

A.7 INSTRUCTION TIMING

This section describes how one can calculate DSP56000/DSP56001 instruction timing manually using the tables provided in this section. Three complete examples are presented to illustrate the “layered” nature of the tables. Alternatively, the user can obtain the number of instruction program words and the number of oscillator clock cycles required for a given instruction by using the DSP56000/DSP56001 simulator. This method of determining instruction timing information is much faster and much simpler than using the aforementioned tables. This powerful software package is available for the IBM™ PC, Apple Macintosh™, and SUN-4™ workstation.

Table A-6 gives the number of instruction program words and the number of oscillator clock cycles for each instruction mnemonic. Table A-7 gives the number of additional (if any) instruction words and additional (if any) clock cycles for each type of parallel move operation. Table A-8 gives the number of additional (if any) clock cycles for each type of MOVEC operation. Table A-9 gives the number of additional (if any) clock cycles for each type of MOVEP operation. Table A-10 gives the number of additional (if any) clock cycles for each type of bit manipulation (BCHG, BCLR, BSET, and BTST) operation. Table A-11 gives the number of additional (if any) clock cycles for each type of jump (Jcc, JCLR, JMP, JScC, JSCLR, JSET, JSR, and JSSET) operation. Table A-12 gives the number of additional (if any) clock cycles for the RTI and RTS instructions. Table A-13 gives the number of additional (if any) instruction words and additional (if any) clock cycles for each effective addressing mode. Table A-14 gives the number of additional (if any) clock cycles for external data, external program, and external I/O memory accesses.

The number of words per instruction is dependent on the addressing mode and the type of parallel data bus move operation specified. The symbols used reference subsequent tables to complete the instruction word count.

The number of oscillator clock cycles per instruction is dependent on many factors, including the number of words per instruction, the addressing mode, whether the instruction fetch pipe is full or not, the number of external bus accesses, and the number of wait states inserted in each external access. The symbols used reference subsequent tables to complete the execution clock cycle count.

All tables are based on the following assumptions:

1. All instruction cycles are counted in **oscillator clock cycles**.
2. The instruction fetch pipeline is **full**.

IBM is a trademark of International Business Machines.
Macintosh is a trademark of Apple Computer Corporation
SUN-4 is a trademark of Sun Microsystems, Inc.

3. There is no contention for **instruction** fetches. Thus, external program instruction fetches are assumed not to have to contend with external data memory accesses.
4. There are no wait states for **instruction** fetches done sequentially (as for non-change-of-flow instructions), but they are taken into account for change-of-flow instructions which flush the pipeline such as JMP, Jcc, RTI, etc.

To better understand and use the aforementioned tables, three examples are presented prior to the actual tables. These examples attempt to illustrate the “layered” nature of the tables.

Example 1: Arithmetic Instruction with Two Parallel Moves

Problem: Calculate the number of 24-bit instruction program words and the number of oscillator clock cycles required for the instruction (located in internal program memory):

MACR –X0,X0,A X1,X:(R6)– Y0,Y:(R0)+

where Operating Mode Register (OMR) = \$02 (normal expanded memory map)
 Bus Control Register (BCR) = \$1135
 R6 Address Register = \$0052 (internal X memory)
 R0 Address Register = \$0523 (external Y memory)

Solution: To determine the number of instruction program words and the number of oscillator clock cycles required for the given instruction, the user should perform the following operations:

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in Table A-6.

According to Table A-6, the MACR instruction will require (1+mv) instruction program words and will execute in (2+mv) oscillator clock cycles. The term “mv” represents the additional (if any) instruction program words and the additional (if any) oscillator clock cycles that may be required over and above those needed for the basic MACR instruction due to the parallel move portion of the instruction.

2. Evaluate the “mv” term using Table A-7.

The parallel move portion of the MACR instruction consists of an XY memory move. According to Table A-7, the parallel move portion of the instruction will require mv=0 additional instruction program words and mv=(ea+axy) additional oscillator clock cycles. The term “ea” represents the number of additional (if any) oscillator clock cycles that are required for the effective addressing move specified in the parallel move portion of the instruction. The term “axy” represents the number of additional (if any) oscillator clock cycles that are required to access an **XY** memory operand.

3. Evaluate the “ea” term using Table A-13.

The parallel move portion of the MACR instruction consists of an XY memory move which uses both address register banks (R0–R3 and R4–R7) in generating the effective addresses of the XY memory operands. Thus, the two effective address operations occur in parallel, and the larger of the two “ea” terms should be used. The X memory move operation uses the “postdecrement by 1” effective addressing mode. According to Table A-13, this operation will require ea=0 additional oscillator clock cycles. The Y memory move operation uses the “postincrement by 1” effective addressing mode. According to Table A-13, this operation will also require ea=0 additional oscillator clock cycles. Thus, using the maximum value of “ea”, the effective addressing modes used in the parallel move portion of the MACR instruction will require ea=0 additional oscillator clock cycles.

4. Evaluate the “axy” term using Table A-14.

The parallel move portion of the MACR instruction consists of an XY memory move. According to Table A-14, the term “axy” depends upon where the referenced X and Y memory locations are located in the DSP56000/DSP56001 memory space. **External** memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the DSP56000/DSP56001 bus control register (BCR). Thus, assuming that the 16-bit bus control register contains the value \$1135, external X memory accesses require wx=1 wait state of additional oscillator clock cycle while external Y memory accesses require wy=1 wait state or additional oscillator clock cycle. For this example, the X memory reference is assumed to be an **internal** reference; the Y memory reference is assumed to be an **external** reference. Thus, according to Table A-14, the XY memory reference in the parallel move portion of the MACR instruction will require axy=wy=1 additional oscillator clock cycle.

5. Compute final results.

Thus, based upon the assumptions given for Table A-6 and those listed in the problem statement for Example 1, the instruction

MACR –X0,X0,A X1,X:(R6)– Y0,Y:(R0)+

will require

$$\begin{aligned} & (1+mv) \\ & = (1+0) \\ & = 1 \qquad \text{instruction program word} \end{aligned}$$

and will execute in

$$\begin{aligned} & = (2+mv) \\ & = (2+ea+axy) \\ & = (2+ea+wy) \\ & = (2+0+1) \qquad \text{oscillator clock cycles.} \\ & = 3 \end{aligned}$$

Note that if a similar calculation were to be made for a MOVEC, MOVEM, MOVEP, or MOTOROLA

one of the bit manipulation (BCHG, BCLR, BSET, or BTST) instructions, the use of Table A-7 would no longer be appropriate. For one of these cases, the user would refer to Table A-8, Table A-9, or Table A-10, respectively.

Example 2: Jump Instruction

Problem: Calculate the number of 24-bit instruction program words and the number of oscillator clock cycles required for the instruction

JLC (R2+N2)

where	Operating Mode Register (OMR)	= \$02 (normal expanded memory map),
	Bus Control Register (BCR)	= \$2246,
	R2 Address Register	= \$1000 (external P memory), and
	N2 Address Register	= \$0037.

Solution: To determine the number of instruction program words and the number of oscillator clock cycles required for the given instruction, the user should perform the following operations:

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in Table A-6.

According to Table A-6, the Jcc instruction will require (1+ea) instruction program words and will execute in (4+jx) oscillator clock cycles. The term “ea” represents the number of additional (if any) instruction program words that are required for the effective address of the Jcc instruction. The term “jx” represents the number of additional (if any) oscillator clock cycles required for a jump-type instruction.

2. Evaluate the “jx” term using Table A-11.

According to Table A-11, the Jcc instruction will require $jx=ea+(2 * ap)$ additional oscillator clock cycles. The term “ea” represents the number of additional (if any) oscillator clock cycles that are required for the effective addressing mode specified in the Jcc instruction. The term “ap” represents the number of additional (if any) oscillator clock cycles that are required to access a P memory operand. Note that the “+(2 * ap)” term represents the two program memory instruction fetches executed at the end of a one-word jump instruction to refill the instruction pipeline.

3. Evaluate the “ea” term using Table A-13.

The JLC (R2+N2) instruction uses the “indexed by offset Nn” effective addressing mode. According to Table A-13, this operation will require ea=0 additional instruction program words and ea=2 additional oscillator clock cycles.

