DSP96002

32-BIT
DIGITAL SIGNAL PROCESSOR
USER'S MANUAL

Motorola, Inc.
Semiconductor Products Sector
DSP Division
6501 William Cannon Drive, West
Austin, Texas 78735-8598

SECTION 1
DSP96002 INTRODUCTION

This manual describes the first member of a family of dual-port IEEE floating point programmable CMOS processors. The family concept defines a core as the Data ALU, Address Generation Unit, Program Controller and associated Instruction Set. The On-Chip Program Memory, Data Memories and Peripherals support many numerically intensive applications and minimize system size and power dissipation; however, they are not considered part of the core.

The first family member is the DSP96002. The main characteristics of the DSP96002 are support of IEEE 754 Single Precision (8 bit Exponent and 24 bit Mantissa) and Single Extended Precision (11 bit Exponent and 32 bit Mantissa) Floating-Point and 32 bit signed and unsigned fixed point arithmetic, coupled with two identical external memory expansion ports. Its features are listed below.

DSP96002 Features

- IEEE 745 Standard SP (32-bit) and SEP (44 bit) Arithmetic
- 16.5 Million Instructions per Second (Mips) with a 33 Mhz clock
- 49.5 Million Floating Point Instructions per Second (MFLOPS) peak with a 33 Mhz clock
- Single-Cycle 32 x 32 Bit Parallel Multiplier
- Highly Parallel Instruction Set with Unique DSP Addressing Modes
- Nested Hardware Do Loops
- Fast Auto-Return Interrupts
- 2 Independent On-Chip 512 x 32 Bit Data RAMs
- 2 Independent On-Chip 1024 x 32 Bit Data ROMs
- Off-Chip Expansion to 2 x 2^{32} 32-Bit Words of Data Memory
- On-Chip 1,024 x 32 Bit Program RAM
- On-Chip 64 x 32 Bit Bootstrap ROM
- Off-Chip Expansion to 2^{32} 32-Bit Words of Program Memory
- Two Identical External Memory Expansion Ports
- Two 32-Bit Parallel Host MPU/DMA Interfaces
- On-Chip Two-Channel DMA Controller
- On-Chip Emulator

SECTION 2
SIGNAL DESCRIPTION AND BUS OPERATION

2.1 PINOUT

The functional signal groups of the DSP96002 are shown in Figure 2-2, and are described in the following sections. A pin allocation summary is shown in Figure 2-1. Specific pinout and timing information is available in the DSP96002 Technical Data Sheet (DSP96002/D).

2.1.1 Package

The DSP96002 is available in a 223 pin PGA package. There are 176 signal pins (including 5 spares), 17 power pins and 30 ground pins. All packaging information is available in the data sheet.

2.1.2 Interrupt And Mode Control (4 Pins)

$\overline{\text{RESET}}$ (Reset) - active low, Schmitt trigger input. $\overline{\text{RESET}}$ is internally synchronized to the input clock (CLK). When asserted, the chip is placed in the reset state and the internal phase generator is reset. The Schmitt trigger input allows a slowly rising input (such as a capacitor charging) to reliably reset the chip. If $\overline{\text{RESET}}$ is deasserted synchronous to the input clock (CLK), exact startup timing is guaranteed, allowing multiple processors to startup synchronously and operate together in "lock-step". When the $\overline{\text{RESET}}$ pin is deasserted, the initial chip operating mode is latched from the MODA, MODB and MODC pins.

MODA/$\overline{\text{IRQA}}$ (Mode Select A/External Interrupt Request A) - active low input, internally synchronized to the input clock (CLK). MODA/$\overline{\text{IRQA}}$ selects the initial chip operating mode during hardware reset and becomes a level sensitive or negative edge triggered, maskable interrupt request input during normal instruction processing. MODA, MODB and MODC select one of 8 initial chip operating modes, latched into the operating mode register (OMR) when the $\overline{\text{RESET}}$ pin is deasserted. If $\overline{\text{IRQA}}$ is asserted synchronous to the input clock (CLK), multiple processors can be resynchronized using the WAIT instruction and asserting $\overline{\text{IRQA}}$ to exit the wait state. If the processor is in the STOP standby state and $\overline{\text{IRQA}}$ is asserted, the processor will exit the STOP state.

CPU Pins	**Pins**
Reset and IRQs	4
Clock Input	1
OnCE Port	4
CPU Spare	1
Quiet Power	4
Quiet Ground	4
CPU Subtotal	18

Power/Ground Planes	**Pins**
Package Noisy Power Plane	2
Package Noisy Ground Plane	5
Package Quiet Power Plane	1
Package Quiet Ground Plane	1
Power/Ground Plane Subtotal	9

Port A/B	**Each Port Pins**	**Both Ports Pins**
Data Bus	32	64
Address Bus	32	64
Data Power	2	4
Data Ground	4	8
Address Power	2	4
Address Ground	4	8
Addr/Data Subtotal	76	152

Port A/B	**Each Port Pins**	**Both Ports Pins**
Bus Control Signals	17	34
Bus Control Spare	2	4
Bus Control Power	1	2
Bus Control Ground	2	4
Control Subtotal	22	44

Pinout Summary	**Pins**
CPU Pins	18
Package Power/Ground Planes	9
Port A/B Pins	
Data and Address	152
Bus Control	44
TOTALS	223

Figure 2-1. DSP96002 Functional Group Pin Allocation

MODB/$\overline{\text{IRQB}}$ (Mode Select B/External Interrupt Request B) - active low input, internally synchronized to the input clock (CLK). MODB/$\overline{\text{IRQB}}$ selects the initial chip operating mode during hardware reset and becomes a level sensitive or negative edge triggered, maskable interrupt request input during normal instruction processing. MODA, MODB and MODC select one of 8 initial chip operating modes, latched into the operating mode register (OMR) when the $\overline{\text{RESET}}$ pin is deasserted. If $\overline{\text{IRQB}}$ is asserted synchronous to the input clock (CLK), multiple processors can be resynchronized using the WAIT instruction and asserting $\overline{\text{IRQB}}$ to exit the wait state.

MODC/$\overline{\text{IRQC}}$ (Mode Select C/External Interrupt Request C) - active low input, internally synchronized to the input clock (CLK). MODC/$\overline{\text{IRQC}}$ selects the initial chip operating mode dur-

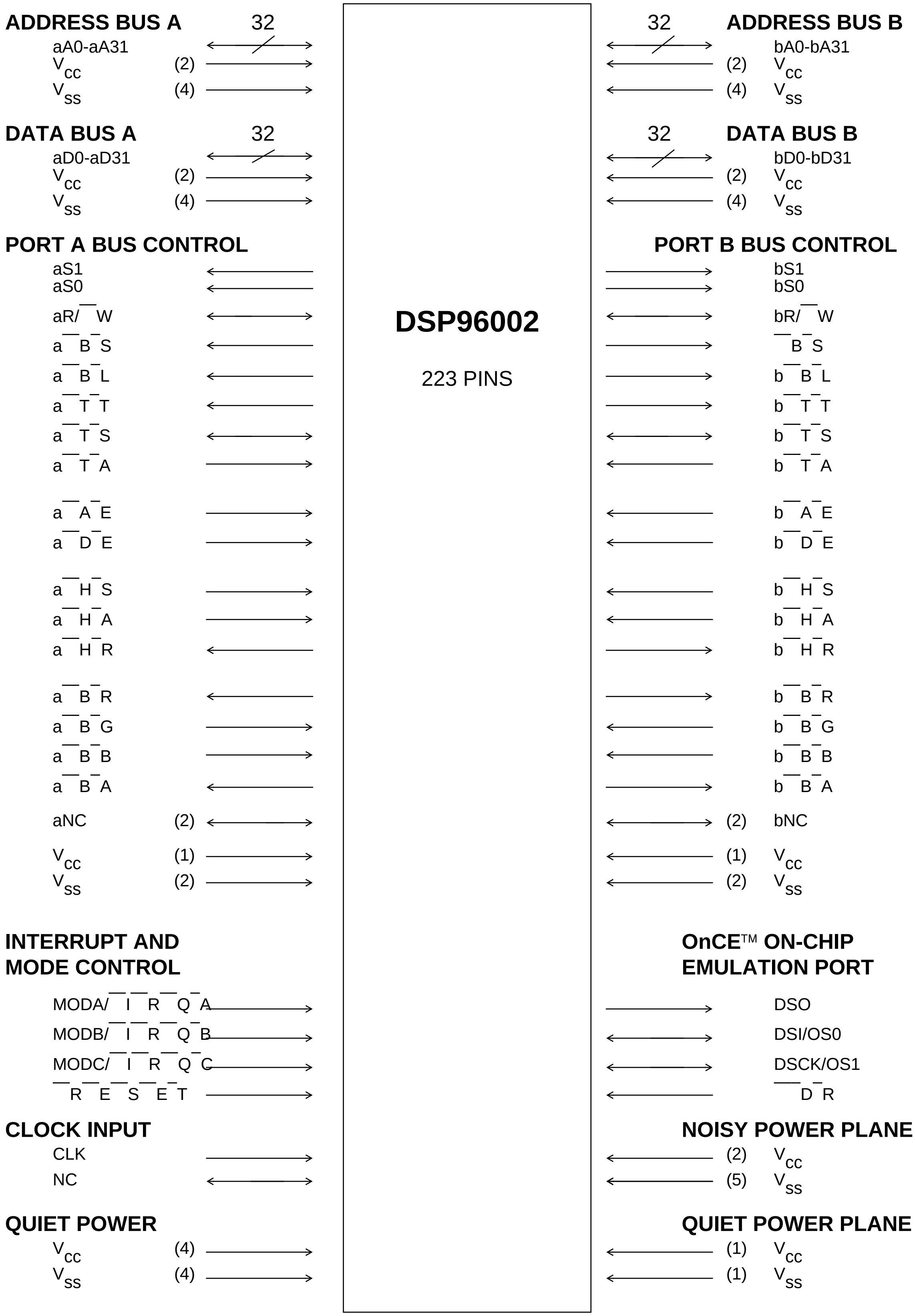


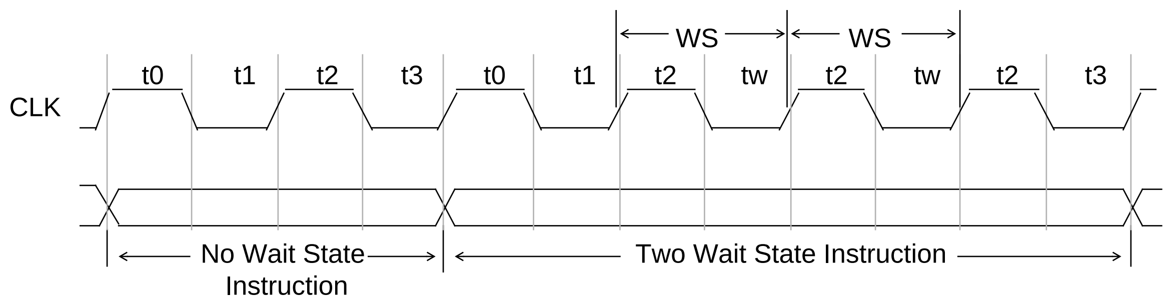
Figure 2-2. DSP96002 Functional Signal Groups

OnCE™ is a trademark of Motorola Inc.

ing hardware reset and becomes a level sensitive or negative edge triggered, maskable interrupt request input during normal instruction processing. MODA, MODB and MODC select one of 8 initial chip operating modes, latched into the operating mode register (OMR) when the $\overline{RESET}$ pin is deasserted. If $\overline{IRQC}$ is asserted synchronous to the input clock (CLK), multiple processors can be resynchronized using the WAIT instruction and asserting $\overline{IRQC}$ to exit the wait state.

2.1.3 Power and Clock (39 Pins)

CLK (Clock Input) - active high input, high frequency processor clock. Frequency is twice the instruction rate. An internal phase generator divides CLK into four phases (t0, t1, t2 and t3) which is the basic instruction execution cycle. Additional tw phases are optionally generated to insert wait states (WS) into instruction execution. A wait state is formed by pairing a t2 and tw phase. CLK should be continuous with a 46-54% duty cycle.



Quiet VCC (4) (Power) - isolated power for the CPU logic. Must be tied to all other chip power pins externally. User must provide adequate external decoupling capacitors.

Quiet VSS (4) (Ground) - isolated ground for the CPU logic. Must be tied to all other chip ground pins externally. User must provide adequate external decoupling capacitors.

Address Bus VCC(4) (Power) - isolated power for sections of address bus I/O drivers. Must be tied to all other chip power pins externally. User must provide adequate external decoupling capacitors.

Address Bus VSS(8) (Ground) - isolated ground for sections of address bus I/O drivers. Must be tied to all other chip ground pins externally. User must provide adequate external decoupling capacitors.

Data Bus VCC(4) (Power) - isolated power for sections of data bus I/O drivers. Must be tied to all other chip power pins externally. User must provide adequate external decoupling capacitors.

Data Bus VSS(8) (Ground) - isolated ground for sections of data bus I/O drivers. Must be tied to all other chip ground pins externally. User must provide adequate external decoupling capacitors.

Bus Control VCC(2) (Power) - isolated power for the bus control I/O drivers. Must be tied to all other chip power pins externally. User must provide adequate external decoupling capacitors.

Bus Control VSS(4) (Ground) - isolated ground for the bus control I/O drivers. Must be tied to all other chip ground pins externally. User must provide adequate external decoupling capacitors.

2.1.4 On-chip Emulator Interface (OnCE) (4 Pins)

$\overline{DR}$ (Debug Request) - The debug enable input provides a means of entering the debug mode of operation from the external command controller. This pin when asserted causes the DSP96002 to finish the current instruction being executed, save the instruction pipeline information, enter the debug mode and wait for commands to be entered from the debug serial input line.

DSCK/OS1 (Debug Serial Clock/Chip Status 1) - The DSCK/OS1 pin, when configured as an input, is the pin through which the serial clock is supplied to the OnCE. The serial clock provides pulses required to shift data into and out of the OnCE serial port. When output (not in Debug Mode), this pin in conjunction with the OS0 pin, provides information about the chip status.

DSI/OS0 (Debug Serial Input/Chip Status 0) - The DSI/OS0 pin, when configured as an input, is the pin through which serial data or commands are provided to the OnCE controller. The data received on the DSI pin will be recognized only when the DSP 96002 has entered the debug mode of operation. When configured as an output (not in Debug Mode), this pin in conjunction with the OS1 pin, provides information about the chip status.

DSO (Debug Serial Output) The debug serial output provides the data contained in one of the OnCE controller registers as specified by the last command received from the external command controller. When a trace or breakpoint occurs this line will be asserted for one T cycle to indicate that the chip has entered the debug mode and is waiting for commands.

2.1.5 Port A and Port B (162 Pins)

Port A and Port B are identical in pinout and function. The following pin descriptions apply to both ports. Each port may be a bus master and each port has a host interface which can be accessed on demand.

The pins are specified for a 50 pf load and two external TTL loads. Derating curves will be provided specifying performance up to 250 pf capacitive loads.

A0-A31 (Address Bus) - three-state, active high outputs when a bus master. When not a bus master, A2-A5 are active high inputs, A0-A1 and A6-A31 are three-stated. As inputs, A2-A5 may change asynchronous relative to the input clock (CLK). A2-A5 are host interface address inputs which are used to select the host interface register. When a bus master, A0-A31 specify the address for external program and data memory accesses. If there is no external bus activity, A0-A31 remain at their previous values. When a bus master, the Address Enable ($\overline{AE}$) input acts as an output enable control for A0-A31. When a bus master, A0-A31 are stable whenever the transfer strobe $\overline{TS}$ is asserted

and may change only when $\overline{TS}$ is deasserted. A0-A31 are three-stated during hardware reset.

D0-D31 (Data Bus) - three-state, active high, bidirectional input/outputs when a bus master or not a bus master. The Data Enable ($\overline{DE}$) input acts as an output enable control for D0-D31. As a bus master, the data lines are controlled by the CPU instruction execution or the DMA controller. D0-D31 are also the Host Interface data lines. If there is no external bus activity, D0-D31 are three-stated. D0-D31 are also three-stated during hardware reset.

S1,S0 (Space Select) - three-state, active low outputs when a bus master, three-stated when not a bus master. Timing is the same as the address lines A0-A31. S1 and S0 are three-stated during hardware reset.

These signals can be viewed in different ways, depending on how the external memories are mapped. They support the trend toward splitting memory spaces among ports and mapping multiple memory spaces into the same physical memory locations. Several examples are given in Figure 2-3 . The encoding S1:S0=11 may be used to place external memories in their low power standby mode.

S1	**S0**	**MEMORY SPACE**
1	1	No access
1	0	P access
0	1	X access
0	0	Y access

R/$\overline{W}$ (Read/Write)- three-state, active low output when a bus master, active low input when not a bus master. Bus master timing is the same as the DSP96002 address lines, giving

EXTERNAL MEMORY AND MAPPING	**S1 FUNCTION**	**S0 FUNCTION**
P only	—	$\overline{PS}$
X only	$\overline{DS}$	—
Y only	$\overline{DS}$	—
X and Y mapped as 1 or 2 spaces	$\overline{DS}$	X/$\overline{Y}$
P and X mapped as 2 spaces	$\overline{DS}$	$\overline{PS}$
P and Y mapped as 1 space	$\overline{PS}$/$\overline{DS}$	$\overline{PS}$ and $\overline{DS}$
P, X, and Y mapped as 1 space	$\overline{PS}$/$\overline{DS}$	—

Figure 2-3. Program and Data Memory Select Encoding

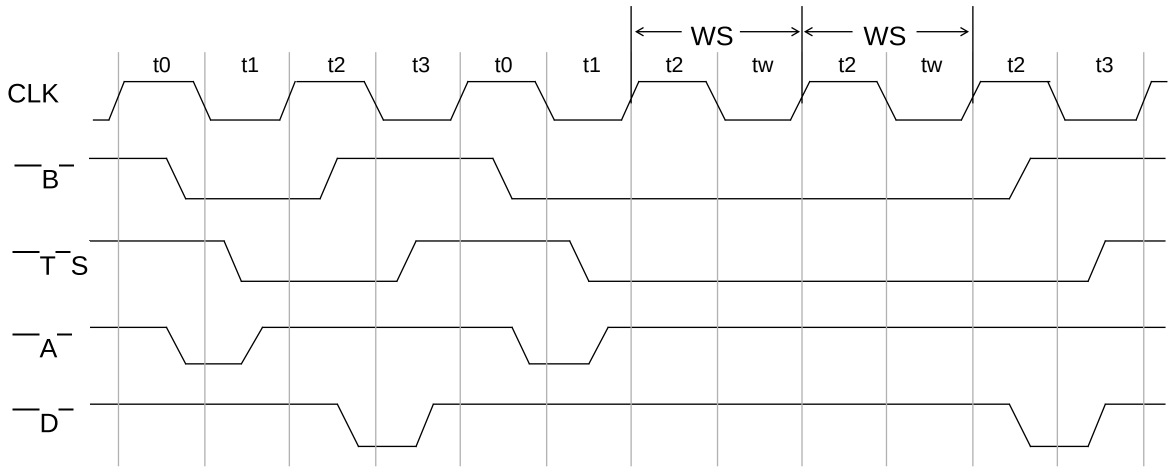
an "early write" signal for DRAM interfacing. R/$\overline{W}$ is high for a read access and is low for a write access. The R/$\overline{W}$ pin is also the Host Interface read/write input. As an input, R/$\overline{W}$ may change asynchronous relative to the input clock. R/$\overline{W}$ goes high if the external bus is not used during an instruction cycle. R/$\overline{W}$ is three-stated during hardware reset.

$\overline{BS}$ (Bus Strobe) - three-state, active low output when a bus master, three-stated when not a bus master. Asserted at the start of a bus cycle (providing an "early bus start" signal for DRAM interfacing) and deasserted at the end of the bus cycle. The early negation provides an "early bus end" signal useful for external bus control. If the external bus is not used during an instruction cycle, $\overline{BS}$ remains deasserted until the next external bus cycle. $\overline{BS}$ is three-stated during hardware reset.

$\overline{TT}$ (Transfer Type) - three-state, active low output when a bus master, three-stated when not a bus master. When a bus master, $\overline{TT}$ is controlled by an on-chip page circuit (see Section seven). $\overline{TT}$ is asserted when a fast access memory mode (page, static column, nibble or serial shift register) is detected. If the external bus is not used during an instruction cycle or a fault is detected by the page circuit during an external access, $\overline{TT}$ remains deasserted. The parameters of the page circuit fault detection are user programmable. $\overline{TT}$ is three-stated during hardware reset.

$\overline{TS}$ (Transfer Strobe) - three-state, active low output when a bus master, active low input when not a bus master. When a bus master, $\overline{TS}$ is asserted to indicate that the address lines A0-A31, S1, S0, $\overline{BS}$, $\overline{BL}$ and R/$\overline{W}$ are stable and that a bus read or bus write transfer is taking place. During a read cycle, input data is latched inside the DSP96002 on the rising edge of $\overline{TS}$. During a write cycle, output data is placed on the data bus after $\overline{TS}$ is asserted. Therefore $\overline{TS}$ can be used as an output enable control for external data bus buffers if they are present. If the external bus is not used during an instruction cycle, $\overline{TS}$ remains deasserted until the next external bus cycle. An external flip-flop can delay $\overline{TS}$ if required for slow devices or more address decoding time. The $\overline{TS}$ pin is also the Host Interface transfer strobe input used to enable the data bus output drivers during host read operations and to latch data inside the Host Interface during host write operations. As an input, $\overline{TS}$ may change asynchronous relative to the input clock. Write data is latched inside the Host Interface on the rising edge of $\overline{TS}$. $\overline{TS}$ is three-stated during hardware reset.

When a bus master, the combination of $\overline{BS}$ and $\overline{TS}$ can be decoded externally to determine the status of the current bus cycle and to generate hardware strobes useful for latching address and data signals. The encoding is shown in Figure 2-4.



$\overline{BS}$	$\overline{TS}$	Bus Status	Strobe Generation Application
1	1	Idle	
0	1	Cycle Start	Address Strobe ($\overline{AS}$)
0	0	Wait	
1	0	Cycle End	Data Strobe ($\overline{DS}$)

Figure 2-4. Bus Status Encoding

$\overline{TA}$ (Transfer Acknowledge) - active low input. If the DSP96002 is the bus master and either there is no external bus activity or the DSP96002 is not the bus master, the $\overline{TA}$ input is ignored by the core. The $\overline{TA}$ input is a synchronous "DTACK" function which can extend an external bus cycle indefinitely. $\overline{TA}$ must be asserted and deasserted synchronous to the input clock (CLK) for proper operation. $\overline{TA}$ is sampled on the falling edge of the input clock (CLK). Any number of wait states (0, 1, 2, ..., infinity) may be inserted by keeping $\overline{TA}$ deasserted. In typical operation, $\overline{TA}$ is deasserted at the start of a bus cycle, is asserted to enable completion of the bus cycle and is deasserted before the next bus cycle. The current bus cycle completes one clock period after $\overline{TA}$ is asserted synchronous to CLK. The number of wait states is determined by the $\overline{TA}$ input or by the Bus Control Register (BCR), whichever is longer. The BCR can be used to set the minimum number of wait states in external bus cycles. If $\overline{TA}$ is tied low (asserted) and no wait states are specified in the BCR register, zero wait states will be inserted into external bus cycles.

$\overline{AE}$ (Address Enable) - active low input, must be asserted and deasserted synchronous to the input clock (CLK) for proper operation. If a bus master, $\overline{AE}$ is asserted to enable the A0-A31 address output drivers. If $\overline{AE}$ is deasserted, the address output drivers are three-stated. If not a bus master, the address output drivers are three-stated regardless of whether $\overline{AE}$ is asserted or deasserted. The function of $\overline{AE}$ is to allow multiplexed bus systems to be implemented. Examples are a multiplexed address/data bus such as the NuBus™ used in the Macintosh II™ or a multiplexed address1/address2 bus used with dual port memories such as dynamic VRAMs. Note that there must be at least one undriven CLK period between enables for multiplexed buses to allow one bus to three-state before another bus is enabled. External control is responsible for this timing. For non-multiplexed systems, $\overline{AE}$ should be tied low.

$\overline{DE}$ (Data Enable) - active low input, must be asserted and deasserted synchronous to the input clock (CLK) for proper operation. If a bus master or the Host interface is being read, $\overline{DE}$ is asserted to enable the D0-D31 data bus output drivers. If $\overline{DE}$ is deasserted, the data bus output drivers are three-stated. If not a bus master, the data bus output drivers are three-stated regardless of whether $\overline{DE}$ is asserted or deasserted. Read-only bus cycles may be performed even though $\overline{DE}$ is deasserted. The function of $\overline{DE}$ is to allow multiplexed bus systems to be implemented. Examples are a multiplexed address/data bus such as the NuBus™ used in the Macintosh II™ or a multiplexed data1/data2 bus used for long word transfers with one 32 bit wide memory. Note that there must be at least one undriven CLK period between enables for multiplexed buses to allow one bus to three-state before another bus is enabled. External control is responsible for this timing. For non-multiplexed systems, $\overline{DE}$ should be tied low.

$\overline{HS}$ (Host Select) - active low input, may change asynchronous to the input clock. $\overline{HS}$ is asserted low to enable selection of the Host Interface functions by the address lines A2-A5. If $\overline{TS}$ is asserted when $\overline{HS}$ is asserted, a data transfer will take place with the Host Interface. Note that both $\overline{HS}$ and $\overline{HA}$ must be tied high to disable the Host Interface. When $\overline{HA}$ is asserted, $\overline{HS}$ is ignored.

$\overline{HA}$ (Host Acknowledge) - active low input, may change asynchronous to the input clock. $\overline{HA}$ is used to acknowledge either an interrupt request or a DMA request to the host interface. When the host interface is not in DMA mode, asserting $\overline{TS}$ when $\overline{HA}$ and $\overline{HR}$ are asserted will enable the contents of the host interface interrupt vector

NuBus is a trademark of Texas Instruments, Inc.
Macintosh II is a trademark of Apple Computer, Inc.

register (IVR) onto the data bus outputs D0-D31. This provides an interrupt acknowledge capability compatible with MC68000 family processors.

If the host interface is in DMA mode, $\overline{HA}$ is used as a DMA transfer acknowledge input and it is asserted by an external device to transfer data between the Host Interface registers and an external device. In DMA read mode, $\overline{HA}$ is asserted to read the Host Interface RX register on the data bus outputs D0-D31. In DMA write mode, $\overline{HA}$ is asserted to strobe external data into the Host Interface TX register. Write data is latched into the TX register on the rising edge of $\overline{HA}$.

$\overline{HR}$ (Host Request) - active low output, never three-stated. The host request $\overline{HR}$ is asserted to indicate that the host interface is requesting service - either an interrupt request or a DMA request - from an external device.

The $\overline{HR}$ output may be connected to interrupt request input $\overline{IRQA}$, $\overline{IRQB}$, or $\overline{IRQC}$ of another DSP96002. The DSP96002 on-chip DMA Controller channel can select the interrupt request input as a DMA transfer request input.

$\overline{BR}$ (Bus Request) - active low output, never three-stated. $\overline{BR}$ is asserted when the CPU or DMA is requesting bus mastership. $\overline{BR}$ is deasserted when the CPU or DMA no longer needs the bus. $\overline{BR}$ may be asserted or deasserted independent of whether the DSP96002 is a bus master or a bus slave. Bus "parking" allows $\overline{BR}$ to be deasserted even though the DSP96002 is the bus master. See the description of bus "parking" in the $\overline{BA}$ pin description. The RH bit in the Bus Control Register (see Section seven) allows $\overline{BR}$ to be asserted under software control even though the CPU or DMA does not need the bus. $\overline{BR}$ is typically sent to an external bus arbitrator which controls the priority, parking and tenure of each DSP96002 on the same external bus. $\overline{BR}$ is only affected by CPU or DMA requests for the external bus, never for the internal bus. During hardware reset, $\overline{BR}$ is deasserted and the arbitration is reset to the bus slave state.

$\overline{BG}$ (Bus Grant) – active low input. $\overline{BG}$ must be asserted/ deasserted synchronous to the input clock (CLK) for proper operation. $\overline{BG}$ is asserted by an external bus arbitration circuit when the DSP96002 may become the next bus master. When $\overline{BG}$ is asserted, the DSP96002 must wait until $\overline{BB}$ is deasserted before taking bus mastership. When $\overline{BG}$ is deasserted, bus mastership is typically given up at the end of the current bus cycle. This may occur in the middle of an instruction which requires more than one external bus cycle for execution. Note that indivisible read-modify-write instructions

(BSET, BCLR, BCHG) will not give up bus mastership until the end of the current instruction. $\overline{BG}$ is ignored during hardware reset.

$\overline{BA}$ (Bus Acknowledge) - Open drain, active low output. When deasserting $\overline{BA}$, the DSP96002 drives $\overline{BA}$ high during half a CLK cycle and then disables the active pull-up. In this way, only a weak external pull-up resistor is required to hold the line high. $\overline{BA}$ may be directly connected to $\overline{BB}$ in order to obtain the same functionality as the MC68040 $\overline{BB}$ pin. When $\overline{BG}$ is asserted, the DSP96002 becomes the pending bus master. It waits until $\overline{BB}$ is negated by the previous bus master, indicating that the previous bus master is off the bus. The pending bus master asserts $\overline{BA}$ to become the current bus master. $\overline{BA}$ is asserted when the CPU or DMA has taken the bus and is the bus master. While $\overline{BA}$ is asserted, the DSP96002 is the owner of the bus (the bus master). When $\overline{BA}$ is negated, the DSP96002 is a bus slave. $\overline{BA}$ may be used as a three-state enable control for external address, data and bus control signal buffers. $\overline{BA}$ is three-stated during hardware reset.

Note that a current bus master may keep $\overline{BA}$ asserted after ceasing bus activity, regardless of whether $\overline{BR}$ is asserted or deasserted. This is called "bus parking" and allows the current bus master to use the bus repeatedly without re-arbitration until some other device wants the bus.

The current bus master keeps $\overline{BA}$ asserted during indivisible read-modify-write bus cycles, regardless of whether $\overline{BG}$ has been deasserted by the external bus arbitration unit. This form of "bus locking" allows the current bus master to perform atomic operations on shared variables in multitasking and multiprocessor systems. Current instructions which perform indivisible read-modify-write bus cycles are BCLR, BCHG and BSET.

$\overline{BB}$ (Bus Busy) - active low input, must be asserted and deasserted synchronous to the input clock (CLK) for proper operation. $\overline{BB}$ is deasserted when there is no bus master on the external bus. In multiple DSP96002 systems, all $\overline{BB}$ inputs are tied together and are driven by the logical AND of all $\overline{BA}$ outputs. $\overline{BB}$ is asserted by a pending bus master (directly or indirectly by $\overline{BA}$ assertion) to indicate that it is now the current bus master. $\overline{BB}$ is deasserted by the current bus master (directly or indirectly by $\overline{BA}$ negation) to indicate that it is off the bus and is no longer the bus master. The pending bus master monitors the $\overline{BB}$ signal until it is deasserted. Then the pending bus master asserts $\overline{BA}$ to become the current bus master, which asserts $\overline{BB}$ directly or indirectly.

$\overline{BL}$ (Bus Lock) - active low output, never three-stated. Asserted at the start of an external indivisible Read-Modify-Write (RMW) bus cycle (providing an "early bus start" signal for DRAM interfacing) and deasserted at the end of the write bus cycle. $\overline{BL}$ remains asserted between the read and write bus cycles of the RMW bus sequence. $\overline{BL}$ can be used to indicate that special memory timing (such as RMW timing for DRAMs) may be used or to "resource lock" an external multi-port memory for secure semaphore updates. The early negation provides an "early bus end" signal useful for external bus control. If the external bus is not used during an instruction cycle, $\overline{BL}$ remains deasserted until the next external indivisible RMW bus cycle. $\overline{BL}$ also remains deasserted if the external bus cycle is not an indivisible RMW bus cycle or if there is an internal RMW bus cycle. The only instructions which automatically assert $\overline{BL}$ are a BSET, BCLR or BCHG instruction which accesses external memory. $\overline{BL}$ can also be asserted by setting the LH bit in the BCR register (see Section seven). $\overline{BL}$ is deasserted during hardware reset.

2.1.6 Reserved Pins

There are 5 spare pins reserved for future use.

2.2 BUS OPERATION

The external bus timing is defined by the operation of the Address Bus, Data Bus and Bus Control pins described in paragraph 2.1.5. The DSP96002 external ports are designed to interface with a wide variety of memory and peripheral devices, high speed static RAMs, dynamic RAMs and video RAMs as well as slower memory devices. External bus timing is controlled by the $\overline{TA}$ control signal and by the Bus Control Registers (BCR) which are described in Section seven. The BCR and $\overline{TA}$ control the timing of the bus interface signals. Insertion of wait states is controlled by the BCR to provide constant bus access timing, and by $\overline{TA}$ to provide dynamic bus access timing. The number of wait states is determined by the $\overline{TA}$ input or by the BCR, whichever is longer.

2.2.1 Synchronous Bus Operation

Synchronous external bus cycle consists of at least 4 internal clock phases. See the DSP96002 Technical Data Sheet (DSP96002/D) for the specification of the internal clock phases. Each synchronous external memory access requires the following procedure:

3:3. The external memory address is defined by the Address Bus A0-A31 and the Memory Reference Select signals S1 and S0. These signals change in the first phase of the external bus cycle. The Memory Reference Select signals have the same timing as the Address Bus and may be used as additional address lines. The Address and Memory Reference signals are also used to generate chip select signals for the appropriate memory chips. These chip select signals change the memory chips from low power standby mode to active mode and begin the read access time. This allows slower memories to be used since the chip select signals are address-based rather than read or write enable-based.

3:4. When the Address and Memory Reference signals are stable, the data transfer is enabled by the Transfer Strobe $\overline{TS}$ signal. $\overline{TS}$ is asserted to "qualify" the Address and Memory Reference signals as stable and to perform the read or write data transfer. $\overline{TS}$ is asserted in the second phase of the bus cycle.

3:5. Wait states are inserted into the bus cycle controlled by a wait state counter or by $\overline{TA}$, whichever is longer. The wait state counter is loaded from the Bus Control Register. If the wait state number determined by these two factors is zero, no wait states are inserted into the bus cycle and $\overline{TS}$ is deasserted in the fourth phase. If the wait state number determined is W, then W wait states are inserted into the instruction cycle. Each wait state introduces one Tc delay.

3:6. When the Transfer Strobe $\overline{TS}$ is deasserted at the end of a bus cycle, the data is latched in the destination device. At the end of a read cycle, the DSP96002 latches the data internally. At the end of a write cycle, the external memory latches the data. The Address signals remain stable until the first phase of the next external bus cycle to minimize power dissipation. The Memory Reference signals S1 and S0 are deasserted during periods of no bus activity and the data signals are three-stated.

3.6.1 Static RAM Support

Static RAM devices can be easily interfaced to the DSP96002 bus timing. There are two basic techniques - $\overline{CS}$ controlled writes and $\overline{WE}$ controlled writes.

3. 6.1.1 $\overline{CS}$ Controlled Writes

This form of static interface uses the memory chip select ($\overline{CS}$) as the write strobe. The DSP96002 R/$\overline{W}$ signal is used as an early read/write direction indication. Proper data buffer enable control on RAMs without a separate output enable ($\overline{OE}$) input must use this form to avoid multiple data buffers colliding on the data bus. The interface schematic is shown in Figure 2-5.

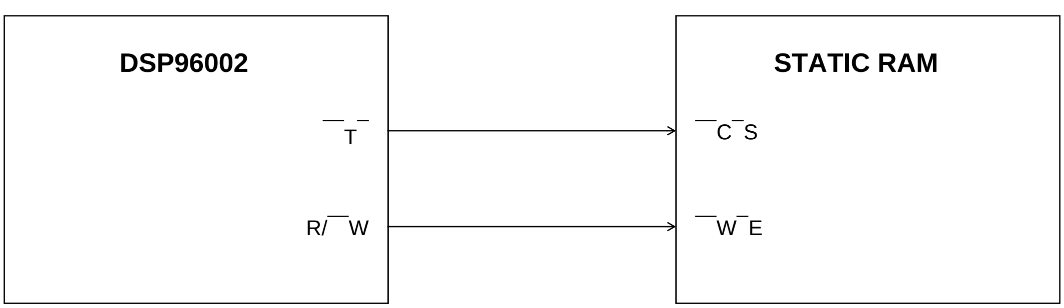


Figure 2-5. $\overline{CS}$ Controlled Writes Interface to Static RAM

The disadvantage of this technique is that access time is measured from $\overline{TS}$ instead of from the address or $\overline{BS}$. Hence faster memories are required.

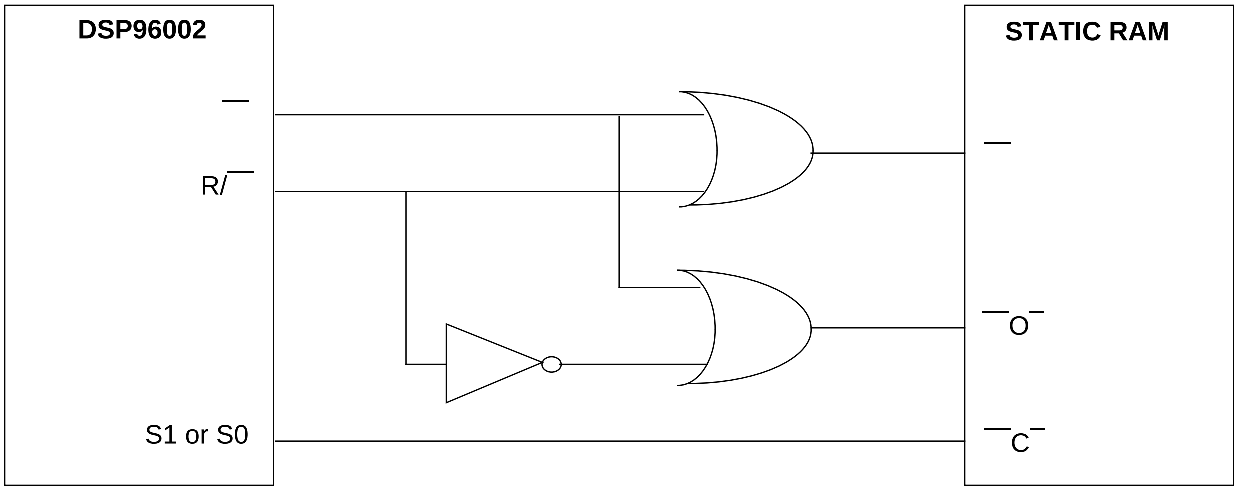


Figure 2-6. $\overline{WE}$ Controlled Writes Interface To Static RAM

3. 6.1.2 $\overline{WE}$ Controlled Writes

This form of static interface uses the memory write enable ($\overline{WE}$) as the write strobe. The DSP96002 R/$\overline{W}$ signal is used to form a late read/write indication by gating it with $\overline{TS}$. This form is the one used by the 56000/1 bus interface. Proper data buffer enable control requires a separate output enable ($\overline{OE}$) input on the memory to avoid multiple data buffers colliding on the data bus. The interface schematic is shown in Figure 2-6.

The advantage of this technique is that access time is measured from S1, S0 or addresses instead of $\overline{TS}$. Hence slower memories can be used. The disadvantage of this technique is that the write data hold will be shortened because the $\overline{WE}$ signal is delayed by the OR gate.

3.6.2 Dynamic RAM and Video RAM Support

Modern dynamic memory (DRAM) and video memory (VRAM) are becoming the preferred choice for a wide variety of computing systems based on

4:7. Cost per bit due to dynamic storage cell density.

4:8. Packaging density due to multiplexed address and control pins.

4:9. Improved performance relative to static RAMs due to fast access modes (page, static column, nibble and serial shift (VRAM)).

4:10. Commodity pricing due to high volume production.

The Port A/B bus control signals are designed for efficient interface to DRAM/VRAM devices in both random read/write cycles and fast access modes such as those listed above. The bus control signal timing is specified relative to the external clock (CLK) to enable synchronous control by an external state machine. An on-chip page circuit controls the $\overline{TT}$ pin, indicating to the external state machine when a slow or fast access is being made. The page circuit operation and programming is described in Section seven.

4.11 BUS HANDSHAKE AND ARBITRATION

Bus transactions are governed by a single bus master. Bus arbitration determines which device becomes the bus master. The arbitration logic implementation is system dependent, but must result in at most one device becoming the bus master (even if multiple devices request bus ownership). The arbitration signals permit simple implementation of a variety of bus arbitration schemes (e.g. fairness, priority, etc.). External logic must be provided by the system designer to implement the arbitration scheme.

