external ROM may be accessed, depending on the state of the EXM pin (and the address, for the DSP16A).
4. Postmodify the value of the pt register by either 1 or i, selected by the X field:

   X = 0 → *pt++ X = 1 → *pt++i

Bit		15				11	10	9	8	5	4	3			0
Field	a0	1	1	0	0	1	D	S	F1		X	0	0	0	0
	a1	1	1	0	1	1	D	S	F1		X	0	0	0	0

Words: 1
Cycles: 2 (1 cycle if in cache)
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 1

F1 aT[l] = Y (multiply/ALU operation with parallel load of accumulator register)

perform operation F1; then
copy *rN to aT (or aTl); then
modify rN by M

This instruction performs the following three operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

The value of S can be 0 to select a0 or 1 to select a1. The value of $\overline{aT}$ can be 0 to select a1 or 1 to select a0. Since aD and aT must be different accumulators, aD will be the opposite of aT. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

www.DataSheet4U.com

2. Access the internal RAM location pointed to by rN, and write this value to the aT (or aTl) register. $\overline{aT}$ is defined as the opposite of D for this instruction. Therefore, if the F1 field selects writing to aD, aD will be the opposite of aT. rN is specified by the two most significant bits of the Y field:

   00 - r0 01 - r1 10 - r2 11 - r3

   The X field selects y or yl:

   X = 0 → yl X = 1 → y

3. Postmodify the value of rN, where the postmodification is specified the two least significant bits of the Y field:

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN−−
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8	5	4	3	0
Field	1	0	1	1	1	$\overline{aT}$	S	F1		X	Y	

Words: 1
Cycles: 1
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 1a

F1 Y = y[l]

(multiply/ALU operation with parallel store of y register)

perform operation F1;
(*rN) ← (y) or (yl); then
modify rN

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS − p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS − p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS − y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS − y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

www.DataSheet4U.com

2. Write the value of y or yl to the internal RAM location pointed to by rN, where N is specified by the two most significant bits of the Y field:

   00 - r0 01 - r1 10 - r2 11 - r3

   The X field selects y or yl:

   X = 0 → yl X = 1 → y

3. Postmodify the value of rN, where the postmodification is specified by the two least significant bits of the Y field:

2 LSBs of Y	Action	Symbol
00	no action	*rN
01	postincrement	*rN++
10	postdecrement	*rN−−
11	postincrement by (j)	*rN++j

Code 11, in this case, means add the current value of the j register to rN (after accessing *rN).

Bit	15				11	10	9	8	5	4	3	0
Field	1	0	1	0	0	D	S	F1		X	Y	

Words: 1
Cycles: 2
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 1

F1 Z : y[I] (multiply/ALU operation with compound data move)

perform operation F1; then
temp ← (y) or (yl); then
(y) or (yl) ← (*rN); then
modify rN (first action); then
(*rN) ← temp; then
modify rN (second action)

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible F1 operations are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS − p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS − p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS − y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS − y	

The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

www.DataSheet4U.com

2. Save either the y or yl register into an internal temporary location (temp). The X field select y or yl:

   X = 0 → yl X = 1 → y

3. Access the internal RAM location pointed to by rN, and write this value into the y (or yl) register. rN is specified by the 2 most significant bits of the Z field:

   00 - r0 01 - r1 10 - r2 11 - r3

4. Postmodify the value of rN by the first action described by the two least significant bits of the Z field (described below).

5. Write the value saved in the temporary register (temp) to the memory location now pointed to by rN.

6. Postmodify the value of rN by the second action described by the two least significant bits of the Z field. The available options for the postmodification are specified as follows:

Symbol	2 LSBs of Z	First Action	Second Action
*rNzp	00	no action	postincrement
*rNpz	01	postincrement	no action
*rNm2	10	postdecrement	postincrement by 2
*rNjk	11	postincrement by (j)	postincrement by (k)

Code 11, in this case, means add the current value of the j (or) k register to rN (after accessing *rN).