4. Evaluate the “ap” term using Table A-14.

According to Table A-14, the term “ap” depends upon where the referenced P memory location is located in the DSP56000/DSP56001 memory space. **External** memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the DSP56000/DSP56001 bus control register (BCR). Thus, assuming that the 16-bit bus control register contains the value \$2246, external **P** memory accesses require $wp=4$ wait states or additional oscillator clock cycles. For this example, the P memory reference is assumed to be an **external** reference. Thus, according to Table A-14, the Jcc instruction will use the value $ap=wp=4$ oscillator clock cycles.

5. Compute final results.

Thus, based upon the assumptions given for Table A-6 and those listed in the problem statement for Example 2, the instruction

JLC (R2+N2)

will require

$$\begin{aligned} &=(1+ea) \\ &=(1+0) \\ &= 1 \qquad \text{instruction program word} \end{aligned}$$

and will execute in

$$\begin{aligned} &=(4+jx) \\ &=(4+ea+(2 * ap)) \\ &=(4+ea+(2 * wp)) \\ &=(4+2+(2 * 4)) \quad \text{oscillator clock cycles.} \\ &= 14 \end{aligned}$$

Example 3: RTI Instruction

Problem: Calculate the number of 24-bit instruction program words and the number of oscillator clock cycles required for the instruction

RTI

where Operating Mode Register (OMR) = \$02 (normal expanded memory map),
 Bus Control Register (BCR) = \$0012, and,
 Return Address (on the stack) = \$0100 (internal P memory).

Solution: To determine the number of instruction program words and the number of oscillator clock cycles required for the given instruction, the user should perform the following operations:

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in Table A-6.

According to Table A-6, the RTI instruction will require one instruction program word and

will execute in $(4+rx)$ oscillator clock cycles. The term “rx” represents the number of additional (if any) oscillator clock cycles required for an RTI or RTS instruction.

2. Evaluate the “rx” term using Table A-12.

According to Table A-12, the RTI instruction will require $rx=(2 * ap)$ additional oscillator clock cycles. The term “ap” represents the number of additional (if any) oscillator clock cycles that are required to access a P memory operand. Note that the term “ $(2 * ap)$ ” represents the two program memory instruction fetches executed at the end of an RTI or RTS instruction to refill the instruction pipeline.

3. Evaluate the “ap” term using Table A-14.

According to Table A-14, the term “ap” depends upon where the referenced P memory location is located in the DSP56000/DSP56001 memory space. **External** memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the DSP56000/DSP56001 bus control register (BCR). Thus, assuming that the 16-bit bus control register contains the value \$0012, external P memory accesses require $wp=1$ wait state or additional oscillator clock cycles. For this example, the P memory reference is assumed to be an **internal** reference. This means that the return address (\$0100) pulled from the system stack by the RTI instruction is in internal P memory. Thus, according to Table A-14, the RTI instruction will use the value $ap=0$ additional oscillator clock cycles.

4. Compute final results.

Thus, based upon the assumptions given for Table A-6 and those listed in the problem statement for Example 3, the instruction

RTI

will require

1 instruction program word

and will execute in

$$\begin{aligned}
 &(4+rx) \\
 &=(4+(2 * ap)) \\
 &=(4+(2 * 0)) \\
 &= 4 \quad \text{oscillator clock cycles}
 \end{aligned}$$

Note that the “ap” term present in Table A-8 for the P memory move entry represents the wait state spent when accessing the program memory during DATA read or write and does not refer to instruction fetches.

Table A-6 Instruction Timing Summary (see Note 3)

Mnemonic	Instruction Program Words	Osc. Clock Cycles	Notes	Mnemonic	Instruction Program Words	Osc. Clock Cycles	Notes
ABS	1 + mv	2 + mv		MAC	1 + mv	2 + mv	
ADC	1 + mv	2 + mv		MACR	1 + mv	2 + mv	
ADD	1 + mv	2 + mv		MOVE	1 + mv	2 + mv	
ADDL	1 + mv	2 + mv		MOVEC	1 + ea	2 + mvc	
ADDR	1 + mv	2 + mv		MOVEM	1 + ea	6 + ea + ap	
AND	1 + mv	2 + mv		MOVEP	1 + ea	4 + mvp	
ANDI	1	2		MPY	1 + mv	2 + mv	
ASL	1 + mv	2 + mv		MPYR	1 + mv	2 + mv	
ASR	1 + mv	2 + mv		NEG	1 + mv	2 + mv	
BCHG	1 + ea	4 + mvb		NOP	1	2	
BCLR	1 + ea	4 + mvb		NORM	1	2	
BSET	1 + ea	4 + mvb		NOT	1 + mv	2 + mv	
BTST	1 + ea	4 + mvb		OR	1 + mv	2 + mv	
CLR	1 + mv	2 + mv		ORI	1	2	
CMP	1 + mv	2 + mv		REP	1	4 + mv	
CMPM	1 + mv	2 + mv		RESET	1	4	
DIV	1	2		RND	1 + mv	2 + mv	
DO	2	6 + mv		ROL	1 + mv	2 + mv	
ENDDO	1	2		ROR	1 + mv	2 + mv	
EOR	1 + mv	2 + mv		RTI	1	4 + rx	
Jcc	1 + ea	4 + jx		RTS	1	4 + rx	
JCLR	2	6 + jx		SBC	1 + mv	2 + mv	
JMP	1 + ea	4 + jx		STOP	1	n/a	1
JScC	1 + ea	4 + jx		SUB	1 + mv	2 + mv	
JSCLR	2	6 + jx		SUBL	1 + mv	2 + mv	
JSET	2	6 + jx		SUBR	1 + mv	2 + mv	
JSR	1 + ea	4 + jx		SWI	1	8	
JSSET	2	6 + jx		Tcc	1	2	
LSL	1 + mv	2 + mv		TFR	1 + mv	2 + mv	
LSR	1 + mv	2 + mv		TST	1 + mv	2 + mv	
LUA	1	4		WAIT	1	n/a	2

Note 1: The STOP instruction disables the internal clock oscillator. After clock turn on, an internal counter counts 65,536 clock cycles (if bit 6 in the OMR is clear) before enabling the clock to the internal DSP circuits. If bit 6 in the OMR is set, only six clock cycles are counted before enabling the clock to the external DSP circuits.

Note 2: The WAIT instruction takes a minimum of 16 cycles to execute when an internal interrupt is pending during the execution of the WAIT instruction.

Note 3: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ap" term should be added, and, to each two-word instruction, a "+(2*ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

Table A-7 Parallel Data Move Timing

Parallel Move Operation	+ mv Words	+ mv Cycles	Comments
No Parallel Data Move	0	0	
I Immediate Short Data	0	0	
R Register to Register	0	0	
U Address Register Update	0	0	
X: X Memory Move	ea	ea + ax	See Note 1
X:R X Memory and Register	ea	ea + ax	See Note 1
Y: Y Memory Move	ea	ea + ay	See Note 1
R:Y Y Memory and Register	ea	ea + ay	See Note 1
L: Long Memory Move	ea	ea + axy	
X:Y: XY Memory Move	0	ea + axy	
LMS(X) LMS X Memory Moves	0	ea + ax	See Notes 1,2
LMS(Y) LMS Y Memory Moves	0	ea + ay	See Notes 1,2

Note 1: The ax or ay term does not apply to MOVE IMMEDIATE DATA.

Note 2: The ea term does not apply to ABSOLUTE ADDRESS and IMMEDIATE DATA.

Table A-8 MOVEC Timing Summary (see Note 2)

MOVEC Operation	+ mvc Cycles	Comments
Immediate Short → Register	0	
Register ↔ Register	0	
X Memory ↔ Register	ea + ax	See Note 1
Y Memory ↔ Register	ea + ay	See Note 1
P Memory ↔ Register	4 + ea + ap	

Note 1: The ax or ay term does not apply to MOVE IMMEDIATE DATA.

Note 2: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ ap" term should be added, and to each two-word instruction, a "(2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

Table A-9 MOVEP Timing Summary (see Note 2)

MOVEP Operation	+ mvp Cycles	Comments
Register↔ Peripheral	aio	
X Memory↔ Peripheral	ea + ax + aio	See Note 1
Y Memory↔ Peripheral	ea + ay + aio	See Note 1
P Memory↔ Peripheral	2 + ea + ap + aio	

Note 1: The ax or ay term does not apply to MOVE IMMEDIATE DATA.

Note 2: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ ap" term should be added, and to each two-word instruction, a "(2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

Note that the "ap" term present in Table A-9 for the P memory move entry represents the wait states spent when accessing the program memory during DATA read or writer operations and does not refer to instruction fetches.