4.11.1 Bus Arbitration Signals

Four signals are provided for bus arbitration. Three of them are considered as local arbitration signals and one as system arbitration signal. The local arbitration signals run between a potential bus master and the arbitration logic. The local signals are $\overline{BR}$, $\overline{BG}$, and $\overline{BA}$; $\overline{BB}$ is a system arbitration signal. These signals are described below.

$\overline{BR}$ — Bus Request - Asserted by the requesting device to indicate that it wants to use the bus, and is held asserted until it no longer needs the bus. This includes time when it is the bus master as well as when it is not the bus master.

$\overline{BG}$ — Bus Grant - Asserted by the bus arbitration controller to signal the requesting device that it is the bus master elect. $\overline{BG}$ is valid only when the bus is not busy (Bus Busy signal described below).

$\overline{BA}$ — Bus Acknowledge - Asserted by the device (bus master) that received the bus ownership from the bus arbitration controller. The master holds $\overline{BA}$ asserted for the duration of its bus possession. $\overline{BA}$ indicates whether the device is a bus master or a bus slave. When asserted, $\overline{BA}$ indicates that the device is the bus master. $\overline{BA}$ may be used as a three-state enable control for external address, data and bus control signal buffers.

$\overline{BB}$ — Bus Busy - The system arbitration signal $\overline{BB}$ is monitored by all potential bus masters and is derived from the local bus signal $\overline{BA}$. This signal controls the hand-over of bus ownership by the bus master at the end of bus possession. Typically $\overline{BB}$ is the wired-OR of all bus acknowledgments. $\overline{BB}$ is asserted if the Bus Acknowledge signal is asserted by the bus master.

4.11.2 The Arbitration Protocol

The bus is arbitrated by a central bus arbitrator, using individual request/grant lines to each bus master. The arbitration protocol can operate in parallel with bus transfer activity so that the bus hand-over can be made without much performance penalty.

The arbitration sequence occurs as follows:

5:12. All candidates for bus ownership assert their respective $\overline{BR}$ signals as soon as they need the bus.

5:13. The arbitration logic designates a bus master-elect by asserting the $\overline{BG}$ signal for that device.

5:14. The master-elect tests $\overline{BB}$ to ensure that the previous master has relinquished the bus. If $\overline{BB}$ is deasserted, then the master-elect asserts $\overline{BA}$, which designates the device as the new bus master. If a higher priority bus request occurs before the $\overline{BB}$ signal was deasserted, then the arbitration logic may replace the current master-elect with the higher priority candidate. However, only one $\overline{BG}$ signal must be asserted at one time.

5:15. The new bus master begins its bus transfers after the assertion of $\overline{BA}$.

5:16. The arbitration logic signals the current bus master to relinquish the bus by deasserting $\overline{BG}$ at any time. A DSP96002 bus master releases its ownership (deasserts $\overline{BA}$) after completing the current external bus access. If an instruction is executing a Read-Modify-Write external access, a DSP96002 master asserts the $\overline{BL}$ signal and will only relinquish the bus (and deassert $\overline{BL}$) after completing the entire Read-Modify-Write sequence. When the current bus master deasserts $\overline{BA}$, the $\overline{BB}$ signal must also be deasserted because the next bus master-elect has received its $\overline{BG}$ signal and is waiting for $\overline{BB}$ to be deasserted before claiming ownership.

The DSP96002 has 2 control bits and one status bit, located in the Bus Control Registers (see Section 7) to permit software control of the $\overline{BR}$ and $\overline{BL}$ signals, and to verify when the chip is the bus master. If the RH bit in the BCR register is cleared, the DSP96002 asserts its $\overline{BR}$ signal only as long as requests for bus transfers are pending or being attempted. If the RH bit is set, $\overline{BR}$ will remain asserted. If the LH bit in the BCR register is cleared, the DSP96002 asserts its $\overline{BL}$ signal only during a read-modify-write bus access. If the LH bit is set, $\overline{BL}$ will remain asserted.

5.16.1 Arbitration Scheme

The bus arbitration scheme is implementation dependent. The diagram in Figure 2-7 illustrates a common method of implementing the bus arbitration scheme. The arbitration logic determines the device priorities and assigns bus ownership depending on those priorities.

An implementation of a bus arbitration scheme may hold $\overline{BG}$ asserted, for example, to the current bus owner if none of the other devices are requesting the bus. As a consequence, the current bus master may keep $\overline{BA}$ asserted after ceasing bus activity, regardless of whether $\overline{BR}$ is asserted or deasserted. This situation is called "bus parking" and allows the current bus master to use the bus repeatedly without re-arbitration until some other device requests the bus.

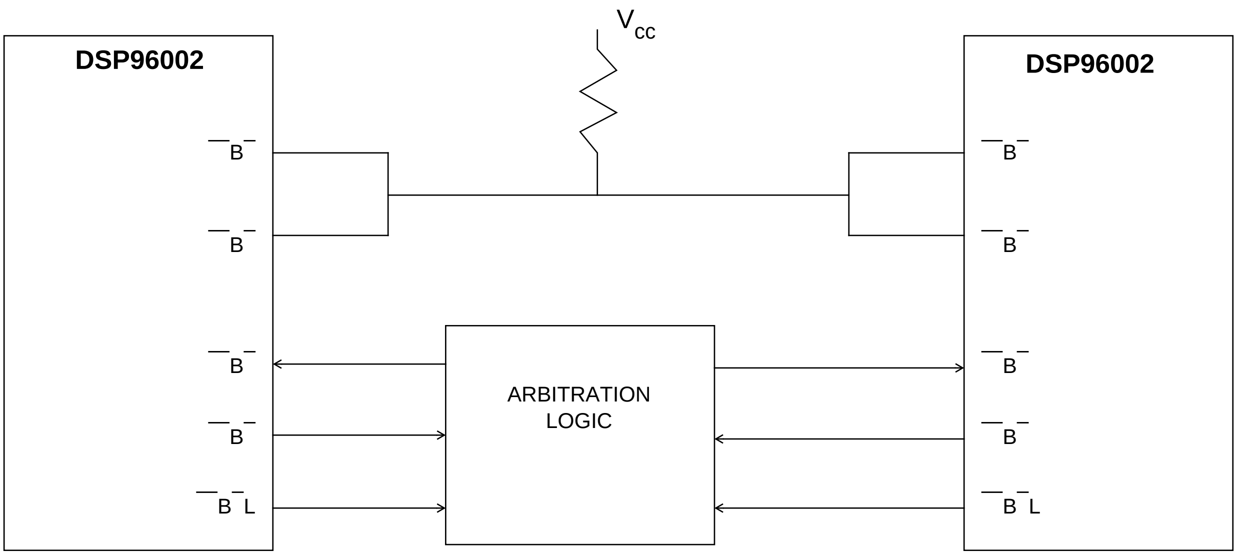


Figure 2-7. Bus Arbitration Scheme

5.16.2 Bus Handshake Unit

The bus handshake unit in the DSP96002 is implemented within a finite state machine. It consists of two external outputs ($\overline{BR}$, $\overline{BA}$), two external inputs ($\overline{BG}$, $\overline{BB}$) and three internal inputs (ext_acc_req, end_of_sequence, RH) (see Figure 2-8). The ext_acc_req signal is asserted when one or more requests for external bus access are pending, and remains asserted as long as the transfers are being executed. The end_of_sequence signal is asserted at the last bus cycle of the current sequence.

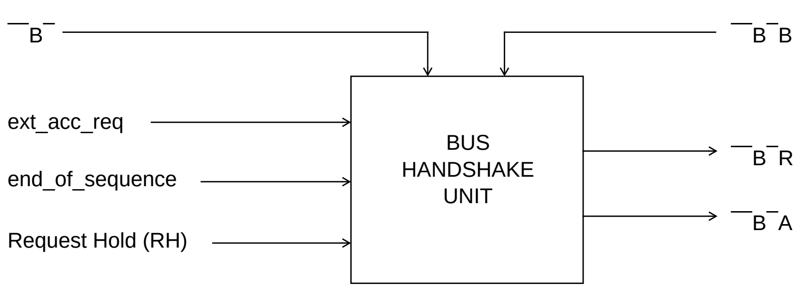


Figure 2-8. Bus Handshake Unit

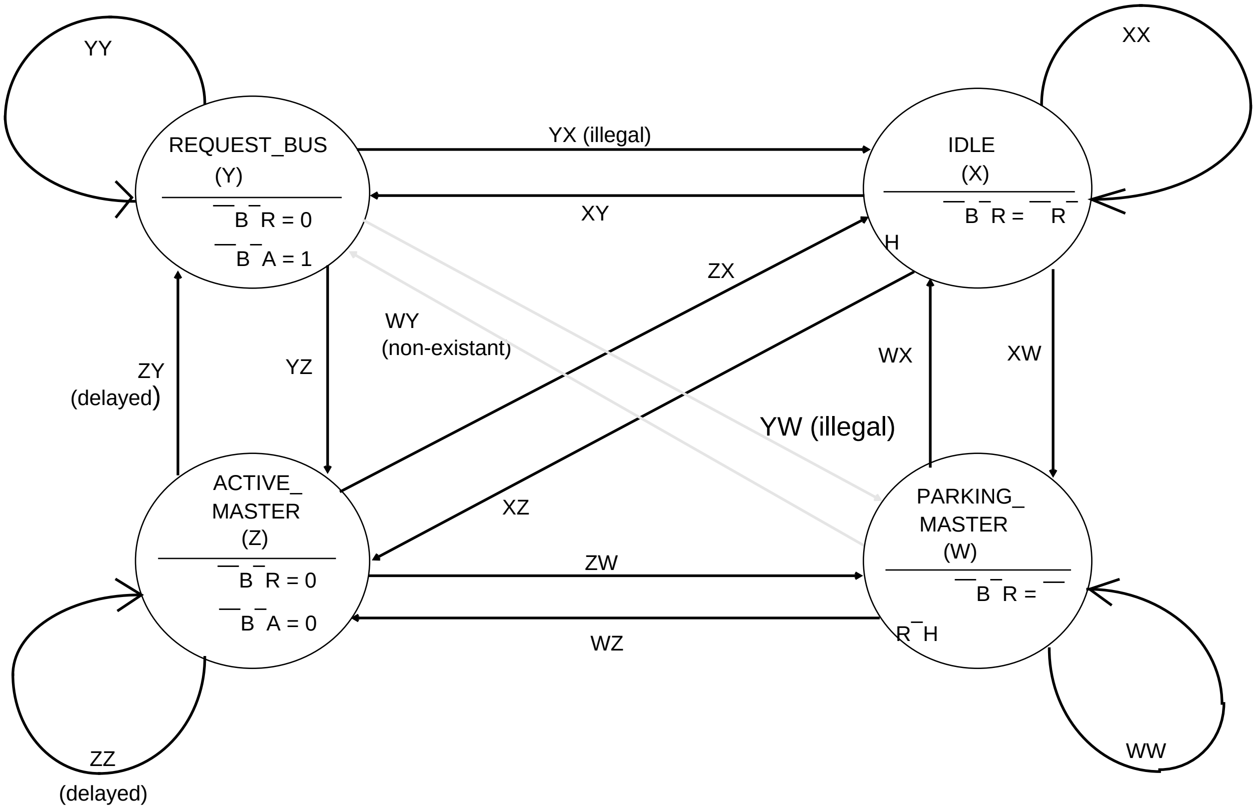


Figure 2-9. Bus Handshake State Diagram

Likewise, when executing the read part of a RMW access, the end_of_sequence signal is deasserted. This signal is used to give up bus ownership if $\overline{BG}$ is deasserted during bus transfers. The state machine which controls the bus handshake is illustrated in Figure 2.9.

The transition arcs are labeled by two letters which denote its source and destination states. The equations of the transition arcs are described as follows:

XX = ^ext_acc_req & ^(^$\overline{BG}$ & $\overline{BB}$)
XY = ext_acc_req & ^(^$\overline{BG}$ & $\overline{BB}$)
XZ = ext_acc_req & (^$\overline{BG}$ & $\overline{BB}$)
XW = ^ext_acc_req & (^$\overline{BG}$ & $\overline{BB}$)

YX = ^ext_acc_req & ^(^$\overline{BG}$ & $\overline{BB}$) (note 1)
YY = ext_acc_req & ^(^$\overline{BG}$ & $\overline{BB}$)
YZ = ext_acc_req & (^$\overline{BG}$ & $\overline{BB}$)
YW = ^ext_acc_req & (^$\overline{BG}$ & $\overline{BB}$) (note 1)

ZX = ^ext_acc_req & $\overline{BG}$
ZY = ext_acc_req & $\overline{D}$ $\overline{BG}$ & end_of_sequence (note 3)

ZZ = ^end_of_sequence v (ext_acc_req & ^$\overline{DBG}$) (note 3)

ZW = ^ext_acc_req & ^$\overline{BG}$

WX = ^ext_acc_req & $\overline{BG}$
WY = NON-EXISTENT ARC (note 2)
WZ = ext_acc_req

WW = ^ext_acc_req & ^$\overline{BG}$

Notes:
1. Illegal arcs in DSP96002 since once the request of the bus is pending, it will not be canceled before the execution of the access.
2. Non-existent arc since if ext_acc_req arrives together with the negation of $\overline{BG}$, the device becomes active master and begins its bus transfers.
3. $\overline{DBG}$ is $\overline{BG}$ delayed by one phase. This is done to provide a response to the ext_acc_req signal when it is asserted at the same phase together with $\overline{BG}$ negation.

5.16.3 Bus Arbitration Example Cases

5.16.3.1 Case 1 – Normal

If the device requesting mastership asserts $\overline{BR}$: the arbiter asserts the requesting devices' $\overline{BG}$ and $\overline{BB}$ is deasserted indicating the bus is not busy. The requesting device will assert $\overline{BA}$.

5.16.3.2 Case 2 – Bus Busy

If the device requesting mastership asserts $\overline{BR}$: the arbiter responds by asserting the requesting devices' $\overline{BG}$; however, the bus is busy because $\overline{BB}$ is asserted. The requesting device will not assert $\overline{BA}$ until $\overline{BB}$ is deasserted.

5.16.3.3 Case 3 – Low Priority

If the device requesting mastership asserts $\overline{BR}$: the arbiter withholds asserting the requesting devices' $\overline{BG}$ because a higher priority device requested the bus. $\overline{BA}$ of the requesting device will not be asserted.

5.16.3.4 Case 4 – Default

If a device does not request the bus and it is not in the bus parking state but rather it is in the idle state: the arbiter, by design (i. e., default), asserts $\overline{BG}$. $\overline{BA}$ will remain deasserted.

5.16.3.5 Case 5 – Bus Lock during RMW

If the device requesting mastership asserts $\overline{BR}$ and the arbiter asserts the requesting devices' $\overline{BG}$ and $\overline{BB}$ is deasserted, then the requesting device will assert $\overline{BA}$. If a read-modify-write (RMW) instruction which accesses external memory is being executed, and the bus arbiter deasserts $\overline{BG}$, then $\overline{BA}$ will remain asserted until the entire RMW instruction completes execution. $\overline{BA}$ will then be deasserted thereby relinquishing the bus. Note that during external RMW instruction execution, $\overline{BL}$ is asserted. In general, the $\overline{BL}$ signal can be used to ensure that a multiport memory can only be written by one master at a time. That is, referring to Figure 2-10, $\overline{BL}$ can be input from DSP #1to the memory controller which prevents $\overline{TA}$ from being asserted by the controller (thereby suspending the memory access by DSP #2) until DSP #1 completes its RMW access.

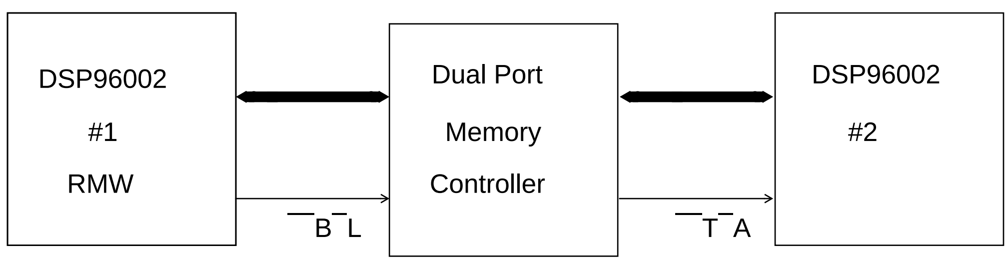


Figure 2-10. Bus Lock During RMW

5.16.3.6 Case 6 – Bus Park

The device requesting mastership asserts $\overline{BR}$; the arbiter asserts the requesting devices' $\overline{BG}$ and $\overline{BB}$ is deasserted indicating the bus is not busy – the requesting device will assert $\overline{BA}$. When the requesting device no longer requires the bus it will deassert $\overline{BR}$; if the bus arbiter leaves $\overline{BG}$ asserted because other requests are not pending, then $\overline{BA}$ will remain asserted. This condition is called bus parking and eliminates the need for the last bus master to rearbitrate for the bus during its next external access.

SECTION 3
CHIP ARCHITECTURE

3.1 INTRODUCTION

The DSP96002 architecture is a 32-bit highly-parallel multiple-bus IEEE floating-point processor. The architecture is designed to accommodate various IC family members with different memory and on-chip peripheral requirements while maintaining a standard programmable core. The overall chip architecture is presented and detailed block diagrams of the Data ALU and Address Generation Unit AGU) core architecture are described.

3.2 DSP96002 BLOCK DIAGRAM

The major components of the DSP96002 are

- Data Buses
- Address Buses
- Data ALU
- Address Generation Unit
- X Data Memory
- Y Data Memory
- Program Control and System Stack
- Program Memory
- Port A and Port B External Bus Interfaces
- Internal Bus Switch and Bit Manipulation Unit
- I/O Interfaces

An overall block diagram of the DSP96002 architecture is shown in Figure 3-1.

3.2.1 Data Buses

Data movement on the chip occurs over five bidirectional 32-bit buses, X Data Bus (XDB), Y Data Bus (YDB), Global Data Bus (GDB), the DMA Data Bus (DDB) and the Program Data Bus (PDB). The X and Y data buses may also be treated by certain instructions as one 64-bit data bus by concatenation of XDB and YDB. Data transfer between the Data ALU and the X Data Memory and Y Data Memory occur over the X Data Bus and Y Data Bus. These are kept local on the chip to maximize speed and minimize power. The direct memory access data transfers occur over the DMA Data Bus. Program memory data transfers and instruction fetches occur over the Program Data Bus. All other data transfers occur over the Global Data Bus.

Figure 3-1. DSP96002 Block Diagram

3.2.2 Address Buses

Addresses are specified for internal X Data Memory and Y Data Memory on two unidirectional 32-bit buses, X Address Bus (XAB) and Y Address Bus (YAB). Internal address bus sizes depend on the amount of internal memory implemented. External memory spaces for each port, A and B, are addressed via a single 32-bit unidirectional address bus driven by a three input multiplexer that can select the X Address Bus (XAB), the Y Address Bus (YAB) or the Program Address Bus (PAB). On-chip peripherals and the DMA Controller are memory mapped in the internal X memory space. When zero wait state external memory is used, one instruction cycle is needed for each external memory access.

The XAB, YAB and PAB are dual access buses in the sense that one instruction cycle contains two slots, the one slot is dedicated to the on-chip DMA transfers and the second is used for the core transfers.

3.2.3 Data ALU

The Data ALU performs all of the arithmetic and logical operations on data operands. The Data ALU consists of ten 96-bit general purpose registers, a 32-bit barrel shifter, a 32-bit adder, and a 32-bit parallel multiplier. Data ALU registers may be read or written over the XDB and YDB as 32 or 64-bit operands. The Data ALU is capable of multiplication, addition, subtraction, format conversion, shifting and logical operations in one instruction cycle. Data ALU source operands may be 32 or 96-bits and originate from the general purpose register file. Data ALU results are always stored in one of the general purpose registers. Floating-point Data ALU operations always have a 96-bit result. Integer (fixed-point) Data ALU operations have a 32 or 64-bit result.

The Data ALU fully implements the IEEE Standard 754 for binary floating-point arithmetic. The operations are supported in three data formats: 32-bit two's-complement fixed-point, 32-bit unsigned-magnitude fixed-point and 44-bit IEEE single extended precision floating-point. All the floating-point computations are performed using the single extended precision format and the results are automatically rounded to single precision or single extended precision numbers as programmed. All four IEEE rounding modes (round to zero, round to nearest, round to plus infinity and round to minus infinity) are supported for all floating-point operations and conversions. The IEEE gradual underflow with denormalized numbers is supported by the IEEE mode. In the IEEE mode, if input operand(s) or output result(s) are denormalized numbers, additional instruction cycles are required to process these numbers per the IEEE standard. A "Flush to Zero" mode is also provided which forces all floating point result underflows to zero (all denormalized input operands are considered as being zero). The Flush to Zero mode never requires any additional instruction cycles.

Refer to Section 3.3 for a detailed description of the Data ALU architecture.

3.2.4 AGU

The AGU performs all of the address storage and effective address calculations necessary to address data operands in memory and it is used by both the core and the on-chip DMA Controller. The AGU operates in parallel with other chip resources to minimize address generation overhead. The AGU contains eight Address Registers (R0-R7), eight Offset Registers (N0-N7), and eight Modifier Registers (M0-M7). The Address Registers are 32-bit registers which may contain any address or data. Each Address Register may be accessed for output to the XAB, YAB, and PAB. The modifier and offset registers are 32-bit registers which are normally used to control updating of the address registers.

AGU registers may be read or written over the Global Data Bus as 32-bit operands. The AGU can generate two 32-bit addresses every instruction cycle - one for any two of the XAB, YAB or PAB. The AGU can directly address 4,294,967,296 locations on the XAB and 4,294,967,296 locations on the YAB - a total capability of 8,589,934,592 32-bit data words. Refer to Section 3.4 for a detailed description of the AGU architecture.

3.2.5 X Data Memory

The X Data Memory may contain both data RAM and ROM. The X Data RAM is a 32-bit wide internal memory and occupies the lowest 512 locations in X Memory Space. The X Data ROM is also a 32-bit wide internal memory and occupies 1024 locations in X Memory Space. Addresses are received from the XAB and data transfers occur on the XDB. The X memory is a dual-access memory in the sense that it may be accessed twice during a cycle: once by the core and once by the DMA. X memory may be expanded off chip.

3.2.6 Y Data Memory

The Y Data Memory may contain both data RAM and ROM. The Y Data RAM is a 32-bit wide internal memory and occupies the lowest 512 locations in Y Memory Space. The Y Data ROM is also a 32-bit wide internal memory and occupies 1024 locations in Y Memory Space. Addresses are received from the YAB and data transfers occur on the YDB. The Y memory is dual-access memory in the sense that it may be accessed twice during a cycle: once by the core and once by the DMA. Y memory may be expanded off chip.

3.2.7 Program Control and System Stack

The Program Control logic performs instruction prefetch, instruction decoding and exception processing. A 32-bit program counter (PC) register can address 4,294,967,296 locations in Program Memory Space.

The System Stack is a separate internal RAM which stores the PC and the status register (SR) for subroutine calls and long interrupts. The stack will also store the loop counter (LC) and the loop address register (LA) in addition to the PC and SR registers for program looping. The System Stack is in Stack Memory Space and its address is always inherent and implied by the current instruction. The stack RAM is 64-bits wide and 15 locations "deep". When a subroutine call or long interrupt occurs, the contents of the PC and SR registers are stored (pushed) on the "top" location in the System Stack. When a return from subroutine occurs, the contents of the "top" location in the System Stack are copied (pulled) to the PC. When a return from interrupt occurs, the contents of the "top" location in the System Stack are copied (pulled) to the PC and SR.

An interrupt will cause the processor to enter the exception processing state. Upon entering this state, the current instruction in decode will execute normally, unless it is the first word of a two-word instruction, in which case it will be aborted, and re-fetched at the completion of exception processing. The next two fetch addresses are supplied by the interrupt controller. During these fetches the PC is not updated.

If one of the words fetched by the interrupt controller is a jump to subroutine, a long interrupt routine is formed, and a context switch is performed using the stack. If neither interrupt instruction word causes a change of control flow, then the two interrupt instructions fetched constitute a fast interrupt routine. In this case, the stack is not used, and interrupt service concludes with the execution of the instructions contained within the two words. Fetching then resumes using the PC. The fast interrupt routine provides minimum overhead exception processing. This mechanism is commonly used to move data between memory and an I/O device.

For more details on the behavior of interrupts, see Section 8.

The system stack is also used to implement no-overhead hardware program loops. When a program loop is initiated with the execution of a DO instruction, the following events occur:

- the current 32-bit loop counter (LC) and 32-bit loop address register (LA) are pushed onto the system stack to allow nested loops.
- the LC and LA registers are initialized with values specified in the DO instruction.
- the address of the first instruction in the program loop and the current status register contents are transferred onto the system stack.
- the loop flag bit in the status register is set.

The loop flag bit is set when a program loop is in progress and enables the end of loop detection (comparison between the PC and LA registers, discussed below). The loop flag bit is pulled from the system stack when a loop is terminated and indicates if the terminated loop was a nested loop.

A program loop begins execution after the DO instruction and continues until the program address fetched equals the loop address register contents (last address of program loop). The contents of the loop counter are then tested for one. If the loop counter is not one, the loop counter is decremented and the top location in the stack RAM is read (but not pulled) into the PC to return to the start of the loop. If the loop counter is one, the program loop is terminated by incrementing the PC, reading the previous loop flag bit from the top location in the stack into the status register, purging the stack (pulling the top location and discarding the contents) and pulling the LA and LC registers off the stack and restoring the respective registers. When terminating a loop the loop flag, LA and LC registers as well as the system stack pointer are restored.

3.2.8 Program Memory

The Program Memory consists of a 1,024 location by 32-bit RAM. Addresses are received from the program control logic (usually the PC). The Program Memory may contain instructions, constants, and data tables which are fixed at assembly time. The Program Memory is a dual-access memory in the sense that it may be accessed twice during a cycle: once by the core and once by the DMA. Program Memory may be expanded off-chip. Program RAM may be written to download instructions. The bootstrap ROM also appears in Program Memory space during the bootstrap mode. See Section 9.

3.2.9 External Bus Interfaces

The DSP96002 has two identical external bus interfaces. Each bus interface has a 32-bit wide address bus and a 32-bit wide data bus, and may be used to access external Data Memory, Program Memory or I/O devices. Separate select lines control access to the memory spaces. A Port Select control register permits assigning sections of each memory space to each external bus interface port. Refer to Section 2 and Section 9 for a detailed description of the external bus interface.

3.2.10 Internal Bus Switch and Bit Manipulation Unit

The Internal Bus Switch performs data transfers from one internal bus to another.

The Bit Manipulation Unit performs bit manipulation operations on memory and register operands on the XDB, YDB, and GDB.

3.2.11 I/O Interfaces

The on-chip I/O interfaces are intended to minimize system chip count and "glue" logic in many DSP96002 applications. Each I/O interface has its own control, status and data registers and is treated as memory-mapped I/O by the DSP96002. Each interface has several dedicated interrupt vector addresses and control bits to enable/disable interrupts. This minimizes the overhead associated with servicing the device since each interrupt source has its own service routine.

The DSP96002 provides the following I/O interfaces: two identical 32-bit parallel Host MPU/DMA Interface peripherals are provided on the DSP96002, one connected to External Bus Interface A and the other to External Bus Interface B; a two-channel DMA Controller.

3.2.11.1 Host Interfaces

The DSP96002 provides a Host MPU/DMA Interface for each of its external bus interface ports. Each Host Interface (HI) is a 8-, 16-, 24- or 32-bit wide parallel port which may be connected directly to the data bus of a host processor. The host processor may be any of a number of popular microcomputers or micropro-

cessors, another DSP96002 or DMA hardware. The HI appears as a memory mapped peripheral occupying 16 words in the host processor address space. Separate transmit and receive data registers are double-buffered to allow the DSP96002 and host processor to efficiently transfer data at high speed. Host processor communication with the HI is accomplished using standard Host processor data move instructions and addressing modes. Handshake flags are provided for polled or interrupt-driven data transfers.

3.2.11.2 DMA Controller

The DMA Controller performs all the address storage and effective address calculations necessary to address the DMA source and destination operands. The DMA controller operates in parallel with other chip resources to minimize data or program transfers overhead. The DMA controller contains one Source Address Register, one Source Offset Register, one Source Modifier Register, one Destination Address Register, one Destination Offset Register and one Destination Modifier Register for each channel.

In addition there are two control registers per channel. The Transfer Count down counter, decremented after each transfer, contains the number of DMA transfers remaining to be done. The DMA Control/Status Register controls the DMA activities and contains the DMA status. All DMA registers are mapped into the X memory space. The AGU is shared by the DMA for the source and destination address calculations. The DMA addressing modes are: linear, bit reversed and modulo. For more details see Section 7.5.

3.3 DATA ALU BLOCK DIAGRAM

The major components of the Data ALU are

- Data ALU Register File
- Multiply Unit
- Adder Unit
- Logic Unit
- Format Converter
- Divide and Square Root Unit
- Controller and Arbitrator

A block diagram of the Data ALU architecture is shown in Figure 3-2.

D0, D1, D2, D3, D4, D5, D6, D7, D8 and D9 are 96-bit registers which serve as the Data ALU general purpose register file. Every register is divided into three portions: high, middle, and low, each 32-bits wide. The registers may be treated as ten 96-bit registers Dn (Dn.H:Dn.M:Dn.L), n=0,1,..,9 for floating-point source and/or destination operands. These floating point registers receive inputs from the Multiplier, the Adder, and the Subtracter and supply a source data register of the same form. Most Data ALU floating-point operations specify the 96-bit registers as source and/or destination operands. However, D8 and D9 are never destinations of a Data ALU operation.

The data is stored in the registers in double precision floating-point format. Each register may be read or written over the XDB or YDB as a floating-point operand. A format conversion is automatically performed when a Dn register is written with an operand of a different floating-point format. This can occur when writing Dn from the XDB or YDB as a result of a single precision floating-point MOVE. If a single precision operand is written to a floating point data register, the middle portion of the data register is written with the mantissa portion of the word operand, the low portion is zeroed and the high portion is written with the exponent portion of the word operand.

Figure 3-2. Data ALU Block Diagram Data ALU Register File (D0-D9)

The registers may also be treated as thirty 32-bit registers Dn.H, Dn.M, Dn.L, n=0,1,..,9. Each register may be read or written over the XDB or YDB as a word operand. When an individual 32-bit register is written over the XDB or YDB, no format conversion takes place and only the designated register is affected. The low portion of the registers, Dn.L, is used as source and/or destination for most integer operations. In this case the integer registers supply an operand for the Multiplier and the Adder/Subtracter while receiving an input from the Multiplier and the Adder/subtracter. Note that in the case of integer multiplication the result will be 64-bits wide and will be stored in both middle and low portions of the destination register.

3.3.1 Multiply Unit

The Multiplier is one of the two arithmetic processing units of the Data ALU and performs all the floating-point multiplications as well as signed/unsigned fixed-point (integer) multiplications on the data operands.

For the floating-point multiplication the Multiplier accepts two 44-bit input operands, and outputs one 44-bit result. The operation of the floating-point Multiplier occurs independently and in parallel with the operation of the floating-point Adder and with the XDB and YDB activity. For the fixed-point multiplication the Multiplier accepts two 32-bit input operands, and outputs one 64-bit result. The operation of the fixed point Multiplier occurs independently and in parallel with the XDB and YDB activity. The Data ALU registers can be used by the programmer to implement Data ALU pipelines.

The Multiplier is implemented in asynchronous logic and all multiplication operations occur in one instruction cycle. Latches are provided on the Multiplier input operand buses to avoid race conditions. The major components of the Multiply Unit are listed below.

- Multiplier Array
- Multiplier Control Recoder
- Exponent Adder

3.3.1.1 Multiplier Array

The multiplier array is a 32 X 32-bit asynchronous, parallel multiplier with 64-bit result. The multiplier array is based on the modified Booth's algorithm. The array performs signed/unsigned fixed-point multiplications with an integer data representation and floating-point multiplications using a 32-bit mantissa. The multiplier array performs automatic rounding to 32-bit result mantissa for the floating-point multiplications according to the IEEE Standard 754 for single extended precision. If rounding to IEEE single precision is specified (explicitly by the instruction or implicitly by the MR register), the result is rounded to 24-bit mantissa according to IEEE Standard 754 for single precision. The four IEEE rounding modes are supported; the rounding mode is specified by the rounding mode bits R1, R0 in the IER register.

3.3.1.2 Multiplier Control Recoder

The multiplier control decoder directs the operation of the Multiplier array and performs multiplier operand recoding for the modified Booth's algorithm multiplication.

3.3.1.3 Exponent Adder

The Exponent Adder is an 11-bit adder which serves as an adder for the exponents of the two operands of the multiplication. It actually computes the sum between the two input exponents and subtracts the bias. The resultant exponent is stored in the high portion of the destination register.

3.3.2 Adder Unit

The Adder is the second arithmetic processing unit of the Data ALU and performs all signed/unsigned integer fixed-point add, subtract and shift operations on the data operands as well as floating-point add, subtract and add-subtract. The floating-point add-subtract operation consists of a simultaneous add and subtract performed on the same input operands. This operation is useful for implementing FFT's (any Radix or type) and other transforms.

The operation of the floating-point Adder/Subtracter occurs independently and in parallel with the operation of the floating-point Multiplier and with the XDB and YDB activity.

The operation of the fixed-point Adder occurs independently and in parallel with the XDB and YDB activity. The Data ALU registers provide pipelining for both Data ALU Adder inputs and outputs.

All operations inside the Adder occur in one instruction cycle. Latches are provided on the Adder input operand buses to avoid race conditions. The major components of the Adder are

- Add Unit
- Subtract Unit
- Barrel Shifter and Normalization Unit
- Exponent Comparator and Update Unit
- Special Function Unit

3.3.2.1 Add Unit

The Add Unit is a high speed 32-bit asynchronous adder used in all floating-point non-multiply operations delivering a 32-bit result. The Add Unit performs automatic rounding to 32-bit result mantissa for the floating-point add/subtract according to the IEEE Standard for single extended precision arithmetic. If rounding to IEEE single precision is specified, the result is rounded to 24-bit mantissa according to the IEEE Standard for single precision arithmetic. The type of rounding is specified by the rounding mode bits in the MR register.

Two input operands are received on two internal data buses which are the 32-bit mantissas and are supplied to the Add Unit after the process of mantissa alignment required by a floating-point addition. The output of the Add Unit is delivered to the rounding unit which produces the result that is stored in the destination register.

3.3.2.2 Subtract Unit

The Subtract Unit is a high speed 32-bit asynchronous adder/subtracter used in all floating-point non-multiply operations as well as all fixed-point operations delivering a 32-bit result. The Subtract Unit performs automatic rounding to 32-bit result mantissa for the floating-point add/subtract according to the IEEE Standard for single extended precision arithmetic. If rounding to IEEE single precision is specified, the result is rounded to 24-bit mantissa according to the IEEE Standard for single precision arithmetic. The type of rounding is specified by the rounding mode bits in the MR register.

Two input operands are received on two internal data buses which are the 32-bit mantissas and are supplied to the Subtract Unit after the process of mantissa alignment required by a floating-point subtraction. For fixed-point operations the two input operands are supplied on the same data buses. The output of the Subtract Unit is delivered, in case of floating-point operations, to the rounding unit.

The Subtract Unit delivers the result in the middle portion of the destination register in case of floating-point operations and in the low portion of the destination register in case of integer operations.

3.3.2.3 Barrel Shifter and Normalization Unit

The Barrel Shifter is a 32-bit asynchronous parallel bidirectional (left-right) multibit shifter used in most floating-point operations and in arithmetic and logical shifting operations delivering a 32-bit result. When used in floating-point operations its main task is to provide operand alignment for add/subtract operations and post normalization of the final result. When used in fixed-point shifts the Barrel Shifter performs the following operations:

- single and multibit arithmetic shift left or right (ASL #n, ASR #n)
- single and multibit logical shift left or right (LSL #n, LSR #n)

Linkages are provided to shift in/out the condition code carry (C) bit.

3.3.2.4 Exponent Comparator and Update Unit

EXC is an 11-bit subtracter which compares the exponents of the two operands of the add/subtract operations. It receives its inputs on the AEIA and AEIB buses from the high portion of the registers and delivers as result the largest exponent and the difference between the exponents. The exponent difference is delivered to the barrel shifter which uses this information for the mantissa alignment process required by the floating point add/subtract operations. The largest exponent is delivered to exponent update units which may update it according to the result of the postnormalization process. The final result is supplied on the AEOA and/or AEOS buses and stored in the high portion of the destination register(s).

3.3.3 Logic Unit

The logic unit in the Data ALU performs the logical operations AND, ANDC, OR, ORC, EOR, NOT, ROR and ROL on Data ALU integer registers. It also performs the SPLIT, SPLITB, JOIN, JOINB, EXT and EXTB field manipulation instructions. The logic unit is 32-bits wide and operates on data in the low portion of the registers. The high and middle portions of the registers are not affected.

3.3.4 Divide and Square Root Unit

The Divide and Square Root Unit supports execution of the divide and square root operations. These operations are done using iterative algorithms that require an initial seed (first approximation) of 1/x and sqr(1/x).

3.3.5 Controller and Arbitrator

The controller and arbitrator unit (CA) supplies the control signals required by the processing units of the Data ALU and register file and is responsible for the full implementation of the IEEE standard. For the latter task the actions taken by the controller and arbitrator are determined by the FZ bit in the SR register. In the "Flush-to-Zero" mode, all denormalized input operands are considered as being zero and all denormalized results are "flushed to zero". Denormalized numbers include floating point zero. In the "IEEE" mode, all denormalized input operands are correctly used in calculations and denormalized results are computed and stored correctly, according to the IEEE standard. The DSP96002 is not able to perform operations on denormalized numbers in a single cycle when in IEEE mode, except for operations done in the floating point adder when the operand is a denormalized number in SEP. The controller and arbitrator unit is responsible for generating the appropriate sequence that deals with such situations.

When detecting denormalized numbers as input operands, the controller and arbitrator unit will add one extra cycle for entering the IEEE Mode procedure and afterwards it will add extra cycles, one for each denormalized input operand(s). These extra cycles are used for normalizing the input operand. After the normalization, the operand is stored in a temporary format which has a negative biased exponent ("wrapped format") but which is not available to the user. The original value of the operand in the source register is however not affected. During the IEEE Mode procedure the activity of the chip is suspended and it is resumed after all the input operands have been normalized. When detecting denormalized numbers as output results, the controller and arbitrator unit will enter the IEEE Mode Procedure and will add extra cycles, one for each denormalized output result.

3.4 AGU

The major components of the AGU are

- Address Register Files
- Offset Register Files
- Modifier Register Files
- Temporary Address Registers
- Modulo Arithmetic Units
- Address Output Multiplexers

A block diagram of the AGU is shown in Figure 3-3.

3.4.1 Address Register Files

Each of two Address Register Files consists of four 32-bit registers. The two files contain the address registers R0-R3 and R4-R7 respectively, which usually contain addresses used as pointers to memory. Each register may be read or written by the Global Data Bus. High speed access to the XAB and YAB is required to allow maximum access time for the internal and external X Data Memory, Y Data Memory, and Program Memory. Each address register may be used as input to its associated modulo arithmetic unit for a register update calculation. Each register may be written by the Global Data Bus or by the output of its respective modulo arithmetic unit. The registers accessed by the Global Data Bus and the Modulo Arithmetic Unit are not required to be the same. A separate write enable is provided for each register.

CAUTION

Due to pipelining, if an address register R is the destination of a MOVE instruction, the new contents will not be available for use as a pointer until the second following instruction.

3.4.2 Offset Register Files

Each of two Offset Register Files consists of four 32-bit registers. The two files contain the offset registers N0-N3 and N4-N7 respectively, and usually hold offset values used to update address pointers but can hold data. Each offset register may be read or written by the Global Data Bus. Each offset register is read when the same number address register is read and used as input to its associated modulo arithmetic unit. A read address selects the offset register to be read to the Modulo Arithmetic Unit during an instruction cycle. The registers accessed by the Global Data Bus and the Modulo Arithmetic Unit are not required to be the same. A separate write enable is provided for each register.

CAUTION

Due to pipelining, if an offset register N is the destination of a MOVE instruction, the new contents will not be available for use in address calculations until the second following instruction.

3.4.3 Modifier Register Files

Each of two Modifier Register Files consists of four 32-bit registers. The two files contain the modifier registers M0-M3 and M4-M7 respectively, and usually specify the type of modification made to an address reg-

Figure 3-3. AGU Block Diagram

ister during address register update calculations but they can hold data. Each modifier register may be read or written by the Global Data Bus. Each modifier register is automatically read when the same number address register is read and used as input to its associated modulo arithmetic unit. The registers accessed by the Global Data Bus and the Modulo Arithmetic Unit are not required to be the same. A separate write enable is provided for each register. Each modifier register is set to $FFFFFFFF during a processor reset.