Bit	15				11	10	9	8	5	4	3	0
Field	1	0	1	0	1	D	S	F1		X	Z	

Words: 1
Cycles: 2
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 2

F1 Z : aT[I]

(multiply/ALU operation with parallel compound accumlator move)

perform operation F1; then
temp ← (aT) or (aTI); then
(aT) or (aTI) ← (*rN); then
modify rN (first action);
(*rN) ← temp;
modify rN (second action)

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

   The value of S can be 0 to select a0 or 1 to select a1. The value of $\overline{aT}$ can be 0 to select a1 or 1 to select a0. Since aD and aT must be different accumulators, aD will be the opposite of aT. Flags are modified based on the value computed by the ALU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Save either the aT or aTI register into an internal temporary location (temp). $\overline{aT}$ is defined as the opposite of D for this instruction. Therefore, if the F1 field selects writing to aD, aD will be the opposite of aT since the $\overline{aT}$ field must read/write aT, and vice versa. Note that if aS in the F1 operation is the same as aT, the value used in the F1 operation will be the old value, due to pipelining. The X field selects aT or aTI:

   X = 0 → aTI X = 1 → aT

www.DataSheet4U.com

3. Access the internal RAM location pointed to by rN, and write this value to the aT (or aTI) register. rN is specified by the two most significant bits of the Z field:

   00 - r0 01 - r1 10 - r2 11 - r3

4. Postmodify the value of rN by the first action described by the two least significant bits of the Z field (described below).

5. Write the value saved in the temporary register (temp) to the memory location now pointed to by rN.

6. Postmodify the value of rN by the second action described by the two least significant bits of the Z field. The available options for the postmodification are specified as follows:

Symbol	2 LSBs of Z	First Action	Second Action
*rNzp	00	no action	postincrement
*rNpz	01	postincrement	no action
*rNm2	10	postdecrement	postincrement by 2
*rNjk	11	postincrement by (j)	postincrement by (k)

Code 11, in this case, means add the current value of the j (or) k register to rN (after accessing *rN).

Bit	15				11	10	9	8	5	4	3	0
Field	0	0	1	0	1	$\overline{aT}$	S	F1		X	Z	

Words: 1
Cycles: 2
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 2a

F1 Z : y x = *pt++[i]

(multiply/ALU operation with compound data move and parallel load of x register)

perform operation F1; then
temp ← (y); then
(y) ← (*rN); then
modify rN (first action); then
(*rN) ← temp; then
modify rN (second action); then
(x) ← (*pt); then
(pt)= (pt)+ [1 or i]

This instruction performs the following operations (effectively in sequence):

1. The operation F1 is performed. The possible operations for F1 are:

F1	Operation	
0000	aD = p	p = x*y
0001	aD = aS + p	p = x*y
0010		p = x*y
0011	aD = aS – p	p = x*y
0100	aD = p	
0101	aD = aS + p	
0110	NOP	
0111	aD = aS – p	
1000	aD = aS | y	
1001	aD = aS ^ y	
1010	aS & y	
1011	aS – y	
1100	aD = y	
1101	aD = aS + y	
1110	aD = aS & y	
1111	aD = aS – y	

   The value of S can be 0 to select a0 or 1 to select a1. The value of D can be 0 to select a0 or 1 to select a1. Flags are modified based on the value computed by the DAU. Note: for all diadic operations involving the y register, y is sign-extended to 36 bits before performing the operation (this includes logical operations).

2. Save the y register into an internal temporary location (temp).

3. Access the internal RAM location pointed to by rN, and write this value into the y register. rN is specified by the two most significant bits of the Z field:

   00 - r0 01 - r1 10 - r2 11 - r3

www.DataSheet4U.com

4. Postmodify the value of rN by the first action described by the two least significant bits of the Z field (described below).
5. Write the value saved in the temporary register (temp) to the memory location now pointed to by rN.
6. Postmodify the value of rN by the second action described by the two least significant bits of the Z field. The available options for the postmodification are specified as follows:

Symbol	2 LSBs of Z	First Action	Second Action
*rNzp	00	no action	postincrement
*rNpz	01	postincrement	no action
*rNm2	10	postdecrement	postincrement by 2
*rNjk	11	postincrement by (j)	postincrement by (k)

Code 11, in this case, means add the current value of the j (or) k register to rN (after accessing *rN).

7. Access the ROM location pointed to by pt, and write this value into the x register. Either internal or external ROM may be accessed, depending on the state of the EXM pin (and the address, for the DSP16A).
8. Postmodify the value of the pt register by either 1 or i, selected by the X field:

X = 0 → *pt++ X = 1 → *pt++i

Bit	15				11	10	9	8	5	4	3	0
Field	1	1	1	0	1	D	S	F1		X	Z	

Words: 1
Cycles: 2
Group: Multiply/ALU
Addressing: Register Indirect, Register
Flags affected: All
Interruptible: Yes
Cacheable: Yes
Format: 2