Table A-10 Bit Manipulation Timing Summary (see Note 2)

Bit Manipulation Operation	+ mvb Cycles	Comments
Bxxx Peripheral	2 * aio	See Note 1
Bxxx X Memory	ea + (2 * ax)	See Note 1
Bxxx Y Memory	ea + (2 * ay)	See Note 1
Bxxx Register Direct	0	See Note 1
BTST Peripheral	aio	
BTST X Memory	ea + ax	
BTST Y Memory	ea + ay	

Note 1: Bxxx = BCHG, BCLR, or BSET.

Note 2: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ ap" term should be added, and to each two-word instruction, a "(2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

Table A-11 Jump Instruction Timing Summary

Jump Instruction Operation	+ jx Cycles	Comments
Jbit Register Direct	2 * ap	See Note 1
Jbit Peripheral	aio + (2 * ap)	See Note 1
Jbit X Memory	ea + ax + (2 * ap)	See Note 1
Jbit Y Memory	ea + ay + (2 * ap)	See Note 1
Jxxx	ea + (2 * ap)	See Note 2

Note 1: Jbit = JCLR, JSCLR, JSET, and JSSET

Note 2: Jxxx = Jcc, JMP, JSc, and JSR

All one-word jump instructions execute TWO program memory fetches to refill the pipeline, which is represented by the “+(2 * ap)” term.

All two-word jumps execute THREE program memory fetches to refill the pipeline, but one of those fetches is sequential (the instruction word located at the jump instruction 2nd word address+1), so it is not counted as per assumption 4. If the jump instruction was fetched from a program memory segment with wait states, another “ap” should be added to account for that third fetch.

Table A-12 RTI/RTS Timing Summary

Operation	+ rx Cycles
RTI	2 * ap
RTS	2 * ap

The term “2 * ap” come from the two instruction fetches done by the RTI/RTS instruction to refill the pipeline.

Table A-13 Addressing Mode Timing Summary

Effective Addressing Mode	+ ea Words	+ ea Cycles
Address Register Indirect		
No Update	0	0
Postincrement by 1	0	0
Postdecrement by 1	0	0
Postincrement by Offset Nn	0	0
Postdecrement by Offset Nn	0	0
Indexed by Offset Nn	0	2
Predecrement by 1	0	2
Special		
Immediate Data	1	2
Absolute Address	1	2
Immediate Short Data	0	0
Short Jump Address	0	0
Absolute Sort Address	0	0
I/O Short Address	0	0
Implicit	0	0

Table A-14 Memory Access Timing Summary

Access Type	X Mem Access	Y Mem Access	P Mem Access	I/O Access	+ ax Cycle	+ ay Cycle	+ ap Cycle	+ aio Cycle	+ axy Cycle
X:	Int	—	—	—	0	—	—	—	—
X:	Ext	—	—	—	wx	—	—	—	—
Y:	—	Int	—	—	—	0	—	—	—
Y:	—	Ext	—	—	—	wy	—	—	—
P:	—	—	Int	—	—	—	0	—	—
P:	—	—	Ext	—	—	—	wp	—	—
I/O:	—	—	—	Int	—	—	—	0	—
I/O:	—	—	—	Ext	—	—	—	wio	—
L: XY:	—	Int	—	—	—	—	—	—	0
L: XY:	Int	Ext	—	—	—	—	—	—	wy
L: XY:	Ext	Int	—	—	—	—	—	—	wx
L: XY:	Ext	Ext	—	—	—	—	—	—	2 + wx + wy

Note 1: wx = external X memory access wait states
 wy = external Y memory access wait states
 wp = external P memory access wait states
 wio = external I/O memory access wait states

Note 2: wx, wy, wp, and wio are programmable from 0 - 15 wait states in the port A bus control register (BCR).

A.8 INSTRUCTION SEQUENCE RESTRICTIONS

Due to the pipelined nature of the DSP core processor, there are certain instruction sequences that are forbidden and will cause undefined operation. Most of these restricted sequences would cause contention for an internal resource, such as the stack register. The DSP assembler will flag these as assembly errors.

Most of the following restrictions represent very unusual operations which probably would never be used but are listed only for completeness.

Note: The DSP56000/DSP56001 macro assembler is designed to recognize all restrictions and flag them as errors at the source code level. Since many of these are instruction sequence restrictions, they cannot be flagged as errors at the object code level such as when using the DSP56000/DSP56001 simulator's single-line assembler. Therefore, if any changes are made at the object code level using the simulator, the user should always re-assemble his program at the source code level using the DSP56000/DSP56001 macro assembler to verify that no restricted instruction sequences have been generated.

A.8.1 Restrictions Near the End of DO Loops

Proper DO loop operation is not guaranteed if an instruction **starting** at address **LA-2, LA-1, or LA** specifies one of the **program controller registers** SR, SP, SSL, LA, LC, or (implicitly) PC as a **destination** register. Similarly, the SSH register may not be specified

as a **source or destination** register in an instruction starting at address **LA-2, LA-1, or LA**. Additionally, the SSH register cannot be specified as a **source** register in the **DO** instruction itself, and **LA** cannot be used as a **target** for **jumps to subroutine** (i.e., JSR, JScc, JSSET, or JSCLR to LA). The following instructions cannot **begin** at the indicated position(s) near the end of a DO loop:

At LA-2, LA-1, and LA	DO BCHG LA, LC, SR, SP, SSH, or SSL BCLR LA, LC, SR, SP, SSH, or SSL BSET LA, LC, SR, SP, SSH, or SSL BTST SSH JCLR/JSET/JSCLR/JSSET SSH MOVEC from SSH MOVEM from SSH MOVEP from SSH MOVEC to LA, LC, SR, SP, SSH, or SSL MOVEM to LA, LC, SR, SP, SSH, or SSL MOVEP to LA, LC, SR, SP, SSH, or SSL ANDI MR ORI MR
At LA	any two-word instruction Jcc JMP JScc JSR REP RESET RTI RTS STOP WAIT
Other Restrictions	DO SSH,xxxx JSR to (LA) whenever the loop flag (LF) is set JScc to (LA) whenever the loop flag (LF) is set JSCLR to (LA) whenever the loop flag (LF) is set JSSET to (LA) whenever the loop flag (LF) is set

This restriction applies to the situation in which the DSP56000/DSP56001 simulator's single-line assembler is used to change the **last** instruction in a DO loop from a one-word instruction to a two-word instruction. All changes made using the simulator should be reassembled at the **source code** level using the DSP56000/DSP56001 macro assembler to verify that no restricted instruction sequences have been generated.

Note: Due to pipelining, if an address register (R0–R7, N0–N7, or M0–M7) is changed using a move-type instruction (LUA, Tcc, MOVE, MOVEC, MOVEM, MOVEP, or parallel move), the new contents of the destination address register will not be available for use during the **following** instruction (i.e., there is a single instruction cycle pipeline delay). This restriction also applies to the situation in which the **last** instruction in a **DO** loop changes an address register **and** the **first** instruction at the **top** of the DO loop uses that same address register. The **top** instruction becomes the **following** instruction because of the loop construct. The assembler will generate a warning if this condition is detected.