CAUTION

Due to pipelining, if a modifier register M is the destination of a MOVE instruction, the new contents will not be available for use in address calculations until the second following instruction.

3.4.4   Temporary Address Registers

There are two kinds of temporary registers in the AGU: TempR (high and low) and TempN (high and low). The temporary address registers, TempR Low and TempR High, are 32-bit registers which provide temporary storage for an absolute address loaded from the Program Data Bus or for the output of the respective modulo arithmetic units. The modulo arithmetic unit output is loaded into the TempR registers during the pre-update cycle of the indexed by offset addressing mode and the LEA instruction. In each of these cases, an address register is accessed, updated by its respective modulo arithmetic unit, and stored in TempR in

one instruction cycle. In the following cycle, the contents of TempR are used to address X or Y memory. For all absolute addressing modes, the address of the operand is written into TempR and then used to address X, Y, or P memory.

The temporary address registers TempN Low and TempN High are 32-bit registers which provide temporary storage for the PC loaded from the Program Address Bus and it is used in case of the PC relative addressing mode. They may also be loaded from the Program Data Bus in case of Long or Short Displacement addressing mode.

3.4.5 Modulo Arithmetic Units

A block diagram of one modulo arithmetic unit is shown in Figure 3-4. The two modulo arithmetic units are identical. Each contains a 32-bit full adder (called offset adder) which may add one, minus one, the contents of the respective offset register N or the two's complement of N, to the contents of the selected address register. A second full adder (called modulo adder) adds the summed result of the first full adder to a modulo value M or minus M, where M is stored in the respective modifier register. A third full adder (called reverse carry adder) adds the constant one, minus one, the offset N (stored in the respective offset register) or minus N to the selected address register with the carry propagating in the reverse direction, i. e. from the most significant bit to the least. The offset adder and the reverse carry adder are in parallel and share common inputs. The only difference between them is that the carry propagates in opposite directions. Test logic, which consists of a modifier decoder, two carry multiplexers, and some control logic, determines which of the three summed outputs of the full adders is output to its associated address register file or temporary register.

Each modulo arithmetic unit can update one address register, Rn, from its respective address register file during one instruction cycle. It is capable of performing linear, reverse carry, and modulo arithmetic. The contents of the selected modifier register specifies the type of arithmetic to be used in an address register update calculation. The modifier value is decoded in the modulo arithmetic unit and affects the unit's operation. The modulo arithmetic unit's operation is data-dependent and requires execution cycle decoding of the selected modifier register contents. The modulo arithmetic unit performs three operations in parallel:

1. The output of the offset adder gives the result of linear arithmetic (e.g. Rn+1; Rn+Nn) and is selected as the modulo arithmetic unit's output for linear arithmetic addressing modifiers and PC relative addressing modes.
2. The reverse carry adder performs the required operation for reverse carry arithmetic and its output is selected as the modulo arithmetic unit's output for reverse carry addressing modifiers. Reverse carry arithmetic is useful for 2**K point Radix 2 FFT addressing. For modulo arithmetic, the modulo arithmetic unit will perform the function (Rn+/-N) modulo M where N can be one, minus one, or the contents of the offset register Nn.
3. If the modulo operation requires wraparound for modulo arithmetic, the summed output of the modulo adder will give the correct updated address register value; otherwise, if wraparound is not necessary, the output of the offset adder gives the correct result.

The test logic determines which output address to select. Modulo arithmetic units are shared by the DMA and the AGU and they are time multiplexed.

3.4.6 Address Output Multiplexers

The address output multiplexers select the source for the XAB, YAB, and PAB. They allow the XAB, YAB, or PAB address outputs to originate from either R0-R3, R4-R7, or from TempR Low or TempR High. The

address output multiplexers are shared by the DMA and the AGU. The output multiplexers are time multiplexed – the first half instruction cycle is assigned to DMA transfers while the second half cycle is assigned to core transfers.

Figure 3-4. Modulo Arithmetic Unit Block Diagram

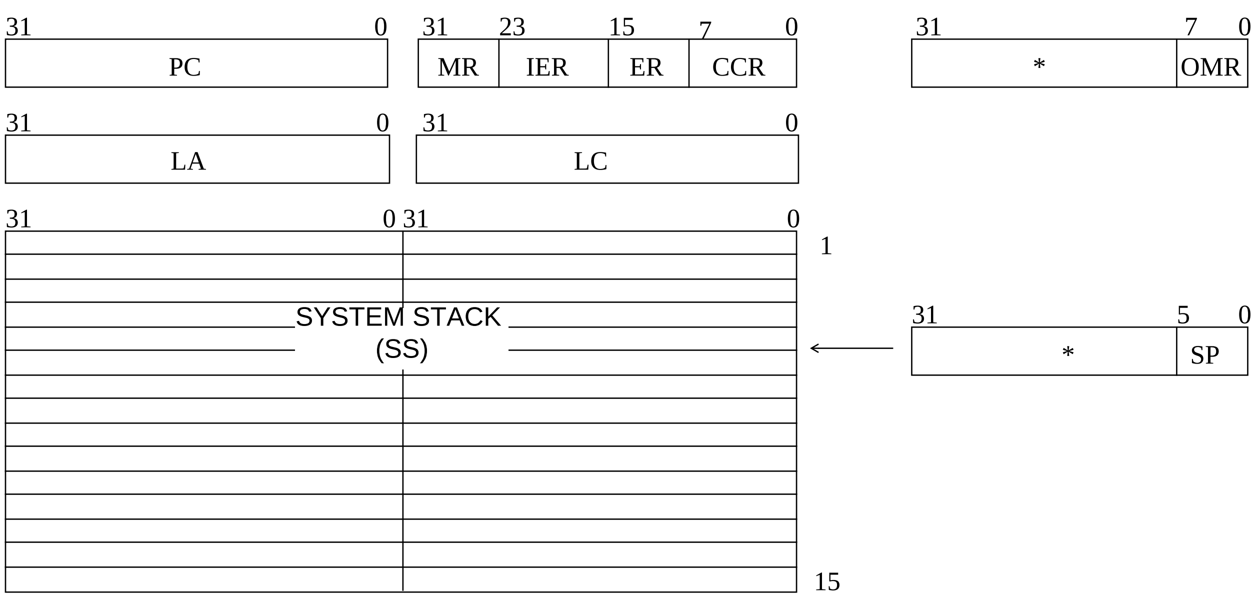
SECTION 4
SOFTWARE ARCHITECTURE

4.1 PROGRAMMING MODEL

The programmer can view the DSP96002 architecture as three execution units operating in parallel. The three execution units are the

- Data ALU
- Address Generation Unit
- Program Controller

The DSP96002 instruction set has been designed to allow flexible control of these parallel processing resources. Many instructions allow the programmer to keep each unit busy, thus enhancing program execution speed. The programming model is shown in Figure 4-1 and Figure 4-2, and is described in the following sections.



Program Controller* - Reserved bits: always read as zero, should be written with zero for future compatibility.

Figure 4-1. DSP96002 Programming Model - Program Controller

DATA ALU

95		0	
D9.H	D9.M	D9.L	D9
D8.H	D8.M	D8.L	D8
D7.H	D7.M	D7.L	D7
D6.H	D6.M	D6.L	D6
D5.H	D5.M	D5.L	D5
D4.H	D4.M	D4.L	D4
D3.H	D3.M	D3.L	D3
D2.H	D2.M	D2.L	D2
D1.H	D1.M	D1.L	D1
D0.H	D0.M	D0.L	D0
31 0	31 0	31 0	

ADDRESS GENERATION UNIT

31 0		31 0		31 0	
	M7		N7		R7
	M6		N6		R6
	M5		N5		R5
	M4		N4		R4
	M3		N3		R3
	M2		N2		R2
	M1		N1		R1
	M0		N0		R0

Figure 4-2. DSP96002 Programming Model – Data ALU and Address Generation Unit

4.2 DATA ALU REGISTER FILE (D0-D9)

The ten registers, D0-D9, are 96-bits wide and may be treated as thirty independent 32-bit registers or as ten 96-bit floating-point registers. Each 96-bit register is divided into three sub-registers: high, middle and low. Each sub-registers may be addressed individually by specifying the register number and the name of the sub-registers (e.g. D0.H, D0.M, D0.L). The low sub-register is used as source and destination for the integer operations. When writing to or reading from a sub-register no format conversion is performed.

The 96-bit registers Dn (n=0,...,9) are developed by the concatenation of Dn.H:Dn.M:Dn.L forming a floating-point data register. The data representation in a floating-point data register is always in an internal representation of the IEEE double precision format. When writing a register with a single or double precision

floating point number a format conversion to/from the internal representation takes place. The format conversion is performed automatically and is transparent to the user.

The registers serve as input pipeline registers between the XDB and YDB and the multiplier and/or adder. They are used as Data ALU source and/or destination operands allowing also new operands to be loaded for the next instruction while the register contents are used by the current instruction. They may also be read back out to the appropriate data bus to implement memory delay operations and save/restore operations for interrupt service routines.

4.2.1 Data ALU Auxiliary Registers (D8, D9)

D8 and D9 are two 96-bit data registers which are mainly present to permit a four instruction Radix-2 FFT butterfly. Operations with these registers are limited. They may be source operands only in multiply operations and source or destination operands in MOVE instructions. These registers are useful for extra multiplier input registers, pipelining registers, holding constants for compilers and temporary storage.

4.2.2 Data ALU General Purpose Registers (D0-D7)

D0, D1, D2, D3, D4, D5, D6 and D7 are eight general purpose data registers in the sense that MOVE instructions and arithmetic operations do not differentiate between them. They are used as Data ALU source and destination operands for most of the Data ALU instructions.

4.3 ADDRESS REGISTER FILES (R0-R3 AND R4-R7)

The eight address registers, R0-R7, are 32-bits wide and may contain addresses or general purpose data. The 32-bit address in a selected address register is used in the calculation of the effective address of an operand. This address may point to data directly or may be modified by a register offset. Most addressing modes modify the selected address register in a read-modify-write fashion. Typically, the address register is accessed, used as input to its associated modulo arithmetic unit, modified by the arithmetic unit and written back into the selected register. The form of address register modification performed by the modulo arithmetic unit is controlled by the contents of the offset and modifier registers discussed below. The contents of an address register may be transferred to/from an effective address held in a temporary address register.

4.4 OFFSET REGISTER FILES (N0-N3 AND N4-N7)

The eight offset registers, N0-N7, are 32-bits wide and may contain offset values used to increment and decrement address registers in address register update calculations or they may be used for general purpose storage. In addition, the contents of an offset register may be used to step through a table at some rate for waveform generation or may specify the offset into a table or the base of the table. An offset register will be accessed for an address register update calculation involving an address register of the same number (i.e., N0 is accessed when R0 is to be updated, N1 for R1, etc.).

4.5 MODIFIER REGISTER FILES (M0-M3 AND M4-M7)

The eight modifier registers, M0-M7, are 32-bits wide and may contain values which specify address arithmetic types used in address register update calculations (i.e., linear, reverse carry, and modulo) or they may be used for general purpose storage. When specifying modulo arithmetic, a modifier register will also specify the modulo value to be used. Refer to Section 5.8 for a description of the modifier types. A modifier reg-

ister will be accessed for an address register update calculation involving an address register of the same number (i.e., M0 is accessed when R0 is to be updated, M1 for R1, etc.). Each modifier register is set to $FFFFFFFF on processor reset which specifies the default value for linear arithmetic register update calculations.

4.6 PROGRAM COUNTER (PC)

This 32-bit register contains the address of the next location to be fetched from Program Memory Space. The PC may point to instructions, data operands or addresses of operands. References to this register are always inherent and are implied by most instructions. This special purpose address register is stacked when program looping is initiated, jump to subroutine is performed, and when interrupts occur except for fast interrupts (refer to Section 8.3).

4.7 STATUS REGISTER (SR)

The SR is a 32-bit register consisting of an 8-bit Mode register (MR), an 8-bit IEEE Exception register (IER), an 8-bit Exception register (ER) and an 8-bit Condition Code register (CCR).

The MR bits are only affected by processor reset, exception processing, the DO, DOR, ENDDO, ILLEGAL, RTI, RTR, FTRAPcc and TRAPcc instructions and by instructions which directly reference the MR register.

The IER bits are affected by processor reset, by instructions which directly reference the IER register and by the Data ALU floating-point operations. The IER contains the IEEE Rounding Mode control and the five exceptions flags as defined by the IEEE 754 standard. The five exception flags are "sticky" and the only way in which they can be cleared is by hardware reset or by the user writing the IER register. The purpose of making bits sticky is to prevent them from accidentally being cleared before being processed or used later by other instructions. The standard definition of the IER bits and the complete IER exception flag computation rules are given in Section A.5. It is strongly recommended that users of the DSP96002 obtain and comprehend the ANSI/IEEE Standard 754-1985 so that the full advantage of the standard can be realized.

The ER bits are affected by processor reset, by instructions which directly reference the ER register and by the Data ALU floating-point operations. The ER reflects the exceptions produced as a result of the execution of the last instruction. The standard definition of the ER bits and the complete ER bit computation rules are given in Section A.4.

The CCR contains flags that reflect the status produced by Data ALU instructions currently executing. The CCR bits are affected by Data ALU operations and by instructions which directly reference the CCR register. The standard definition of the CCR bits and the complete CCR bit computation rules are given in Section A.3.

The SR register is stacked when program looping is initialized, jump or branch to subroutine is performed, and when interrupts occur except for fast interrupts (refer to Section 8). The SR format is shown in Figure 4-3, and is described below.

4.7.1 CCR Carry (C) Bit 0

The carry bit is set if a carry is generated in an integer addition or if a borrow is generated in an integer subtraction. The carry bit is also modified by bit manipulation, rotate, and shift integer instructions as well as by the Address Generation Unit operation when executing MOVETA instructions. The carry bit is not affected by floating-point instructions. The C bit is cleared during processor reset.

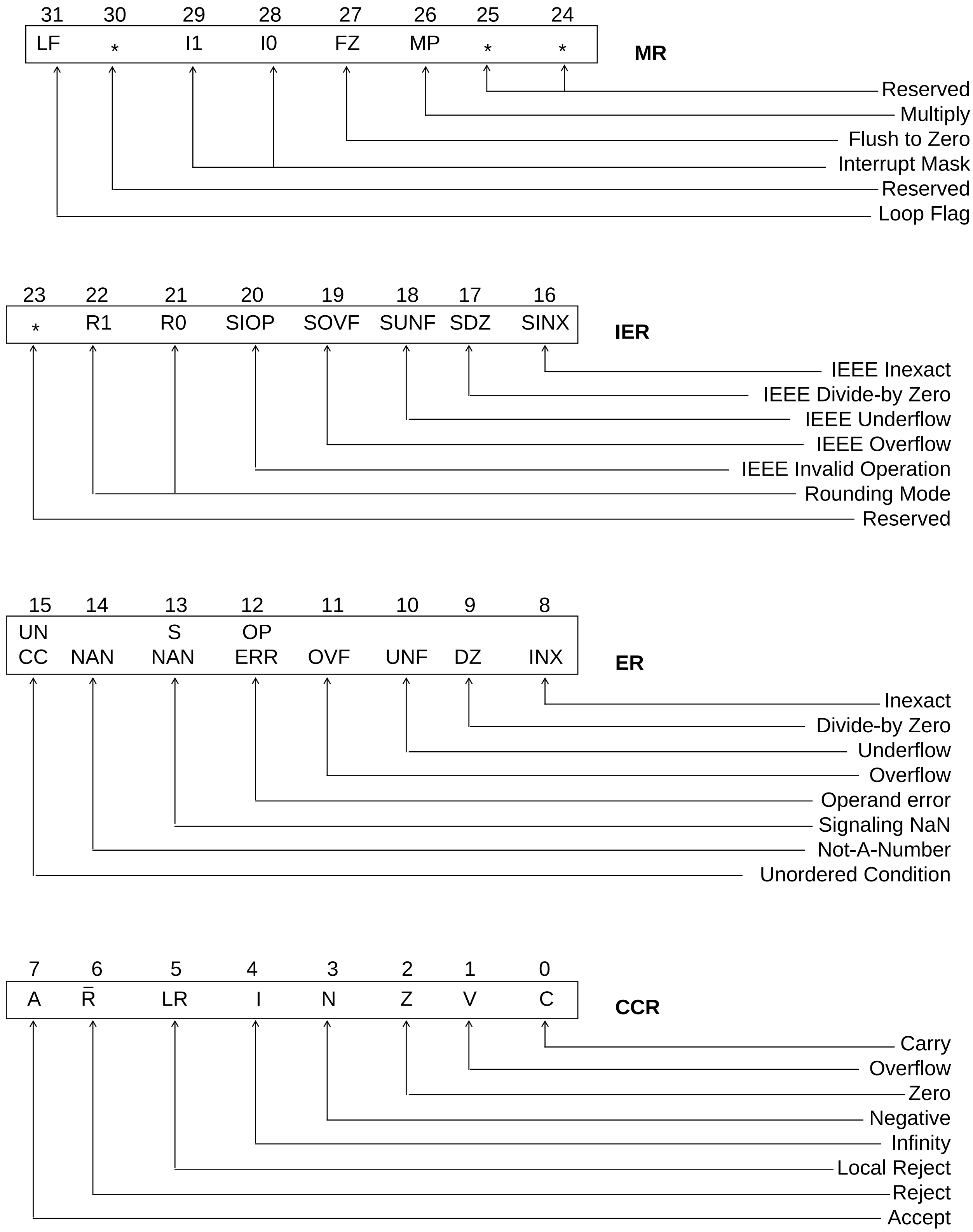


Figure 4-3. SR Format

4.7.2 CCR Overflow (V) Bit 1

The integer overflow bit is set if an arithmetic overflow occurred in a fixed point operation. This means that the result is not representable in the destination size. The V bit is not affected by floating point operations unless they have a fixed point result. The overflow bit is also modified by Address Generation Unit operation when executing MOVETA instructions. The V bit is cleared during processor reset.

4.7.3 CCR Zero (Z) Bit 2

The zero bit is set if the result equals plus or minus zero in a floating point or zero in a fixed point operation. The zero bit is also modified by Address Generation Unit operation when executing MOVETA instructions. The Z bit is cleared during processor reset.

4.7.4 CCR Negative (N) Bit 3

The negative bit is set if the result is negative in a floating point or zero in a fixed point operation. The negative bit is also modified by Address Generation Unit operation when executing MOVETA instructions. The N bit is cleared during processor reset.

4.7.5 CCR Infinity (I) Bit 4

The infinity bit is set if the result of a floating-point operation is infinity. The I bit is not affected by fixed point operations. The I bit is cleared during processor reset.

4.7.6 CCR Local Reject (LR) Bit 5

The local reject bit is used for trivial reject testing of floating point or fixed point operands in graphics applications. The LR bit is cleared during processor reset.

4.7.7 CCR Reject ($\overline{R}$) Bit 6

The global reject bit is used for trivial reject testing of floating point or fixed point operands in graphics applications. The $\overline{R}$ bit is cleared during processor reset.

4.7.8 CCR Accept (A) Bit 7

The accept bit is used for trivial accept testing of floating point or fixed point operands of floating point or fixed point operands in graphics applications. The A bit is cleared during processor reset.

4.7.9 ER Inexact (INX) Bit 8

The inexact bit is set if a floating-point result is inexact. This occurs when the **mantissa** of the intermediate result from the Data ALU operation is rounded to the specified precision. If the rounded mantissa transferred to the Dn register differs from the unrounded intermediate result mantissa, a loss of accuracy has occurred and the INX bit will be set. The INX bit is not affected by fixed point operations. The INX bit is cleared during processor reset.

4.7.10 ER Divide-by-Zero (DZ) Bit 9

The DZ flag in the DSP96002 can be set by software as part ofo an FDIV routine. No single DSP96002 instruction can set the DZ flag. The DZ bit is cleared during processor reset and during all floating-point instructions.

4.7.11 ER Underflow (UNF) Bit 10

The underflow bit is set if a result of a floating-point operation is too small to be **represented** in a floating-point data register (i. e., strictly between $\pm 2^{E_{min}}$). The test is done on the exponent before rounding. A denormalized result will set the UNF bit. The UNF bit is not affected by fixed point operations. The UNF bit is cleared during processor reset.

4.7.12 ER Overflow (OVF) Bit 11

The overflow bit is set if a floating-point result is too large to be represented in a floating-point data register with the specified rounding precision as a normalized result. The test is done on the **exponent after** rounding the **mantissa** (i. e., the result with its mantissa rounded $\geq 1.0 \times 2^{E_{max}+1}$). Depending on the rounding mode and the sign of the result, a decision is made as to what the returned result will be. This returned result is the final rounded result. For example, the largest positive SP result which does not set OVF is $7F7FFFFF for all rounding modes. Note that a positive overflow of a finite number with round to minus infinity also returns $7F7FFFFF but sets OVF (see **Section C.1.5.1** – **General** for additional information on the rounding modes) . The OVF bit is not affected by fixed point operations. The OVF bit is cleared during processor reset.

4.7.13 ER Operand Error (OPERR) Bit 12

The operand error bit is set if an operation has no mathematical interpretation for the given operands. Examples of operations which set the OPERR bit are $(+\infty)+(-\infty)$, $0\times\infty$, and $\sqrt{-n}$. The OPERR bit is not affected by fixed point operations. The OPERR bit is cleared during processor reset.

4.7.14 ER Signaling NaN (SNAN) Bit 13

The signaling NaN bit is set when a signaling NaN is involved in an arithmetic floating-point operation. For example, “FABS.S D” where D is an SNaN will set the SNaN bit and return a quiet NaN. The SNAN bit is not affected by fixed point operations. The SNAN bit is cleared during processor reset. One example of where signaling NaN can be used is to give a known value to uninitialized memory which can be used to flag the user.

4.7.15 ER Not-a-Number (NAN) Bit 14

The Not-a-Number bit is set if the result of a floating-point operation is a NaN. For example, the DSP96002 sets the NaN bit as the result of operations which set the OPERR bit (i. e., the default result of invalid operations). The NAN bit is not affected by fixed point operations but is affected by some conversion instructions. For example, “INT D” where D is a NaN will return the fixed point value $FFFFFFFF and set the NaN bit. The NAN bit is cleared during processor reset.

4.7.16 ER Unordered Condition (UNCC) Bit 15

The unordered condition bit is set if a non-aware floating-point conditional instruction (FBcc, FJcc, FIFcc, etc) is executed when the NaN bit is set (the unordered condition). The result of the condition tested by an instruction depends on being able to represent the operand on the real number line. By definition, if the operand is a NaN, it cannot be ordered or represented on the real number line and therefore the UNCC bit will be set. UNCC is not affected by fixed point operations. The UNCC bit is cleared during processor reset.

4.7.17 IER IEEE Inexact Flag (SINX) Bit 16

The IEEE inexact flag is the IEEE flag for trap disabled operations that is set when the rounded result of an operation is not exact or if it overflows without an overflow trap (i. e., the INX bit is set by the current or a previous instruction). The SINX flag is cleared during processor reset.

4.7.18 IER IEEE Divide-by-Zero Flag (SDZ) Bit 17

The IEEE division by zero flag is the IEEE flag for trap disabled operations and is set if the dividend is a finite nonzero number and the divisor is zero (i. e., the DZ bit is set by the current or a previous instruction). The SDZ flag is cleared during processor reset.

4.7.19 IER IEEE Underflow Flag (SUNF) Bit 18

The IEEE underflow flag is the IEEE flag for trap disabled operations and is set when both tininess (UNF is set) and loss of accuracy (INX is set) have been detected (i. e., the INX bit and the UNF bit were set simultaneously in the current or a previous instruction). The SUNF flag is cleared during processor reset.

4.7.20 IER IEEE Overflow Flag (SOVF) Bit 19

The IEEE overflow flag is the IEEE flag for trap disabled operations and is set when the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result if the exponent range were unbounded (i. e., the OVF bit is set by the current or a previous instruction). The SOVF flag is cleared during processor reset.

4.7.21 IER IEEE Invalid Operation Flag (SIOP) Bit 20

The IEEE invalid operation flag is the IEEE flag for trap disabled operations and is set if an operand is invalid for the operation to be performed (i. e., the OPERR bit is set by the current or a previous instruction). The SIOP flag is cleared during processor reset.

4.7.22 IER Rounding Mode (R0-R1) Bits 21,22

The rounding mode bits R1 and R0 specify the way in which inexact results should be rounded in floating point operations. The rounding mode bits are cleared during processor reset.

R1	R0	Rounding Mode
0	0	Round to Nearest Even (default)
0	1	Round toward Zero
1	0	Round toward -Infinity
1	1	Round toward +Infinity

The Data ALU performs rounding of the result to the precision specified by the instruction. The DSP96002 supports only single extended and single precision results. The DSP96002 implements all four rounding modes specified by the IEEE standard. These modes are round to nearest (RN), round toward zero (RZ), round toward plus infinity (RP) and round toward minus infinity (RM). The rounding definitions are listed below.

RN Round to Nearest Even (default) - In this mode the representable value nearest to the infinitely precise value will be delivered as result. If the two nearest values are equally near, the one with the least significand bit equal to zero (even) will be the result – e. g., 1.65 rounds to 1.6 whereas 1.75 rounds to 1.8.

RZ Round Toward Zero - In this mode the result will be the value closest to, and no greater in magnitude than the infinitely precise result. This mode is sometimes called "truncation mode" or "chopped mode" since the bits to the right of the rounding point are discarded – e. g., 1.65 rounds to 1.6 and -1.65 rounds to -1.6.

RM Round Toward Minus Infinity - In this mode the result will be the value closest to, and no greater than the infinitely precise result (possibly minus infinity) – e. g., 1.65 rounds to 1.6 and -1.65 rounds to -1.7.

RP Round Toward Plus Infinity - In this mode the result will be the value closest to, and no less than the infinitely precise result (possibly plus infinity) – e. g., 1.65 rounds to 1.7 and -1.65 rounds to -1.6.

4.7.23 Reserved Status (Bits 23,24,25)

These bits are reserved for future expansion and will read as zero during read operations. They should be written with zero for future compatibility.

4.7.24 MR Multiply Precision Control (MP) Bit 26

The multiply precision control bit specifies the output precision of the multiply operation in the FMPY//FADD, FMPY//FADDSUB and FMPY//FSUB instructions. If MP is cleared, then the output precision of the multiply operation is determined by the accompanying instruction (FADD, FADDSUB or FSUB). If MP is set, then the output precision of the multiply operation is the maximum precision supported by the hardware (single extended precision in theDSP96002). MP is cleared during processor reset.

For example, if MP=0 and the accompanying instruction is FADD.S, then the multiply output precision will be single precision. If MP=1 and the accompanying instruction is FADD.S, then the multiply output precision will be single extended precision. If the accompanying instruction is FADD.X, then the multiply output precision will be single extended precision independently of the state of MP.

MP	Multiply Precision Control
0	Output Precision Determined By The Accompanying Instruction
1	Maximum Output Precision (SEP in theDSP96002)

4.7.25 Flush to Zero (FZ) Bit 27

The Flush to Zero bit specifies one of two modes for handling floating-point underflow - the IEEE gradual underflow mode using denormalized numbers and the Flush to Zero mode. If FZ is cleared, floating-point underflows are processed in full conformance to the IEEE 754-1985 floating-point standard, resulting in the possible generation of denormalized numbers. If a Data ALU source operand or result is a denormalized number, the IEEE underflow mode may insert additional instruction cycles for normalization and denormal-

ization, respectively. If FZ is set, floating-point underflows are flushed to zero. Any denormalized source operand is considered as zero (with the sign of the denormalized source operand) and any underflowed results are flushed to zero (with the sign of the original underflowed result). Cleared during processor reset.

FZ	Description
0	IEEE Gradual Underflow with Denormalized Numbers (default)
1	Flush to Zero

4.7.26 MR Interrupt Masks (I1-I0) Bits 28,29

The interrupt mask bits I1 and I0 reflect the current priority level of the processor and indicate the interrupt priority level (IPL) needed for an interrupt source to interrupt the processor. The current priority level of the processor may be changed under software control. The interrupt mask bits are set during processor reset.

I1	I0	Exceptions Permitted	Exceptions masked
0	0	IPL 0,1,2,3	None
0	1	IPL 1,2,3	IPL 0
1	0	IPL 2,3	IPL 0,1
1	1	IPL 3	IPL 0,1,2

4.7.27 Reserved Status (Bit 30)

This bit is reserved for future expansion and will read as one during read operations. It should be written with one for future compatibility.

4.7.28 MR Loop Flag (LF) Bit 31

The loop flag bit is set when a program loop is in progress and enables the circuitry which detects the end of a program loop. The loop flag is the only SR bit which is restored when terminating a program loop. Stacking and restoring the loop flag when initiating and exiting a program loop, respectively, allow the nesting of program loops. The loop flag is cleared during a processor reset.

4.8 LOOP COUNTER (LC)

The loop counter is a special 32-bit counter used to specify the number of times to repeat a hardware program loop. This register is stacked by a DO instruction and unstacked by end of loop processing or by execution of an ENDDO instruction. When the end of a hardware program loop is reached, the contents of the loop counter register are tested for one. If the loop counter is one, the program loop is terminated and the LC register is loaded with the previous LC contents stored on the stack. If the counter is not one, it is decremented by 1 and the program loop is repeated. The loop counter may be read under program control. This allows the number of times a loop has been executed to be determined during execution. LC is also used in the REP instruction.

4.9 LOOP ADDRESS REGISTER (LA)

The loop address register indicates the location of the last instruction word in a program loop. This register is stacked by a DO instruction and unstacked by end of loop processing or by execution of an ENDDO instruction. When the instruction word at the address contained in this register is fetched, the contents of LC

are checked. If it is not one, the LC is decremented, and the next instruction is taken from the address at the top of the system stack; otherwise the PC is incremented, the loop flag is restored (pulled from stack), the stack is purged, the LA and LC registers are pulled from the stack and restored and instruction execution continues normally. The LA register is a 32-bit read/write register written into by a DO instruction and is read by the system stack for stacking the register.

4.10 SYSTEM STACK (SS)

The system stack is a separate internal RAM 15 locations "deep" and divided into two banks: High (SSH) and Low (SSL) each 32-bits wide. SSH stores the PC or LA contents; SSL stores the LC or SR contents.

The PC and SR registers are pushed on the stack for subroutine calls and long interrupts (see Section 8). These registers are pulled from the stack for subroutine returns using the RTS instruction and for interrupt returns that use the RTI instruction. The system stack is also used for storing the address of the beginning instruction of a hardware program loop as well as the SR, LA and LC register contents just prior to the start of the loop. This allows nesting of DO loops.

Up to 15 long interrupts, 7 DO loops, or 15 JSRs or combinations of these can be accommodated by the Stack. Care must be taken when approaching the stack limit. When the Stack limit is exceeded the data to be stacked will be lost and a non-maskable Stack Error interrupt will occur.

4.11 STACK POINTER (SP)

The stack pointer register (SP) is a 32-bit register that indicates the location of the top of the system stack and the status of the stack (underflow and overflow error conditions). The stack pointer is referenced implicitly by some instructions (DO, ENDDO, REP, JSR, RTI, etc.) or directly by the MOVEC, MOVEI, MOVEM, MOVEP and MOVES instructions. The stack pointer register format is shown in Figure 4-4. Note that the stack pointer register is implemented as a six bit counter which addresses (selects) a fifteen location stack with its four least significant bits. The possible stack values are shown in Figure 4-5 and are described below.

4.11.1 Stack Pointer (SP) Bits 0,1,2,3

The stack pointer (SP) points to the last used place on the stack. Immediately after hardware reset these bits are cleared (SP=0), indicating that the stack is empty.

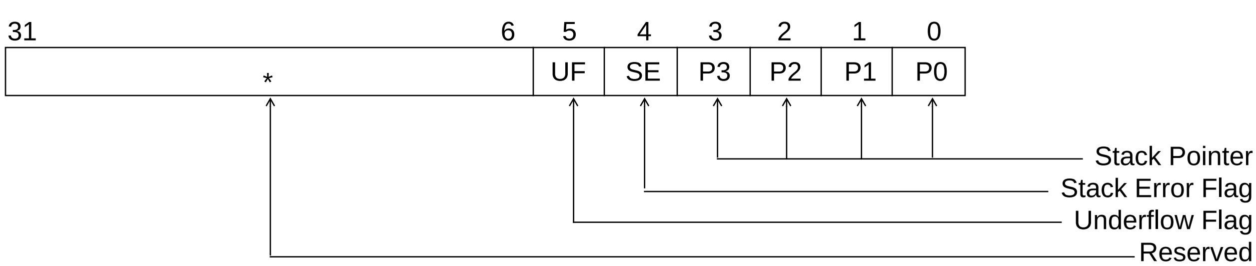


Figure 4-4. Stack Pointer Format

UF	SE	P3	P2	P1	P0	Description
1	1	1	1	1	0	Stack Underflow condition after double pull.
1	1	1	1	1	1	Stack Underflow condition.
0	0	0	0	0	0	Stack Empty (reset). Pull causes underflow.
0	0	0	0	0	1	Stack location 1. Double pull causes underflow.
0	0	0	0	1	0	Stack location 2.
.	.	.	.	.	.	
.	.	.	.	.	.	
.	.	.	.	.	.	
0	0	1	1	0	1	Stack location 13.
0	0	1	1	1	0	Stack location 14. Double push causes overflow.
0	0	1	1	1	1	Stack location 15. (Stack full). Push causes overflow.
0	1	0	0	0	0	Stack overflow condition.
0	1	0	0	0	1	Stack overflow condition after double push.

Figure 4-5. Stack Pointer Values

Data is pushed onto the stack by incrementing SP by one then writing the item at the new stack location SP. An item is pulled off the stack by copying it from location SP and then decrementing SP by one. Move instructions that read the SSH implicitly decrement the SP, and move instructions that write the SSH implicitly increment the SP. This facilitates managing the stack under software control. Since each location that the stack points to is 64 bits wide, it must be accessed by two move instructions. The first move should be to/from the SSL and then the second move should be to/from the SSH to automatically trigger a SP increment/decrement.

4.11.2 Stack Error flag (SE) Bit 4

The Stack Error flag (SE) indicates that a stack error has occurred. The transition of SE from 0 to 1 causes the priority level 3 Stack Error exception (see Section 8).

When the stack is completely full, the Stack Pointer reads 001111, and any operation that pushes data to the stack will cause a stack error exception to occur and the stack register will read 010000 (or 010001 if an implied double push occurs).

Any implied pull operation with SP=0 will cause a Stack Error exception (see Section 8), and the SP will read all ones (or 111110 if an implied double pull occurs). As shown in Figure 4-5, the SE bit is set.

Once set, the SE flag remains so until a move or bit instruction that directly references the Stack Pointer explicitly clears the SE flag. The SE flag is also cleared by hardware reset. When SP=0 (stack empty), no stack level is selected. Instructions which read the stack without SP post-decrement (REP SSL, MOVEC when SSL is specified as source, etc.) do not cause a stack error exception and the data read will be indeterminate. Instructions which write the stack without SP pre-increment (MOVEC when SSL is specified as destination, etc.) do not cause a stack error exception and no stack registers are altered.

4.11.3 Underflow flag (UF) Bit 5

The Underflow flag (UF) is set when a stack underflow occurs. The UF flag is cleared when a stack overflow occurs. While the SE flag remains set, the UF flag does not change with Stack Pointer operations caused by instructions that refer implicitly to the Stack Pointer such as RTI, RTS, DO, ENDDO, JSR, etc. The UF flag is cleared by hardware reset (see Figure 4-5). Implicit stack pointer operations that do not produce a stack error (i.e. do not set SE) will always clear UF as long as SE is not set.

4.11.4 Unimplemented Stack Pointer Register bits (Bits 6-31)

Any unimplemented stack pointer register bits are reserved for future expansion and read as zero during DSP96002 read operations. They should be written with zero for future compatibility.

4.12 OPERATING MODE REGISTER (OMR)

The operating mode register (OMR) is a 32-bit register which defines the current chip operating mode of the processor. The OMR bits are only affected by processor reset and by instructions which directly reference the OMR.

The operating mode register format is shown in Figure 4-6 and is described below.

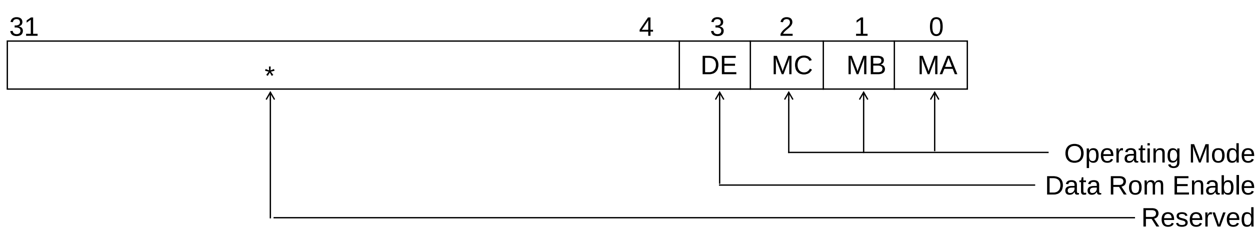


Figure 4-6. Operating Mode Register Format

4.12.1 Chip Operating Mode (Bits 0,1,2)

The operating mode bits MA, MB and MC determine if the internal program RAM is enabled and the startup procedure when the chip leaves the RESET state. These bits are loaded from the external Mode Select pins MODC, MODB and MODA respectively when the $\overline{\text{RESET}}$ pin is negated. After the DSP96002 leaves the RESET state, MC, MB and MA may be changed under program control. See Section 9 for more details on the chip operating modes.

4.12.2 Data ROM Enable (Bit 3)

The Data ROM Enable (DE) bit enables the two on-chip 512x32 Data ROMs located at address \$00000400 to \$000007FF in the X and Y memory spaces. When DE is cleared, the \$00000200 to \$000007FF space is part of the external X and Y data spaces and the on-chip Data ROMs are disabled (see the DSP96002 data memory maps in Section 9.2 for additional details).

4.12.3 Reserved Operating Mode Register (Bits 4-31)

These operating mode register bits are reserved for future expansion and will read as zero during DSP96002 read operations. They should be written with zero for future compatibility.

SECTION 5
DATA ORGANIZATION AND ADDRESSING MODES

5.1 OPERAND SIZES

Operand sizes are defined as follows: a byte is 8 bits long, a short word is 16 bits long, a word is 32 bits long and a long word is 64 bits long. For floating-point operations the operand sizes are defined as follows: a single real is 32 bits long, a double real is 64 bits long and a register operand is 96 bits long. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation.

5.2 DATA ORGANIZATION IN MEMORY

Program memory is 32 bits wide and supports 32-bit instruction words and instruction extension words.

The X and Y data memories are each 32 bits wide and support word and single real operands. The X and Y memories may be referenced as a single 64-bit wide memory space (the "L" space) to support long word and double real operands.

5.2.1 Integer Memory Data Formats

The DSP96002 supports four integer memory data formats:

- Signed Word Integer - 32 bits wide with two's complement representation.
- Signed Long Word Integer - 64 bits wide with two's complement representation.
- Unsigned Word Integer - 32 bits wide with unsigned magnitude representation.
- Unsigned Long Word Integer - 64 bits wide with unsigned magnitude representation.

The bit weighting for signed integers is presented in Figure 5-1. The bit weighting for unsigned integers is presented in Figure 5-2.

The DSP96002 does not support direct operations on Long Word Integers but they can be produced as result of some ALU operations or as a result of a Long Move.

5.2.2 Floating-point Memory Data Formats

The DSP96002 supports two floating-point memory data formats: Single Precision (32 bits) and Double Precision (64 bits), both fully complying with the IEEE Standard 754 for Binary Floating-Point Arithmetic. The memory formats for floating-point operands supported by DSP96002 are shown in Figure 5-3. The memory format for single and double real operands which conform to the IEEE 754 standard are shown below. Note that the stored exponent (e) is unsigned (i. e., biased positive) and positioned in the significant bits above those for the mantissa. By doing this, data can be ordered (sorted) by an integer machine which

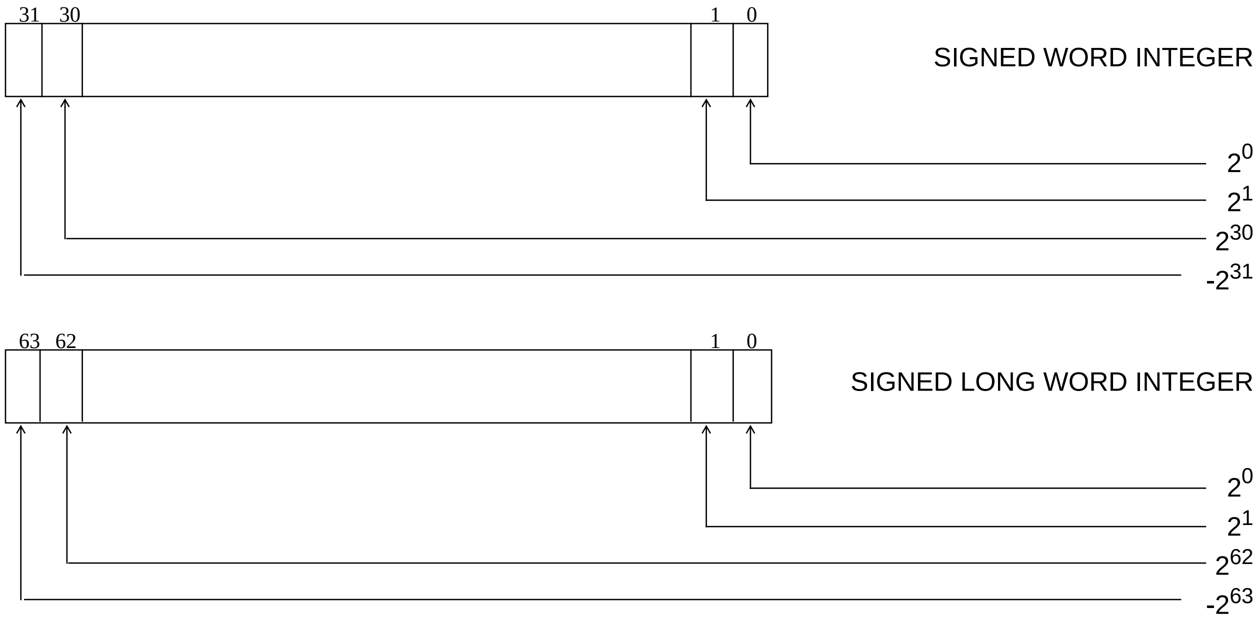


Figure 5-1. Bit Weighting and Alignment of Signed Integer Operands

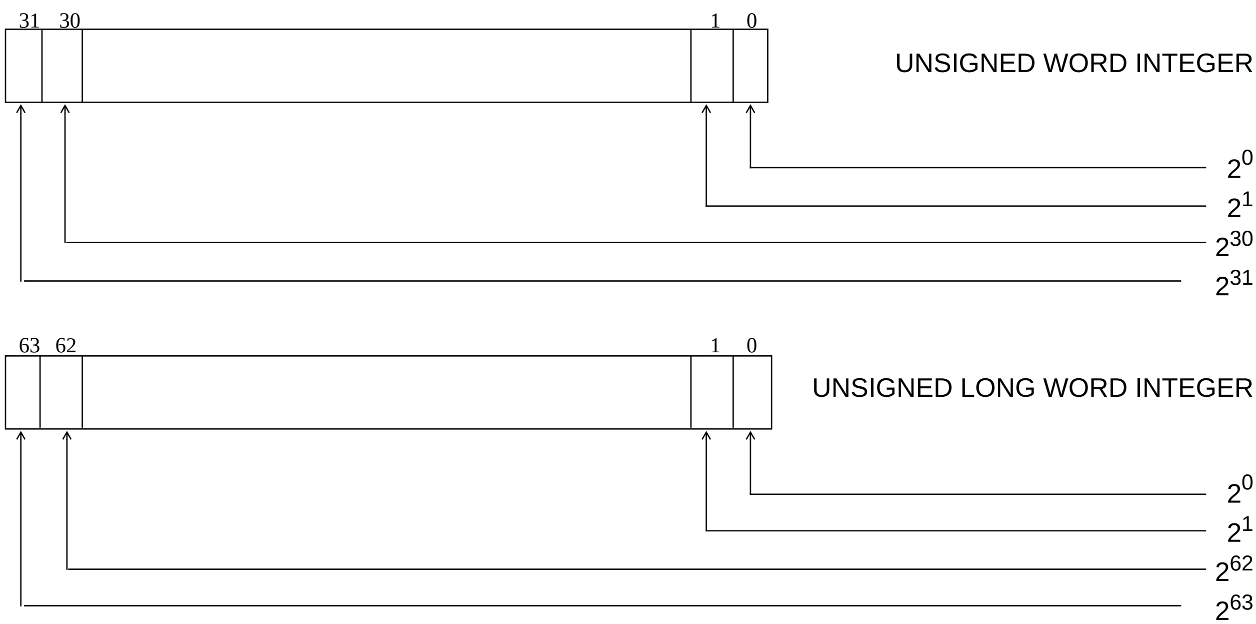


Figure 5-2. Bit Weighting and Alignment of Unsigned Integer Operands

is not aware that the data is represented in a floating point format. The range of the unbiased exponent, E, is every integer between E_{min} and E_{max}, inclusive ($-E_{min} \leq E \leq E_{max}$). For single precision (SP), E_{min} = -126 while E_{max} = +127; for double precision (DP), E_{min} = -1022 while E_{max} = +1023. For both SP and DP, E_{min}-1 is reserved to encode ±0 and denormalized numbers while E_{max}+1 is used to encode ±∞ and NaN's.

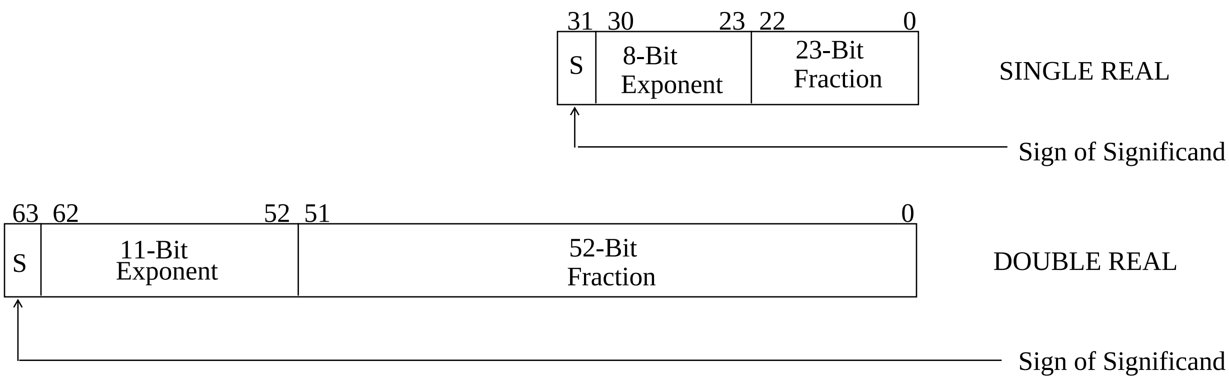
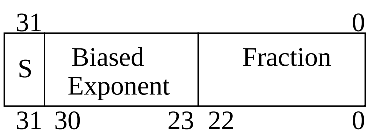


Figure 5-3. Memory Format for floating-point Operands

5.2.2.1 IEEE Single Precision Real Memory Format Summary



Field Size (in bits):

s = Sign 1
e = Biased Exponent 8
f = Fraction 23

Interpretation of Sign:

Positive Mantissa: s = 0
Negative Mantissa: s = 1

Normalized Numbers:

Represents real numbers in the form $(-1)^{s} \times 2^{(E+127)} \times 1.f$
E .... unbiased exponent $-126 \leq E \leq +127$
Bias of e .... +127 ($7F)
e = E + bias .... $0 \leq e \leq 254$ ($FE)
f .... Zero or Non-Zero
Mantissa................ 1.f

Denormalized Numbers:

Represents real numbers in the form $(-1)^{s} \times 2^{(E_{min}-1+127)} \times 0.f$
Bias of e .... +127 (\$7E)
e 0 (\$00)
f....................... Non-Zero
Mantissa................ 0.f

Signed Zeros:

Represents real zeroes in the form $(-1)^{s} \times 2^{(E_{min}-1+127)} \times 0.0$
Bias of e .... +127 (\$7F)
e 0 (\$00)
f....................... Zero
Mantissa................ 0.f = 0.00...00

Signed Infinities:

Represents real infinities in the form $(-1)^{s} \times 2^{(E_{max}+1+127)} \times 1.0$
Bias of e .... +127 (\$7F)
e .... 255 (\$FF)
f....................... Zero
Mantissa .... 1.f+1.00...00

NaNs (Not-a-Number):

Represents NaNs as $2^{(E_{max}+1+127)} \times 1.\mathbf{f}$
s .... Don't care
Bias of e .... n.a.
e .... 255(\$FF)
f .... Non-Zero: 11...11 Internal (legal) QNaN
1x...xx recognized QNaN
0x...xx SNaN

5.2.2.2 Double Precision Real Memory Format Summary

63		0
S	Biased Exponent	Fraction
63	62 — 52	51 — 0

Field Size (in bits):

s = Sign 1
e = Biased Exponent 11
f = Fraction 52

Interpretation of Sign:

Positive Mantissa: s = 0
Negative Mantissa: s = 1

Normalized Numbers:

Represents real numbers in the form $(-1)^{s} \times 2^{(E+1023)} \times 1.f$
E .... unbiased exponent $-1022 \le E \le +1023$
Bias of e .... +1023 ($3FF)
e + E + bias .... $0 \le e \le 2046$ ($7FE)
f .... Zero or Non-Zero
Mantissa................ 1.f

Denormalized Numbers:

Represents real numbers in the form $(-1)^{s} \times 2^{(E_{min}-1+1023)} \times 0.f$
E_{min}.................... -1022
Bias of e .... +1023 ($3FF)
e 0 ($000)
f .... Non-Zero
Mantissa................ 0.f

Signed Zeros:

Represents real zeroes in the form $(-1)^{s} \times 2^{(E_{min}-1+1023)} \times 0.0$
Bias of e .... +1023 ($3FF)
e 0 ($000)
f .... Zero
Mantissa................ 0.f = 0.00...00

Signed Infinities:

Represents infinities in the form $(-1)^{s} \times 2^{(E_{max}+1+1023)} \times 1.0$
Bias of e .... n.a.
e 2047 ($7FF)
f .... Zero
Mantissa................ 1.f = 1.00...00

NaNs (Not-a-Number):

Represents NaNs as $2^{(E_{max}+1+1023)} \times 1.f$
s .... Don't care
Bias of e .... n.a.
e 2047 ($7FF)
f .... Non-Zero: 11...11 Internal (legal) QNaN
1x...xx Recognized QNaN
0x...xx SNaN

5.3 DATA ORGANIZATION IN REGISTERS

5.3.1 Data ALU Registers

The thirty Data ALU registers are 32 bits wide and may be accessed as word operands. Sets of 2 Data ALU registers may be concatenated to form ten 64 bits registers which may be accessed as long words. The least significant bit (LSB) is the right-most bit (bit 0) and the most significant bit (MSB) is bit 31 or 63 for integer operands.

Sets of 3 Data ALU registers may be concatenated to form ten 96 bit registers which may be accessed as single real or double real operands. Floating-point operands are always represented in an internal double precision format, described below.

5.3.1.1 Internal floating-point Data Format

All DSP96002 internal floating-point operations are performed using single extended precision. All operands are converted to the internal double precision format when written into a Data ALU register. The internal double precision floating-point format used in the ten floating-point data registers is shown in Figure 5-4.

95	94	93	92 75	74 64	63	62 11	10 0
S	U	V	Zero	Biased Exponent	I	Fraction	Zero

- S is the sign of the mantissa.

- U is the single precision unnormalized tag.

- V is the single extended precision unnormalized tag.

- Biased Exponent is a 11 bit number which is essentially the 11 bit double precision biased exponent.

- Zero are bits that are always cleared by floating-point operations and floating-point moves.

- I is the integer part of the mantissa.

- Fraction is a 52 bit field representing the fractional part of the mantissa.

Figure 5-4. Data Format in the Floating Point Registers

When a result of an internal operations (which is a single extended precision number in the DSP96002) is written into a Data ALU register or when writing single or double precision numbers represented in one of the memory data formats to a Data ALU register as a result of a MOVE operation, automatic format conversion to the internal double precision representation is performed. Thus, mixed mode arithmetic is implicitly supported.

Since the DSP96002 implements single extended precision internal calculations, the Fraction part in the register may contain actually only 31 significand bits for single extended precision results or 23 significand bits for single precision results. However, if a double precision MOVE is performed, a 52 bit fraction will be written into the register but, if the same register is used as a floating-point operand, only the 31 most significand bits of the fraction will actually be used while the remaining bits are ignored by the Data ALU, resulting in a truncation error toward zero. Therefore, for future compatibility, only single extended precision data should be moved with the double precision data moves.

5.3.1.2 Internal Double Precision Format Summary

Field Size (in bits):
s = Sign 1

e = Biased Exponent 11

95	94	93	92 – 75	74 – 64	63	62 – 11	10 – 0
S	U	V	Zero	Biased Exponent	I	Fraction	Zero

u = U tag 1
v = V tag 1
i = Integer Part 1
f = Fraction 52
z = Unused bits......... 29

Interpretation of Unused Bits:
Input .... Don’t Care
Output.................. All Zeros

Unused bits should be written with zero for future compatibility.

Interpretation of Sign:
Positive Mantissa: s = 0
Negative Mantissa: s = 1

Normalized Numbers:
Represents real numbers in the form $(-1)^{s} x\ 2^{(e-1023)}$ x 1.f
Bias of e .... +1023 ($3FF)
e .... 0 < e < 2047 ($7FF)
i 1
f .... Zero or Non-Zero
Mantissa................ i.f = 1.f

Denormalized Numbers:
Represents real numbers in the form $(-1)^{s} x\ 2^{(-1022)}$x 0.f
Bias of e .... +1022 ($3FE)
e 0 ($000)
i 0
f .... Non-Zero
Mantissa................ i.f = 0.f

Signed Zeros:
Bias of e .... n.a.
e 0 ($000)
i 0
f .... Zero
Mantissa................ i.f = 0.00...00

Signed Infinities:
Bias of e .... n.a.
e 2047 ($7FF)
i 1
f .... Zero

Mantissa................ i.f = 1.00...00

NaNs (Not-a-Number):
s .... Don't care
Bias of e .... n.a.
e 2047 ($7FF)
i 1
f .... Non-Zero
Mantissa................ i.f: 1.11...11 Legal QNaN
1.1x...xx QNaN
1.0x...xx SNaN

5.3.2 Address Generation Unit (AGU) Registers

The notation Rn will be used to designate one of the 8 address registers R0-R7. The notation Nn will be used to designate one of the 8 address offset registers N0-N7. The notation Mn will be used to designate one of the 8 address modifier registers M0-M7. The eight AGU address registers R0-R7 support address or data operands of 32 bits. The eight AGU offset registers N0-N7 support offsets of 32 bits or may support address or data operands of 32 bits. The eight AGU modifier registers M0-M7 support modifiers of 32 bits or may support address or data operands of 32 bits.

5.3.3 Program Control Registers

The operating mode register (OMR) is 32 bits wide and may be accessed as a byte or word operand. The status register (SR) is 32 bits wide with the system mode register (MR) occupying the high-order 8 bits, the IEEE exception register (IER) occupying the next 8 bits, the exception register (ER) occupying the following 8 bits and the user condition code register (CCR) occupying the low-order 8 bits. The SR register may be accessed as a word operand. The MR, IER, ER and CCR registers may be accessed as byte operands. The loop counter register (LC), loop address register (LA), system stack pointer (SP), system stack high (SSH), and system stack low (SSL) are 32 bits wide and may be accessed as word operands.

The program counter register (PC) is a special 32-bit wide program control register. It is always referenced implicitly as a word operand.

The system stack is 64 bits wide and supports the concatenated PC and SR registers (PC:SR) for subroutine calls, interrupts and program looping, and also supports the concatenated LA and LC registers (LA:LC) for program looping.

5.4 NOT-A-NUMBER IMPLEMENTATION

When created by the DSP96002, Quiet Not-a-Numbers (QNaNs) represent the result of operations that have no mathematical interpretation (e.g. zero multiplied by infinity) or the result of operations involving a NaN operand as input.

Two different types of NaNs are implemented, differentiated by the most significand bit (MSB) of the fraction. NaNs with the most significant bit of the fraction set to one are quiet NaNs (QNaNs), also called non-signaling NaNs. NaNs with the most significant fraction bit equal to zero are signaling NaNs (SNaNs). The DSP96002 never creates a SNaN as a result of an operation.

The DSP96002 legal QNaN is defined as follows:

- It has the same pattern for all precisions.
- All bits of the fraction are set to one.
- The biased exponent is set to all ones.
- The sign bit is cleared.
- In the internal floating-point format, the I bit is always set to one; note that if the I bit is set to zero, the pattern is not recognized as a legal pattern by the Data ALU hardware, and operations on these bit patterns may yield unexpected results.

The IEEE specification defines the manner in which NaNs are handled when used as inputs to an operation. If a SNaN is used as an input, it requires that a QNaN be returned as the result if traps are disabled, which is the case for the DSP96002. The DSP96002 handles operations with SNaNs by generating the legal QNaN as a result. If QNaNs are used as input, it requires that one of the input QNaNs be returned as a result. The DSP96002 can only return the legal QNaN, and therefore, to be fully IEEE compatible, the only QNaN that should be used is the legal QNaN.

5.5 AUTOMATIC FLOATING-POINT FORMAT CONVERSIONS

There are two kinds of automatic floating-point format conversions within the DSP96002:

1. Conversion of a floating-point operand in any memory data format to the double precision internal data format of a floating-point data register. This is done when moving data from an external (to the Data ALU) location into a Data ALU floating-point register.
2. Conversion of a floating-point operand in the internal data format of a floating-point data register to any memory data format. This is done when moving data from a Data ALU floating-point register to an external (to the Data ALU) location.

5.5.1 Conversion to the Double Precision Internal Data Format

Since the internal data format used by the DSP96002 Data ALU is double precision, all external floating-point operands are converted to double precision values before writing them into a Data ALU floating-point register. The conversion is actually a "bit rearranging" operation using the procedure shown in Figure 5-5.

When converting a single precision number to the internal register data format, the implicit bit is revealed and stored as an explicit bit in the register. If the number to be converted is a denormalized single precision floating-point number, the U tag will be set indicating an unnormalized number. If such a number is to be used as an operand for floating-point operations, two cases arise depending on the state of the FZ (Flush-to-Zero) bit in the SR. In the Flush-to-Zero mode, the operand will be considered as zero in calculations. However, the data stored in the register will not be affected (unless the register is also the destination of the current operation). In the IEEE mode, the operand will be first "corrected" by adding to the execution cycle extra cycles for normalization. However, the data stored in the register will not be affected (unless the register is also the destination of the current operation).

When converting a double precision number to the internal register data format, the implicit bit is revealed and stored as an explicit bit in the register. If the number to be converted is a denormalized double precision (SEP in the DSP96002) floating-point number, the V tag will be set. If such a number is to be used as an operand for floating-point operations, two cases arise depending on the state of the FZ (Flush-to-Zero) bit in the SR. In the Flush-to-Zero mode, the operand will be considered as zero in calculations. However, the data stored in the register will not be affected (unless the register is also the destination of the current operation). In the IEEE mode, multiply operands will be first "wrapped" by adding to the execution cycle extra cycles for normalization. However, the data stored in the register will not be affected (unless the

Single Precision → Double Precision
Memory Format   Internal Format

```
31 → 95   S
     94   U - SET IF DENORMALIZED, CLEARED OTHERWISE
     93   V - CLEARED
     92   CLEARED
      .
     75   CLEARED
30 → 74
     73   SET IF NAN OR INFINITY, CLEARED IF ZERO, INV(BIT 30) OTHERWISE
     72   SET IF NAN OR INFINITY, CLEARED IF ZERO, INV(BIT 30) OTHERWISE
     71   SET IF NAN OR INFINITY, CLEARED IF ZERO, INV(BIT 30) OTHERWISE
29 → 70
 .  →  .
23 → 64
     63   I - CLEARED IF DENORM. OR ZERO, SET OTHERWISE
22 → 62
 .  →  .
 0 → 40
     39   CLEARED
      .    .
      0   CLEARED
```

Double Precision → Double Precision
Memory Format   Internal Format

```
63 → 95   S
     94   U - CLEARED
     93   V - SET IF DENORMALIZED, CLEARED OTHERWISE
     92   CLEARED
      .
     75   CLEARED
62 → 74
 .  →  .
52 → 64

     63   I - CLEARED IF DENORM. OR ZERO, SET OTHERWISE
51 → 62
 .  →  .
 0 → 11
     10   CLEARED
      .    .
      0   CLEARED
```

Figure 5-5. Conversion to Double Precision Internal Data Format

register is also the destination of the current operation). The DSP96002 does not support double precision. It does support single extended precision.

5.5.2 Conversion to the Memory Formats

Conversions from the internal double precision format to either of the two memory floating-point formats is performed whenever a data register is to be stored in memory or any other location external to the Data ALU. The conversion is actually a "bit rearranging" operation performed automatically by the MOVE instructions, and it is only responsible for collecting the required bits from the register and constructing the 32 or 64-bit data field to be stored in memory. This will produce correct results only if the data in the register is in a precision equal to the specified MOVE precision. For example, for single precision MOVEs the data must be already rounded to single precision.

Precision conversion to single precision (not format conversion) is accomplished by specifying an appropriate rounding operation (this may be an explicit instruction like FTFR.S or an implicit operation like FADD.S). The result after rounding is still stored in the internal double precision format; however, MOVE instructions that read it out of the Data ALU do not alter the value due to bit rearrangement. Figure 5-6 shows the bit rearrangement procedure performed by the MOVE instructions.

If a double precision value is to be rounded to single precision and the rounded result should yield a denormalized number, two different actions may be performed depending on FZ (Flush-to-Zero) bit in the SR. In the Flush-to-Zero mode, the result will be stored as zero in the register. In the IEEE mode, the operand will be first "corrected" by adding to the execution cycle extra cycles for denormalization. However, the data stored in the register will be in the internal double precision format and the U-tag will be set. The U-tag indicates that if another Data ALU operation will use this result as an operand, extra cycles should be added for operand normalization before actually using it.

5.6 OPERAND REFERENCES

The DSP96002 separates operand references into four classes: program, stack, register, and memory references. The type of operand reference(s) required for an instruction is specified by both the opcode field and the data bus movement field of the instruction (see Section 6.3). All operand reference types may not be used with all instructions.

5.6.1 Program References

Program references (called P references) are references to 32-bit wide program memory space and are usually instruction reads. Instructions or data operands may be read from or written to program memory space using the Move Program Memory (MOVEM), Move Peripheral Data (MOVEP), and Move Absolute Short (MOVES) instructions. Program references may be internal or external memory references depending on the address and the chip operating mode.

5.6.2 Stack References

Stack references (called S references) are references to a separate 64-bit wide internal memory space (System Stack) used implicitly to store the PC and SR registers for subroutine calls, interrupts and returns. In addition to the PC and SR registers, the LA and LC registers are stored on the stack when a program loop is initiated. The stack space address is always implied by the instruction. Data is written to stack memory space to save the processor state and is read from the stack to restore the processor state.

Double Precision Internal Format	→	**Single Precision Memory Format**
95	→	31
94		
.		
75		
74	→	30
73		
72		
71		
70	→	29
.	→	.
64	→	23
63		
62	→	22
.	→	.
40	→	0
39		
.		
0		

Double Precision Internal Format	→	**Double Precision Memory Format**
95	→	63
94		
75		
74	→	62
.	→	.
64	→	52
63		
62	→	51
.	→	.
11	→	0
10		
0		

Figure 5-6. Conversion from Internal Format to Memory Formats

5.6.3 R Register References

Register references (called R references) are references to the Data ALU, Address Generation Unit and Program Controller registers. Data may be read from one register and written into another register.

5.6.4 Memory References

Memory references are references to the 32-bit wide X or Y memory spaces and may be internal or external memory references depending on the effective address of the operand in the data bus movement field of the instruction. Data may be read or written from any address in either memory space.

5.6.4.1 X Memory References

The operand is in X memory space and is a word reference. Data may be read from memory to a register or from a register to memory.

5.6.4.2 Y Memory References

The operand is in Y memory space and is a word reference. Data may be read from memory to a register or from a register to memory.

5.6.4.3 L Memory References

L memory space references both X and Y memory spaces with one operand address. L memory space is developed by the concatenation (X:Y) of X and Y memory spaces. The data operand is a long word reference. The high-order word of the operand is in X memory; the low-order word of the operand is in Y memory. Data may be transferred between memory and concatenated registers (i.e., Dn.M:Dn.L) or double precision registers (i.e., Dn.D).

5.6.4.4 XY Memory References

XY memory space references both X and Y memory spaces with two operand addresses. One word operand is in X memory space and one word operand is in Y memory space.

5.6.4.4.1 Two independent addresses

Two independent addresses are used to access two word operands. Two effective addresses in the instruction are used to derive two independent operand addresses - one operand address may reference X memory space or Y memory space and the other operand address must reference the other memory space. One of the two effective addresses specified in the instruction must reference one of the address registers R0-R3, and the other effective address must reference one of the address registers R4-R7. Addressing modes are restricted to no-update and post-update by +1, -1, and +N addressing modes. Refer to Section 5.7 for a description of the addressing modes. Each effective address provides independent read/write control for its memory space. Data may be read from memory to a register or from a register to memory.

5.6.4.4.2 One common address

One common address is used to access two word operands. One effective address in the instruction is used to derive two indentical operand addresses referencing X and Y memory spaces. The effective address specified in the instruction references one of the address registers R0-R7. All address register indirect addressing modes may be used. Refer to Section 5.7 for a description of the addressing modes. The effective address provides a common read/write control for both memory spaces. Data may be read from memory to a register or from a register to memory.

5.7 ADDRESSING MODES

The DSP96002 instruction set contains a full set of operand addressing modes. All address calculations are performed in the Address Generation Unit to minimize execution time and loop overhead.

Addressing modes specify whether the operand(s) is in a register or memory and provide the specific address of the operand(s). An effective address in an instruction will specify an addressing mode, and for some addressing modes the effective address will further specify an address register. In addition, address register indirect modes require additional address modifier information which is not encoded in the instruction. The address modifier information is specified in the selected address modifier register(s). All memory references require one address modifier and the XY memory reference requires one or two address modifiers. The definition of certain instructions implies the use of specific registers and the addressing modes used.

Address register indirect modes require an offset and a modifier register for use in address calculations. These registers are implied by the address register specified in an effective address in the instruction word. Each offset register Nn and each modifier register Mn is assigned to an address register Rn having the same register number n. Thus the assigned registers are M0;N0;R0, M1;N1;R1, M2;N2;R2, M3;N3;R3, M4;N4;R4, M5;N5;R5, M6;N6;R6 and M7;N7;R7. The address register Rn is used as the address register, the offset register Nn is used to specify an optional offset and the modifier register Mn is used to specify an addressing mode modifier.

The addressing modes are grouped into three categories: register direct, address register indirect and special. These addressing modes are described below. Refer to Figure 5-7 for a summary of the addressing modes and operand references.

5.7.1 Register Direct Modes

These effective addressing modes specify that the operand is in one (or more) of the 30 Data ALU registers, 10 floating-point registers, 24 address registers or 7 control registers.

5.7.1.1 Data or Control Register Direct

The operand is in one, two or three Data ALU register(s) as specified in a portion of the data bus movement field in the instruction. This addressing mode is also used to specify a control register operand for special instructions. This reference is classified as a register reference.

5.7.1.2 Address Register Direct

The operand is in one of the 24 address registers specified by an effective address in the instruction. This reference is classified as a register reference.

CAUTION:

Due to pipelining, if an address register (Mn, Nn, or Rn) is changed with a MOVE instruction, the new contents will not be available for use as a pointer until the second following instruction.

5.7.2 Address Register Indirect Modes

The effective address in the instruction specifies the address register Rn and the address calculation to be performed. These addressing modes specify that the operand(s) is in memory and provide the specific address of the operand(s). When an address register is used to point to a memory location, the addressing mode is called address register indirect. The term indirect is used because the operand is not the address register itself, but the contents of the memory location pointed to by the address register. A portion of the data bus movement field in the instruction specifies the memory reference to be performed. The type of address arithmetic used is specified by the address modifier register Mn.

5.7.2.1 No Update (Rn)

The address of the operand is in the address register Rn. The contents of the Rn register are unchanged. The Mn and Nn registers are ignored. This reference is classified as a memory reference.

5.7.2.2 Postincrement by 1 (Rn)+

The address of the operand is in the address register Rn. After the operand address is used, it is incremented by 1 and stored in the same address register. The type of arithmetic used to increment Rn is determined by Mn. The Nn register is ignored. This reference is classified as a memory reference.

5.7.2.3 Postdecrement by 1 (Rn)-

The address of the operand is in the address register Rn. After the operand address is used, it is decremented by 1 and stored in the same address register. The type of arithmetic used to increment Rn is determined by Mn. The Nn register is ignored. This reference is classified as a memory reference.

5.7.2.4 Postincrement by Offset Nn (Rn)+Nn

The address of the operand is in the address register Rn. After the operand address is used, it is incremented (added) by the contents of the Nn register and stored in the same address register. The content of Nn is treated as a 2’s complement number and can therefore be interpreted as signed or unsigned (see **Section 5.8.1**). The contents of the Nn register are unchanged. The type of arithmetic used to increment Rn is determined by Mn. This reference is classified as a memory reference.

5.7.2.5 Postdecrement by Offset Nn (Rn)-Nn

The address of the operand is in the address register Rn. After the operand address is used, it is decremented (subtracted) by the contents of the Nn register and stored in the same address register. The content of Nn is treated as a 2’s complement number and can therefore be interpreted as signed or unsigned (see **Section 5.8.1**). The contents of the Nn register are unchanged. The type of arithmetic used to increment Rn is determined by Mn. This reference is classified as a memory reference.

5.7.2.6 Indexed by Offset Nn (Rn+Nn)

The address of the operand is the sum of the contents of the address register Rn and the contents of the address offset register Nn. The content of Nn is treated as a 2’s complement number and can therefore be interpreted as signed or unsigned (see **Section 5.8.1**). The contents of the Rn and Nn registers are un-

changed. The type of arithmetic used to increment Rn is determined by Mn. This reference is classified as a memory reference.

5.7.2.7 Predecrement by 1 -(Rn)

The address of the operand is the contents of the address register Rn decremented by 1. Before the operand address is used, it is decremented (subtracted) by 1 and stored in the same address register. The type of arithmetic used to increment Rn is determined by Mn. The Nn register is ignored. This reference is classified as a memory reference.

5.7.2.8 Long displacement (Rn+Label)

This addressing mode requires one word (label) of instruction extension. The address of the operand is the sum of the contents of the address register Rn and the extension word. The contents of the Rn register is unchanged. The type of arithmetic used to increment Rn is determined by Mn. The Nn register is ignored. This reference is classified as a memory reference.

5.7.3 PC Relative Modes

In the PC relative addressing modes, the address of the operand is obtained by adding a displacement, represented in two's complement format, to the value of the program counter (PC). The PC always point to the address of the next instruction, so PC relative addressing with zero displacement will produce the address of the following instruction.

5.7.3.1 Long Displacement PC Relative

This addressing mode requires one word of instruction extension. The address of the operand is the sum of the contents of the PC and the extension word.

5.7.3.2 Short Displacement PC Relative

The short displacement occupies 15 bits in the instruction operation word. The displacement is first sign extended to 32 bits and then added to the PC to obtain the address of the operand.

5.7.3.3 Address Register PC Relative

The address of the operand is the sum of the contents of the address register Rn and the PC. The Mn and Nn registers are ignored.

5.7.4 Special Address Modes

The special address modes do not use an address register in specifying an effective address. These modes specify the operand or the address of the operand in a field of the instruction or they implicitly reference an operand.

5.7.4.1 Immediate Data

This addressing mode requires one word of instruction extension. The immediate data is a word operand in the extension word of the instruction. This reference is classified as a program reference.

5.7.4.2 Immediate Short Data

The 8-, 16-, or 19-bit operand is in the instruction operation word. The 8-bit operand is used for ANDI and ORI instructions and it is zero extended. The 16-bit operand is used for immediate move to register and it is sign extended (interpreted as signed integer). The 19-bit operand is used for DO and REP instructions and it is zero extended. This reference is classified as a program reference.

5.7.4.3 Absolute Address

This addressing mode requires one word of instruction extension. The address of the operand is in the extension word. This reference is classified as a memory reference and a program reference.

5.7.4.4 Absolute Short Address

For the Absolute Short addressing mode the address of the operand occupies 7 bits in the instruction operation word and it is zero extended. This reference is classified as a memory reference.

5.7.4.5 Short Jump Address

The operand occupies 15 bits in the instruction operation word. The address is sign extended to 32 bits to use the same format for jumps and relative branches. This reference is classified as a program reference.

5.7.4.6 I/O Short Address

For the I/O short addressing mode the address of the operand occupies 7 bits in the instruction operation word and it is one extended. I/O short is used with the bit manipulation and move peripheral data instructions.

5.7.4.7 Implicit Reference

Some instructions make implicit reference to the program counter (PC), system stack (SSH, SSL), loop address register (LA), loop counter (LC)or status register (SR). The registers implied and their use is defined by the individual instruction descriptions (Appendix A).

5.7.5 Addressing Modes Summary

Figure 5-7 contains a summary of the addressing modes discussed in the previous paragraphs.

5.8 ADDRESS MODIFIER TYPES

The DSP96002 Address Generation Unit supports linear, modulo and bit-reversed address arithmetic for all address register indirect modes. Address modifiers determine the type of arithmetic used to update addresses. Address modifiers allow the creation of data structures in memory for FIFOs (queues), delay lines, circular buffers, stacks and bit-reversed FFT buffers. Data is manipulated by updating address registers

(pointers) rather than moving large blocks of data. The contents of the address modifier register Mn defines the type of address arithmetic to be performed for addressing mode calculations, and for the case of modulo arithmetic, the contents of Mn also specifies the modulus. All address register indirect modes may be used with any address modifier type. Each address register Rn has its own modifier register Mn associated with it.

5.8.1 Linear Modifier

The address modification is performed using normal 32-bit (modulo 4,294,967,296) linear arithmetic (two's complement). A 32-bit offset Nn, or immediate data (+1, -1, or a displacement value) may be used in the address calculations. The range of values may be considered as signed (Nn from -2,147,483,648 to +2,147,483,647) or unsigned (Nn from 0 to +4,294,967,295). There is no arithmetic differences between these two data representations. Addresses are normally considered unsigned, data is normally considered signed.

5.8.2 Reverse Carry Modifier

The address modification is performed by propagating the carry in the reverse direction, i.e., from the MSB to the LSB. This is equivalent to bit-reversing the contents of Rn and the offset value Nn, adding normally and then bit-reversing the result. If the (Rn)+Nn addressing mode is used with this address modifier, and Nn contains the value 2^{K-1} (a power of two), then postincrementing by Nn is equivalent to bit-reversing the K LSBs of Rn, incrementing Rn by 1, and bit-reversing the K LSBs of Rn. This address modification is useful for 2^K point FFT addressing. The range of values for Nn is 0 to +4,294,967,295. This allows bit-reversed addressing for FFTs up to 8,589,934,592 points.

As an example, consider a 1024 point FFT with real data stored in X memory and imaginary data stored in Y memory. Then Nn would contain the value 512 and postincrementing by +N would generate the address sequence 0, 512, 256, 768, 128, 640, ... This is the scrambled FFT data order for sequential frequency points from 0 to 2*pi. For proper operation the reverse carry modifier restricts the base address of the bit reversed data buffer to an integer multiple of 2^K, such as 1024, 2048, 3072, etc. The use of addressing modes other than postincrement by Nn is possible but may not provide a useful result.

5.8.3 Modulo Modifier

The address modification is performed modulo M, where M is permitted to range from 2 to +16,777,216. Modulo M arithmetic causes the address register value to remain within an address range of size M defined by a lower and upper address boundary. The value M-1 is stored in the modifier register Mn, thus allowing a modulo size range from 2 to 16,777,216. The lower boundary (base address) value must have zeroes in the k LSBs, where $2^k >= M$, and therefore must be a multiple of 2^k. The upper boundary is the lower boundary plus the modulo size minus one (base address plus M-1).

For example, to create a circular buffer of 24 stages, M is chosen as 24 and the lower address boundary must have its 5 LSBs equal to zero ($2^k >= 24$, thus $k >= 5$). The Mn register is loaded with the value 23 (m-1). The lower boundary may be chosen as 0, 32, 64, 96, 128, 160, etc. The upper boundary of the buffer is then the lower boundary plus 23.

The address pointer is not required to start at the lower address boundary and may begin anywhere within the defined modulo address range. In fact, the location of Rn determines the lower and upper boundaries.

Addressing Mode	Modifier MMM	P	S	C	D	A	X	Y	L	XY
Register Direct										
Data or Control Register	No			x	x					
Address Register	No					x				
Address Modifier Register	No					x				
Address Offset Register	No					x				
Address Register Indirect										
No Update	No	x					x	x	x	x
Postincrement by 1	Yes	x					x	x	x	x
Postdecrement by 1	Yes	x					x	x	x	x
Postincrement by Offset Nn	Yes	x					x	x	x	x
Postdecrement by Offset Nn	Yes	x					x	x	x	
Indexed by Offset Nn	Yes	x					x	x	x	
Predecrement by 1	Yes	x					x	x	x	
Long Displacement	Yes						x	x	x	
PC Relative										
Long Displacement	No	x								
Short Displacement	No	x								
Address Register	No	x								
Special										
Immediate Data	No	x								
Absolute Address	No	x					x	x	x	
Absolute Short Address	No						x	x	x	
Immediate Short Data	No	x								
Short Jump Address	No	x								
I/O Short Address	No						x	x		
Implicit	No	x	x	x						

where MMM = address modifier
P = program reference
S = stack reference
C = Program Controller register reference
D = Data ALU register reference
A = Address Generation Unit register reference
X = X memory reference
Y = Y memory reference
L = L memory reference
XY = XY memory reference

Figure 5-7. Addressing Modes Summary

On the DSP96002, the upper and lower boundaries are not explicitly needed. If the address register pointer increments past the upper boundary of the buffer (base address plus M-1) it will wrap around to the base address. If the address decrements past the lower boundary (base address) it will wrap around to the base address plus M-1.

If an offset Nn is used in the address calculations, the 32-bit value ∫Nn∫ must be less than or equal to M for proper modulo addressing. This is because a single modulo wrap around is detected. If ∫Nn∫ is greater than M, the result is data dependent and unpredictable except for the special case where Nn=L*(2^k), a multiple of the block size, 2^k, where L is a positive integer. Note that the offset Nn must be a positive two's complement integer. For this case the pointer Rn will be incremented using linear arithmetic to the same relative address L blocks forward in memory. Similarly, for the (Rn)-Nn addressing mode the pointer Rn will be decremented, using linear arithmetic, L blocks backward in memory. For the normal case where ∫Nn∫ is less than or equal to M, the modulo arithmetic unit will automatically wrap the address pointer around by the required amount. This type of address modification is useful in creating circular buffers for FIFOs (queues), delay lines and sample buffers up to 16,777,216 words long. It is also used for decimation, interpolation, and waveform generation. The special case of (Rn)+/-Nn with Nn=L*(2^k) is useful for performing the same algorithm on multiple buffers, for example implementing a bank of parallel filters. The range of values for Nn is -2,147,483,648 to +2,147,483,647 although all values are not useful when modulo addressing as described above.

5.8.4 Multiple Wrap-Around Modulo Modifier

The address modification is performed modulo M, where M may be any power of 2 in the range from 2^1 to 2^{23}. Modulo M arithmetic causes the address register value to remain within an address range of size M defined by a lower and upper address boundary. The value M-1 is stored in the modifier register Mn least significant 24 bits while the 8 most significant bits are set to $FF. The lower boundary (base address) value must have zeroes in the k LSBs, where 2^k = M , and therefore must be a multiple of 2^k. The upper boundary is the lower boundary plus the modulo size minus one (base address plus M-1).

For example, to create a circular buffer of 32 stages, M is chosen as 32 and the lower address boundary must have its 5 LSBs equal to zero (2^k = 32, thus k = 5). The Mn register is loaded with the value $FF00001F. The lower boundary may be chosen as 0, 32, 64, 96, 128, 160, etc. The upper boundary of the buffer is then the lower boundary plus 31.

The address pointer is not required to start at the lower address boundary and may begin anywhere within the defined modulo address range (between the lower and upper boundaries). If the address register pointer increments past the upper boundary of the buffer (base address plus M-1) it will wrap around to the base address. If the address decrements past the lower boundary (base address) it will wrap around to the base address plus M-1. If an offset Nn is used in the address calculations, the 32-bit value ∫Nn∫ is not required to be less than or equal to M for proper modulo addressing since multiple wrap around is supported for (Rn)+Nn, (Rn)-Nn and (Rn+Nn) address updates (multiple wrap-around cannot occur with (Rn)+, (Rn)- and -(Rn) addressing modes). The range of values for Nn is -2,147,483,648 to +2,147,483,647.

This type of address modification is useful for decimation, interpolation and waveform generation since the multiple wrap-around capability may be used for argument reduction.

5.8.5 Address Modifier Type Encoding Summary

Figure 5-8 contains a summary of the address modifier types discussed in the previous paragraphs.