A.8.2 Other DO Restrictions

Due to pipelining, the DO instruction must not be **immediately preceded** by any of the following instructions:

Immediately before DO

BCHG LA, LC, SSH, SSL, or SP
BCLR LA, LC, SSH, SSL, or SP
BSET LA, LC, SSH, SSL, or SP
MOVEC to LA, LC, SSH, SSL, or SP
MOVEM to LA, LC, SSH, SSL, or SP
MOVEP to LA, LC, SSH, SSL, or SP
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH

A.8.3 ENDDO Restrictions

Due to pipelining, the ENDDO instruction must not be **immediately preceded** by any of the following instructions:

Immediately before ENDDO

BCHG LA, LC, SR, SSH, SSL, or SP
BCLR LA, LC, SR, SSH, SSL, or SP
BSET LA, LC, SR, SSH, SSL, or SP
MOVEC to LA, LC, SR, SSH, SSL, or SP
MOVEM to LA, LC, SR, SSH, SSL, or SP
MOVEP to LA, LC, SR, SSH, SSL, or SP
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH
ANDI MR
ORI MR
REP

A.8.4 RTI and RTS Restrictions

Due to pipelining, the RTI and RTS instructions must not be **immediately preceded** by any of the following instructions:

Immediately before RTI BCHG SR, SSH, SSL, or SP
 BCLR SR, SSH, SSL, or SP
 BSET SR, SSH, SSL, or SP
 MOVEC to SR, SSH, SSL, or SP
 MOVEM to SR, SSH, SSL, or SP
 MOVEP to SR, SSH, SSL, or SP
 MOVEC from SSH
 MOVEM from SSH
 MOVEP from SSH
 ANDI MR or ANDI CCR
 ORI MR or ORI CCR

Immediately before RTS BCHG SSH, SSL, or SP
 BCLR SSH, SSL, or SP
 BSET SSH, SSL, or SP
 MOVEC to SSH, SSL, or SP
 MOVEM to SSH, SSL, or SP
 MOVEP to SSH, SSL, or SP
 MOVEC from SSH
 MOVEM from SSH
 MOVEP from SSH

A.8.5 SP and SSH/SSL Manipulation Restrictions

In addition to all the above restrictions concerning MOVEC, MOVEM, MOVEP, SP, SSH, and SSL, the following MOVEC, MOVEM, and MOVEP restrictions apply:

Immediately before MOVEC from SSH or SSL BCHG to SP
 BCLR to SP
 BSET to SP

Immediately before MOVEM from SSH or SSL BCHG to SP
 BCLR to SP
 BSET to SP

Immediately before MOVEP from SSH or SSL BCHG to SP
 BCLR to SP
 BSET to SP

Freescale Semiconductor, Inc.

Immediately before MOVEC from SSH or SSL	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before MOVEM from SSH or SSL	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before MOVEP from SSH or SSL	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JCLR #n,SSH or SSL,xxxx	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JSET #n,SSH or SSL,xxxx	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JSCLR #n,SSH or SSL,xxxx	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JSSET #n,SSH or SSL,xxxx	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JCLR #n,SSH or SSL,xxxx	BCHG to SP BCLR to SP BSET to SP
Immediately before JSET #n,SSH or SSL,xxxx	BCHG to SP BCLR to SP BSET to SP
Immediately before JSCLR from SSH or SSL,xxxx	BCHG to SP BCLR to SP BSET to SP
Immediately before JSSET from SSH or SSL,xxxx	BCHG to SP BCLR to SP BSET to SP

Also, the instruction MOVEC SSH,SSH is illegal.

A.8.6 R, N, and M Register Restrictions

If an address register (R0–R7, N0–N7, or M0–M7) is changed with a move-type instruction (LUA, Tcc, MOVE, MOVEC, MOVEM, MOVEP, or parallel move), the new contents of the destination address register will **not** be available for use as a pointer during the **following** instruction (i.e., there is a single instruction cycle pipeline delay). This does not apply to address registers that are updated as part of an addressing mode update.

Note: This restriction also applies to the situation in which the **last** instruction in a **DO** loop changes an address register using a move-type instruction **and** the **first** instruction at the **top** of the DO loop uses that same address register. The **top** instruction becomes the **following** instruction because of the loop construct. The DSP assembler will generate a warning if this condition is detected.

A.8.7 Fast Interrupt Routines

The following instructions may not be used in a fast interrupt routine:

In a fast interrupt routine

DO
ENDDO
RTI
RTS
MOVEC to LA, LC, SSH, SSL, SP, or SR
MOVEM to LA, LC, SSH, SSL, SP, or SR
MOVEP to LA, LC, SSH, SSL, SP, or SR
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH
ORI MR or ORI CCR
ANDI MR or ANDI CCR
STOP
SWI
WAIT

A.8.8 REP Restrictions

The REP instruction can repeat any single-word instruction except the REP instruction itself and any instruction that changes program flow. The following instructions are not allowed to follow an REP instruction:

Immediately after REP DO
Jcc
JCLR
JMP
JSET
JScC
JSCLR
JSR
JSSET
REP
RTI
RTS
STOP
SWI
WAIT
ENDDO

Also, an REP instruction cannot be the **last** instruction in a DO loop (at LA).

A.9 INSTRUCTION ENCODING

This section summarizes instruction encoding for the DSP56000/DSP56001 instruction set. The instruction codes are listed in nominally descending order. The symbols used in decoding the various fields of an instruction are identical to those used in the Opcode section of the individual instruction descriptions. The user should always refer to the actual instruction description for complete information on the encoding of the various fields of that instruction.

Section A.9.1 gives the encodings for (1) various groupings of registers used in the instruction encodings, (2) condition code combinations, (3) addressing, and (4) addressing modes.

Section A.9.2 gives the encoding for the parallel move portion of an instruction. These 16-bit partial instruction codes may be combined with the 8-bit data ALU opcodes listed in Section A.9.3 to form a complete 24-bit instruction word.

Section A.9.3 gives the complete 24-bit instruction encoding for those instructions which do not allow parallel moves.

Section A.9.4 gives the encoding for the data ALU portion of those instructions which allow parallel data moves. These 8-bit partial instruction codes may be combined with the 16-bit parallel move opcodes listed in Section A.9.1 to form a complete 24-bit instruction word.

Section A.9.5 contains instruction encodings for nonsensical instructions (called insane instructions) for which encodings exist but which cause problems such as writing two sources to one destination.

A.9.1 Partial Encodings for Use in Instruction Encoding

Table A-15 Single-Bit Register Encodings

Code	d*	e	f	Where:
0	A	X0	Y0	d = 2 Accumulators in Data ALU
1	B	X1	Y1	e = 2 Registers in Data ALU
				f = 2 Registers in Data ALU

* For class II encodings for R:Y and X:R, see Table A - 16

Table A-16 Single-Bit Special Register Encodings

d	X:R Class II Opcode	R:Y Class II Opcode
0	A → X:<ea> X0 → A	Y0 → A A → Y:<ea>
1	B → X:<ea> X0 → B	Y0 → B B → Y:<ea>

Table A-17 Double-Bit Register Encodings

Code	DD	ee	ff
00	X0	X0	Y0
01	X1	X1	Y1
10	Y0	A	A
11	Y1	B	B

Where: DD = 4 registers in data ALU
 ee = 4 XDB registers in data ALU
 ff = 4 YDB registers in data ALU

Table A-18 Triple-Bit Register Encodings

Code	DDD	LLL	FFF	NNN	TTT	GGG
000	A0	A10	M0	N0	R0	*
001	B0	B10	M1	N1	R1	SR
010	A2	X	M2	N2	R2	OMR
011	B2	Y	M3	N3	R3	SP
100	A1	A	M4	N4	R4	SSH
101	B1	B	M5	N5	R5	SSL
110	A	AB	M6	N6	R6	LA
111	B	BA	M7	N7	R7	LC

* Reserved

Where: DDD : 8 accumulators in data ALU

LLL: 8 extended-precision registers in data ALU; LLL field is encoded as LOLL

FFF: 8 address modifier registers in address ALU

NNN: 8 address offset registers in address ALU

TTT: 8 address registers in address

GGG: 8 program controller registers

Table A-19(a) Four-Bit Register Encodings for 12 Registers in Data ALU

D	D	D	D	Description
0	0	X	X	Reserved
0	1	D	D	Data ALU Register
1	D	D	D	Data ALU Register

Table A-19(b) Four-Bit Register Encodings for 16 Condition Codes

Mnemonic	C	C	C	C	Mnemonic	C	C	C	C
CC(HS)	0	0	0	0	CS(LO)	1	0	0	0
GE	0	0	0	1	LT	1	0	0	1
NE	0	0	1	0	EQ	1	0	1	0
PL	0	0	1	1	MI	1	0	1	1
NN	0	1	0	0	NR	1	1	0	0
EC	0	1	0	1	ES	1	1	0	1
LC	0	1	1	0	LS	1	1	1	0
GT	0	1	1	1	LE	1	1	1	1

Table A-20 Five-Bit Register Encodings for 28 Registers in Data ALU and Address ALU

e d	e d	e or d	e d	e d	Description
0	0	0	0	X	Reserved
0	0	0	1	X	Reserved
0	0	1	D	D	Data ALU Register
0	1	D	D	D	Data ALU Register
1	0	T	T	T	Address ALU Register
1	1	N	N	N	Address Offset Register

Where: eeeee = source
 ddddd = destination

Table A-21 Six-Bit Register Encodings for 43 Registers On-Chip

d	d	d	d	d	d	Description
0	0	0	0	X	X	Reserved
0	0	0	1	D	D	Data ALU Register
0	0	1	D	D	D	Data ALU Register
0	1	0	T	T	T	Address ALU Register
0	1	1	N	N	N	Address Offset Register
1	0	0	F	F	F	Address Modifier Register
1	0	1	X	X	X	Reserved
1	1	0	X	X	X	Reserved
1	1	1	G	G	G	Program Controller Register

Table A-22 Write Control Encoding

W	Operation
0	Read Register or Peripheral
1	Write Register or Peripheral

Table A-23 Memory Space Bit Encoding

S	Operation
0	X Memory
1	Y Memory

Table A-24 Program Controller Register Encoding

E	E	Register
0	0	MR Mode Register
0	1	CCR Condition Code Register
1	0	OMR Operating Mode Register
1	1	— Reserved

Table A-25 Condition Code and Address Encoding

Code	Code Definition
c c c c	16 Condition Code Combinations
b b b b b	5-Bit Immediate Data
iiii iii	8-Bit Immediate Data (int, frac, mask)
iiii iii x x x x h h h h	12-Bit Immediate Data (iiii iii hhhh)
a a a a a	6-Bit Absolute Short (Low) Address
p p p p p	6-Bit Absolute I/O (High) Address
a a a a a a a a a a	12-Bit Fast Absolute Short (Low) Address

Table A-26 Effective Addressing Mode Encoding

M M M R R R	Code Definition
0 0 0 r r r	Post - N
0 0 1 r r r	Post + N
0 1 0 r r r	Post - 1
0 1 1 r r r	Post + 1
1 0 0 r r r	No Update
1 0 1 r r r	Indexed + N
1 1 1 r r r	Pre - 1
1 1 0 0 0 0	Absolute Address
1 1 0 1 0 0	Immediate Data

RRR = three unencoded bits R0, R1, R2

MMM = three unencoded bits M0, m1, m2

NOTES:

(1) R2 is 0 for low register bank and 1 for the high register bank.

(2) M2 is 0 for all post update modes and 1 otherwise.

(3) M1 is 0 for update by register offset and 1 for update by one.

(4) M0 is 0 for minus and 1 for plus.

(5) For X and Y moves, rr is a subfield or rrr with equations: $r2 = \overline{R2}$.

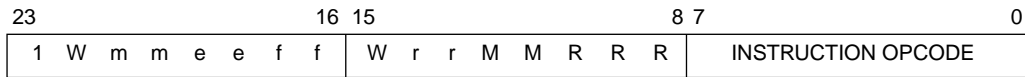
(6) For rr field, r1 is bit 14; r0 is bit 13.

(7) For X and Y moves, mm is a subfield of mmm with equations: $M2 = (\overline{M1} \vee \overline{M0})$ $m2 = (\overline{m1} \vee \overline{m0})$.

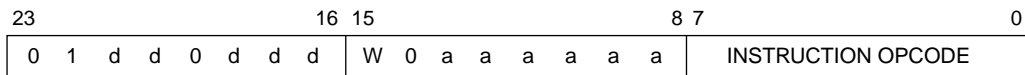
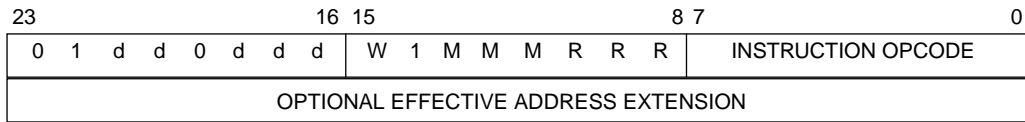
(8) For mm field, m1 is bit 21; m0 is bit 20. For MM field, M1 is bit 12; M0 is bit 11.

A.9.2 Instruction Encoding for the Parallel Move Portion of an Instruction

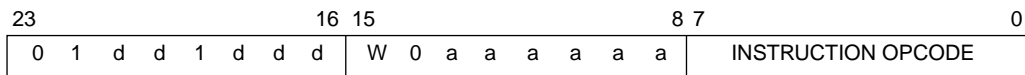
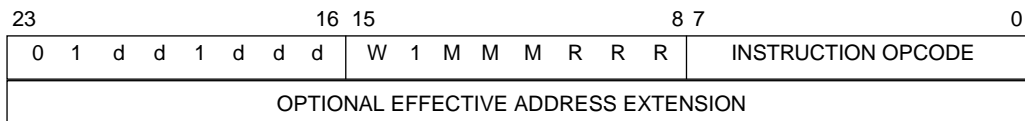
X: Y: Parallel Data Move



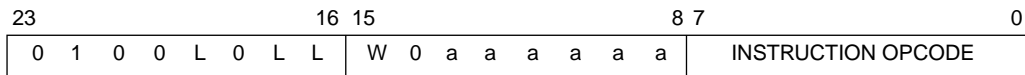
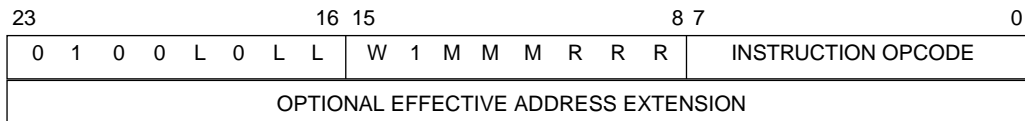
X: Parallel Data Move



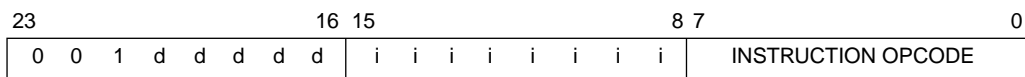
Y: Parallel Data Move



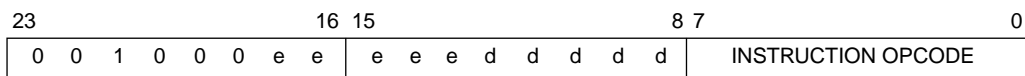
L: Parallel Data Move



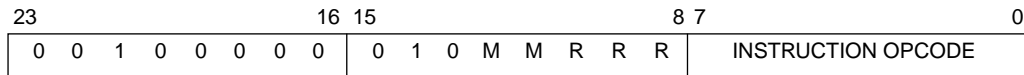
I: Immediate Short Parallel Data Move



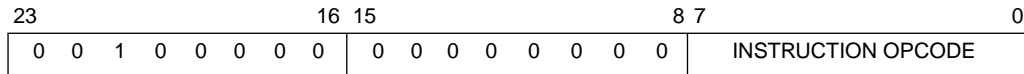
R: Register to Register Parallel Data Move



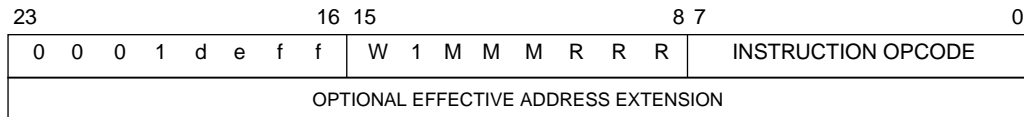
U: Address Register Update Parallel Data Move



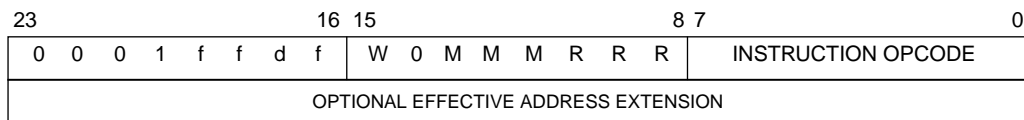
Parallel Data Move NOP



R:Y Parallel Data Move



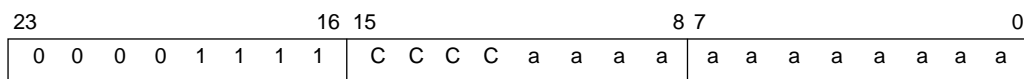
X:R Parallel Data Move



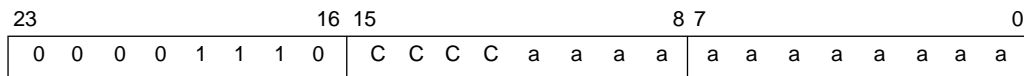
A.9.3 Instruction Encoding for the Parallel Move Portion of an Instruction

Note: For following bit class instructions bbbbb = 11bbb is reserved:
JSSET, JSCLR, JSET, JCLR, BTST, BCHG, BSET, and BCLR.