Modifier MMMMMM M M	Address Calculation Arithmetic
0 0 0 0 0 0 0 0	Reverse Carry (Bit Reversed Update)
0 0 0 0 0 0 0 1	Modulo 2
0 0 0 0 0 0 0 2	Modulo 3
.	. .
.	. .
.	. .
0 0 F F F F F E	Modulo 16,777,215 ((2**24)-1)
0 0 F F F F F F	Modulo 16,777,216 (2**24)
0 1 x x x x x x	reserved
0 2 x x x x x x	reserved
.	. .
.	. .
.	. .
F D x x x x x x	reserved
F E x x x x x x	reserved
F F 0 0 0 0 0 0	reserved
F F 0 0 0 0 0 1	Multiple Wrap-Around Modulo 2
F F 0 0 0 0 0 3	Multiple Wrap-Around Modulo 4
F F 0 0 0 0 0 7	Multiple Wrap-Around Modulo 8
F F 3 F F F F F	Multiple Wrap-Around Modulo 2**22
F F 7 F F F F F	Multiple Wrap-Around Modulo 2**23
F F F F F F F F	Linear (Modulo 2**32)

where MMMMMMMM = Modifier Register Contents in Hex

Figure 5-8. Address Modifier Summary

SECTION 6
INSTRUCTION SET AND EXECUTION

6.1 INTRODUCTION

This chapter introduces the DSP96002 instruction set and instruction format. The complete range of instruction capabilities combined with the flexible addressing modes described in Chapter 5 provide a very powerful assembly language for digital signal processing and graphics algorithms. The instruction set has been designed to allow efficient coding for high-level language compilers and yet be easily programmed in assembly language.

As indicated by the programming model in Chapter 4, the DSP96002 architecture can be viewed as three execution units operating in parallel (Data ALU, Address Generation Unit and Program Controller). The goal of the instruction set is to keep each of these units busy during each instruction cycle. This achieves maximum throughput and minimum use of program memory.

6.2 INSTRUCTION GROUPS

The instruction set is divided into the following groups:

- Floating-Point Arithmetic (38)
- Fixed-Point Arithmetic (30)
- Logical (13)
- Bit Manipulation (4)
- Loop (4)
- Move (9)
- Program Control (35)

Each instruction group is described in the following sections. Detailed information on each of the 133 instructions is given in Appendix A.

6.2.1 Floating-Point Arithmetic Instructions

All floating-point arithmetic instructions operate on the 96-bit Data ALU registers. The floating-point arithmetic instructions are register-based (register direct addressing modes used for operands) and execute within the Data ALU. This means that the X Data Bus, Y Data Bus and the Global Data Bus are free for optional parallel move operations. This allows new data to be pre-fetched for use in following instructions and results calculated by previous instructions to be stored. Floating-point instructions always execute in a single instruction cycle in the Flush-to-Zero mode. Floating-point instructions execute in a single instruc-

tion cycle in the IEEE mode if denormalized numbers are not detected, otherwise additional instruction cycles will be required. See Figure 6-1 for a list of the thirty eight floating point arithmetic instructions.

FABS.S	Absolute Value (Single Precision)
FABS.X	Absolute Value (Single Extended Precision)
FADD.S	Add (Single Precision)
FADD.X	Add (Single Extended Precision)
FADDSUB.S	Add and Subtract (Single Precision)
FADDSUB.X	Add and Subtract (Single Extended Precision)
FCLR	Clear a Floating-Point Operand
FCMP	Compare
FCMPG	Graphics Compare with Trivial Accept/Reject Flags
FCMPM	Compare Magnitude
FCOPYS.S	Copy Sign (Single Precision)
FCOPYS.X	Copy Sign (Single Extended Precision)
FGETMAN	Get Mantissa
FINT	Convert to Floating-Point Integer
FLOAT.S	Integer to SP Floating-Point Conversion
FLOAT.X	Integer to SEP Floating-Point Conversion
FLOATU.S	Unsigned Integer to SP Floating-Point Conversion
FLOATU.X	Unsigned Integer to SEPFloating-Point Conversion
FLOOR	Convert to Floating-Point Integer Round to -Infinity
FMPY FADD.S	Multiply and Add (Single Precision)
FMPY FADD.X	Multiply and Add (Single Extended Precision)
FMPY FADDSUB.S	Multiply, Add and Subtract (Single Precision)
FMPY FADDSUB.X	Multiply, Add and Subtract (Single Extended Precision)
FMPY FSUB.S	Multiply and Subtract (Single Precision)
FMPY FSUB.X	Multiply and Subtract (Single Extended Precision)
FMPY.S	Multiply (Single Precision)
FMPY.X	Multiply (Single Extended Precision)
FNEG.S	Change Sign (Single Precision)
FNEG.X	Change Sign (Single Extended Precision)
FSCALE.S	Scale a Floating-Point Operand (Single Precision)
FSCALE.X	Scale a Floating-Point Operand (Single Extended Precision)
FSEEDD	Reciprocal Approximation
FSEEDR	Square Root Reciprocal Approximation
FSUB.S	Subtract (Single Precision)
FSUB.X	Subtract (Single Extended Precision)
FTFR.S	Transfer Floating-Point Register (Single Precision)
FTFR.X	Transfer Floating-Point Register (Single Extended Precision)
FTST	Test a Floating-Point Operand

Figure 6-1. Floating-Point Arithmetic Instructions

6.2.2 Fixed-Point Arithmetic Instructions

The fixed-point arithmetic instructions perform all operations within the Data ALU. Arithmetic instructions are register-based (register direct addressing modes used for operands) so that the Data ALU operation indicated by the instruction does not use the X Data Bus, the Y Data Bus, or the Global Data Bus. This allows for parallel data movement over these buses during most Data ALU operations. This allows new data to be pre-fetched for use in following instructions and results calculated by previous instructions to be stored. Fixed-point arithmetic instructions execute in one instruction cycle. See Figure 6-2 for a list of the thirty fixed-point arithmetic instructions.

ABS	Absolute Value
ADD	Add
ADDC	Add with Carry
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
CLR	Clear an Operand
CMP	Compare
CMPG	Graphics Compare with Trivial Accept/Reject Flags
DEC	Decrement by one
EXT	Sign Extend 16-Bit To 32-Bit
EXTB	Sign Extend 8-Bit To 32-Bit
GETEXP	Get Exponent
INC	Increment by One
INT	Floating-Point to Integer Conversion
INTRZ	Floating-Point to Integer Conversion Round to Zero
INTU	Floating-Point to Unsigned Integer Conversion
INTURZ	Floating-Point to Un. Integer Conversion Round to Zero
JOIN	Join Two 16-Bit Integers
JOINB	Join Two 8-Bit Integers
MPYS	Signed Multiply
MPYU	Unsigned Multiply
NEG	Negate
NEGC	Negate with Carry
SETW	Set an Operand
SPLIT	Extract a 16-Bit Integer
SPLITB	Extract an 8-Bit Integer
SUB	Subtract
SUBC	Subtract with Carry
TFR	Transfer Data ALU Register
TST	Test an Operand

Figure 6-2. Fixed-Point Arithmetic Instructions

6.2.3 Logical Instructions

The logical instructions perform all of the logical operations, except ANDI and ORI, within the Data ALU. Logical instructions are register-based like the arithmetic instructions discussed previously. Optional data transfers may be specified in parallel with most logical instructions – over the X and Y data buses or over the Global Data Bus. This allows new data to be pre-fetched for use in following instructions and results calculated in previous instructions to be stored. These instructions execute in one instruction cycle. See Figure 6-3 for a list of the thirteen logical instructions.

AND	Logical AND
ANDC	Logical AND with Complement
ANDI	AND Immediate to Control Register *
BFIND	Find Leading One
EOR	Logical Exclusive OR
LSL	Logical Shift Left
LSR	Logical Shift Right
NOT	Logical Complement
OR	Logical Inclusive OR
ORC	Logical Inclusive OR with Complement
ORI	OR Immediate to Control Register *
ROL	Rotate Left
ROR	Rotate Right

* These instructions do not allow parallel data moves.

Figure 6-3. Logical Instructions

6.2.4 Bit Manipulation Instructions

The bit manipulation instructions test the state of any single bit in a data memory location or register and then optionally sets, clears, or inverts the bit. The Carry bit in the CCR register will contain the result of the bit test. Parallel moves are not allowed with any of these instructions. See Figure 6-4 for a list of the four bit manipulation instructions.

BCLR	Bit Test and Clear
BSET	Bit Test and Set
BCHG	Bit Test and Change
BTST	Bit Test

Figure 6-4. Bit Manipulation Instructions

6.2.5 Loop Instructions

The loop instructions control hardware looping by initiating a program loop and setting up looping parameters, or by "cleaning" up the system stack when terminating a loop. Initialization includes saving registers used by a program loop (LA and LC) on the system stack so that program loops can be nested. The address of the first instruction in a program loop is also saved to allow no-overhead looping. See Figure 6-5 for a list of the four loop instructions.

DO	Start Hardware Loop
DOR	Start PC Relative Hardware Loop
ENDDO	Exit from Hardware Loop
REP	Repeat Next Instruction

Figure 6-5. Loop Instructions

6.2.6 Move Instructions

The move instructions perform data movement over the X and Y Data Buses, over the Global Data Bus and over the Program Data Bus. Address Generation Unit instructions are also included among the following move instructions. See Figure 6-6 for a list of the nine move instructions.

LEA	Load Effective Address
LRA	Load PC Relative Address
MOVE	Move Data Register(s)
MOVETA	Move Data Register(s) and Test Address
MOVEC	Move Control Register
MOVEI	Move Immediate
MOVEM	Move Program Memory
MOVEP	Move Peripheral Data
MOVES	Move Absolute Short

Figure 6-6. Move Instructions

6.2.7 Program Control Instructions

The program control instructions include jumps, conditional jumps, branches, conditional branches and other instructions which affect the PC and system stack. Branch instructions allow PC relative displacements needed for position independent code. See Figure 6-7 for a list of the thirty five program control instructions.

Bcc	Branch Conditionally
BRA	Branch Always
BRCLR	Branch if Bit Clear
BRSET	Branch if Bit Set
BScc	Branch to Subroutine Conditionally
BSCLR	Branch to Subroutine if Bit Clear
BSR	Branch to Subroutine
BSSET	Branch to Subroutine if Bit Set
DEBUG	Enter Debug Mode
FBcc	Branch Conditionally
FBScc	Branch to Subroutine Conditionally (Floating-Point Condition)
FFcc	Conditional Data ALU Operation without CCR Update
FFcc.U	Conditional Data ALU Operation with CCR Update
FJcc	Jump Conditionally
FJScc	Jump to Subroutine Conditionally
FTRAPcc	Conditional Software Interrupt
IFcc	Conditional Data ALU Operation without CCR Update
IFcc.U	Conditional Data ALU Operation with CCR Update
ILLEGAL	Illegal Instruction Interrupt
Jcc	Jump Conditionally
JCLR	Jump if Bit Clear
JMP	Jump
JScc	Jump to Subroutine Conditionally
JSCLR	Jump to Subroutine if Bit Clear
JSET	Jump if Bit Set
JSR	Jump to Subroutine
JSSET	Jump to Subroutine if Bit Set
NOP	No Operation
RESET	Reset Peripheral Devices
RTI	Return from Interrupt
RTR	Return from Subroutine and Restore Status Register
RTS	Return from Subroutine
STOP	Stop Processing (low power stand-by)
TRAPcc	Conditional Software Interrupt
WAIT	Wait for Interrupt (low power stand-by)

Figure 6-7. Program Control Instructions

6.3 INSTRUCTION FORMAT

Because of the multiple bus structure and the parallelism of the DSP96002, up to 3 data transfers may be specified in the instruction word - one on the X Data Bus, one on the Y Data Bus and one within the Data ALU. A fourth data transfer is generally implied and occurs in the Program Controller (instruction word fetch, program looping control, etc.). Each data transfer will involve a source and a destination.

In an instruction word, one or more "effective addresses" may be specified. An effective address defines the way in which an operand location is derived. The effective address will include an addressing mode and may also include a selected register. The addressing mode selects the address update to be used (see Section 5.7). The register specified may be the location of an operand or it may be an address register used to calculate the address of an operand. Certain instructions imply the use of specific registers and do not specify effective addresses for these registers.

The DSP96002 instructions consist of one or two 32-bit words - an operation word and an optional effective address extension word. The instruction and its length are specified by the first word of the instruction. The general format of the operation word is shown in Figure 6-8.

Most instructions specify data movement on the X and Y data buses and Data ALU operations in the same operation word. The DSP96002 is designed to perform each of these operations in parallel. The data bus movement field provides the operand reference type, the direction of transfer and the effective address(es) for data movement on the X and Y data buses. The operand reference type selects the type of memory or register reference to be made. The data bus movement field may require additional information to fully specify the operand for certain addressing modes. An effective address extension word following the operation word is used to provide immediate data, an absolute address or a displacement if required.

The opcode field of the operation word specifies the Data ALU operation or the Program Controller operation to be performed and any additional operands required by the instruction. Only those Data ALU and Program Controller operations which can accompany data bus movement activity will be specified in the opcode field of the instruction. Other Data ALU and Program Controller operations and all Address Generation Unit operations will be specified in an instruction word with a different format. These include operation words which contain short immediate data or short absolute addresses.

The assembly language source code for a typical one word instruction is shown below. The source code is organized into up to six fields.

(Multiplier) **(Adder/Subtracter)**

31 — 14	13 — 0
DATA BUS MOVE FIELD	OPCODE
OPTIONAL EFFECTIVE ADDRESS EXTENSION	

Figure 6-8. Instruction Word - General Format

Opcode	**Operands**	**Opcode**	**Operands**	**X Bus Data**	**Y Bus Data**
FMPY	D0,D5,D2	FSUB.S	D7,D3	X:(R0)+,D0.S	Y:(R4)+,D5.S

The first Opcode field indicates the Data ALU, Address Generation Unit, Bit Manipulation Unit, or Program Controller operation to be performed. The first Operands field specifies the operands to be used by the opcode specified in the first Opcode field.

The second Opcode field indicates a floating-point adder/subtracter operation in the Data ALU whenever parallel operation of the floating point adder/subtracter and multiplier is required. The second Operands

field specifies the operands to be used by the adder/subtracter opcode. One of the Opcode fields must always be included in the source code.

The X Bus Data field specifies an optional data transfer over the X Bus and the addressing mode to be used. The Y Bus Data field specifies an optional data transfer over the Y Bus and the addressing mode to be used. The address space qualifiers X:, Y: and L: indicate which address space is being referenced.

The DSP96002 offers parallel processing of the Data ALU, Address Generation Unit and Program Controller. For the instruction word above, the DSP96002 will perform the designated floating-point multiplier operation (Data ALU), the designated floating-point adder/subtracter operation (Data ALU), the data transfers specified with address register updates (Address Generation Unit), and will also decode the next instruction and fetch an instruction from program memory (Program Controller) all in one instruction cycle. When an instruction is more than one word in length, an additional instruction execution cycle is required.

Most instructions involving the Data ALU are register-based (all operands are in Data ALU registers) and allow the programmer to keep each parallel processing unit busy. An instruction which is memory-oriented (such as a bit manipulation instruction) or that causes a control flow change (such as a jump) prevents the use of parallel processing resources during its execution.

6.4 INSTRUCTION EXECUTION

Instruction execution is pipelined to allow most instructions to execute at a rate of one instruction every instruction cycle. However, certain instructions will require additional time to execute. These include instructions which are longer than one word, instructions which use an addressing mode that requires more than one cycle, instructions which make use of the global data bus more than once, and instructions which cause a control flow change. In the latter case a cycle is needed to clear the pipeline.

6.4.1 Instruction Processing

Pipelining allows the fetch-decode-execute operations of an instruction to occur during the fetch-decode-execute operations of other instructions. While an instruction is executing, the next instruction to be executed is decoded, and the instruction to follow the instruction being decoded is fetched from program memory. If an instruction is two words in length, the additional word will be fetched before the next instruction is fetched. Figure 6-9 demonstrates pipelining; F1, D1 and E1 refer to the fetch, decode and execute operations, respectively, of the first instruction. The third instruction contains an instruction extension word and takes two cycles to execute.

Each instruction requires a minimum of 12 clock phases to be fetched, decoded, and executed. A new instruction may be started after four phases. Two word instructions require a minimum of 16 phases to execute and a new instruction may start after eight phases.

	F1	F2	F3	F3e	F4	F5	F6	.	.	.
		D1	D2	D3	D3e	D4	D5	.	.	.
			E1	E2	E3	E3e	E4	.	.	.
Instruction Cycle:	1	2	3	4	5	6	7	.	.	.

Figure 6-9. Instruction Pipelining

6.4.2 Memory Access Processing

One or more of the DSP96002 memory sources (X data memory, Y data memory and program memory) may be accessed during the execution of an instruction. Each of these memory sources may be internal or external to the DSP96002. Three address buses (XAB, YAB and PAB) and four data buses (XDB, YDB, PDB and GDB) are available for internal memory core (as opposed to DMA) accesses during one instruction cycle.

The DSP96002 has two external expansion ports (Port A and Port B), that function as extensions of the internal address and data buses for external memory accesses. If all memory sources are internal to the DSP96002, one or more of the three memory sources may be accessed in one instruction cycle (i.e., program memory access or program memory access plus an X, Y, XY or L memory reference; refer to Section 5.6 for a description of operand references). However, when one or more of the memories are external to the DSP96002, and the external memories are located in the same expansion port, memory references may require additional instruction cycles.

If, in one instruction cycle, more than one external access is required on the same port, the accesses will be made with the following priority:

1. X memory.
2. Y memory.
3. Program memory.
4. DMA.

SECTION 7
EXPANSION PORTS AND I/O PERIPHERALS

7.1 INTRODUCTION

The upper 128 locations of the X and Y Data memories are defined as the I/O space. The Y memory I/O space is wholly external, while the X memory I/O space is internal. The X memory I/O space is used to address the I/O Interface registers as well as the bus, port select and interrupt control registers. Both I/O spaces may be accessed by regular X and Y memory MOVE instructions. The MOVEP instructions offer I/O short addressing and memory to memory move capability for easy data transfers with the I/O mapped registers.

The on-chip I/O peripherals are intended to minimize system chip count and "glue" logic in many applications. Each I/O interface has its own control, status and data registers memory-mapped into the X memory I/O space. Each interface has several dedicated interrupt vector addresses and control bits to enable/disable interrupts. This minimizes the overhead associated with servicing the device since each interrupt source has its own service routine.

Three on-chip peripherals are provided in the DSP96002:

- a 32-bit parallel Host MPU/DMA Interface connected to Port A.
- a 32-bit parallel Host MPU/DMA Interface connected to Port B.
- a two-channel DMA Controller.

7.2 EXPANSION PORTS CONTROL

The DSP96002 has two external expansion ports (Port A and Port B). Each port has a bus control register where memory wait states may be specified, parameter and control bits for a page circuit dedicated to DRAM/VRAM memory support are located, and control bits for direct software control of $\overline{BR}$ and $\overline{BL}$ pins are found.

7.2.1 Bus Control Registers (BCRA and BCRB)

There are 2 identical BCR registers, one for each port. The Bus Control Registers (BCRx) may be programmed to insert wait states in a bus cycle during external memory accesses. They are also used to program the Page Fault circuitry and for direct software control of the $\overline{BR}$ and $\overline{BL}$ pins.

31															16	
RH	LH	BS	XE	YE	PE	SF1	SF0	MF	NS	**	**	P3	P2	P1	P0	Port A Bus Control Register (BCRA) X:$FFFFFFFE

15 12	11 8	7 4	3 0
External X Memory Wait Control	External Y Memory Wait Control	External Prog Memory Wait Control	External I/O Memory Wait Control

31															16	
RH	LH	BS	XE	YE	PE	SF1	SF0	MF	NS	**	**	P3	P2	P1	P0	Port B Bus Control Register (BCRB) X:$FFFFFFFD

15 12	11 8	7 4	3 0
External X Memory Wait Control	External Y Memory Wait Control	External Prog Memory Wait Control	External I/O Memory Wait Control

** – reserved, read as zero, should be written with zero for future compatibility.

Figure 7-1. DSP96002 Bus Control Registers (BCRA and BCRB)

7.2.1.1 BCRx Wait Control Fields (Bits 0-15)

The BCRx Wait Control fields specify the number of wait states to be inserted in the bus cycle for an external X memory, Y memory, program memory or I/O access. Four bits are available in the control register for each type of external memory access. Each 4 bit field can specify up to 15 wait states. The Wait Control fields are set to '$F' (15 wait states) during hardware reset. See Section 2 for a description of the interaction between the wait states determined by the BCR and wait states generated due to the $\overline{TA}$ pin. Neither software reset, nor page circuit personal reset, affect BCRx.

7.2.1.2 BCRx Page Size (P3–P0) Bits 16-19

These bits define the page size for page fault operation. P3-P0 are set to '1010' by hardware reset. See Section 7.2.2 on Page Circuit Operation.

P3-P0	Page Size
0000	1
0001	2
0010	4
0011	8
0100	16
0101	32
0110	64
0111	128
1000	256
1001	512
1010	1,024 (Reset value)
1011	2,048
1000	4,096
1101	8,192
1110	16,384
1111	32,768

7.2.1.3 BCRx Reserved bits (Bits 20, 21)

These reserved bits read as zero and should be written with zero for future compatibility.

7.2.1.4 BCRx Non-Sequential Fault Enable (NS) Bit 22

Non-sequential fault detection is enabled if the NS control bit is set. Non-sequential faults are ignored by the page circuit if the NS control bit is cleared. See Section 7.2.2 on Page Circuit Operation. Cleared by hardware reset.

7.2.1.5 BCRx Bus Mastership Fault Enable (MF) Bit 23

Bus mastership fault detection is enabled if the MF control bit is set. Bus mastership faults are ignored by the page circuit if the MF control bit is cleared. See Section 7.2.2 on Page Circuit Operation. Cleared by hardware reset.

7.2.1.6 BCRx Memory Space Fault Enable (SF1-SF0) Bits 24-25

Memory space faults based on changes in S1 and/or S0 are enabled by SF1 and SF0, respectively. If SF1(SF0) is set, changes in S1(S0) will cause a memory space fault. If SF1(SF0) is cleared, changes in S1(S0) are ignored by the page circuit. See Section 7.2.2 on Page Circuit Operation. SF1 and SF0 are cleared by hardware reset.

7.2.1.7 BCRx Program Memory Fault Enable (PE) Bit 26

If the Program Memory Fault Enable bit PE is set, the page fault circuit will monitor program memory bus cycles. If PE is set and a fault is detected during a program memory bus cycle, $\overline{\mathrm{TT}}$ will be deasserted. If PE is set and no fault is detected during a program memory bus cycle, $\overline{\mathrm{TT}}$ will be asserted. If PE is cleared, the page fault circuit will be inactive for program memory bus cycles and $\overline{\mathrm{TT}}$ will remain deasserted. PE is cleared by hardware reset.

PE	$\overline{\mathrm{TT}}$ Pin Activity for P Space
0	Deasserted
1	Active

7.2.1.8 BCRx Y Data Memory Fault Enable (YE) Bit 27

If the Y Data Memory Fault Enable bit YE is set, the page fault circuit will monitor Y Data memory bus cycles. If YE is set and a fault is detected during a Y Data memory bus cycle, $\overline{\mathrm{TT}}$ will be deasserted. If YE is set and no fault is detected during a Y Data memory bus cycle, $\overline{\mathrm{TT}}$ will be asserted. If YE is cleared, the page fault circuit will be inactive for Y Data memory bus cycles and $\overline{\mathrm{TT}}$ will remain deasserted. YE is cleared by hardware reset.

YE	$\overline{\mathrm{TT}}$ Pin Activity for Y Space
0	Deasserted
1	Active

7.2.1.9 BCRx X Data Memory Fault Enable (XE) Bit 28

If the X Data Memory Fault Enable bit XE is set, the page fault circuit will monitor X Data memory bus cycles. If XE is set and a fault is detected during a X Data memory bus cycle, $\overline{TT}$ will be deasserted. If XE is set and no fault is detected during a X Data memory bus cycle, $\overline{TT}$ will be asserted. If XE is cleared, the page fault circuit will be inactive for X Data memory bus cycles and $\overline{TT}$ will remain deasserted. XE is cleared by hardware reset.

XE	$\overline{TT}$ Pin Activity for X Space
0	Deasserted
1	Active

7.2.1.10 BCRx Bus State (BS) Bit 29

The read-only Bus State status bit BS is set if the DSP96002 is currently the bus master. If the DSP96002 is not the bus master, BS is cleared. Cleared by hardware reset.

7.2.1.11 BCRx Bus Lock Hold Control (LH) Bit 30

If the Bus Lock Hold control bit LH is set, the $\overline{BL}$ pin is asserted even if no read-modify-write access is occurring. If LH is cleared, the $\overline{BL}$ pin will only be asserted during a read-modify-write external access. Cleared by hardware reset.

7.2.1.12 BCRx Bus Request Hold Control (RH) Bit 31

If the Bus Request Hold control bit RH is set, the $\overline{BR}$ pin is asserted even though the CPU or DMA does not need the bus. If RH is cleared, the $\overline{BR}$ pin will only be asserted if an external access is being attempted or pending. Cleared by hardware reset.

7.2.2 Page Circuit Operation

The goal of the page circuit is to allow designers to achieve static RAM performance with low cost, dynamic RAM memory systems. With its internal page detection circuitry, the DSP96002 can achieve zero wait state performance using the fast access modes available on DRAM/VRAM devices. Without internal page detection circuitry, zero wait state performance would not be possible. Example memories are:

Device	Size	Mode
MCM514256A	256K x 4	Page
MCM51L1000A	1Meg x 1	Page
MCM514258A	256K x 4	Static Column
MCM511002A	1Meg x 1	Static Column

When a bus master, the page circuit is active when the CPU or DMA accesses the external bus using the P, X or Y memory spaces (S1:S0=10, 01 or 00). The page circuit uses the transfer type ($\overline{TT}$) output pin to indicate the type of external bus access. The page circuit asserts the transfer type ($\overline{TT}$) pin when an

external memory may use a fast access mode (page, static column, nibble or serial shift) during the current bus cycle. The page circuit must be programmed with the characteristics of the external memory which allow fast access modes. When the external memory cannot use a fast access mode in the current bus cycle, $\overline{TT}$ remains deasserted.

The page circuit selectively compares the address, memory space selection and bus mastership of a previously latched bus cycle C’ to the same attributes of the current bus cycle C based on the memory parameters programmed by the user in the Bus Control Register. Note that the previously latched bus cycle C’ may not be immediately prior to the current bus cycle, depending on the memory space mapping. The attributes of the current and previous bus cycle are defined in Figure 7-2, and the page circuit programming parameters are defined in Figure 7-3. These parameters (or functional equivalents) are user programmable in the Bus Control Register. Hardware, software, or page circuit personal reset (generated when PE, XE, and YE are clear) will reset the page circuit.

C	C’	Bus Access Attributes
A	A’	Address A0-A31
S	S’	Space Select S0-S1
M	M’	Bus Mastership $\overline{BA}$

Figure 7-2. Bus Access Attributes

Name	Memory Parameter	Random Port(D/VRAM)	Serial Port (VRAM)
P3-P0	Log2(page size)	number of rows (4 if nibble mode)	serial reg. size
NS	Non-Sequential Fault	yes if nibble mode	yes
MF	Bus Mastership Fault	depends on system	depends on system
SF1	Memory Space Fault 1	depends on system	depends on system
SF0	Memory Space Fault 0	depends on system	depends on system
PE	P Space Enable	depends on system	depends on system
XE	X Space Enable	depends on system	depends on system
YE	Y Space Enable	depends on system	depends on system

Figure 7-3. Page Circuit Programming Parameters

Once the memory parameters are programmed in the page circuit, the $\overline{TT}$ pin will provide information about the current external bus cycle based on information latched in the page circuit about a previous external bus cycle. The page circuit is capable of detecting the following faults:

Page Fault - $\overline{TT}$ is deasserted if the current address A is not in the same memory page as the latched address A’. The page size for the random access port of a DRAM or VRAM is typically the number of rows. The page size parameter P is equal to the number of row address lines latched into the memory when the row address strobe is asserted. Typical page sizes for page or static column mode RAMs are 256, 1024, etc. The page size for nibble mode RAMs is 4.

Non-Sequential Fault - ¯¯T¯T is deasserted if the current address A is not the increment (+1) of the latched address A'. The non-sequential fault is enabled if the NS control bit is set, otherwise disabled. Nibble mode accesses on the random port or serial accesses on the serial port can cause non-sequential faults. Page and static column mode RAMs cannot have non-sequential faults and NS should be cleared. The page circuit checks for non-sequential faults for addresses that are inside the defined page.

Bus Mastership Fault - ¯¯T¯T is deasserted if the current bus cycle is the first external bus cycle since becoming the bus master. The first external bus cycle by any bus master typically is not a fast access mode since other bus masters may have accessed the same external memory. This also ensures that the first external bus cycle after hardware reset deasserts ¯¯T¯T. The bus mastership fault is enabled if the MF control bit is set, otherwise disabled. It is possible that certain multiple processor systems may want to disable this feature if the external memory is allocated to a particular processor.

Memory Space (Physical Memory) Faults- ¯¯T¯T is deasserted if the current bus cycle accesses a different memory space than the previously latched bus cycle. This is useful if the space select pins S1 or S0 are used as address lines to the external memory. In this case, the user is mapping the same address in different memory spaces to DIFFERENT physical memory locations. If the space select pins S1 and S0 are not being used as address lines to the external memory, the user is mapping the same address in different memory spaces to the SAME physical memory location so changes in memory space should be ignored. This is an example of the "single memory space" mentality prevalent in systems executing high level languages like C.

Memory space faults based on changes in S1 and/or S0 are enabled by the SF1 and SF0 control bits, respectively. If SF1(SF0) is set, changes in S1(S0) will cause a memory space fault and deassert ¯¯T¯T. If SF1(SF0) is cleared, changes in S1(S0) are ignored. The user memory mapping and memory space change detection for each SF1 and SF0 combination are given in Figure 7-4a.

Note that both the current bus cycle C and the previously latched bus cycle C' represent accesses to one of the three memory spaces. The S1:S0=11 combination will never appear as a current or latched memory space value, since it means that no access is being done (S1:S0 = 00 ⇒ Y, S1:S0 = 01 ⇒ X, S1:S0 = 10 ⇒ P).

There is one combination (PX) missing from this encoding - where P and X share the same addresses. Since this combination cannot directly use S1 or S0 as address lines, its use will not be as popular and its implementation would require control on a "per-space" basis instead of the "per-pin" basis as shown above.

This discussion assumes that if S1 and/or S0 are used as address lines, they are introduced as high order address lines above the page size boundary. If S1 and/or S0 are introduced as low order addresses below the page size boundary, proper page fault operation can be achieved by adjusting the page size but the non-sequential fault detection cannot be used. Therefore, it is recommended that S1 and S0 only be used as high order address lines above the page size boundary. An example system with SF1:SF0 = 10 to detect shifts between program and data spaces is shown in Figure 7-4b.

7.2.2.1 Memory Space Enables and Page Fault Circuit Personal Reset

The page fault circuit is enabled if the current bus cycle is in a user selected memory space. Separate memory space enable control bits (PE, XE and YE) are provided so the user can select the memory space(s) which the page fault circuit monitors. If a memory space enable bit (PE, XE and/or YE) is set, the page fault circuit is active if the current bus cycle is in that memory space. If a memory space enable bit is cleared, the page circuit is inactive for that bus cycle and ¯¯T¯T remains deasserted. If all three memory space enables are set, the page circuit is active for all external bus cycles.

SF1	SF0	Memory Spaces Mapped To Same Physical Address	Memory Space Changes Detected as Faults
0	0	PXY share same addresses	none
0	1	PY share same addresses	P→X, X→P, X→Y, Y→X
1	0	XY share same addresses	P→X, X→P, P→Y, Y→P
1	1	none, all addresses unique	P→X, X→P, X→Y, Y→X, P→Y, Y→P

Figure 7-4a. Memory Space Change Detection

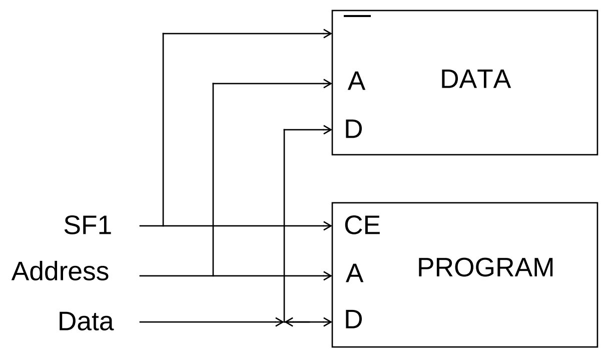


Figure 7-4b. Using SF1 to Physically separate Data and Program Spaces

If the current bus cycle is in an enabled memory space, the $\overline{\mathrm{TT}}$ pin is controlled by comparison of the current bus cycle and the previously latched bus cycle and the current bus cycle information (A, S) is latched at the end of the bus cycle. Thus the current bus cycle information becomes the previously latched bus cycle information for comparison in the next enabled external bus cycle. The encoding of the memory space enables is shown in Figure 7-5.

The page circuit normally monitors addresses intended for one external physical memory. However, if multiple memory spaces are mapped into one physical memory at either the same or different addresses, then the page circuit must monitor multiple memory spaces. These memory space enable bits allow the user to indicate which memory spaces should be monitored. Also if multiple memory spaces are mapped into different physical memories which are not accessed in an "interleaved" manner, one page circuit can serve multiple external physical memories by being enabled for more than one memory space. Non-interleaved accesses with multiple external physical memories are typical of systems where the main external bus activity is block-oriented DMA transfers.

If all three memory space enable bits are cleared, the page circuit is in the Personal Reset state. While in the Personal Reset state, the page circuit is inactive, $\overline{\mathrm{TT}}$ remains deasserted for all external bus cycles, and no bus cycle information is latched. The first bus cycle after re-enabling the page circuit always has $\overline{\mathrm{TT}}$ deasserted since no previous bus cycle information is available for comparison.

PE	XE	YE	$\overline{\mathrm{TT}}$ Pin Activity for P Space	X Space	Y Space	Current Bus Cycle Latched for P Space	X Space	Y Space
0	0	0	Deasserted	Deasserted	Deasserted	No	No	No
0	0	1	Deasserted	Deasserted	Active	No	No	Yes
0	1	0	Deasserted	Active	Deasserted	No	Yes	No
0	1	1	Deasserted	Active	Active	No	Yes	Yes
1	0	0	Active	Deasserted	Deasserted	Yes	No	No
1	0	1	Active	Deasserted	Active	Yes	No	Yes
1	1	0	Active	Active	Deasserted	Yes	Yes	No
1	1	1	Active	Active	Active	Yes	Yes	Yes

Figure 7-5. Memory Space Enables Encoding

7.2.2.2 Refresh Faults

There is no internal support for refresh timers, refresh address counters or refresh faults which should deassert $\overline{\mathrm{TT}}$. The page circuit assumes that refresh does not exist and therefore $\overline{\mathrm{TT}}$ must be interpreted by the external memory controller based on its knowledge of refresh timing and external bus activity. The use of multiple processors with the same external DRAM/VRAM indicates that the memory controller is the best place to enforce refresh priorities. With the variety of refresh techniques based on the expected memory activity, the external memory controller state machine is the best place to have global control over refresh timing and arbitration caused by multiple access conflicts. At the end of each external bus cycle, the external memory controller should determine if it should begin a refresh cycle. If yes, it will disable the transfer acknowledge $\overline{\mathrm{TA}}$ signal to ensure that the DSP96002 waits if it begins an external access. Once the refresh is completed, the external memory controller must remember to ignore the $\overline{\mathrm{TT}}$ signal for the next memory cycle so that a fast access mode is not used. The external state machine should cancel (ignores) the effect of the $\overline{\mathrm{TT}}$ signal in the next external bus cycle after any hardware refresh operation. Note that if fast interrupts are used to implement a software refresh, refresh looks like a memory read cycle so no special treatment of $\overline{\mathrm{TT}}$ is needed.

7.2.2.3 $\overline{\mathrm{RAS}}$, $\overline{\mathrm{CAS}}$ and SC Timeout Faults

Since DRAM/VRAM devices are dynamic, there are maximum limits on the $\overline{\mathrm{RAS}}$ and $\overline{\mathrm{CAS}}$ low time which must be observed. To effectively use the fast access modes with the DSP96002, the external state machine must keep $\overline{\mathrm{RAS}}$ asserted between bus cycles for page, nibble and static column modes. $\overline{\mathrm{CAS}}$ must remain asserted between bus cycles for static column mode only. However, if no external access occurs after the external state machine is ready for a fast access mode, there is a possibility that $\overline{\mathrm{RAS}}$ or $\overline{\mathrm{CAS}}$ may "timeout". This is because the idle memory state must be "$\overline{\mathrm{RAS}}$ active" to use the fast access modes with the DSP96002 non-burst, random address bus cycles. The DSP96002 does not provide any internal support for $\overline{\mathrm{RAS}}$ or $\overline{\mathrm{CAS}}$ timeouts. The external state

machine is responsible for ensuring that $\overline{RAS}$ or $\overline{CAS}$ timeouts do not occur. Since typical $\overline{RAS}$ and $\overline{CAS}$ timeouts are 10-100 µsec, one of the simplest solutions is to perform a hardware refresh which deasserts both $\overline{RAS}$ and $\overline{CAS}$. If refresh is performed often enough, $\overline{RAS}$ and $\overline{CAS}$ timeout will never happen.

The serial port of VRAM devices is clocked by a serial clock SC. Since the serial shift register is dynamic, there is a minimum frequency at which the shift register must be clocked to refresh its contents. This frequency is typically about 20 kHz (50 µsec refresh period). The DSP96002 does not provide any internal support for SC timeouts. The external state machine is responsible for ensuring that SC timeouts do not occur.

If an SC timeout does occur, the external state machine cancels (ignores) the effect of the $\overline{TT}$ signal in the next external bus cycle to force a reload of the serial shift register. Fortunately, future 1Mbit VRAMs are being specified with static shift registers so the SC timeout problem should go away.

7.2.2.4 DMA Accesses

External DMA accesses to P, X or Y memory spaces are normal bus cycles and cannot be distinguished from CPU read/write cycles. Therefore DMA accesses can use the $\overline{TT}$ pin and do not need any special treatment by external hardware.

7.2.2.5 Multiple Memory Banks

Multiple memory banks exist when there are more external memories than needed just to cover the 32-bit data bus size. In this case, the external memory controller typically selects between banks by enabling one of several row address strobe ($\overline{RAS}$) signals or column address strobe ($\overline{CAS}$) signals based on several address lines. Since changes from one memory bank to another will cause a page fault, multiple memory banks are allowed and no special treatment is required.

7.2.2.6 Multiple Memory Controllers

Multiple memory controllers may exist to support fast access modes with multiple external physical memories. Since the page circuit can monitor multiple memory spaces and detect or ignore changes in memory spaces, multiple memory controllers are allowed and no special treatment is required.

7.3 EXPANSION PORTS SELECTION

Every memory space (X, Y and P) is divided into 8 equal portions. The division is fixed, that is, the sizes of the portions are fixed at 0.5 gigawords per portion and the address boundaries are fixed. Each portion of each memory space may be individually assigned to one of the external expansion ports (Port A or B). The mapping is controlled by the Port Select Register (PSR).

7.3.1 Port Select Register (PSR)

The Port Select Register is a 32-bit wide read/write register situated in the X I/O memory space. For each portion of each memory space there is a bit in the Port Select Register (PSR): if the bit is cleared, the respective portion goes thorough Port A, and if the bit is set, then it goes thorough Port B. Any memory seg-

ment that is defined as internal remains internal. The Port Select Register format is shown in Figure 7-6 and is described below.

31–24	23							16	15							8	7							0	PSR Port Select Register X:$FFFFFFFC
*	X7	X6	X5	X4	X3	X2	X1	X0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	P7	P6	P5	P4	P3	P2	P1	P0	

* - reserved, read as zeros, should be written with zeros for future compatibility.

X		Y		P	
$FFFFFF7F		$FFFFFFFF		$FFFFFFFF	
	X7		Y7		P7
$E0000000		$E0000000		$E0000000	
	X6		Y6		P6
$C0000000		$C0000000		$C0000000	
	X5		Y5		P5
$A0000000		$A0000000		$A0000000	
	X4		Y4		P4
$80000000		$80000000		$80000000	
	X3		Y3		P3
$60000000		$60000000		$60000000	
	X2		Y2		P2
$40000000		$40000000		$40000000	
	X1		Y1		P1
$20000000		$20000000		$20000000	
	X0		Y0		P0
$00000800 or $00000200		$00000800 or $00000200		$00000400 or $00000000	

Note: X and Y Data Memories lowest external address determined by DE bit in the OMR register. P Memory lowest external address determined by MA, MB and MC bits in the OMR register.