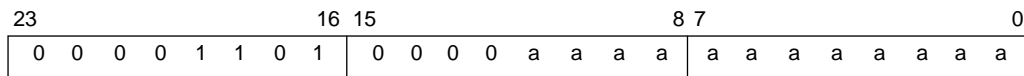
JScC xxx



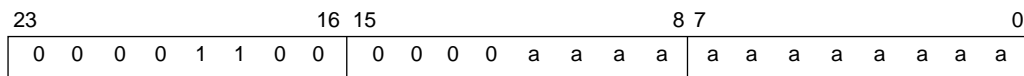
JcC xxx



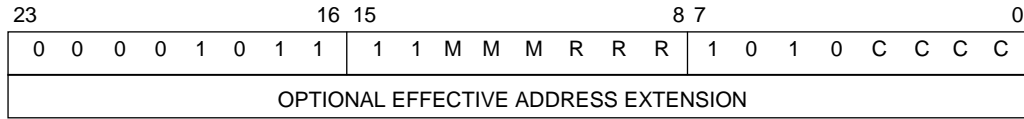
JSR xxx



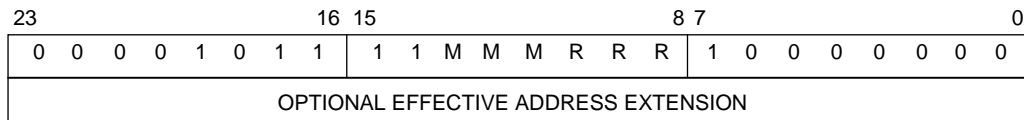
JMP xxx



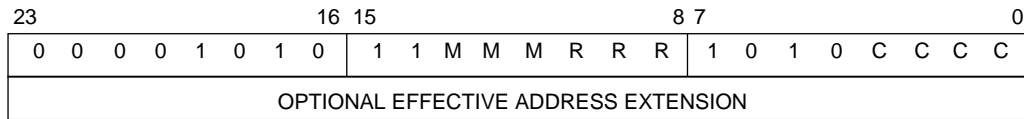
JScC ea



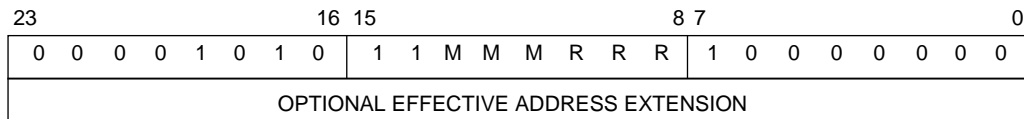
JSR ea



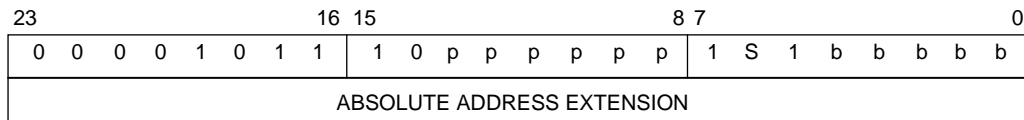
JcC ea



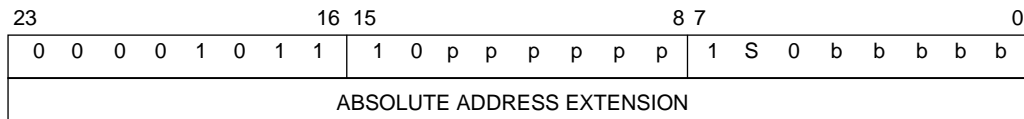
JMP ea



JSSET #n,X:pp,xxxx JSSET #n,Y:pp,xxxx



JSCLR #n,X:pp,xxxx JSCLR #n,Y:pp,xxxx



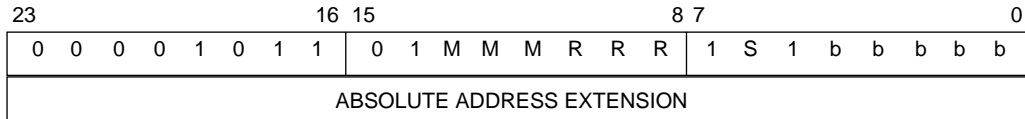
JSET #n,X:pp,xxxx JSET #n,Y:pp,xxxx



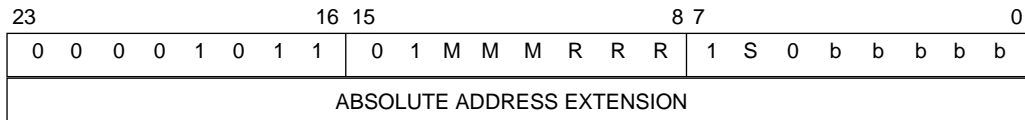
JCLR #n,X:pp,xxxx
JCLR #n,Y:pp,xxxx



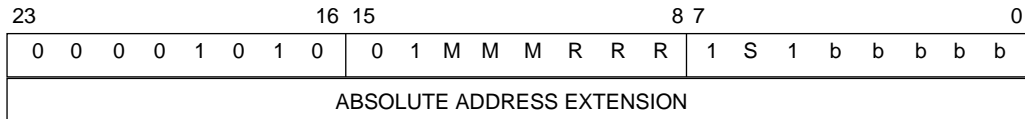
JSSET #n,X:ea,xxxx
JSSET #n,Y:ea,xxxx



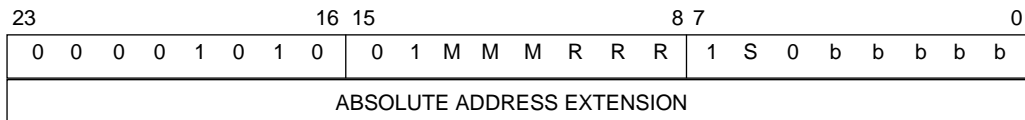
JSCLR #n,X:ea,xxxx
JSCLR #n,Y:ea,xxxx



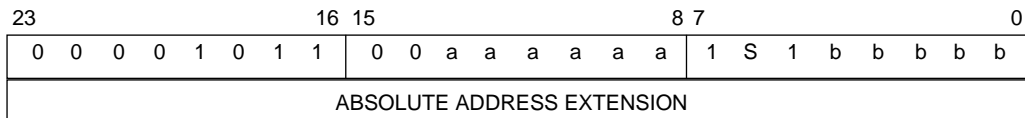
JSET #n,X:ea,xxxx
JSET #n,Y:ea,xxxx



JCLR #n,X:ea,xxxx
JCLR #n,Y:ea,xxxx



JSSET #n,X:aa,xxxx
JSSET #n,Y:aa,xxxx

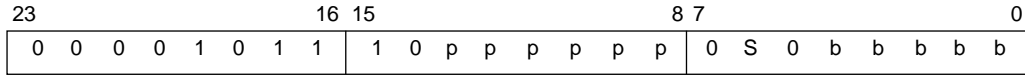


JSCLR #n,X:aa,xxxx

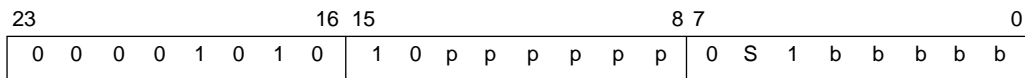
BTST #n,Y:pp



BCHG #n,X:pp
BCHG #n,Y:pp



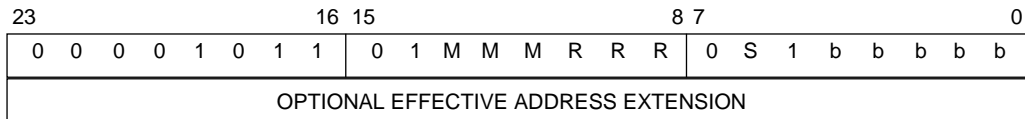
BSET #n,X:pp
BSET #n,Y:pp



BCLR #n,X:pp
BCLR #n,Y:pp



BTST #n,X:ea
BTST #n,Y:ea



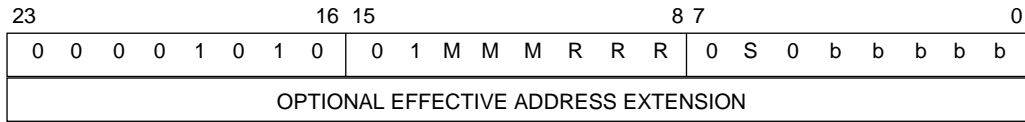
BCHG #n,X:ea
BCHG #n,Y:ea



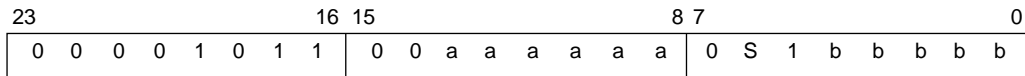
BSET #n,X:ea
BSET #n,Y:ea



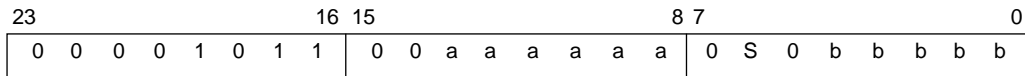
BCLR #n,X:ea
BCLR #n,Y:ea



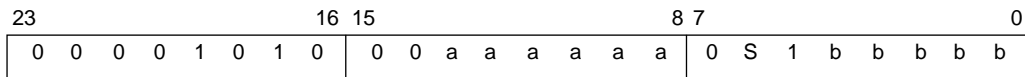
BTST #n,X:aa
BTST #n,Y:aa



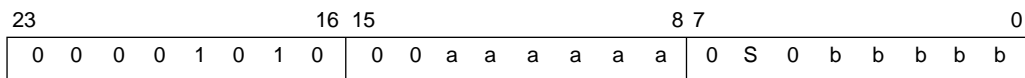
BCHG #n,X:aa
BCHG #n,Y:aa



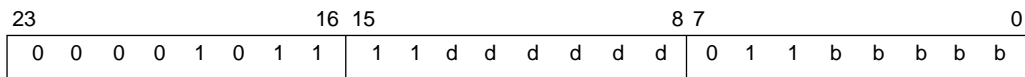
BSET #n,X:aa
BSET #n,Y:aa



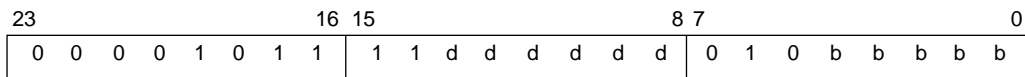
BCLR #n,X:aa
BCLR #n,Y:aa



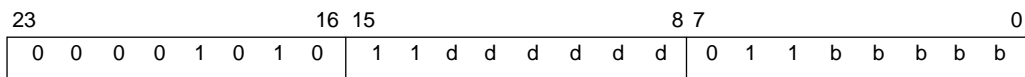
BTST #n,D



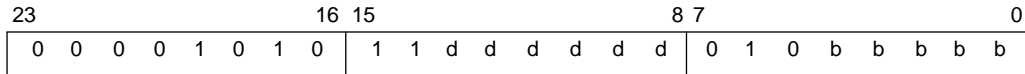
BCHG #n,D



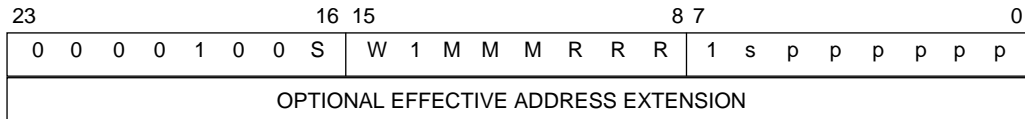
BSET #n,D



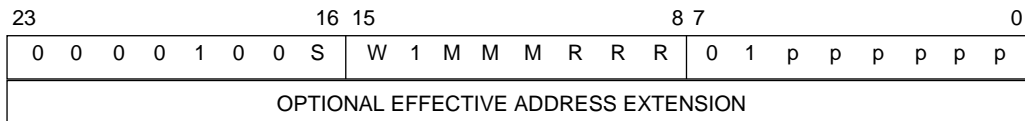
BCLR #n,D



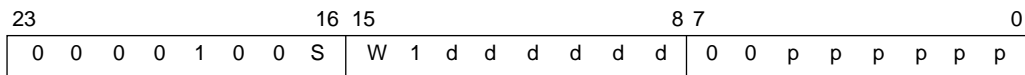
- MOVEP X:ea,X:pp**
- MOVEP Y:ea,X:pp**
- MOVEP #xxxxxx,X:pp**
- MOVEP X:pp,X:ea**
- MOVEP X:pp,Y:ea**
- MOVEP X:ea,Y:pp**
- MOVEP Y:ea,Y:pp**
- MOVEP #xxxxxx,Y:pp**
- MOVEP Y:pp,X:ea**
- MOVEP Y:pp,Y:ea**



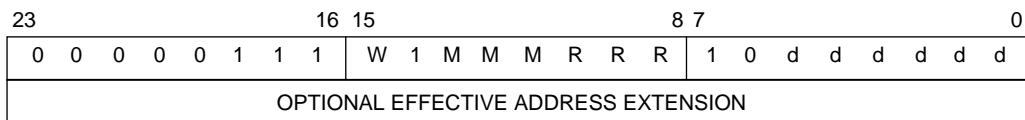
- MOVEP P:ea,X:pp**
- MOVEP X:pp,P:ea**
- MOVEP P:ea,Y:pp**
- MOVEP Y:pp,P:ea**



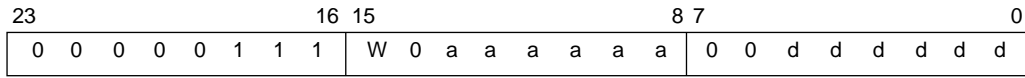
- MOVEP S,X:pp**
- MOVEP X:pp,D**
- MOVEP S,Y:pp**
- MOVEP Y:pp,D**



- MOVE(M) S,P:ea**
- MOVE(M) P:ea,D**



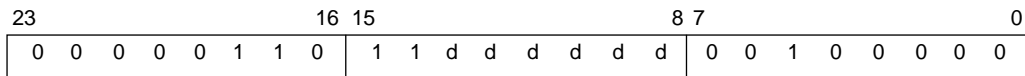
MOVE(M) S,P:aa
MOVE(M) P:aa,D



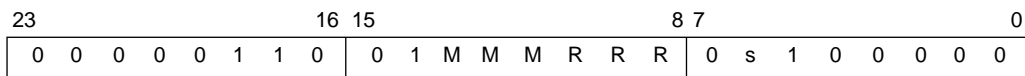
REP #xxx



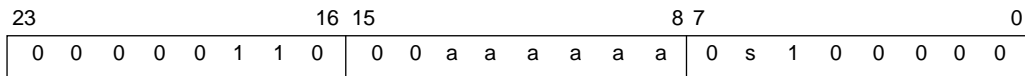
REP S



REP X:ea
REP Y:ea



REP X:aa
REP Y:aa



DO #xxx,expr



DO S,expr

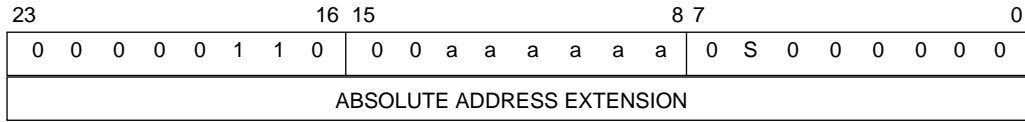


DO X:ea,expr
DO Y:ea,expr

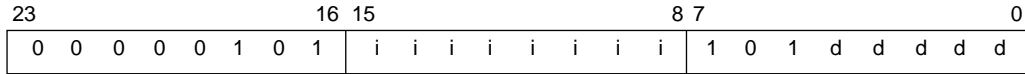


DO X:aa,expr

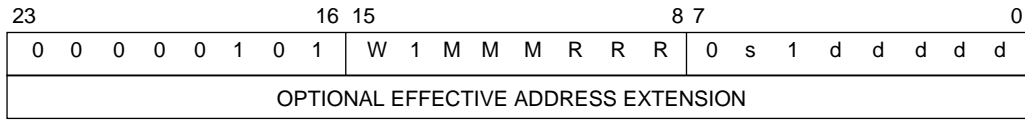
DO Y:aa,expr



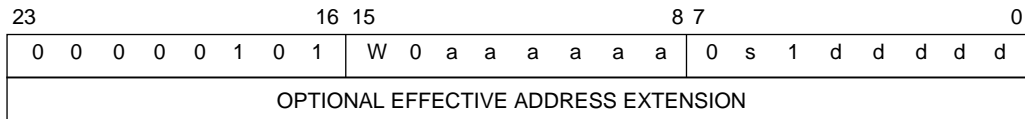
MOVE(C) #xx,D1



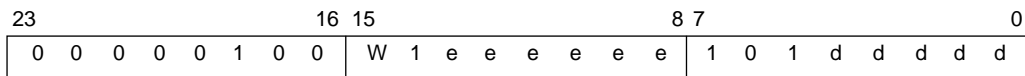
MOVE(C) X:ea,D1
MOVE(C) S1,X:ea
MOVE(C) Y:ea,D1
MOVE(C) S1,Y:ea
MOVE(C) #xxxx,D1