Figure 7-6. DSP96002 Port Select Register (PSR)

7.3.1.1 PSR Program Memory Port Select (P0-P7) Bits 0-7

The Program Memory Port Select control bits (P0-P7) determine the assignment of the 8 Program Memory segments to Port A or B. If the segment bit is cleared, the Program Memory segment is assigned to Port A. If the segment bit is set, the memory segment is assigned to Port B. The memory segment to control bit correlation is shown in Figure 7-6. For example, if the P4 bit is set, then all memory traffic for addresses P:$80000000 to P:$9FFFFFFF will go thorough Port B. During hardware reset, the P0-P7 bits are cleared if the MODA pin was hold low when negating $\overline{RESET}$. P0-P7 are set if the MODA pin was hold high when negating $\overline{RESET}$.

7.3.1.2 PSR Y Data Memory Port Select (Y0-Y7) Bits 8-15

The Y Data Memory Port Select control bits (Y0-Y7) determine the assignment of the 8 Y Data Memory segments to Port A or B. If the segment bit is cleared, the Y Data Memory segment is assigned to Port A. If the segment bit is set, the memory segment is assigned to Port B. The memory segment to control bit correlation is shown in Figure 7-6. For example, if the Y4 bit is set, then all memory traffic for addresses Y:$80000000 to Y:$9FFFFFFF will go thorough Port B. During hardware reset, the Y0-Y7 bits are cleared.

7.3.1.3 PSR X Data Memory Port Select (X0-X7) Bits 16-23

The X Data Memory Port Select control bits (X0-X7) determine the assignment of the 8 X Data Memory segments to Port A or B. If the segment bit is cleared, the X Data Memory segment is assigned to Port A. If the segment bit is set, the memory segment is assigned to Port B. The memory segment to control bit correlation is shown in Figure 7-6. For example, if the X4 bit is set, then all memory traffic for addresses X:$80000000 to X:$9FFFFFFF will go thorough Port B. During hardware reset, the X0-X7 bits are cleared.

7.3.1.4 PSR Reserved Bits (Bits 24-31)

These reserved bits read as zero and should be written with zero for future compatibility.

7.4 HOST INTERFACES

7.4.1 Introduction

The DSP96002 provides a Host MPU/DMA Interface for each of its ports. The Host MPU/DMA Interface provides a 32-bit parallel port to a host processor or DMA controller.

These Host Interfaces (HI) are intended to minimize system chip count and "glue" logic in many computer graphics and other multiprocessing applications. Each HI has its own control, status and data registers and is treated as memory-mapped I/O by the DSP96002. Each interface has several dedicated interrupt vector addresses and control bits to enable/disable interrupts. This minimizes the overhead associated with servicing the interface since each interrupt source has its own service routine.

The HI supports operation in a multiprocessor environment with a set of "host functions". The external device invoking these features is called the "host processor" and may be another DSP96002 processor or a 32-bit microprocessor such as the 68020, 68030, 68040 or 88000. Host processors with 32, 24 or 16-bit data buses may access all status and control bits of the HI. Host processors with an 8-bit data bus should add additional hardware to be able to access all status and control bits.

The HI functions allow:

- a host processor to transfer data having an arbitrary address to/from the DSP96002 without using external shared memory.
- a host processor to interrupt the DSP96002 using multiple interrupt vectors without using external shared memory.
- a host processor (with DMA capability) to transfer data blocks to/from the DSP96002 without using external shared memory.
- an external DMA controller to transfer data blocks to/from the DSP96002 without using external shared memory.
- unbuffered systems with minimum external logic as well as large buffered systems.

The HI connects to the external world thorough the external expansion port and a set of dedicated pins (described in Section 2):

- 32-bit bidirectional data bus D0-D31.
- 5 control lines: R/$\overline{W}$, $\overline{HS}$, $\overline{HA}$, $\overline{TS}$, $\overline{HR}$.
- address lines A2-A5.

The HI appears as a memory mapped peripheral occupying 16 locations in the host processor address space. Separate transmit and receive data registers are double-buffered to allow the DSP96002 and host processor to efficiently transfer data at high speed. Host processor communication with the HI registers is accomplished using standard host processor instructions and addressing modes.

Handshake flags are provided for polled or interrupt-driven data transfers with a host processor.

External DMA controllers (e.g. MC68450) are able to perform block data transfers between the DSP96002 HI and the external host processor memory. For this purpose, a "DMA mode" is provided in the HI. In this mode, the $\overline{\text{HA}}$ pin is used to enable access to the transmit/receive registers in the HI, without regard to the status of the address lines A2-A5.

The host processor can also issue vectored exception requests to the DSP96002 with the host command feature. The host processor may select any of the 256 DSP96002 exception routines to be executed by writing a vector address register. This flexibility allows the host processor programmer to execute a wide number of preprogrammed functions inside the DSP96002. Host exceptions can allow the host processor to read or write DSP96002 registers, X, Y, or Program memory locations and perform control and debugging operations if exception routines are implemented in the DSP96002 to do these tasks.

The DSP96002 views the HI as a memory mapped peripheral occupying four 32-bit words in X data memory space. The DSP96002 may use the HI as a normal memory-mapped peripheral using standard polled or interrupt programming techniques.

7.4.2 HI Reset

The HI is affected by the following types of reset:

HW/SW Reset	Hardware (HW) reset, generated by asserting the $\overline{\text{RESET}}$ pin, or Software (SW) reset, generated by executing the RESET instruction. Status and control bits in the HI are affected as defined in Figure 7-7 and Figure 7-8.
HOST Reset	HI personal reset, generated when the HRES bit in the HCR register is set. Only HI status bits are affected as defined in Figure 7-7 and 7-8. Only the DSP96002 may directly activate the HOST Reset since HRES is located in the DSP96002 side. Note that the HI remains in this state as long as the HRES bit is set. The HRES bit is not self-clearing.
INIT	HI personal reset, generated when the INIT bit in the ICS register is set. Only HI status bits are affected as defined in Figure 7-7 and Figure 7-8. Note that INIT may selectively reset the transmit and/or the receive channel(s) according to the state of the TREQ and RREQ control bits in the ICS register. Also, the INIT bit is self-clearing, in contrast to the HRES bit which requires an explicit clear operation.

7.4.3 HI Operation During Stop

The host processor is able to read/write the HI registers when the DSP96002 is in the Stop state (see Section 8). If the clock is stopped in the middle of a host processor access, the flag setup and data transfer across the HI will be frozen. The transfer and flag setup will finish after the clock is restarted.

If $\overline{HR}$ is used and the host processor reads RX or writes TX when the DSP96002 is in the Stop state, then $\overline{HR}$ will only be deasserted after exiting the Stop state. .

Register Name	**Register Contents**	**HW/SW Reset**	**HOST Reset**	**INIT TREQ=1 RREQ=0**	**INIT TREQ=0 RREQ=1**	**INIT TREQ=1 RREQ=1**	**Comments**
ICS	HMRC	0	0	0	-	0	
	HRST	1	1	-	-	-	
	DMAE	0	-	-	-	-	
	HF3-HF2	0	-	-	-	-	
	HF1-HF0	0	-	-	-	-	
	HREQ	0	Note 1	1	Note 2	1	
	INIT	0	-	0	0	0	
	TYEQ	0	-	-	-	-	
	TREQ	0	-	1	0	1	
	RREQ	0	-	0	1	1	
	TRDY	1	1	1	-	1	
	TXDE	1	1	1	-	1	
	RXDF	0	0	-	0	0	
CVR	HC	0	-	-	-	-	
	HV7-HV0	\$0E	-	-	-	-	port A
		\$0F	-	-	-	-	port B
IVR	IV7-IV0	\$0F	-	-	-	-	
SEM	SEM(15-0)	\$0000	-	-	-	-	

Notes:

1. HREQ = TYEQ + TREQ
2. HREQ = (TYEQ & TRDY) + (TREQ & TXDE)

Symbols:

HW - Hardware Reset caused by asserting the external pin $\overline{RESET}$.
SW - Software Reset caused by executing the RESET instruction.
HOST - Host Personal Reset caused when HRES=1.
INIT - Host Personal Reset caused when INIT=1.
"1" - The bit is set.
"0" - The bit is cleared.
"-" - The bit is not affected.
"+" - Logical OR operation.
"&" - Logical AND operation.

Figure 7-7. Host Interface Reset - Host Processor Side

Register Name	Register Contents	HW/SW Reset	HOST Reset	INIT TREQ=1 RREQ=0	INIT TREQ=0 RREQ=1	INIT TREQ=1 RREQ=1	Comments
HCR	HYWE	0	-	-	-	-	
	HYRE	0	-	-	-	-	
	HXWE	0	-	-	-	-	
	HXRE	0	-	-	-	-	
	HPWE	0	-	-	-	-	
	HPRE	0	-	-	-	-	
	HRES	1	1	-	-	-	
	HF3-HF2	0	-	-	-	-	
	HCIE	0	-	-	-	-	
	HTIE	0	-	-	-	-	
	HRIE	0	-	-	-	-	
HSR	HYWP	0	0	0	-	0	
	HYRP	0	0	0	-	0	
	HXWP	0	0	0	-	0	
	HXRP	0	0	0	-	0	
	HPWP	0	0	0	-	0	
	HPRP	0	0	0	-	0	
	HDMA	0	-	-	-	-	
	HF1-HF0	0	-	-	-	-	
	HCP	0	-	-	-	-	
	HTDE	1	1	-	1	1	
	HRDF	0	0	0	-	0	

Figure 7-8. Host Interface Reset - DSP96002 Side

7.4.4 HI Programming Model

The HI block diagram is shown in Figure 7-9. The HI has two programming models - one for the DSP96002 programmer and one for the external host processor programmer. In most cases, the notation used reflects the DSP96002 perspective. The HI - DSP96002 Programming Model is shown in Figure 7-10. The HI - External Host Processor Programming Model is shown in Figure 7-11. The HI Interrupt Structure is shown in Figure 7-13. The DSP96002 has two HIs. The registers of the two HIs are identical except for the addresses. Their names have an A or B suffix identifying the port they are connected to.

7.4.5 Host Transmit Data Register (HTX) - DSP96002 Side

The Host Transmit register (HTX) is used for DSP96002 to host processor data transfers. The HTX register is viewed as a 32-bit write-only register by the DSP96002. Writing the HTX register clears HTDE. The DSP96002 may program the HTIE bit to cause a Host Transmit Data interrupt when HTDE is set. The HTX register is transferred as 32-bit data to the Receive Register RX if both the HTDE bit and the Receive Data Full RXDF status bit are cleared. This transfer operation sets RXDF and HTDE.

Figure 7-9. HI Block Diagram (One Port)

7	6	5	4	3	2	1	0
**	**	HRES	HF3	HF2	HCIE	HTIE	HRIE

31	....	14	13	12	11	10	9	8
**	**	**	HYWE	HYRE	HXWE	HXRE	HPWE	HPRE

READ/WRITE
HOST CONTROL
REGISTER
HCR

7	6	5	4	3	2	1	0
HDMA	**	**	HF1	HF0	HCP	HTDE	HRDF

31	....	14	13	12	11	10	9	8
**	**	**	HYWP	HYRP	HXWP	HXRP	HPWP	HPRP

READ-ONLY
HOST STATUS
REGISTER
HSR

31	0
32-bit receive data register	

READ-ONLY
HOST RECEIVE
DATA REGISTER
HRX

31	0
32-bit transmit data register	

WRITE-ONLY
HOST TRANSMIT
DATA REGISTER
HTX or HTXC

HOST INTERFACE DSP96002 ADDRESS MAP

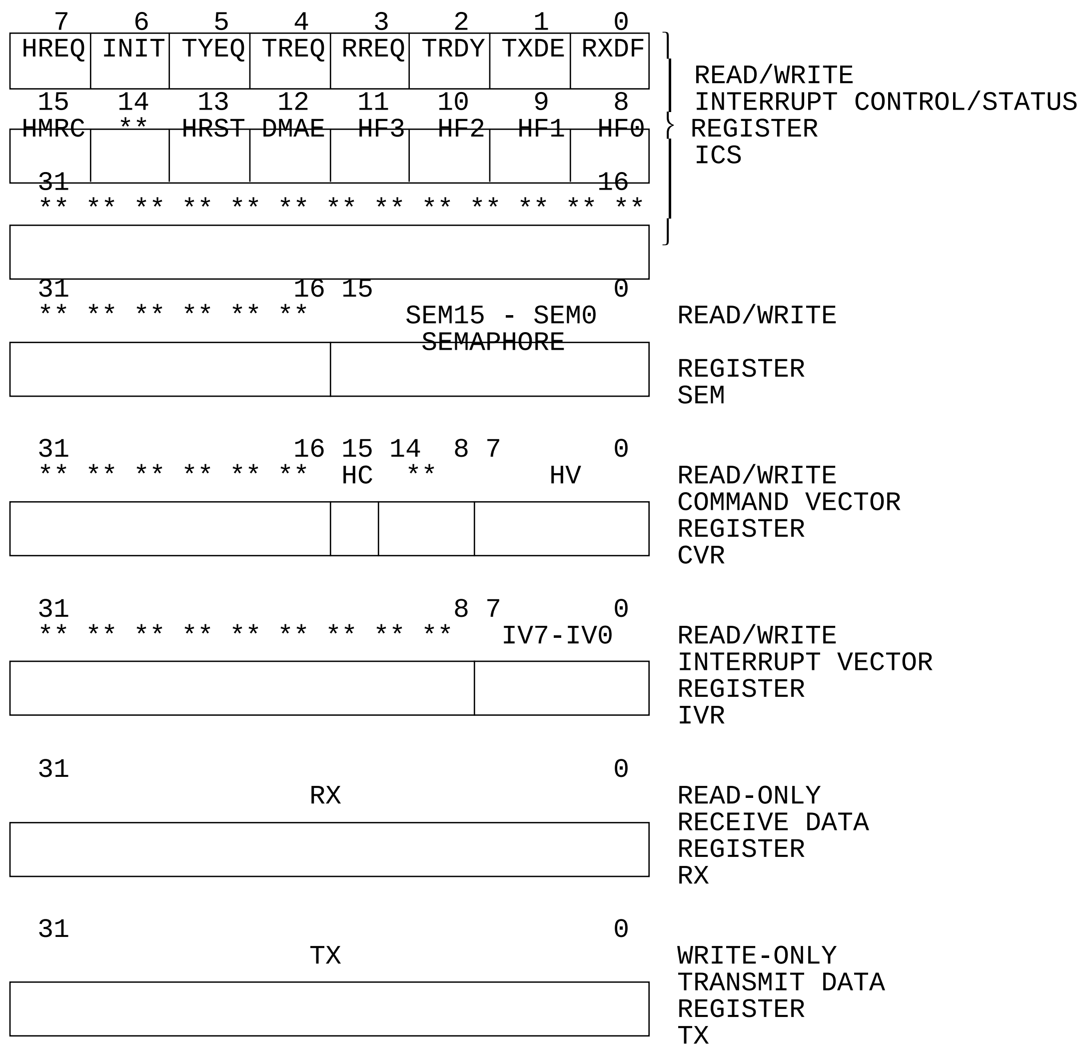
ADDR (HEX)	**DSP96002 READ**	**DSP96002 WRITE**	
X:$FFFFFFEC	HCRA	HCRA	PORT A
X:$FFFFFFED	HSRA	----	
X:$FFFFFFEE	----	HTXCA	
X:$FFFFFFEF	HRXA	HTXA	
X:$FFFFFFE4	HCRB	HCRB	PORT B
X:$FFFFFFE5	HSRB	----	
X:$FFFFFFE6	----	HTXCB	
X:$FFFFFFE7	HRXB	HTXB	

** - reserved, read as zero, should be written with zero for future compatibility.

Figure 7-10. HI - DSP96002 Programming Model

7.4.6 Host Transmit Data Register and HMRC Clear (HTXC) - DSP96002 Side

The Host Transmit register and HMRC Clear (HTXC) is used for DSP96002 to host processor data transfers in conjunction with "TX register write (address) and X/Y/P Memory Read (data) Interrupt" host functions. The HTXC register is viewed as a 32-bit write-only register by the DSP96002. Writing the HTXC register clears HTDE, HPRP, HXRP and HYRP. The HTXC register is transferred as 32-bit data to the Receive Register RX if both the HTDE bit and the Receive Data Full RXDF status bit are cleared. This transfer operation sets RXDF and HTDE, and clears HMRC (See Section 7.4.21.10).



** - reserved, read as zero, should be written with zero
for future compatibility.

Figure 7-11. HI - Host Processor Programming Model

7.4.7 Host Receive Data Register (HRX) - DSP96002 Side

The Host Receive Data register (HRX) is used for host processor to DSP96002 data transfers. The HRX register is viewed as a 32-bit read-only register by the DSP96002. The HRX register is loaded with 32-bit data from the TX register when both the Transmit Data Register Empty TXDE and Host Receive Data Full HRDF bits are cleared. This transfer operation sets TXDE and HRDF. The HRX register contains valid data when the HRDF bit is set. Reading HRX clears HRDF. The DSP96002 may program the HRIE bit to cause a Host Receive Data interrupt when HRDF is set.

$\overline{HR}$	$\overline{HA}$	$\overline{HS}$	$R/\overline{W}$	A5-A2	Host Function
x	1	1	x	xxxx	Host Interface disabled
x	1	0	1	1000	ICS register read
x	1	0	0	1000	ICS register write
x	1	0	1	1001	SEM register read
x	1	0	0	1001	SEM register write
x	1	0	1	1010	RX register read
x	1	0	0	1010	TX register write
x	1	0	x	1011	Reserved
x	1	0	1	1100	IVR register read
x	1	0	0	1100	IVR register write
x	1	0	1	1101	CVR register read
x	1	0	0	1101	CVR register write
x	1	0	x	1110	Reserved
x	1	0	x	1111	Reserved
x	1	0	0	0000	TX register write and Y Memory Write interrupt
x	1	0	0	0001	TX register write and Y Memory Read interrupt
x	1	0	0	0010	TX register write and X Memory Write interrupt
x	1	0	0	0011	TX register write and X Memory Read interrupt
x	1	0	0	0100	TX register write and P Memory Write interrupt
x	1	0	0	0101	TX register write and P Memory Read interrupt
x	1	0	0	011x	Reserved
x	1	0	1	0xxx	Reserved
0	0	x	1	xxxx	IVR read (DMAE=0) - 68K Interrupt Acknowledge
0	0	x	0	xxxx	Reserved (DMAE=0)
1	0	x	x	xxxx	Reserved (DMAE=0)
x	0	x	x	xxxx	RX read (DMA Mode: DMAE=1,TREQ=0,RREQ=1)
x	0	x	x	xxxx	TX write (DMA Mode: DMAE=1,TREQ=1,RREQ=0)
x	0	x	x	xxxx	Reserved (DMA Mode: DMAE=1,TREQ=0,RREQ=0)
x	0	x	x	xxxx	Reserved (DMA Mode: DMAE=1,TREQ=1,RREQ=1)

Figure 7-12. HI Functions

7.4.8 Host Control Register (HCR) - DSP96002 Side

The Host Control Register (HCR) is a 32-bit read/write control register used by the DSP96002 to control the HI interrupts and flags. HCR cannot be accessed by the host processor. HCR is a read/write register to allow the use of bit manipulation instructions on control register bits.

7.4.8.1 HCR Host Receive Interrupt Enable (HRIE) Bit 0

The Host Receive Interrupt Enable (HRIE) bit is used to enable the Host Receive Data interrupt when the Host Receive Data Full (HRDF) status bit in the Host Status register (HSR) is set. When HRIE is cleared, HRDF interrupts are disabled. When HRIE is set, the Host Receive Data interrupt request will occur if HRDF is set. HRIE is cleared by HW/SW reset.

HI Interrupt Sources (96002 side)

INTERRUPT SOURCE	STATUS	MASK	Exception Starting Address Port A	Exception Starting Address Port B
Receive Data Full	HRDF	HRIE	$00000020	$00000030
Transmit Data Empty	HTDE	HTIE	$00000022	$00000032
X Memory Read	HXRP	HXRE	$00000024	$00000034
Y Memory Read	HYRP	HYRE	$00000026	$00000036
P Memory Read	HPRP	HPRE	$00000028	$00000038
X Memory Write	HXWP	HXWE	$0000002A	$0000003A
Y Memory Write	HYWP	HYWE	$0000002C	$0000003C
P Memory Write	HPWP	HPWE	$0000002E	$0000003E
Host Command	HCP	HCIE	2*HV ($00000000-$000001FE)	

Host Processor $\overline{HR}$ Structure

$\overline{HR}$ SOURCE	STATUS	MASK
Receive Data Full	RXDF	RREQ
Transmit Data Empty	TXDE	TREQ
Transmitter Ready	TRDY	TYEQ

Figure 7-13. HI Interrupt Structure

7.4.8.2 HCR Host Transmit Interrupt Enable (HTIE) Bit 1

The Host Transmit Interrupt Enable (HTIE) bit is used to enable the Host Transmit Data interrupt when the Host Transmit Data Empty (HTDE) status bit in the Host Status Register (HSR) is set. When HTIE is cleared, HTDE interrupts are disabled. When HTIE is set, the Host Transmit Data interrupt request will occur if HTDE is set. HTIE is cleared by HW/SW reset.

7.4.8.3 HCR Host Command Interrupt Enable (HCIE) Bit 2

The Host Command Interrupt Enable (HCIE) bit is used to enable Host Command vectored DSP96002 interrupts when the Host Command Pending (HCP) status bit in the Host Status Register (HSR) is set. When HCIE is cleared, HCP interrupts are disabled. When HCIE is set, the Host Command interrupt request will occur if HCP is set. The starting address of this interrupt is determined by the Host Vector (HV). HCIE is cleared by HW/SW reset.

7.4.8.4 HCR Host Flag 2 (HF2) Bit 3

The Host Flag 2 (HF2) bit is used as a general purpose flag for DSP96002 to host processor communication. HF2 may be set or cleared by the DSP96002. HF2 Status can be read in the ICS register by the host processor. HF2 is cleared by HW/SW reset.

7.4.8.5 HCR Host Flag 3 (HF3) Bit 4

The Host Flag 3 (HF3) bit is used as a general purpose flag for DSP96002 to host processor communication. HF3 may be set or cleared by the DSP96002. HF3 Status can be read in the ICS register by the host processor. HF3 is cleared by HW/SW reset.

7.4.8.6 HCR Host Reset (HRES) Bit 5

The Host Reset (HRES) bit is used to reset the status bits of the HI and to initialize the transmit/receive paths to the same state produced by hardware or software reset. The HOST reset (Host Interface personal reset) is generated when HRES is set. The Host Interface exits the HOST reset state after this bit is cleared. HRES is set by HW/SW reset.

7.4.8.7 HCR Reserved bits (Bits 6, 7, 14-31)

These reserved bits read as zero and should be written with zero for future compatibility.

7.4.8.8 HCR Host P Memory Read Interrupt Enable (HPRE) Bit 8

The Host P Memory Read Interrupt Enable (HPRE) bit is used to enable the P Memory Read interrupt when the Host P Memory Read Command Pending (HPRP) status bit in the Host Status Register (HSR) is set. When HPRE is cleared, HPRP interrupts are disabled. When HPRE is set, the Host P Memory Read interrupt request will occur if HPRP is set. The starting address of this interrupt is shown in Figure 7-13. HPRE is cleared by HW/SW reset.

7.4.8.9 HCR Host P Memory Write Interrupt Enable (HPWE) Bit 9

The Host P Memory Write Interrupt Enable (HPWE) bit is used to enable the P Memory Write interrupt when the Host P Memory Write Command Pending (HPWP) status bit in the Host Status Register (HSR) is set. When HPWE is cleared, HPWP interrupts are disabled. When HPWE is set, the Host P Memory Write interrupt request will occur if HPWP is set. The starting address of this interrupt is shown in Figure 7-13. HPWE is cleared by HW/SW reset.

7.4.8.10 HCR Host X Memory Read Interrupt Enable (HXRE) Bit 10

The Host X Memory Read Interrupt Enable (HXRE) bit is used to enable the X Memory Read interrupt when the Host X Memory Read Command Pending (HXRP) status bit in the Host Status Register (HSR) is set. When HXRE is cleared, HXRP interrupts are disabled. When HXRE is set, the Host X Memory Read interrupt request will occur if HXRP is set. The starting address of this interrupt is shown in Figure 7-13. HXRE is cleared by HW/SW reset.

7.4.8.11 HCR Host X Memory Write Interrupt Enable (HXWE) Bit 11

The Host X Memory Write Interrupt Enable (HXWE) bit is used to enable the X Memory Write interrupt when the Host X Memory Write Command Pending (HXWP) status bit in the Host Status Register (HSR) is set. When HXWE is cleared, HXWP interrupts are disabled. When HXWE is set, the Host X Memory Write interrupt request will occur if HXWP is set. The starting address of this interrupt is shown in Figure 7-13. HXWE is cleared by HW/SW reset.

7.4.8.12 HCR Host Y Memory Read Interrupt Enable (HYRE) Bit 12

The Host Y Memory Read Interrupt Enable (HYRE) bit is used to enable the Y Memory Read interrupt when the Host Y Memory Read Command Pending (HYRP) status bit in the Host Status Register (HSR) is set. When HYRE is cleared, HYRP interrupts are disabled. When HYRE is set, the Host Y Memory Read inter-

rupt request will occur if HYRP is set. The starting address of this interrupt is shown in Figure 7-13. HYRE is cleared by HW/SW reset.

7.4.8.13 HCR Host Y Memory Write Interrupt Enable (HYWE) Bit 13

The Host Y Memory Write Interrupt Enable (HYWE) bit is used to enable the Y Memory Write interrupt when the Host Y Memory Write Command Pending (HYWP) status bit in the Host Status Register (HSR) is set. When HYWE is cleared, HYWP interrupts are disabled. When HYWE is set, the Host Y Memory Write interrupt request will occur if HYWP is set. The starting address of this interrupt is shown in Figure 7-13. HYWE is cleared by HW/SW reset.

7.4.9 Host Status Register (HSR) – DSP96002 Side

The Host Status register (HSR) is a 32-bit read-only status register used by the DSP96002 to interrogate status and flags of the HI. It cannot be directly accessed by the host processor.

7.4.9.1 HSR Host Receive Data Full (HRDF) Bit 0

The Host Receive Data Full (HRDF) bit indicates that the Host Receive Data register (HRX) contains data from the host processor, written by the host processor via the host function "TX register write" only. HRDF is set when the data is transferred from the TX register to the HRX register. HRDF is cleared when the Receive Data register HRX is read by the DSP96002. HRDF is cleared by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.9.2 HSR Host Transmit Data Empty (HTDE) Bit 1

The Host Transmit Data Empty (HTDE) bit indicates that the Host Transmit Data register (HTX) is empty and can be written by the DSP96002. HTDE is set when the HTX register is transferred to the RX register. HTDE is cleared when the Transmit Data register HTX is written by the DSP96002. HTDE is set by INIT (RREQ=1), HOST reset, and HW/SW reset.

7.4.9.3 HSR Host Command Pending (HCP) Bit 2

The Host Command Pending (HCP) bit indicates that the host processor has set the HC bit and that a Host Command Interrupt is pending. The HCP bit reflects the status of the HC bit in the Command Vector Register (CVR). HC and HCP are cleared by the DSP96002 exception hardware when the second vector location of the Host Command interrupt is fetched. HCP is cleared by HW/SW reset.

7.4.9.4 HSR Host Flag 0 (HF0) Bit 3

The Host Flag 0 (HF0) bit indicates the state of Host Flag 0 (HF0) in the Interrupt Control Register ICS. HF0 can only be changed by the host processor. HF0 is cleared by HW/SW reset.

7.4.9.5 HSR Host Flag 1 (HF1) Bit 4

The Host Flag 1 (HF1) bit indicates the state of Host Flag 1 (HF1) in the Interrupt Control Register ICS. HF1 can only be changed by the host processor. HF1 is cleared by HW/SW reset.

7.4.9.6 HSR Reserved bits (Bits 5, 6, 14-31)

These status bits are reserved for future expansion and read as zero during DSP96002 read operations.

7.4.9.7 HSR DMA Status (HDMA) Bit 7

The DMA Status bit (HDMA) indicates that the host processor has enabled the external DMA handshake mode of the HI. When HDMA is cleared, it indicates that the DMA Mode is disabled (DMAE=0) in the Interrupt Control Register ICS. When HDMA is set, it indicates that the DMA Mode is enabled (DMAE=1). Cleared by HW/SW reset.

7.4.9.8 HSR Host P Memory Read Command Pending (HPRP) Bit 8

The Host P Memory Read Command Pending (HPRP) bit indicates that the HRX register contains data from the host processor written by the host processor via the host function "TX register write and P Memory Read interrupt". HPRP is set when data is transferred from the TX register to the HRX register. HPRP is cleared when the HTXC register is written by the DSP96002. HPRP is cleared by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.9.9 HSR Host P Memory Write Command Pending (HPWP) Bit 9

The Host P Memory Write Command Pending (HPWP) bit indicates that the HRX and TX registers contain data from the host processor written by the host processor via the host function "TX register write and P Memory Write interrupt". HPWP is set when the host processor writes TX for the second time consecutively using this host function. HPWP is cleared when the HRX register is read twice consecutively (once for address and once for data) by the DSP96002. HPWP is cleared by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.9.10 HSR Host X Memory Read Command Pending (HXRP) Bit 10

The Host X Memory Read Command Pending (HXRP) bit indicates that the HRX register contains data from the host processor written by the host processor via the host function "TX register write and X Memory Read interrupt". HXRP is set when data is transferred from the TX register to the HRX register. HXRP is cleared when the HTXC register is written by the DSP96002. HXRP is cleared by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.9.11 HSR Host X Memory Write Command Pending (HXWP) Bit 11

The Host X Memory Write Command Pending (HXWP) bit indicates that the HRX and TX registers contain data from the host processor written by the host processor via the host function "TX register write and X Memory Write interrupt". HXWP is set when the host processor writes TX for the second time consecutively using this host function. HXWP is cleared when the HRX register is read twice consecutively (once for address and once for data) by the DSP96002. HXWP is cleared by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.9.12 HSR Host Y Memory Read Command Pending (HYRP) Bit 12

The Host Y Memory Read Command Pending (HYRP) bit indicates that the HRX register contains data from the host processor written by the host processor via the host function "TX register write and Y Memory Read

interrupt". HYRP is set when data is transferred from the TX register to the HRX register. HYRP is cleared when the HTXC register is written by the DSP96002. HYRP is cleared by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.9.13 HSR Host Y Memory Write Command Pending (HYWP) Bit 13

The Host Y Memory Write Command Pending (HYWP) bit indicates that the HRX and TX registers contain data from the host processor written by the host processor via the host function "TX register write and Y Memory Write interrupt". HYWP is set when the host processor writes TX for the second time consecutively using this host function. HYWP is cleared when the HRX register is read twice consecutively (once for address and once for data) by the DSP96002. HYWP is cleared by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.10 Receive Register (RX) - Host Processor Side

This 32-bit register receives data from the Host Transmit Data register HTX. The RX register contains valid data when the RXDF bit is set. The host processor may program the Receive Request Enable bit (RREQ), to assert the Host Request $\overline{\text{HR}}$ pin when RXDF is set. This informs the host processor that the Receive Registers RX is full. The RXDF bit is cleared by reading the RX register.

The RX register is viewed by the external host processor as an address in its memory map and may be read by a host processor memory read operation. The RX register may also be read by an external DMA controller (no A2-A5 address required) when the HI is in DMA mode (DMAE=1).

7.4.11 Transmit Register (TX) - Host Processor Side

This 32-bit register sends data to the Host Receive Data register HRX. The TX register contains valid data when the TXDE bit is cleared. The TXDE bit is cleared by writing the TX register. The host processor may program the Transmit Request Enable bit (TREQ) to assert the Host Request $\overline{\text{HR}}$ pin when TXDE is set. This informs the host processor that the TX register is empty.

The Transmit Register (TX) is viewed by the external host processor as address in its memory map and may be written by a host processor memory write operation. The TX register may also be written by an external DMA controller (no A2-A5 address required) when the HI is in DMA mode (DMAE=1).

7.4.12 Command Vector Register (CVR) - Host Processor Side

The 32-bit Host Command Vector Register (CVR) is used by the host processor to request a vectored exception service from the DSP96002. Any exception routine in the DSP96002 may be specified. The Host Command feature is independent of any of the data transfer mechanisms in the HI.

7.4.12.1 CVR Host Vector (HV) Bits 0-7

The eight bit Host Vector (HV) specifies the Host Command exception address indirectly. When the Host Command exception is recognized by the DSP96002 interrupt control logic, the starting address of the exception taken is 2*HV. This allows the host processor to change the exception starting address for the Host Command exception. The host processor can select any of the 256 possible exception routine starting addresses in the DSP96002 by writing the exception routine starting address divided by 2 into HV. This means

that the host processor can force any of the existing exception handlers (IRQA, IRQB, etc.) and can use any of the reserved or otherwise unused starting addresses provided they have been pre-programmed in the DSP96002. The HV is set to a predefined value for each port by HW/SW reset (see Figure 7-7). If HC is set, the host processor should not change HV.

7.4.12.2 CVR Reserved bits (Bits 8-14, 16-31)

Reserved bits are read by the host processor as zeros. They should be written with zero for future compatibility.

7.4.12.3 CVR Host Command (HC) Bit 15

The Host Command bit (HC) is used by the host processor to start execution of Host Command exceptions. Normally the host processor sets HC to request a Host Command exception service from the DSP96002. Setting HC causes HCP (Host Command Pending) to be set in the HSR register. When the Host Command second vector location is fetched, the HC bit is cleared by the HI hardware (interrupt acknowledge). HC is cleared by HW/SW reset.

CAUTION:

The host processor should verify that HC is cleared before attempting to set HC. This is necessary to avoid hardware contention between the host processor set operation and the Host Interface clear operation when receiving the interrupt acknowledge. HC should not be cleared by the host processor.

7.4.13 Interrupt Control/Status Register (ICS) - Host Processor Side

The Interrupt Control/Status Register (ICS) is a 32-bit read/write control and status register used by the host processor to control the HI and verify the current status of the HI. ICS is a read/write register which can be accessed using bit manipulation instructions. The control and status bits are described in the following paragraphs.

7.4.13.1 ICS Receive Data Register Full (RXDF) Bit 0

The read-only Receive Data Register Full (RXDF) bit indicates that the Receive Register RX contains data from the DSP96002 and may be read by the host processor. RXDF is set when the Host Transmit Data Register HTX or HTXC is transferred to the Receive Register RX. RXDF is cleared when RX is read by the host processor. RXDF is cleared by INIT (RREQ=1), HOST reset, and HW/SW reset.

RXDF may be used to assert the Host Request $\overline{HR}$ pin if the Receive Request Enable bit (RREQ) is set. RXDF provides valid status regardless of whether the RXDF interrupt is enabled or not so that polling techniques may be used by the host processor.

7.4.13.2 ICS Transmit Data Register Empty (TXDE) Bit 1

The read-only Transmit Data Register Empty (TXDE) bit indicates that the Transmit Register TX is empty and can be written by the host processor. TXDE is set when the Transmit Register TX is transferred to the Host Receive Data Register (HRX). TXDE is cleared when TX is written by the host processor. TXDE is set by INIT (TREQ=1), HOST reset, and HW/SW reset.

TXDE may be used to assert the Host Request $\overline{HR}$ pin if the Transmit Request Enable bit (TREQ) is set. TXDE provides valid status regardless of whether the TXDE interrupt is enabled or not so that polling techniques may be used by the host processor.

7.4.13.3 ICS Transmitter Ready (TRDY) Bit 2

The read-only Transmitter Ready (TRDY) status bit indicates that both the Transmit Register TX (on the host processor side) and Host Receive Data Register HRX (on the DSP96002 side) are empty. TRDY may be used to assert the Host Request $\overline{HR}$ pin if the Transmitter Ready Request Enable bit (TYEQ) is set. TRDY provides valid status regardless of whether the TRDY interrupt is enabled or not so that polling techniques may be used by the host processor. TRDY is set by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.13.4 ICS Receive Request Enable (RREQ) Bit 3

RREQ is used to enable host processor interrupts/requests via the external Host Request $\overline{HR}$ pin when the Receive Data Register Full (RXDF) status bit is set. When RREQ is cleared, RXDF interrupts are disabled. When RREQ is set, the Host Request $\overline{HR}$ pin will be asserted if RXDF is set.

In DMA Mode (DMAE=1), RREQ must be set or cleared by software to select the direction of DMA transfers. Setting RREQ defines the direction of DMA transfer to be DSP96002 $\rightarrow$ external DMA, and enables the $\overline{HR}$ pin to request these data transfers.

See Figure 7-15 and Figure 7-16 for a summary of the effect of RREQ on the $\overline{HR}$ pin. RREQ is cleared by HW/SW reset.

7.4.13.5 ICS Transmit Request Enable (TREQ) Bit 4

TREQ is used to enable host processor interrupt/requests via the Host Request $\overline{HR}$ pin when the Transmit Data Register Empty (TXDE) status bit is set. When TREQ is cleared, TXDE interrupts are disabled. When TREQ is set, the Host Request $\overline{HR}$ pin will be asserted if TXDE is set.

In DMA Mode (DMAE=1), TREQ must be set or cleared by software to select the direction of DMA transfers. Setting TREQ defines the direction of DMA transfer to be from external DMA→96002, and enables the $\overline{\text{HR}}$ pin to request these data transfers.

See Figure 7-15 and Figure 7-16 for a summary of the effect of TREQ on the $\overline{\text{HR}}$ pin. TREQ is cleared by HW/SW reset.

7.4.13.6 ICS Transmitter Ready Request Enable (TYEQ) Bit 5

TYEQ is used to enable interrupts via the Host Request $\overline{\text{HR}}$ pin when the Transmitter Ready (TRDY) status bit is set. When TYEQ is cleared, TRDY interrupts are disabled. When TYEQ is set, the Host Request $\overline{\text{HR}}$ pin will be asserted if TRDY is set.

See Figure 7-15 for a summary of the effect of TYEQ on the $\overline{\text{HR}}$ pin. TYEQ is cleared by HW/SW reset. In DMA Mode (DMAE=1), TYEQ must be cleared.

7.4.13.7 ICS Initialize (INIT) Bit 6

The INIT bit is used by the host processor to force initialization of the HI hardware. This may or may not be necessary, depending on the software design of the interface.

To correctly initialize the HI, set the INIT bit with the other control bits in ICS which determine the initialization procedure (TREQ, RREQ). All bits may be written in the same command. After setting the INIT bit, the HI starts the initialize procedure, and at the end of the procedure, the HI clears the INIT bit. During the initialize procedure, the host processor should not attempt to read RX, write TX or write the ICS register. The host processor should first ensure that the Initialize procedure has completed, using one of the following techniques:

1. When using the $\overline{\text{HR}}$ pin for handshake, wait until $\overline{\text{HR}}$ is asserted and then start writing/reading data.

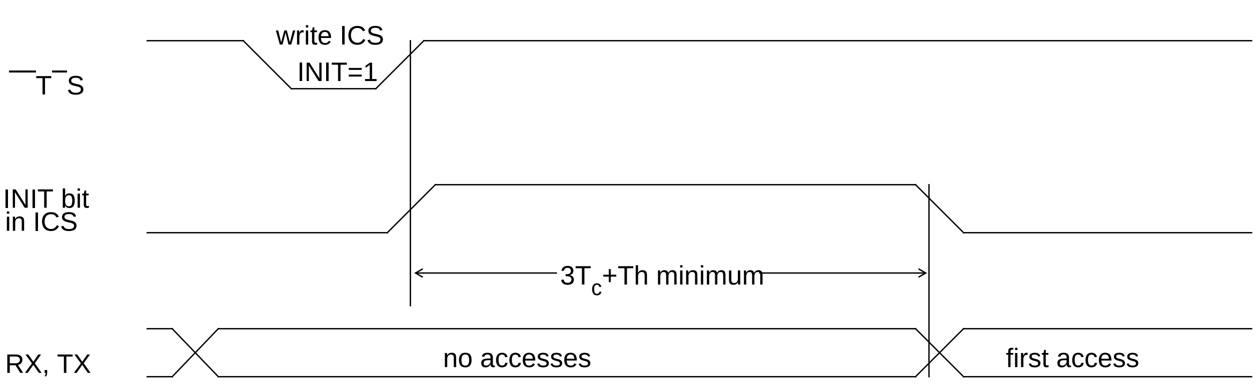


Figure 7-14. Minimum Delay to Ensure Correct INIT Execution

2. When not using the $\overline{HR}$ pin for handshake, use polling of the INIT bit in ICS to make sure it is cleared by the hardware (which means the INIT execution is completed). Then, start writing/reading data.