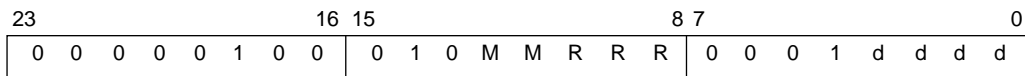
MOVE(C) X:aa,D1
MOVE(C) S1,X:aa
MOVE(C) Y:aa,D1
MOVE(C) S1,Y:aa



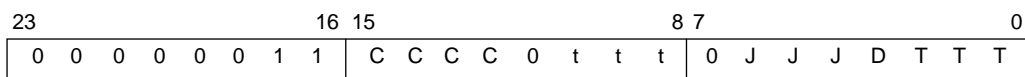
MOVE(C) S1,D2
MOVE(C) S2,D1



LUA ea,D



Tcc S1,D1 S2,D2



Tcc S1,D1

23	16 15	8 7	0
0 0 0 0 0 0 1 0	C C C C 0 0 0 0	0 J J J D 0 0 0	

NORM Rn,D

23	16 15	8 7	0
0 0 0 0 0 0 0 1	1 1 0 1 1 R R R	0 0 0 1 d 1 0 1	

DIV S,D

23	16 15	8 7	0
0 0 0 0 0 0 0 1	1 0 0 0 0 0 0 0	0 1 J J d 0 0 0	

OR(I) #xx,D

23	16 15	8 7	0
0 0 0 0 0 0 0 0	i i i i i i i i	1 1 1 1 1 0 E E	

AND(I) #xx,D

23	16 15	8 7	0
0 0 0 0 0 0 0 0	i i i i i i i i	1 0 1 1 1 0 E E	

ENDDO

23	16 15	8 7	0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 0 0 0 1 1 0 0	

STOP

23	16 15	8 7	0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 0 0 0 0 1 1 1	

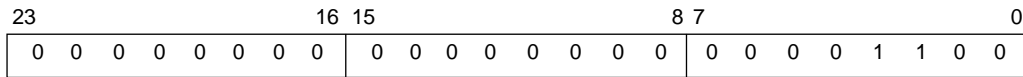
WAIT

23	16 15	8 7	0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 0 0 0 0 1 1 0	

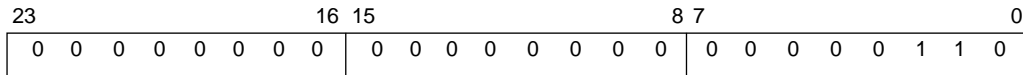
RESET

23	16 15	8 7	0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 0 0 0 0 1 0 0	

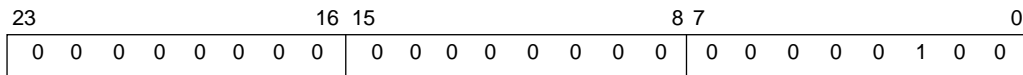
RTS



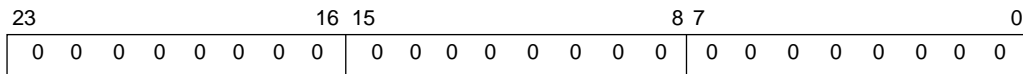
SWI



RTI



NOP



A.9.4 Parallel Instruction Encoding of the Operation Code

The operation code encoding for the instructions which allow parallel moves is divided into the multiply and nonmultiply instruction encodings shown in the following subsection.

Multiply Instruction Encoding

The 8-bit operation code for multiply instructions allowing parallel moves has different fields than the nonmultiply instruction's operation code.

The 8-bit operation code=**1QQQ dkkk** where
 QQQ=selects the inputs to the multiplier
 kkk = three unencoded bits k2, k1, k0
 d = destination accumulator
 d = 0 → A
 d = 1 → B

Table A-27 Operation Code K0-2 Decode

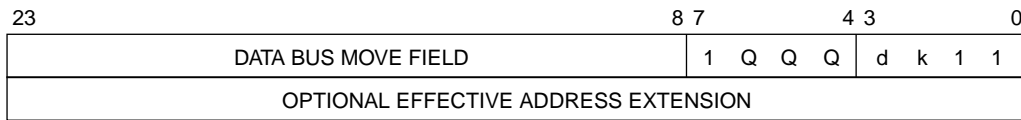
Code	k2	k1	k0
0	positive	mpy only	don't round
1	negative	mpy and acc	round

Table A-28 Operation Code QQQ Decode

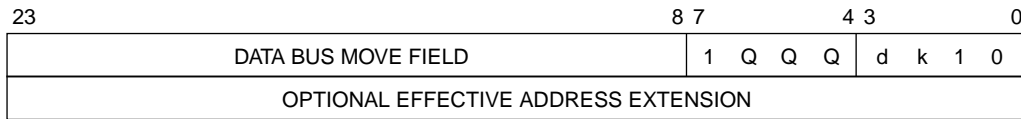
Q	Q	Q	S1	S2
0	0	0	X0	X0
0	0	1	Y0	Y0
0	1	0	X1	X0
0	1	1	Y1	Y0
1	0	0	X0	Y1
1	0	1	Y0	X0
1	1	0	X1	Y0
1	1	1	Y1	X1

NOTE: S1 and S2 are the inputs to the multiplier.

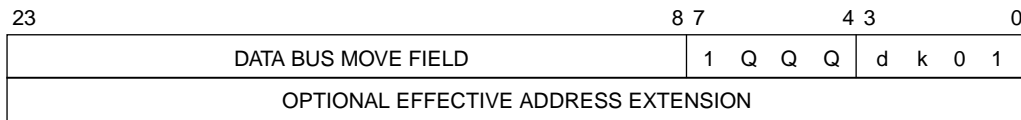
MACR (±) S1,S2,D
MACR (±) S2,S1,D



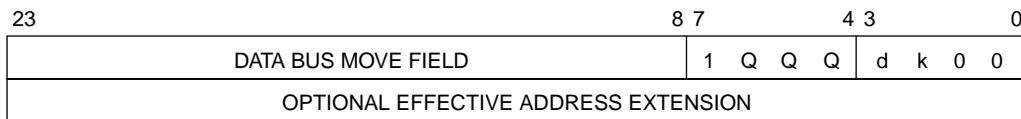
MAC (±) S1,S2,D
MAC (±) S2,S1,D



MPYR (±) S1,S2,D
MPYR (±) S2,S1,D



MPY (±) S1,S2,D
MPY (±) S2,S1,D



CMPM S1,S2

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 J J J	d 1 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

AND S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 1 J J	d 1 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

CMP S1,S2

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 J J J	d 1 0 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

SUB S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 J J J	d 1 0 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

EOR S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 1 J J	d 0 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

OR S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 1 J J	d 0 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

TFR S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 J J J	d 0 0 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ADD S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 J J J	d 0 0 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

SBC S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 J	d 1 0 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ADC S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 J	d 0 0 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ROL D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 1	d 1 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

NEG D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 1	d 1 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

LSL D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 1	d 0 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ASL D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 1	d 0 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ROR D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 0	d 1 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ABS D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 0	d 1 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

LSR D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 0	d 0 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ASR D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 1 0	d 0 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

NOT D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 1	d 1 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

SUBL S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 1	d 1 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

CLR D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 1	d 0 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ADDL S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 1	d 0 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

RND D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 1	d 0 0 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

SUBR S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 0	d 1 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

TST D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 0	d 0 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ADDR S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 0	d 0 1 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

ILLEGAL

23	16 15	8 7	4 3	0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0	0 1 0 1	

MOVE S,D

23	8 7	4 3	0
DATA BUS MOVE FIELD		0 0 0 0	0 0 0 0
OPTIONAL EFFECTIVE ADDRESS EXTENSION			