3. If using neither the $\overline{HR}$ pin for handshake nor polling the INIT bit, wait at least 3Tc+Th after the deassertion of $\overline{TS}$ that wrote ICS, before writing/reading data. This ensures that the INIT is completed. See Figure 7-14.

The type of initialization done depends on the state of TREQ and RREQ. If both TREQ and RREQ are cleared, the INIT procedure will not affect the HI. The effect of the initialization procedure is described in Figure 7-7 and Figure 7-8. The INIT bit is cleared by HW/SW reset.

CAUTION:

The host processor should verify that INIT is cleared before attempting to set INIT. This is necessary to avoid hardware contention between the host processor set operation and the Host Interface clear operation at the end of the INIT procedure. INIT should not be cleared by the host processor.

7.4.13.8 ICS Host Request (HREQ) Bit 7

The read-only Host Request (HREQ) bit indicates the status of the Host Request $\overline{HR}$ pin.

In interrupt mode (DMAE=0):

When the HREQ status bit is cleared, it indicates that the $\overline{HR}$ pin is deasserted and host processor interrupts are not being requested. When the HREQ status bit is set, it indicates that the $\overline{HR}$ pin is asserted indicating that the DSP96002 is interrupting the host processor. The HREQ interrupt request may originate from one or more of 3 sources, selected by their enable bits RREQ, TREQ and TYEQ (See Figure 7-15):

- the RX register or HTX register is full,
- the TX register or HRX register is empty,
- both the TX register (on the host processor side) and the HRX register (on the DSP96002 side) are empty.

In DMA Mode (DMAE=1):

When the HREQ status bit is cleared, it indicates that the $\overline{HR}$ pin is deasserted and no DMA transfers are being requested. When the HREQ status bit is set, it indicates that the $\overline{HR}$ pin is asserted and a DMA transfer request is being made. The DMA transfer request may originate because the Receive Register (RX) is full when the DMA transfer direction is DSP96002 → external DMA, or because the Transmit Register (TX) is empty when the DMA transfer direction is external DMA → DSP96002 (See Figure 7-16).

The condition of RX full and TX empty is indicated by the ICS register RXDF and TXDE status bits, respectively. If the interrupt source has been enabled by the associated request enable bit in the Interrupt Control Register ICS, HREQ will be set if one or more of the 2 enabled interrupt sources is set. HREQ is cleared by HW/SW reset. HREQ is cleared by HOST reset if both TYEQ and TREQ are cleared, and set otherwise. For the effect of INIT on HREQ, see Figure 7-7.

TREQ	RREQ	TYEQ	HREQ flag and $\overline{HR}$ pin
0	0	0	No interrupts (polling).
0	1	0	RX full or HTX full.
1	0	0	TX empty or HRX empty.
1	1	0	RX full, HTX full, TX empty or HRX empty.
x	0	1	TX empty and HRX empty.
x	1	1	All interrupts (no polling).

Figure 7-15. HREQ and $\overline{HR}$ Definition - Interrupt Mode (DMAE=0)

TREQ	RREQ	TYEQ	HREQ flag and $\overline{HR}$ pin
0	0	0	Reserved
0	1	0	DSP96002→DMA Request (RX full)
1	0	0	DMAÆ→DSP96002 Request (TX empty)
1	1	0	Reserved
x	x	1	Reserved

Figure 7-16.
HREQ, $\overline{HR}$ and DMA Transfer Direction Definition - DMA Mode (DMAE=1)

7.4.13.9 ICS Host Flag 0 (HF0) Bit 8

The Host Flag 0 (HF0) bit is used as a general purpose flag for host processor to DSP96002 communication. HF0 may be set or cleared by the host processor. HF0 is cleared by HW/SW reset. The status of HF0 can be read in the HSR, bit 3.

7.4.13.10 ICS Host Flag 1 (HF1) Bit 9

The Host Flag 1 (HF1) bit is used as a general purpose flag for host processor to DSP96002 communication. HF1 may be set or cleared by the host processor. HF1 is cleared by HW/SW reset. The status of HF1 can be read in the HSR, bit 4.

7.4.13.11 ICS Host Flag 2 (HF2) Bit 10

The read-only Host Flag 2 (HF2) bit indicates the state of Host Flag 2 (HF2) in the Host Control Register HCR. HF2 can only be changed by the DSP96002. HF2 is cleared by HW/SW reset.

7.4.13.12 ICS Host Flag 3 (HF3) Bit 11

The read-only Host Flag 3 (HF3) bit indicates the state of Host Flag 3 (HF3) in the Host Control Register HCR. HF3 can only be changed by the DSP96002. HF3 is cleared by HW/SW reset.

7.4.13.13 ICS DMA Mode Enable (DMAE) Bit 12

The DMA Mode Enable bit (DMAE) selects the mode of operation of the HI. When DMAE is set, the HI operates in the DMA Mode. When DMAE is cleared, the DMA Mode is disabled. Cleared by HW/SW reset.

When DMAE is cleared, the HI registers are selected by address lines A2-A5. This mode of operation is appropriate for interfacing with external devices, such as a microprocessor, that are able to supply address-

es. In this mode, the $\overline{\text{HR}}$ pin can be used as an interrupt request to the host processor, and the $\overline{\text{HA}}$ pin may be used to support a 68K family interrupt acknowledge.

When DMAE is set, the HI operates in the DMA Mode. When in DMA Mode, the RX and TX registers are accessed without regard to the address lines A2-A5, permitting data transfers under control of external devices, such as DMA controllers, that do not supply addresses. The $\overline{\text{HR}}$ pin is used as a DMA transfer request to the external DMA controller. The direction of the DMA transfer is selected by TREQ and RREQ. Bidirectional DMA transfers are not supported; the user cannot set both RREQ and TREQ in the DMA mode. Also, TYEQ should remain cleared.

7.4.13.14 ICS Host Reset Status (HRST) Bit 13

The read-only Host Reset Status bit (HRST) may be tested by the host processor to verify the state of the HRES control bit. If HRST is set, the HRES bit is set and the HI is in the reset state. If the HRST bit is cleared, the HRES bit is cleared and the HI operation is enabled. The HRST bit is cleared by clearing HRES. The HRST bit is set by HOST reset and HW/SW reset.

7.4.13.15 ICS Reserved bits (Bits 14, 16-31)

Reserved bits are read by the host processor as zero. They should be written with zero for future compatibility.

7.4.13.16 ICS Host Memory Read Command (HMRC) Bit 15

The read-only Host Memory Read Command status bit (HMRC) may be tested by the host processor to verify when data written to the HTXC register (96002 side) is transferred to the RX register (host processor side).

HMRC is set when the host processor writes into the TX register using the host function "TX register write and X/Y/P Memory Read Interrupt". HMRC is cleared when the HTX register contents which were written, in the DSP96002 side, thorough the HTXC address, are transferred to the RX register in the host processor side. HMRC is cleared by INIT (TREQ=1), HOST reset, and HW/SW reset.

7.4.14 Semaphore Register (SEM) - Host Processor Side

The Semaphore Register (SEM) is a 32-bit read/write register used by the host processor to control the HI allocation in a multiprocessor system and show the current host processor ID.

7.4.14.1 SEM Host Semaphore (SEM0-SEM15) Bits 0-15

The Host Semaphore register bits SEM0-SEM15 are used by host processors for software arbitration of mastership over the HI. This register does not affect the HI operation and only serves as a read/write semaphore repository. All external host processors that compete for mastership over the HI should work according to the same software protocol for handing over the HI from one host processor to another.

Typically, a host processor, before accessing the HI, checks the Semaphore Register to see if the HI is allocated to another host processor. If SEM0-SEM15 are not cleared then the HI is already allocated and the host processor cannot access the HI. If SEM0-SEM15 are cleared then the HI is assumed free and the host processor writes SEM0-SEM15. The host processor can either set just one bit (which will serve as a host

busy semaphore bit), several bits or write the whole 16 bits (which, for example, may be used as host processor ID).

Host processors should use read/modify/write uninterruptable instructions (such as XMEM in the MC88000, CAS in the MC680x0, or BSET in the DSP96002) and examine which host processor has allocated the HI or set the semaphore bit by "bit test and set" instructions. The BSET in the DSP96002 is "uninterruptable" in that it tests the semaphore bit and indicates the results in the status register and then sets the semaphore bit without relinquishing the bus. This combined operation prevents another processor from reading or writing the semaphore bit between the BSET testing and setting operations.

After the present HI "owner" has completed its transfers, it must release the HI (if there are other potential masters capable of host transfers) by clearing the Semaphore Register bits. SEM0-SEM15 are cleared by HW/SW reset.

7.4.14.2 SEM Reserved bits (Bits 16-31)

Reserved bits are read by the host processor as zeros. They should be written with zero for future compatibility.

7.4.15 Interrupt Vector Register (IVR) - Host Processor Side

The Interrupt Vector Register (IVR) is a 32-bit read/write register which contains the exception vector number for use with MC680x0 processor family vectored interrupts.

7.4.15.1 IVR Interrupt Vector (IVR0-IVR7) Bits 0-7

When not in DMA Mode (DMAE=0), the contents of the IVR register may be read to the data bus by asserting $\overline{TS}$ when both $\overline{HR}$ and $\overline{HA}$ are asserted. The contents of the IVR register are initialized to $0F during HW/SW reset. This corresponds to the un-initialized exception vector in the MC68K family.

The IVR register may also be accessed by the host processor as a regular read/write register using the address lines A2-A5 as shown in Figure 7-12.

7.4.15.2 IVR Reserved Bits – Bits 8-31

The upper 24-bits are reserved and are read by the host processor as zeros. They should be written with zero for future compatibility.

7.4.16 HI Interrupts

The HI may request interrupt service from either the DSP96002 core or the external host processor.

The HI interrupt requests to the DSP96002 core are internal and do not require the use of an external interrupt pin. The DSP96002 core services HI interrupts by fetching the appropriate interrupt vector locations (see Section 8). The interrupt service routine must read or write the appropriate HI register to clear the interrupt request (reading HRX to clear HRDF for example). In the case of Host Command interrupts, the interrupt acknowledge from the DSP96002 core, generated when the second interrupt vector location is fetched, will clear the pending interrupt condition.

The HI interrupt requests to the external host processor use the Host Request $\overline{\text{HR}}$ pin. $\overline{\text{HR}}$ is normally connected to a host processor interrupt input. The host processor acknowledges HI interrupts by executing an interrupt service routine. The MC680x0 processor family will assert the $\overline{\text{TS}}$ pin when both $\overline{\text{HR}}$ and $\overline{\text{HA}}$ are asserted to read the exception vector number from the IVR register of the HI. In a multi-DSP96002 system, the HREQ bit in the Interrupt Status Register (ICS) may be tested to determine which DSP96002 HI is the interrupting device and the RXDF, TXDE and TRDY bits may then be tested to determine the interrupt source. The host processor interrupt service routine must read or write the appropriate HI register to clear the interrupt and deassert $\overline{\text{HR}}$.

7.4.17 Host Processor Programmer Considerations

7.4.17.1 Reading RX

When reading the Receive register RX, the host processor programmer should use interrupts or poll the RXDF flag which indicates that data is available. This guarantees that the data in the RX register will be stable.

7.4.17.2 Writing TX

The host processor programmer should not write to the Transmit register TX unless the TXDE bit is set, indicating that the TX register is empty. This guarantees that the HI will transfer stable data to the HRX register on the DSP96002 side.

7.4.17.3 Synchronization of Status Bits from DSP96002 to Host Processor

HC, HMRC, HREQ, HF3, HF2, TRDY, TXDE, and RXDF status bits are set or cleared from the DSP96002 side of the HI and read by the host processor. The host processor is able to read these status bits without regard to the clock rate used by the DSP96002, but there is a chance that the state of the bit could be changing during the read operation. This is generally not a system problem, since, if the bit is changing, the read will indicate that another poll should be taken and the bit will be read correctly in the next pass of the polling routine.

The only potential system problem with the uncertainty of reading any status bits by the Host is when HF3 and HF2 are being used as an encoded pair. For example, if the DSP96002 changes HF3 and HF2 from "00" to "11" there is a very small probability that the host processor could read the bits during the transition and receive "01" or "10" instead of "11". If the combination of HF3 and HF2 has significance, it is recommended that the HF3 and HF2 bits be read twice and checked for consensus.

7.4.17.4 Writing the Host Vector Register

The host processor programmer should change the Host Vector register only when the Host Command bit (HC) is cleared. Clearing HC is a DSP96002 HI task and should not be done by the host programmer. This guarantees that the DSP96002 interrupt control logic will receive a stable vector.

7.4.18 96002 Programmer Considerations

7.4.18.1 Reading Status Bits

HF1, HF0, HCP, HPRP, HPWP, HXRP, HXWP, HYRP, HYWP, HTDE, and HRDF status bits are set or cleared by the host processor side of the HI. These bits are individually synchronized to the DSP96002 clock.

The only system problem with reading status is HF1 and HF0 if they are encoded as a pair, e.g. the four combinations 00, 01, 10, and 11 each have significance. This is because there is a very small probability that the DSP96002 will read the status bits that were synchronized during transition. The solution to this potential problem is to read the bits twice for consensus.

7.4.19 DSP96002 to DSP96002 Data Transfers - Examples

This section presents examples showing the use of the HI and the on-chip DMA Controller for data transfers between two DSP96002 processors. The bus master accesses the slave's HI using regular memory references. The slave's HI registers are memory mapped into the bus master memory space. Note that the bus master HI is not used and that the slave's HI is not in the DMA Mode (DMAE=0).

7.4.19.1 Data Write Using The On-Chip DMA Controllers

This example outlines the steps that a DSP96002 bus master, behaving as host processor, transfers data to a DSP96002 bus slave, thorough the slave's HI. The on-chip DMA Controllers of both DSP96002 proces-

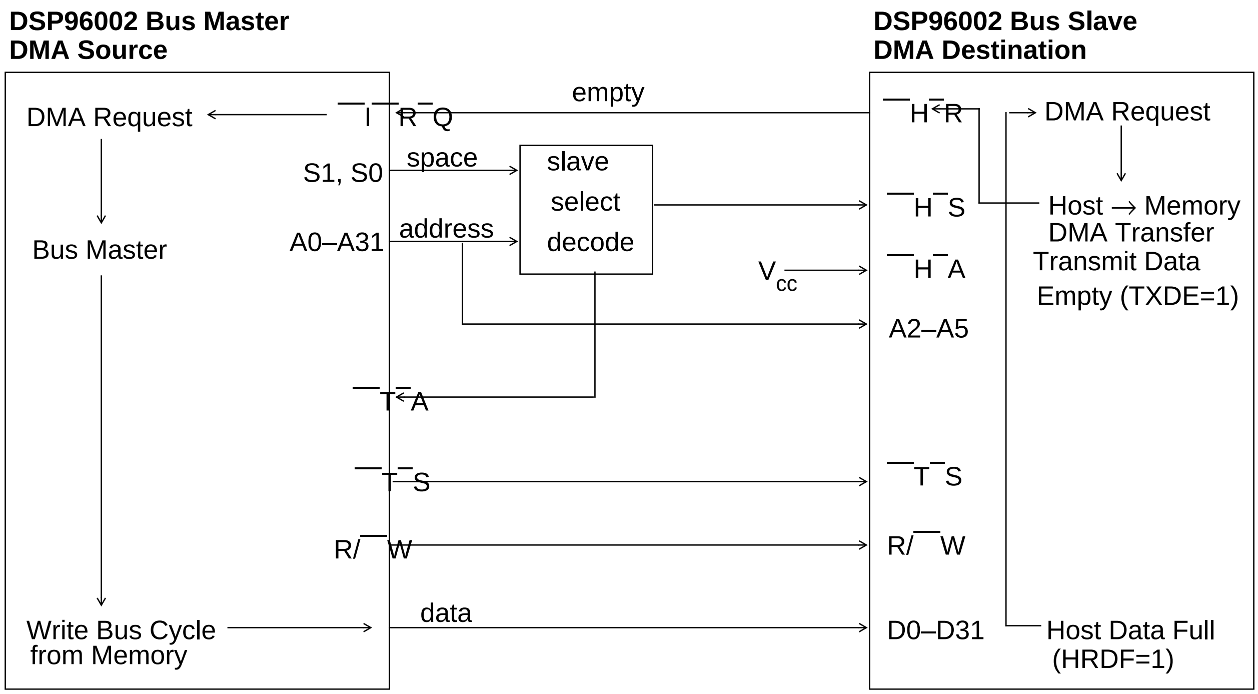


Figure 7-17. DSP96002 to DSP96002 Data Write

sors are used to transfer data without interfering with the local processing in both chips. Figure 7-17 contains a diagram showing the data paths and control lines used for the data transfers.

A data write transfer is initiated when the slave's $\overline{HR}$ signal is asserted, indicating that its HI TX register is empty and ready to receive a data word from the master. The $\overline{HR}$ signal is connected to an $\overline{IRQ}$ pin in the master where this pin is defined as a DMA service request input. When $\overline{HR}$ is asserted, the master DMA Controller transfers the data word from the master's memory to an external address selecting the TX register in the slave's HI as destination. After TX is written (negating $\overline{HR}$), the data is transferred by the HI to the HRX register, setting HRDF and TXDE. Setting TXDE causes $\overline{HR}$ to be asserted if TREQ is set. In the slave's DMA Controller, HRDF is defined as a DMA service request signal. When HRDF is asserted, the slave's DMA Controller initiates a data transfer from HRX to the slave memory, completing the data transfer.

7.4.19.2 Data Read Using The On-Chip DMA Controllers

This example outlines the steps that a DSP96002 bus master, behaving as host processor, transfers data from a DSP96002 bus slave, thorough the slave's HI. The on-chip DMA Controllers of both DSP96002 processors are used to transfer data without interfering with the local processing in both chips. Figure 7-18 contains a diagram showing the data paths and control lines used for the data transfers.

A data read transfer is initiated when the slave's $\overline{HR}$ signal is asserted, indicating that its HI RX register is full and the data is ready to be read by the master. $\overline{HR}$ is connected to an $\overline{IRQ}$ pin in the master

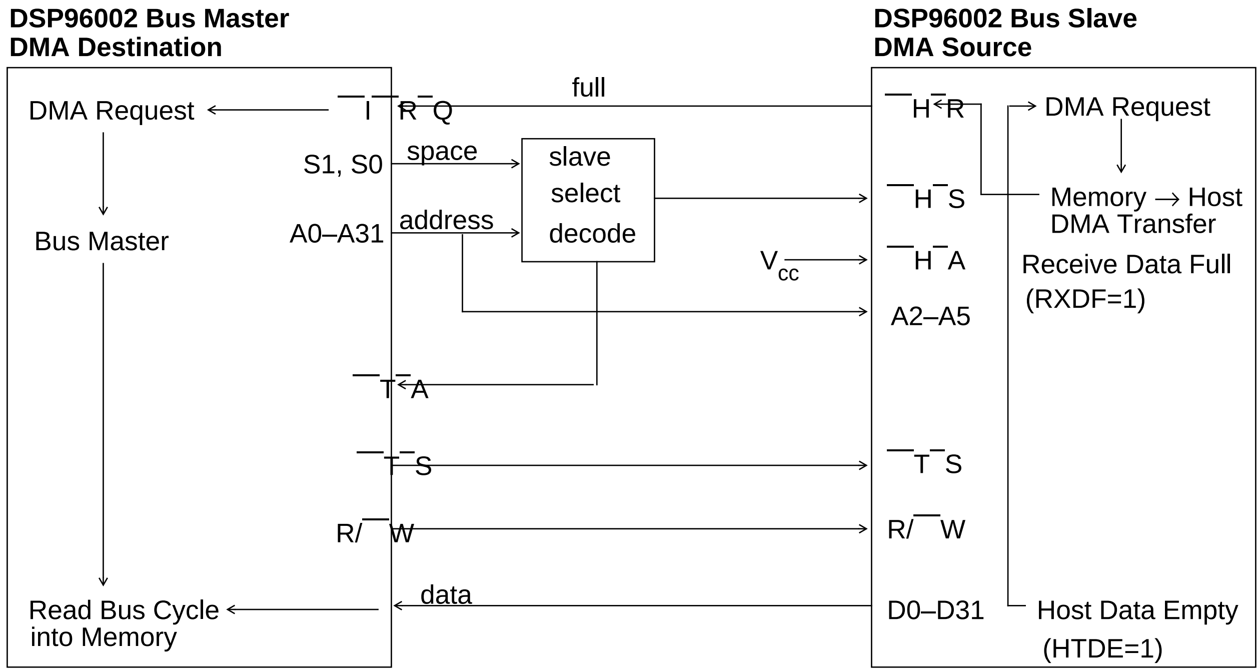


Figure 7-18. DSP96002 to DSP96002 Data Read

where this pin is defined as a DMA service request input. When $\overline{HR}$ is asserted, the master DMA Controller transfers the data word from the external address that selects the RX register in the slave's HI to a master memory location. After RX is read (negating $\overline{HR}$), the HI may transfer the next data word from the HI HTX register, setting HTDE and RXDF. Setting RXDF causes $\overline{HR}$ to be asserted if RREQ is set. In the slave's DMA Controller, HTDE is defined as a DMA service request signal. When HTDE is asserted, the slave's DMA Controller initiates a data transfer from the slave memory to the HTX register, keeping the register full for further data transfers.

7.4.20 External DMA Controller to DSP96002 Data Transfers - Examples

This section presents examples showing the use of the HI and the on-chip DMA Controller for data transfers between a DSP96002 and an external DMA Controller. The external DMA Controller is the bus master and the DSP96002 is the bus slave. The external DMA Controller accesses the DSP96002 HI without supplying an address to select a HI register. Note that the HI is programmed to work in the DMA Mode (DMAE=1).

7.4.20.1 Data Write Using the DSP96002 On-Chip DMA Controller

This example outlines the steps that an external DMA Controller, the bus master, takes to transfer data to a DSP96002 bus slave, thorough the slave's HI. The on-chip DMA Controller of the DSP96002 is used to locally transfer data between the HI and the DSP96002 memory without interfering with core processing. The TREQ and RREQ bits in the ICS register must be programmed to define the direction of data transfer as being from the external DMA Controller to the HI (TREQ=1, RREQ=0). The TYEQ bit in the ICS register

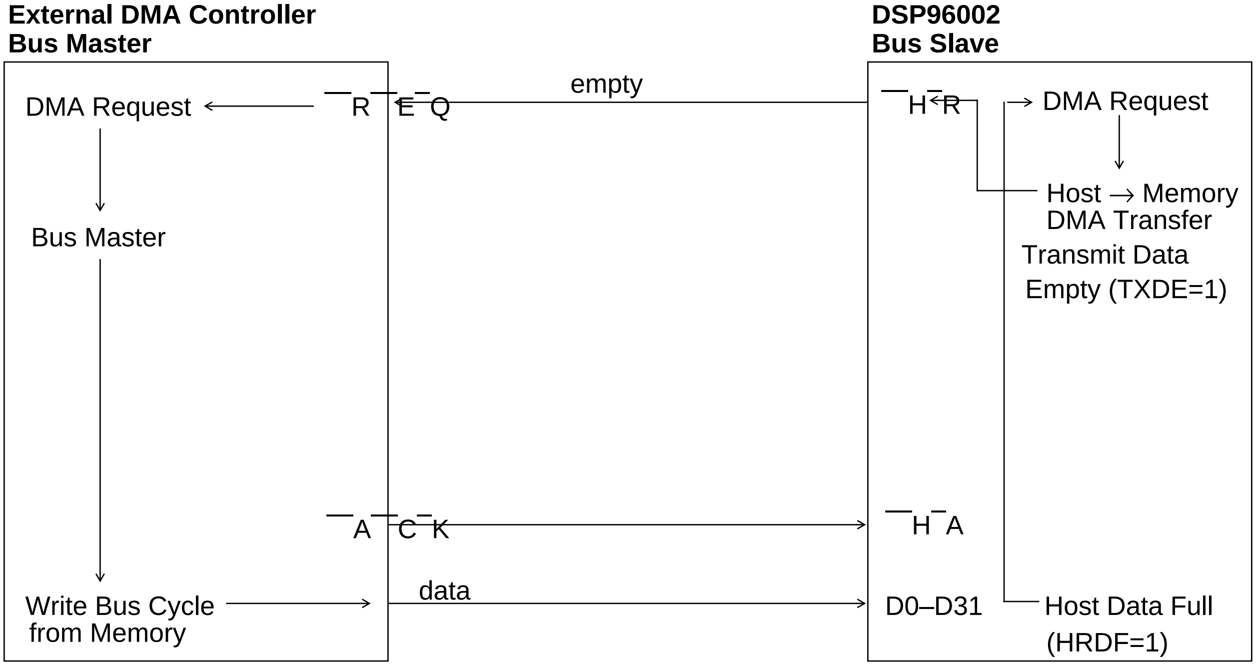


Figure 7-19. External DMA to DSP96002 Data Write

should be cleared. Figure 7-19 contains a diagram showing the data paths and control lines used for the data transfers.

A data write transfer is initiated when the slave's $\overline{HR}$ signal is asserted, indicating that its HI TX register is empty and ready to receive a data word from the master. $\overline{HR}$ is connected to a $\overline{REQ}$ pin in the master which is a DMA service request input. When $\overline{HR}$ is asserted, the external DMA Controller transfers the data word from memory to the TX register in the HI. The TX register is written by asserting $\overline{HA}$ and TREQ=1 and RREQ=0. After TX is written (negating $\overline{HR}$), the data is transferred by the HI to the HRX register, setting HRDF and TXDE. Setting TXDE causes $\overline{HR}$ to be asserted since TREQ is set. In the slave's on-chip DMA Controller, HRDF is defined as a DMA service request signal. When HRDF is set, the slave's on-chip DMA Controller initiates a data transfer from HRX to the slave memory, completing the data transfer.

7.4.20.2 Data Read Using the DSP96002 On-Chip DMA Controller

This example outlines the steps that an external DMA Controller, the bus master, takes to transfer data from a DSP96002 bus slave, thorough the slave's HI. The on-chip DMA Controller of the DSP96002 is used to locally transfer data between the HI and the DSP96002 memory without interfering with core processing. The TREQ and RREQ bits in the ICS register must be programmed to define the direction of data transfer as being from the HI to the external DMA Controller (TREQ=0, RREQ=1). The TYEQ bit in the ICS register should be cleared. Figure 7-20 contains a diagram showing the data paths and control lines used for the data transfers.

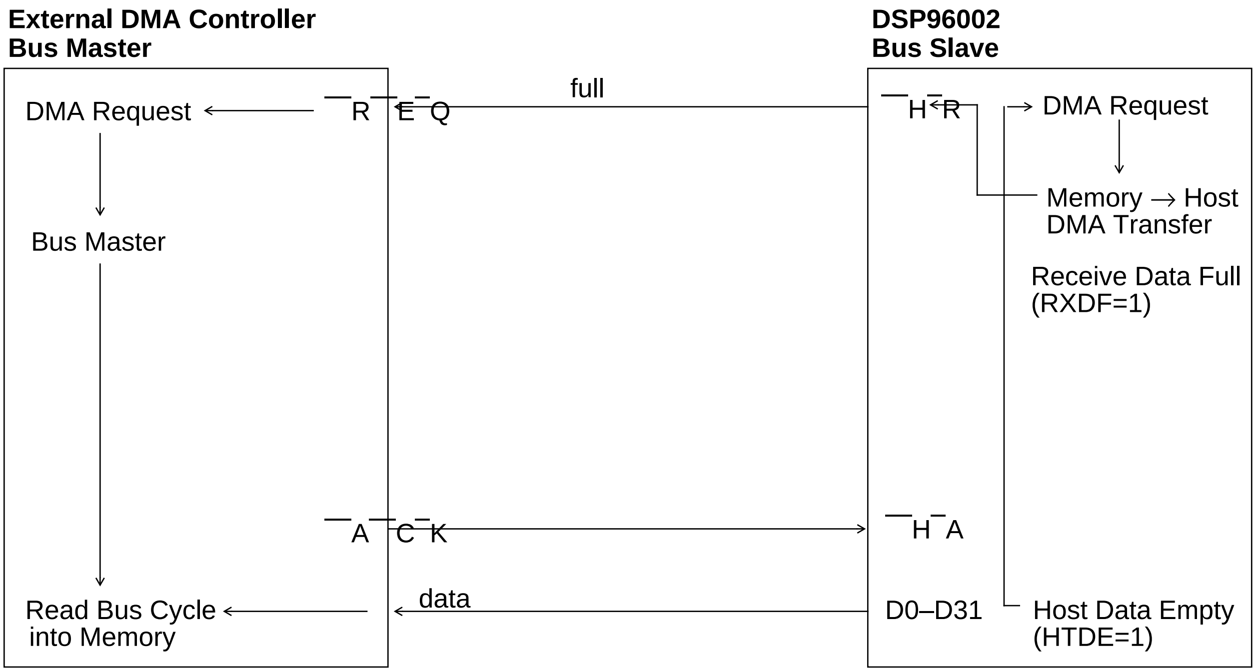


Figure 7-20. DSP96002 to External DMA Data Read

A data read transfer is initiated when the slave's $\overline{HR}$ signal is asserted, indicating that its HI RX register is full and the data is ready to be read by the external DMA Controller. $\overline{HR}$ is connected to a $\overline{REQ}$ pin in the master which is a DMA service request input. When $\overline{HR}$ is asserted, the external DMA Controller transfers the data word from the RX register in the slave's HI to a memory location. The RX register is read by asserting $\overline{HA}$ and TREQ=0 and RREQ=1. After RX is read (negating $\overline{HR}$), the HI may transfer the next data word from the HI HTX register, setting HTDE and RXDF. Setting RXDF causes $\overline{HR}$ to be asserted since RREQ is set. In the slave's on-chip DMA Controller, HTDE is defined as a DMA service request signal. When HTDE is asserted, the slave's on-chip DMA Controller initiates a data transfer from the slave memory to the HTX register, keeping the register full for further data transfers.

7.4.21 HI Performance Analysis and Programming Examples

The following host programming examples show the software needed to support Master-Slave transfers between two DSP96002s. Master processor load, the minimal transfer cycle, and the overhead are estimated. These estimates can vary depending on the addressing mode. In most cases the fastest addressing mode possible was used. Also, it was assumed that the master processor did not loose the bus in the middle of host activity. The HI registers are accessed by the host processor with 0 wait states.

7.4.21.1 Semaphore Control

Whenever a host transfer is to be executed, the host processor must first obtain ownership of the slave's HI. This is done by semaphore control. The following is an example of code used by the host processor to obtain ownership of the HI. The LSB bit of the SEM register is used as a semaphore bit:

```
                                                     clock
                                        words        cycles
SEMA     BSET     #0,Y:SEMR              1             4
         JCS      SEMA                   1             4
         start host activity
         .
         .
         .
         end of host activity
         BCLR      #0,Y:SEMR             1             4
```

The BSET instruction tests the semaphore bit and then sets the bit before releasing the bus. If (1) the bit was already set when tested, the slave is being used by another master and this master enters a loop waiting for the other master to finish and clear the semaphore bit. If (2) the bit was zero when tested, the slave was available and the master can continue to access the slave. Setting the bit with the BSET instruction signals other masters that the slave is now unavailable. After completing the host activity, the current master clears the semaphore bit to allow other masters to access this slave. The minimal overhead for one host transfer is 3 program words and 12 clock cycles. This procedure is not necessary when there can be only one bus master.

7.4.21.2 Host Command Register Read

In this example, both master and slave are DSP96002s. HCVR points to the address of the selected slave CVR register ($\overline{HS}$=0, $\overline{HA}$=1, A5-A2=1101). The master executes the following instruction:

```
         MOVE     Y:HCVR,R0
```

7.4.21.3 Host Command Register Write

In this example, both master and slave are DSP96002s. HCVR points to the slave CVR register ($\overline{HS}$=0, $\overline{HA}$=1, A5-A2=1101). It is recommended to verify, before initiating the Host Command, that the previous host command has been executed (HC bit is cleared). The master executes the following instructions:

```
HCMD     BRSET    15,Y:HCVR,HCMD           ;testing of HC
         MOVE     R0,Y:HCVR
```

7.4.21.4 ICS Register Read

HICSR points to the slave ICS register ($\overline{HS}$=0, $\overline{HA}$=1, A5-A2=1000). The master executes the following instruction:

```
MOVE        Y:HICSR,R0
```

7.4.21.5 ICS Register Write

HICSR points to the slave ICS register ($\overline{HS}$=0, $\overline{HA}$=1, A5-A2=1000). The master executes the following instruction:

```
MOVE        R0,Y:HICSR
```

7.4.21.6 68K Interrupt Acknowledge Sequence

The MC680x0 interrupt acknowledge sequence is as follows:

1. When there is a pending interrupt the 68K must first determine the starting location of the interrupt service routine. The 68K supports the acquisition of this information with the interrupt acknowledge cycle.
2. The 68K interrupt controller generates in response the IACK signal to the interrupting device (96K in this case), which is connected to the 96K $\overline{HA}$ pin.
3. The interrupting device places the vector number on the bus in response to IACK signal from the interrupt controller.

Figure 7-21 shows a flowchart of 68K interrupt acknowledge sequence.

7.4.21.7 IVR Register Read

In this example, the master and slave are two DSP96002s. HIVR points to the slave IVR register ($\overline{HS}$=0, $\overline{HA}$=1, A5-A2=1100). The master executes the following instruction:

```
MOVE        Y:HIVR,R0
```

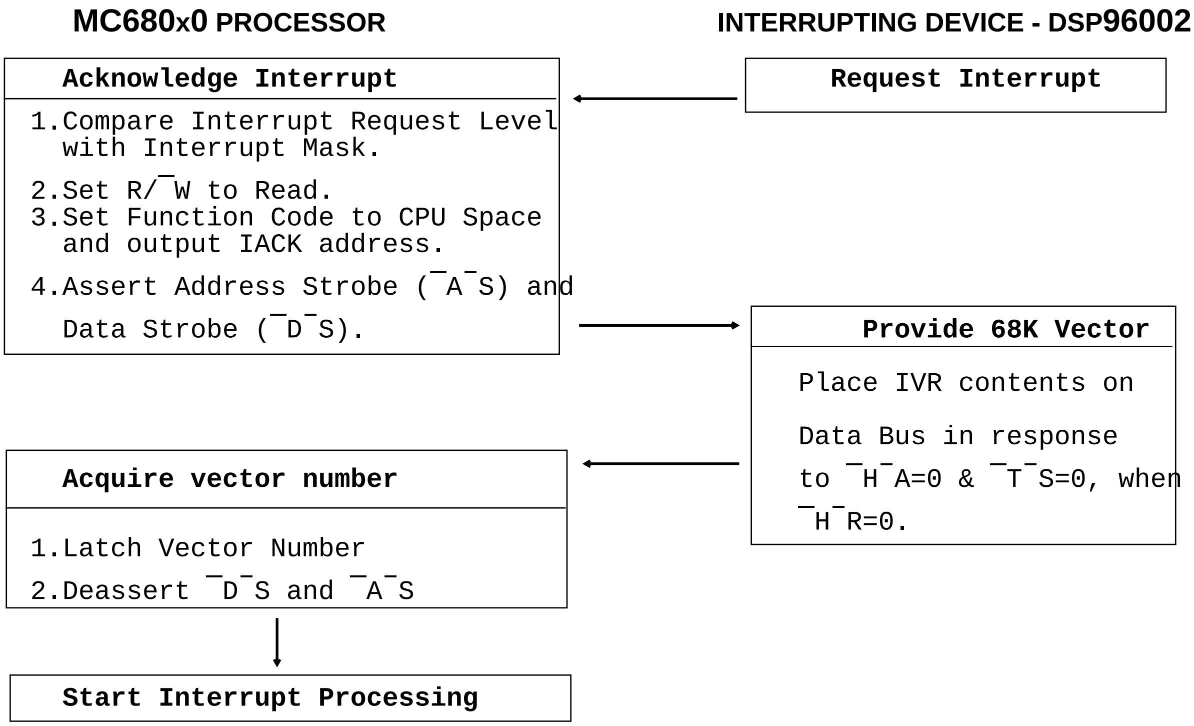


Figure 7-21. 68K Interrupt Acknowledge Sequence

7.4.21.8 68K Interrupt Register Write

HIVR points to the slave IVR register ($\overline{HS}$=0, $\overline{HA}$=1, A5-A2=1100). The master executes the following instruction:

```
MOVE      R0,Y:HIVR
```

7.4.21.9 X/Y/P Memory Write Procedure

The X/Y/P Memory Write procedure enables the host processor to write a data word D into an arbitrary address A located in the DSP96002 memory space. The host processor must execute the following steps:

1. Verify that TX is empty (TXDE=1).
2. Write A into the TX register using the host function "TX register write and X/Y/P Memory Write Interrupt". If HRX is empty, the HI then transfers A to HRX automatically.
3. Verify that TX is empty (TXDE=1).
4. Write D into the TX register using the host function "TX register write and X/Y/P Memory Write Interrupt". This second write initiates the X/Y/P Memory Write interrupt.

5. In the DSP96002 side, the X/Y/P Memory Write interrupt vector should point to a routine that first reads HRX to get the address A, stores A in an address pointer Rn, and then again reads HRX to retrieve the data D and store D into the DSP96002 memory location pointed by Rn.
6. The host processor may test TRDY to see if both A and D were removed from the input double buffer (TX/HRX).

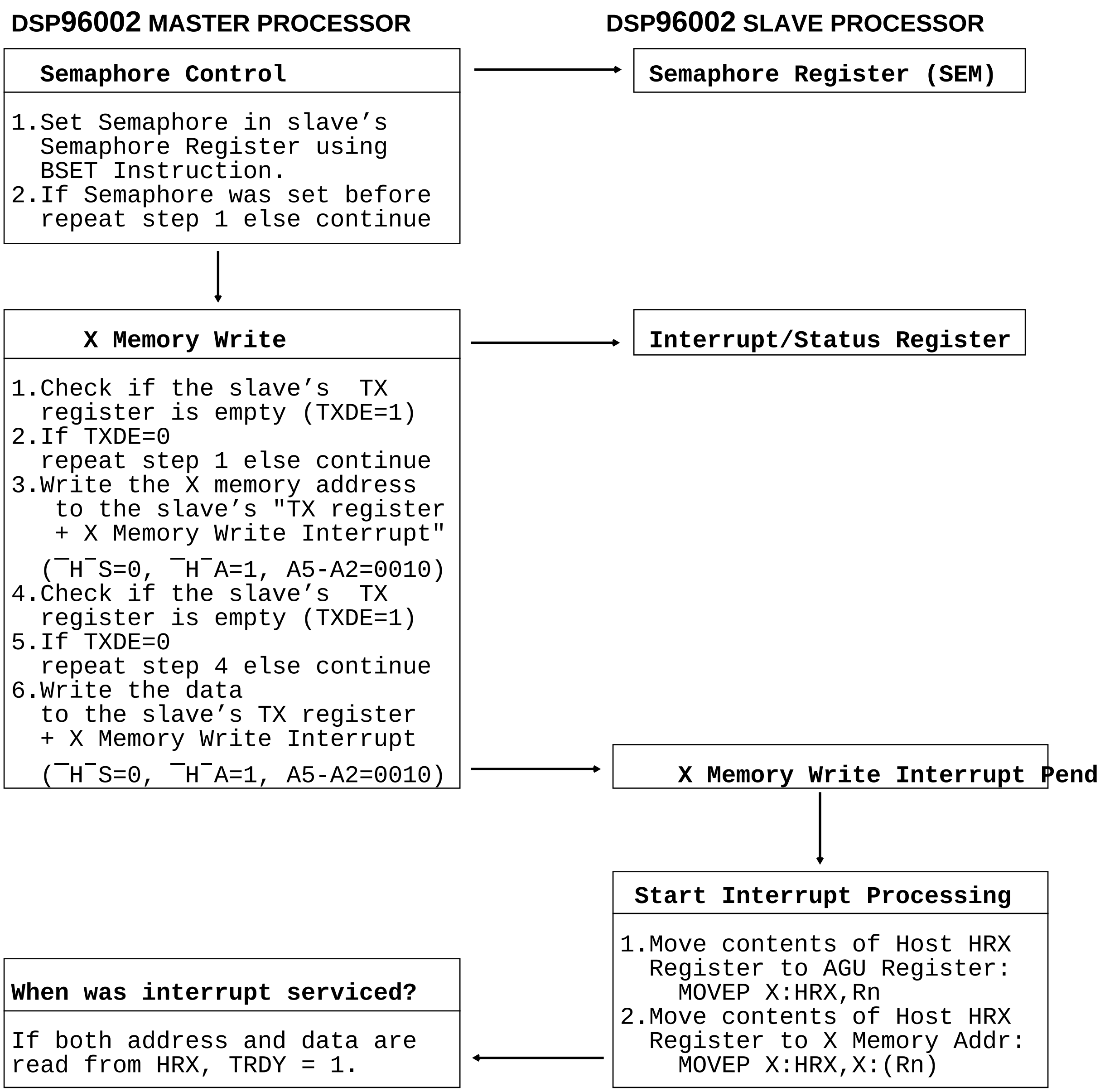


Figure 7-22. X Memory Write Procedure

Figure 7-22 shows a flowchart for X Memory Write.

The following code is executed by the master processor. The R3 register contains the address needed for selecting the "TX register write and X Memory Write interrupt" host function in the slave HI, as defined in Figure 7-12. The R4 register contains the address needed for reading the ICS register of the slave HI. The R1 register contains the target X memory address. The R0 register contains the data to be written to the target X memory address. The master executes the following instructions:

```
                                                   clock
                                      words        cycles
_LOOP1   JCLR     #TXDE,X:(R4),_LOOP1   2             6
         MOVE     R1,X:(R3)             1             2
_LOOP2   JCLR     #TXDE,X:(R4),_LOOP2   2             6
         MOVE     R0,X:(R3)            _1_           _2_
                                        6            16
```

The minimal memory write is 6 program words and 16 clock cycles. The second move triggers the X Memory Write interrupt request in the slave. The interrupt service routine in the slave takes 10-14 clock cycles to execute. If there are other interrupts with higher priority the response to this interrupt may be delayed.

A somewhat faster procedure may be employed by ensuring that sufficient time has elapsed after the writing the address to TX before writing the data to eliminate testing for TXDE=1 as above:

```
                                                   clock
                                      words        cycles
_LOOP    JCLR     #TRDY,X:(R4),_LOOP    2             6
         MOVE     R1,X:(R3)             1             2
         NOP                            1             2
         MOVE     R0,X:(R3)            _1_           _2_
                                        5            12
```

This procedure requires 5 program words and 12 clock cycles. The NOP instruction provides the necessary elapse time between two consecutive TX writes if both master and slave processors are being fed the same clock frequency and duty cycle, otherwise a second NOP instruction should be added to the above code.

7.4.21.10 X/Y/P Memory Read Procedure

The X/Y/P Memory Read procedure enables the host processor to read a data word D from an arbitrary address A located in the DSP96002 memory space. The host processor must execute the following steps:

1. Verify that TX is empty (TXDE=1).
2. Write A into the TX register using the host function "TX register write and X/Y/P Memory Read Interrupt". This sets HMRC. If HRX is empty, the HI then transfers A to HRX automatically and initiates the X/Y/P Memory Read interrupt.
3. In the DSP96002 side, the X/Y/P Memory Read interrupt vector should point to a routine that first reads HRX to get the address A, stores A in an address pointer Rn, reads the memory location pointed to by Rn, and stores the data D in the HTX register using the HTXC address. The data D passes to the RX register (host processor side), HMRC is cleared and RXDF is set (this may assert $\overline{HR}$).

4. The host processor polls the ICS register until HMRC is cleared and then reads the data D from the RX register.

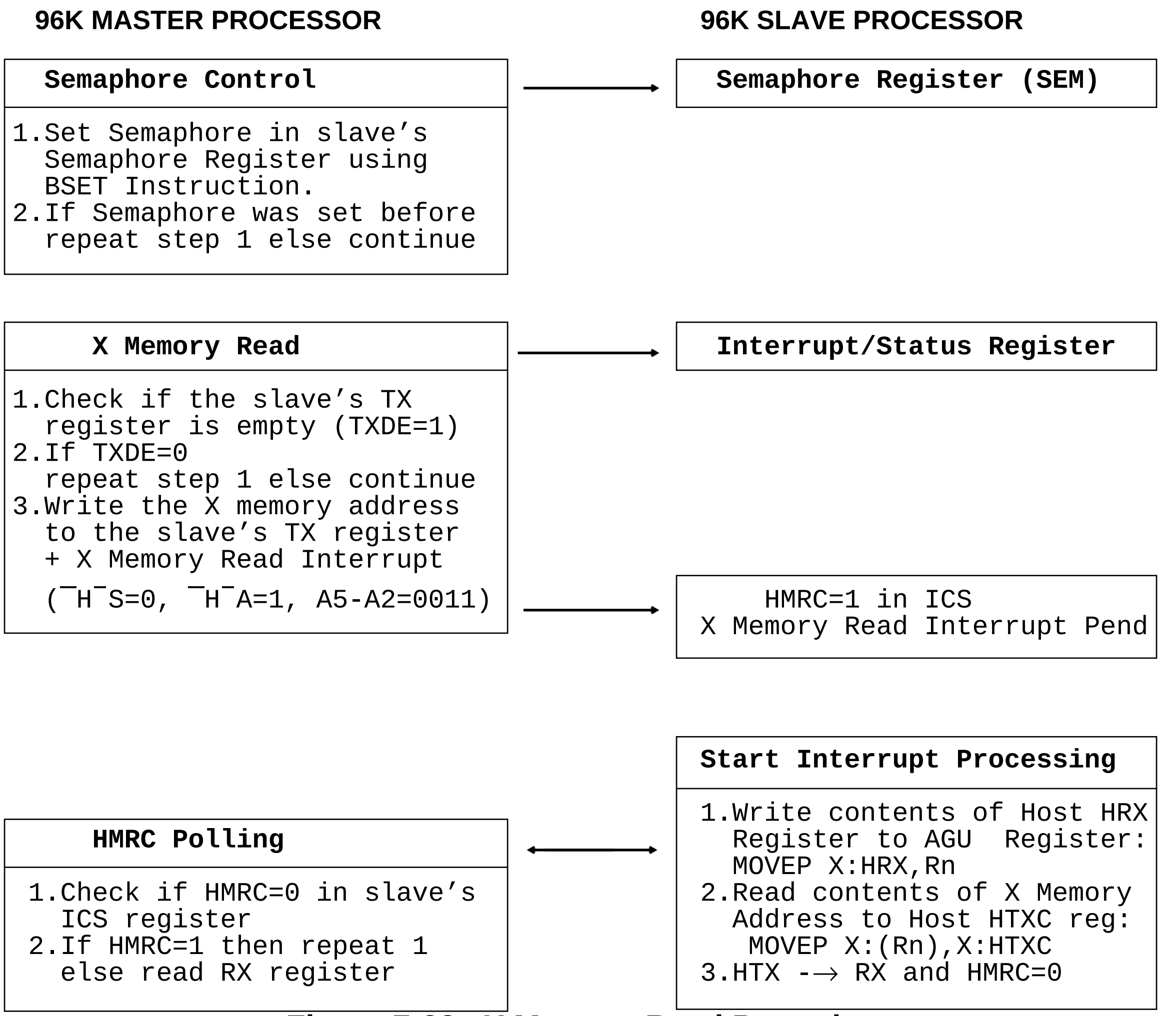


Figure 7-23. X Memory Read Procedure

Figure 7-23 shows a flowchart for X Memory Read.

Following is the code executed by the master processor. The R3 register contains the address needed for selecting the "TX register write and X Memory Read interrupt" host function in the slave HI, as defined in Figure 7-12. The R4 register contains the address needed for reading the ICS register of the slave HI. The R1 register contains the target X memory address. The R2 register contains the address needed for reading the RX register of the slave HI. The required data word is finally stored in D0.s. The master executes the following instructions:

```
                                               clock
                                      words    cycles
_LOOP1   JCLR    #TXDE,X:(R4),_LOOP1    2        6
         MOVE    R1,X:(R3)              1        2

_LOOP2   JCLR    #HMRC,X:(R4),_LOOP2    2        6
         MOVE    X:(R2),D0.S           _1_      _2_
                                        6       16
```

The minimal memory read procedure is 6 program words and 16 clock cycles. The first move triggers the X Memory Read interrupt request in the slave. The interrupt service routine in the slave takes 8-12 clock cycles to execute. If there are other interrupts with higher priority the response to this interrupt may be delayed. Only then can the master continue with the second move to read the data.

7.4.21.11 DSP96002 to DSP96002 Transfers Using On-Chip DMA Controllers

Data transfers done by the on-chip DMA Controllers do not require intervention by the core. Since the DMA has dedicated internal data paths and internal memory slots, no penalty is imposed on execution time of the core processing. However, there is overhead associated with the initialization procedure for the on-chip DMA channels. The following initialization steps have to be done:

1. The master verifies that the slave DMA channel is free by reading the DMA channel Control/Status register. This can be done using the X Memory Read procedure. If the DMA channel is dedicated to this transfer, this step may be bypassed.
2. The master initializes the slave DMA channel. This can done by X Memory Write procedures or more efficiently using a predefined Host Command. If repetitive DMA transfers of data blocks to a predefined address region will be done, the Host Command routine will contain just two instructions: enable DMA channel and load the DMA Counter register.
3. The master initializes its own DMA channel.
4. The master initializes the slave HI.

The entire initialization process may take from less than 12 cycles up to more than a hundred cycles.

For example, in block DMA transfers in a linear array of 96Ks (transferring data only in one direction to fixed predefined addresses), the initialization procedure may be executed only once. Each DMA block transfer will demand just a DMA enable (bit set) and DMA Counter load for both master and slave processors. This may be done in 8-12 cycles using fast interrupts.

The initialization process for system configuration with one master and N slaves is not much longer. If the master makes "constant" DMA transfers then it may have N predefined interrupts while each slave DMA has fixed control register setup. In this case, initialization may be done in less than 20 cycles.

7.5 DMA CONTROLLER

7.5.1 Introduction

The Direct Memory Access (DMA) Controller is an on-chip device that permits data transfers between any two locations in any combination of memory spaces, without intervention of the DSP96002 core. Due to dedicated DMA buses and dual-access internal memories, a high level of isolation is achieved where the DMA operation does not interfere or slow down the core operation. The DMA Controller has two channels, each with its own register set. The DMA Controller registers are read/write registers memory-mapped in the internal I/O memory space (the highest 128 locations in X memory).

The table in Figure 7-24 shows the data transfers that the DMA Controller is capable of. The number of cycles specified in the Figure 7-24 notes are for the operation of one channel using a continuous block transfer.

DMA data transfers			Notes
Int. mem	⟷	Int. mem (different memory space)	#1
Int. mem	⟷	Int. mem (same memory space)	#2
Ext. mem	⟷	Int. mem (different memory space)	#1
Ext. mem	⟷	Int. mem (same memory space)	#2
Ext. mem	⟷	Ext. mem	#3
Int. mem	⟷	Int. I/O (different memory space)	#1
Int. mem	⟷	Int. I/O (same memory space)	#2
Ext. mem	⟷	Int. I/O (different memory space)	#1
Ext. mem	⟷	Int. I/O (same memory space)	#2
Int. I/O	⟷	Int. I/O	#2

Figure 7-24. Direction of DMA Data Transfers

Notes:

1. Two clock cycles for every word.
2. Four clock cycles for every word (the same address bus is used for source and destination).
3. Four clock cycles for every word.

7.5.2 DMA Controller Programming Model

The registers comprising the DMA Controller are shown in Figure 7-25 and Figure 7-26.

Bits 31–0	Register
	DMA Source Modifier Register DSM0 addr X:$FFFFFFDF
	DMA Source Address Register DSR0 addr X:$FFFFFFDE
	DMA Source Offset Register DSN0 addr X:$FFFFFFDD
	DMA Destination Modifier Register DDM0 addr X:$FFFFFFDB
	DMA Destination Address Register DDR0 addr X:$FFFFFFDA
	DMA Destination Offset Register DDN0 addr X:$FFFFFFD9
	DMA Counter DCO0 addr X:$FFFFFFDC

DMA Control/Status Register
DCS0
addr X:$FFFFFFD8

31	30	29	28	27	26	25	24
DE	DIE	*	DTD	*	DTM1	DTM0	DMAP
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
DCP	*	*	*	*	*	*	*
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
*	M6	M5	M4	M3	M2	M1	M0
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
*	*	DSS2	DSS1	DSS0	DDS2	DDS1	DDS0

Figure 7-25. DMA Controller Programming Model - Channel 0

31 — 0	
	DMA Source Modifier Register DSM1 addr X:$FFFFFFD7
	DMA Source Address Register DSR1 addr X:$FFFFFFD6
	DMA Source Offset Register DSN1 addr X:$FFFFFFD5
	DMA Destination Modifier Register DDM1 addr X:$FFFFFFD3
	DMA Destination Address Register DDR1 addr X:$FFFFFFD2
	DMA Destination Offset Register DDN1 addr X:$FFFFFFD1
	DMA Counter DCO1 addr X:$FFFFFFD4

DMA Control/Status Register
DCS1
addr X:$FFFFFFD0

31	30	29	28	27	26	25	24
DE	DIE	*	DTD	*	DTM1	DTM0	DMAP

23	22	21	20	19	18	17	16
DCP	*	*	*	*	*	*	*

15	14	13	12	11	10	9	8
*	M6	M5	M4	M3	M2	M1	M0

7	6	5	4	3	2	1	0
*	*	DSS2	DSS1	DSS0	DDS2	DDS1	DDS0

Figure 7-26. DMA Controller Programming Model - Channel 1

7.5.3 DMA Control/Status Register (DCS)

The DMA Control/Status Register (DCS) is a 32-bit read/write register that controls the DMA operation. Each bit is described in the following paragraphs.

7.5.3.1 DCS DMA Destination Space Control (DDS2-DDS0) Bits 0,1,2

The DMA Destination Space control bits (DDS2-DDS0) specify the memory or I/O space that will be referenced as destination by the DMA. The DDS2-DDS0 bits are cleared by Hardware and Software Reset.

DDS2	DDS1	DDS0	DMA Destination Memory Space
0	0	0	Internal Program Memory
0	0	1	Internal X Data Memory
0	1	0	Internal Y Data Memory
0	1	1	Internal I/O (X Memory Space)
1	0	0	External Program Memory
1	0	1	External X Data Memory
1	1	0	External Y Data Memory
1	1	1	External I/O (Y Memory Space)

7.5.3.2 DCS DMA Source Space Control (DSS2-DSS0) Bits 3,4,5

The DMA Source Space control bits (DSS2-DSS0) specify the memory or I/O space that will be referenced as source by the DMA. The DSS2-DSS0 bits are cleared by Hardware and Software Reset.

DSS2	DSS1	DSS0	DMA Source Memory Space
0	0	0	Internal Program Memory
0	0	1	Internal X Data Memory
0	1	0	Internal Y Data Memory
0	1	1	Internal I/O (X Memory Space)
1	0	0	External Program Memory
1	0	1	External X Data Memory
1	1	0	External Y Data Memory
1	1	1	External I/O (Y Memory Space)

7.5.3.3 DCS Reserved Bits (Bits 6, 7, 15-22, 27, 29)

These bits read as zero and should be written with zero for future compatibility.

7.5.3.4 DCS DMA Request Masks (M0-M6) Bits 8-14

The DMA Request mask bits select the source of DMA requests used to trigger DMA transfers. If a mask bit is set, the corresponding device is selected as the DMA request source. If the mask bit is cleared, the device is ignored. The DMA request sources may be the internal peripherals or external devices requesting service through the $\overline{IRQA}$, $\overline{IRQB}$ and $\overline{IRQC}$ pins. The external inputs behave as edge-triggered synchronous inputs. The mask bits are cleared by Hardware and Software Reset. The internal DMA request sources are produced by ANDing the internal peripheral status bits with DE.

Each requesting device input is first individually ANDed with its respective mask bit (M0,M1,etc) and then all AND outputs are ORed together. The OR output goes to the edge-triggered latch whose output initiates

the DMA transfer. If an input is unmasked, asserting that input will set the latch and initiate a DMA transfer. The DMA state machine clears the latch when accessing the DMA source address. If more than one requesting device input is enabled, the first edge on any input is latched and triggers a DMA transfer, and any other edge that appears before the latch is cleared will be ignored.

DMA Request Mask Bit	Requesting Device
M0	External ($\overline{IRQA}$ pin)
M1	External ($\overline{IRQB}$ pin)
M2	External ($\overline{IRQC}$ pin)
M3	Port A Host Receive Data (HRDF=1)
M4	Port A Host Transmit Data (HTXE=1)
M5	Port B Host Receive Data (HRDF=1)
M6	Port B Host Transmit Data (HTXE=1)

7.5.3.5 DCS DMA Channel Priority (DCP) Bit 23

The DMA Channel Priority (DCP) bit contains the priority level of the DMA channel relative to the other DMA channel. When DMA transfers are pending, the DMA Channel Priority of both channels are compared to decide which channel will be activated. This decision must be made since both channels use common resources such as the DMA ALU, and the address buses. DCP is cleared by Hardware and Software Reset.

If both channels have the same priority then the channels will be active in a round-robin fashion: Channel 0 will be activated to transfer a single data word, followed by Channel 1.

If the channel priorities are different, the channel with highest priority will start executing DMA transfers and will remain doing so as long as there are DMA transfers pending. In the event that the lower priority channel is executing DMA transfers when the higher priority channel receives a transfer request, the lower priority channel will finish the transfer of the current data word and arbitration will again occur.

DCP	DMA Channel Priority
0	Priority 0
1	Priority 1

7.5.3.6 DCS DMA Priority (DMAP) Bit 24

This bit permits setting the DMA priority relative to the core when an external bus access is required. The priority determines, in case of contention between the core and the DMA Controller, whether the DMA will wait or not. If DMAP is cleared, then the DMA will wait until a free slot is available on the external bus. If DMAP is set, the core cycle will be stretched and both core and DMA will access during the same cycle. DMAP is cleared by Hardware and Software Reset.

DMAP	External Access Priority
0	Core
1	Equal

7.5.3.7 DCS DMA Transfer Mode – (DTM1–DTM0) Bits 25,26

DMA Transfer Mode bits (DTM1-DTM0) specify the mode of operation of the DMA channel. DTM1-DTM0 are cleared by Hardware and Software Reset.

When DTM1-DTM0=00, a single block is transferred, the length of the block is determined by the counter, the transfer is initiated by setting the DE bit, and the transfer is completed when the counter decrements to zero.

When DTM1-DTM0=01, a single block is transferred, the length of the block is determined by the counter, the transfer is initiated by the first DMA request after DE is set to 1, and the transfer is completed when the counter decrements to zero.

When DTM1-DTM0=10, a single block is transferred, the length of the block is determined by the counter, each DMA request will transfer a single word while DE=1, and the transfer is completed when the counter decrements to zero.

When DTM1-DTM0=11, a single word is transferred each time a DMA request is received while DE=1. The counter is ignored in this mode.

DTM1	DTM0	Transfer Mode
0	0	Single Block, Trig. by DE Bit, DMA Request Ignored
0	1	Single Block, Trig. by First DMA Request
1	0	Single Block, Word Transfer Trig. by DMA Request
1	1	Single Word, Triggered by DMA Request

7.5.3.8 DCS DMA Transfer Done Status (DTD) Bit 28

The read-only DMA Transfer Done Status bit is set when the last word during a Single Block transfer is stored in the destination, stopping DMA operation. At the same time, DE will be cleared. The last transfer is defined as the one where the DMA Counter reaches zero, or the transfer being done when the DE bit is cleared by the core. If DIE is set (DMA Interrupt enabled), then DTD=1 will cause a DMA interrupt request. When the DMA Interrupt is disabled (DIE=0), the core may verify the DMA status by polling this bit. DTD is set by Hardware and Software Reset. DTD is cleared by setting DE.

7.5.3.9 DCS DMA Interrupt Enable Control Bit (DIE) Bit 30

When the DMA Interrupt Enable (DIE) bit is set, the DMA interrupt occurs when DTD is set. When DIE is cleared, the DMA interrupt is disabled. Cleared by Hardware and Software Reset.

DIE	DMA Interrupt
0	Disabled
1	Enabled

7.5.3.10 DCS DMA Channel Enable Control Bit (DE) Bit 31

The DE bit enables DMA Controller operation. Setting DE will clear DTD. Setting DE will trigger a single block DMA transfer if DTM1-DTM0=00. Setting DE will enable transfers in DMA modes that use a requesting device as trigger. DE is cleared by Hardware and Software Reset, and by end of DMA transfer if a Single

Block transfer mode is selected. Clearing DE during DMA operation will stop the DMA only after the present DMA transfer has been completed (the data is stored in the destination), setting DTD.

DE	DMA Operation
0	Disabled
1	Enabled

7.5.4 DMA Counter (DCO)

The DMA Counter is a read/write 32-bit register that contains the number of DMA data transfers to be done. If the DMA channel is set to Single Block transfer mode then, after each DMA data transfer, the DMA Counter is decremented by one and tested for zero. When the count reaches zero, the DMA Block transfer is done and the DMA channel will stop the data transfers. If the channel is set to Single Word mode (DTM1-DTM0=11), the contents of the DMA counter are ignored since each DMA data transfer is done on demand.

The DMA Counter should not be written while the DMA channel is operating in one of the Block Transfer modes. The DMA Counter may be written only when the channel is disabled (DE=0 and DTD=1), or when in Single Word mode (DTM1-DTM0=11).

7.5.5 DMA Address Registers (DSR and DDR)

The DMA Source Address register (DSR) and the DMA Destination Address register (DDR) are two 32-bit registers that contain the addresses of the source and destination, respectively, for the next DMA transfer. The DMA Address registers are functionally identical to the Address Generation Unit address registers.

7.5.6 DMA Offset Registers (DSN and DDN)

The DMA Source Offset register (DSN) and the DMA Destination Offset register (DDN) are two 32-bit registers that specify the offset values used to update the respective DMA address registers. Each offset register is read when the associated address register is read and used as input to its modulo arithmetic unit. The DMA Offset registers are functionally identical to the Address Generation Unit offset registers.

7.5.7 DMA Modifier Registers (DSM and DDM)

The DMA Source Modifier register (DSM) and the DMA Destination Modifier register (DDM) are two 32-bit registers that specify the type of arithmetic used to update the respective DMA address register during DMA address register update calculations. Each modifier register is read when the associated address register is read and used as input to its modulo arithmetic unit. The DMA Modifier registers are functionally identical to the Address Generation Unit modifier registers. Both DMA modifier registers are set to $FFFFFFFF (linear arithmetic) during a processor reset or software reset.

7.5.8 DMA ALU

The ALU is common to the DMA and Address Generation Unit, and time multiplexed between them. The DMA ALU is hardwired in the (R)+N configuration. Users can increment or decrement by 1 or N by loading the DMA Offset registers accordingly. For example, DMA block transfers with DSP96002 word addressable memory would often load the DMA Offset register with +1. However, interpolation, decimation, and commutation operations could require an arbitrary address offset value N. DMA block transfers with byte address-

able memory would typically load the Offset register with +4 to perform 32-bit aligned accesses. DMA transfers to/from I/O peripherals would load the Offset register with zero to continuously access the same address.

7.5.9 DMA Addressing Modes

The DMA Controller may be programmed for address calculation and updates in the same manner as the registers in the Address Generation Unit. The DMA Modifier registers are completely identical to the Modifier registers M0-M7. In this way, the DMA source and/or destination address registers may be updated using linear, bit-reverse or modulo address calculations. See Section 5.8 for a description of how to program the Modifier registers.

7.5.10 DMA Restrictions

The following are some restrictions that apply to the DMA operation:

1. Source/Destination address area must be wholly internal or external. The DMA cannot handle blocks of data that are partially internal and partially external. These blocks must be handled as two separate blocks, one internal and the other external.

   If the Source/Destination address area is defined as internal, and an address that is greater than the highest internal address is generated by the DMA ALU, the address will wrap around into the internal address space.

   If the Source/Destination address area is defined as external, and an address that is less than the lowest external address is generated by the DMA ALU, the address will access external memory anyway. Note that X and Y Data Memory locations that are always considered as internal by the core may be accessed as external memory locations by the DMA.

2. WAIT and STOP will halt DMA transfers. STOP and WAIT may disable the internal clock in the middle of a DMA transfer. The user should stop DMA transfers before executing the STOP or WAIT instructions. To stop DMA transfers, DE must be cleared. Before executing the STOP or WAIT instruction, the user should poll the DTD bit (or receive a DMA interrupt when DTD is set) to ensure that the present DMA transfer has been completed.

   Note that the use of these instructions already require some kind of software management in multiprocessing systems, since there is no way that the external devices could know that the chip entered the STOP or WAIT state.

3. Only the Host Transmit/Receive Data registers may be accessed by the DMA Controller when specifying source or destination in the internal I/O space.

4. During any (internal or external) read-modify-write core access, the DMA is not permitted to complete or initiate any DMA transfer. The DMA is halted as if it is trying to access an external bus and it is not the bus master.

5. Cases where DMA operation is affected:

   1. If the core is accessing external memory thorough both ports simultaneously, and one or both of the core accesses are delayed due to memory wait, internal DMA transfers will be delayed because the chip clock is generating wait states, freezing internal activity.

   2. If the core is doing one external access and the DMA is also doing an external access thorough the other port, and the DMA access is delayed (for example, due to wait states), the access by the core in the other port is not affected. The DMA has a separate wait mechanism, and in this case the core continues normal execution since the core clock does not enter wait states.

3. If the core is doing one external access and the DMA is also doing an external access thorough the other port, and the core access is delayed, the access by the DMA in the other port is also delayed. This happens because the chip clock generates wait states and the whole chip stops. Also, the arbitration between DMA and core cannot continue if the core is frozen.
4. If one of the DMA channels is accessing external memory thorough a port, and the access is delayed due to bus arbitration or memory wait, the second DMA channel will also stop, since the DMA mechanism does not distinguish between the two channels.
5. If the Data ALU is executing a floating point instruction that requires normalization cycles (IEEE mode with denormalized numbers), the Data ALU may freeze the clock for the other chip sections including the DMA. In this case, the DMA operation will be slowed down.

7.6 I/O MEMORY MAP

Internal I/O peripherals occupy the top 128 locations in X memory space. External I/O peripherals occupy the top 128 locations in Y memory space. Figure 7-27 shows the I/O memory map for the internal I/O peripherals.

X DATA Memory Space

Address	Register
$FFFFFFFF	IPR - Interrupt Priority Register
$FFFFFFFE	BCRA - Port A Bus Control Register
$FFFFFFFD	BCRB - Port B Bus Control Register
$FFFFFFFC	PSR - Port Select Register
:	RESERVED :
$FFFFFFF0	Reserved for OnCE Operation (OGDBR)
$FFFFFFEF	HTXA/HRXA - HOSTA HTX/HRX Register
$FFFFFFEE	HTXCA - HOSTA HTX Reg. and HMRC Clear
$FFFFFFED	HSRA - HOSTA Status Register
$FFFFFFEC	HCRA - HOSTA Control Register
:	RESERVED :
$FFFFFFE7	HTXB/HRXB - HOSTB HTX/HRX Register
$FFFFFFE6	HTXCB - HOSTB HTX Reg. and HMRC Clear
$FFFFFFE5	HSRB - HOSTB Status Register
$FFFFFFE4	HCRB - HOSTB Control Register
:	RESERVED :
$FFFFFFE0	RESERVED
$FFFFFFDF	DSM0 -DMA CH0 Source Modifier Register
$FFFFFFDE	DSR0 -DMA CH0 Source Address Register
$FFFFFFDD	DSN0 -DMA CH0 Source Offset Register
$FFFFFFDC	DCO0 -DMA CH0 Counter Register
$FFFFFFDB	DDM0 -DMA CH0 Destination Modifier Register
$FFFFFFDA	DDR0 -DMA CH0 Destination Address Register
$FFFFFFD9	DDN0 -DMA CH0 Destination Offset Register
$FFFFFFD8	DCS0 -DMA CH0 Control/Status Register
$FFFFFFD7	DSM1 -DMA CH1 Source Modifier Register
$FFFFFFD6	DSR1 -DMA CH1 Source Address Register
$FFFFFFD5	DSN1 -DMA CH1 Source Offset Register
$FFFFFFD4	DCO1 -DMA CH1 Counter Register
$FFFFFFD3	DDM1 -DMA CH1 Destination Modifier Register
$FFFFFFD2	DDR1 -DMA CH1 Destination Address Register
$FFFFFFD1	DDN1 -DMA CH1 Destination Offset Register
$FFFFFFD0	DCS1 -DMA CH1 Control/Status Register
$FFFFFFCF	RESERVED
:	RESERVED :
$FFFFFF80	RESERVED

Figure 7-27. Internal I/O Memory Map

SECTION 8
EXCEPTION PROCESSING

8.1 INTRODUCTION

This section describes the actions of the DSP96002 which are outside the normal processing associated with the execution of instructions. The sequence of actions taken by the DSP96002 on exception conditions is described. Also, the interrupt priority level (IPL) of the processor and interrupt sources is described.

8.2 PROCESSING STATES

The DSP96002 is always in one of five processing states: normal, exception, reset, wait, or stop. The normal processing state is that associated with instruction execution.

8.2.1 Exception Processing State

The exception processing state is associated with interrupts. Exception processing may be internally generated by a software interrupt instruction, by an on-chip peripheral hardware interrupt, or by an error condition. Externally, exception processing can be generated by an interrupt. Exception processing provides an efficient context switch for servicing I/O devices.

8.2.2 Reset Processing State

The reset processing state is entered in response to the external $\overline{RESET}$ pin being asserted. Upon entering the reset state the following actions occur:

- Internal peripheral devices are reset and disabled.
- The modifier registers Mn are set to $FFFFFFFF.
- The Interrupt Priority Register (IPR) is cleared.
- All CCR, ER, IER and MR bits are cleared, except for I1 and I0 in the MR register.
- The interrupt mask bits I1,I0 in the MR register are set.

The DSP96002 remains in the reset state until $\overline{RESET}$ is deasserted. Upon leaving the reset state the chip operating mode bits of the operating mode register are loaded from the external Mode Select pins (MODA, MODB, MODC) and program execution begins at the location described in Section 9.

8.2.3 Wait Processing State

The wait processing state is a low power consumption mode entered by execution of the WAIT instruction. In wait mode, the internal clock is disabled from all internal circuitry except the internal peripherals (the interrupt controller and host interfaces). All internal processing is halted until any unmasked interrupt occurs, the DSP96002 is reset, or $\overline{DR}$ is asserted. If exit from the wait state was caused by asserting $\overline{DR}$, the processor may enter the debug mode (see **Section 10**).

8.2.4 Stop Processing State

The stop processing state is the lowest power consumption mode and is entered by the execution of the STOP instruction. In the stop mode, the clock oscillator is gated off, in contrast to the wait mode where the clock oscillator remains active. All activity in the processor is halted until one of the following actions occurs:

1. A low level is applied to the $\overline{IRQA}$ pin ($\overline{IRQA}$ asserted)
2. A low level is applied to the $\overline{RESET}$ pin ($\overline{RESET}$ asserted)
3. A low level is applied to the $\overline{DR}$ pin.

Either of these actions will gate on the oscillator and, after a clock stabilization delay, clocks to the processor and peripherals will be re-enabled.

When the clocks to the processor and peripherals are re-enabled then the processor will enter the reset processing state if the exit from stop state was caused by a low level on the $\overline{RESET}$ pin.

If the exit from stop state was caused by a low level on the $\overline{IRQA}$ pin then the processor will service the highest priority pending interrupt. If no interrupt is pending (i. e. $\overline{IRQA}$ was deasserted before interrupts were arbitrated) then the processor resumes execution at the instruction following the STOP instruction that caused the entry into the stop state.

If the exit from stop state was caused by a low level on the $\overline{DR}$ pin, the processor may enter the debug mode (see **Section 10**).

8.3 EXCEPTION PROCESSING

Exception processing in a digital signal processing environment is primarily associated with transfer of data between DSP96002 memory or registers and a peripheral device. When an interrupt occurs, a limited context switch must be performed with minimum overhead.

When a hardware interrupt is received, it is synchronized on instruction boundaries so that the first two interrupt instruction words can be inserted into the instruction stream. Suppose that the interrupt is stored in the interrupt pending latch during the current instruction fetch cycle. During the next cycle, which is the decode cycle of the current instruction, the PC will be updated to fetch the next instruction. However, in the following cycle, which is the execution cycle of the current instruction, the address placed on the program address bus (PAB) comes from the appropriate interrupt start address, rather than from the PC. Note that the PC is frozen until exception processing terminates.

Figure 8-1illustrates the effect of the interrupt controller, which is simply to insert two instruction words into the processor's instruction stream.

Int ctl cyc1	i						* i				
Int ctl cyc2		i						i			
Fetch	n3	n4	ii1	ii2	n5	n6	n7	n8	ii3	ii4	
Decode	n2	n3	n4	ii1	ii2	n5	n6	n7	n8	ii3	ii4
Execute	n1	n2	n3	n4	ii1	ii2	n5	n6	n7	n8	ii3

```
  i = interrupt
 ii = interrupt instruction word
  n = normal single word instruction
  *   subsequent interrupts are enabled at this time
```

Figure 8-1. Interrupt Pipeline Operation

The following one-word instructions are aborted when they are fetched in the cycle preceding the fetch of the first interrupt instruction word (n4 or n8 in Figure 8-1): Bcc, BRA, BScc, BSR, FBcc, FBScc, FJcc, FJScc, Jcc, JMP, JScc, JSR, LRA, REP, RESET, RTI, RTR, RTS, STOP, and WAIT.

Two-word instructions are aborted when the first interrupt instruction word fetched will replace the fetch of the second word of the two word instruction (n5 in Figure 8-2).

Aborted instructions are re-fetched again when program control returns from the interrupt routine. The PC is adjusted appropriately prior to the end of the decode cycle of the aborted instruction.

If the first interrupt word fetch occurs in the cycle following the fetch of a one-word instruction not listed above or the second word of a two-word instruction, that instruction will complete normally prior to the start of the interrupt routine.

The following cases have been identified where service of an interrupt might encounter an extra delay:

1. If a long interrupt routine is used to service a (F)TRAPcc interrupt, then the processor priority level is set to 3. Thus, all interrupts except for illegal instruction and stack error are disabled until the (F)TRAPcc service routine terminates with an RTI (unless the (F)TRAPcc service routine software lowers the processor priority level).
2. While servicing an interrupt the next interrupt service will be delayed according to the following rule:

   After the first interrupt instruction word reaches the instruction decoder, at least four more instructions will be decoded before decoding the next first interrupt instruction word (see Figure 8-1). If any one pair of instructions being counted is the REP instruction followed by a instruction to be repeated then the whole "package" is counted as two instructions independently of the number of repeats done.
3. The following instructions are uninterruptable: ILLEGAL, (F)TRAPcc, STOP, WAIT and RESET.
4. The REP instruction and the instruction being repeated are uninterruptable.

8.3.1 Interrupt Instruction Fetch

During an interrupt instruction fetch, instruction words are fetched from the interrupt starting address and interrupt starting address+1 locations.

The interrupt controller generates an interrupt instruction fetch address which points to the first instruction word of a two-word fast interrupt routine. This address is used for the next instruction fetch, instead of the PC, and the interrupt instruction fetch address+1 is used for the subsequent instruction fetch. While the

interrupt instructions are being fetched, the PC is inhibited from being updated. After the two interrupt words have been fetched, the PC is used for any following instruction fetches.

After both interrupt instructions words have been fetched, they are guaranteed to be executed. This is true even if the instruction that is currently being executed is a change of flow instruction (i.e., JMP, JSR, etc.) that would normally ignore the instructions in the pipe. After the interrupt instruction fetch, the PC will point to the instruction that would have been fetched if the interrupt instructions had not been substituted.

8.3.2 Interrupt Instruction Execution

Two types of interrupt routines may be used: fast and long. The fast routine consists of only the two automatically inserted interrupt instruction words. These words can contain any single two-word instruction or any two one-word instructions, except for restrictions listed in Section A.9.2.1. Interrupt instruction execution is considered to be fast if neither of the instructions of the interrupt service routine cause a change of flow. A jump to subroutine within a fast interrupt routine forms a long interrupt. A long interrupt routine is terminated with an RTI instruction to restore the PC and SR from the stack and return to normal program execution. Hardware Reset is a special exception which will normally contain only a JMP instruction at the exception start address.

8.3.2.1 Fast Interrupt Instruction Execution

Execution of a fast interrupt routine always follows the following rules:

1. Status is not saved during a fast interrupt routine; therefore, instructions which modify status should not be used.
2. Fast interrupt routines are never interruptible.
3. The fast interrupt routine may contain any single two-word instruction or any two one-word instructions, except for restrictions listed in Section A.9.2.1.
4. If one of the instructions in the fast routine is a jump to subroutine, then a long interrupt routine is formed.
5. The PC is never updated during a fast interrupt routine.
6. Normal instruction fetching resumes using the PC following the completion of the fast interrupt routine.

Figure 8-3 illustrates the effect of a fast interrupt routine on the instruction pipeline.

Int ctl cyc1	i						* i				
Int ctl cyc2		i						i			
Fetch	n3	n4	ii1	ii2	n4	n5	n6	n7	ii3	ii4	n8
Decode	n2	n3	n4	f1	f2	n4	--	n6	n7	f3	f4
Execute	n1	n2	n3	NOP	f1	f2	n4	--	n6	n7	f3

```
  f = fast interrupt instruction word (non-control-flow-change)
  i = interrupt
 ii = interrupt instruction word
  n = normal single word instruction
 n4 = 2-word instruction
 n5 = 2nd word of n4
  *   subsequent interrupts are enabled at this time
```

Figure 8-2. Example of Aborting a Two Word Instruction Fast Interrupt

Int ctl cyc1	i						* i				
Int ctl cyc2		i						i			
Fetch	n3	n4	ii1	ii2	n5	n6	n7	n8	ii3	ii4	n9
Decode	n2	n3	n4	f1	f2	n5	n6	n7	--	f3	f4
Execute	n1	n2	n3	n4	f1	f2	n5	n6	n7	--	f3

```
  f = fast interrupt instruction word (non-control-flow-change)
  i = interrupt
 ii = interrupt instruction word
  n = normal instruction word
  n7 = 2-word instruction
  n8 = 2nd word of n7
  *   subsequent interrupts are enabled at this time
```

Figure 8-3.
Example Of The Case Of Four Instructions Between Consecutive Vectors

8.3.2.2 Long Interrupt Instruction Execution

A jump to subroutine instruction within a fast interrupt routine forms a long interrupt routine. One-word or two-word jump to subroutine instructions may be used to form a long interrupt routine. The one-word jump to subroutine may be located in either the first or second interrupt vector location. If a conditional one-word jump to subroutine is located in the first interrupt vector location, the instruction in the second vector location will be ignored if the jump condition is true but executed if the jump condition is false. If the one-word jump to subroutine is located in the second interrupt vector location, the instruction in the first vector location will be fetched and executed before executing the jump to subroutine. Execution of a long interrupt routine always follows the following rules:

1. During execution of the jump to subroutine instruction, when it occurs in the first or second interrupt vector location, the following actions occur:
    1. The PC and SR are stacked.

2. The status register is modified as follows: the interrupt mask bits I1, I0 in the MR are updated to mask interrupts of the same or lower priority (except that illegal instruction, stack error and (F)TRAPcc can always interrupt).
3. The PC will be altered by the JSR instruction so that instruction execution will continue with the instructions located in the address pointed to by the JSR instruction.

2. Long interrupt routines are interruptible by higher priority interrupts. The first instruction word of the next interrupt service may reach the decoder only after the decoding of at least four instructions following the decoding of the first instruction of the previous interrupt.
3. The long interrupt routine should be terminated by an RTI, which pulls the PC and SR from the stack.

Int ctl cyc1	i							
Int ctl cyc2		i						
Fetch	n3	n4	ii1	ii2	sr1	sr2	sr3	sr4
Decode	n2	n3	n4	JSRf	NOP	sr1	sr2	sr3
Execute	n1	n2	n3	n4	JSRf	NOP	sr1	sr2

Int ctl cyc1	* i							
Int ctl cyc2		i						
Fetch	sr5	n5	ii3	ii4	n6	n7	n8	n9
Decode	RTI	NOP	n5	ii3	ii4	n6	n7	n8
Execute	sr3	RTI	NOP	n5	ii3	ii4	n6	n7

```
   i = interrupt
  ii = interrupt instruction word
JSRf = fast JSR (one-word JSR instruction)
   n = normal instruction word
  sr = service routine word
   *   subsequent interrupts are enabled at this time
```

Figure 8-4. Long Interrupt Pipeline Action

Figure 8-4 illustrates the effect of a long interrupt routine on the instruction pipeline. A fast JSR (that is, a one-word JSR instruction) is used to form the long interrupt routine. For this example, word 4 of the long interrupt routine is an RTI. A subsequent interrupt is shown to illustrate the uninterruptable nature of the early instructions in the long interrupt routine.

See Figure 8-5 for an example of interrupt service when the instruction that receives the internal interrupt service request is the REP instruction (n3 in Figure 8-5). During the repeated executions of the instruction that follows the REP instruction (n4), instruction fetches are suspended. The fetches will be reactivated only after the loop counter is decremented to one. During the execution of n4, interrupts will not be serviced. When LC finally reaches one, the fetches are reinitiated and the interrupt can be serviced. In Figure 8-5 it can be seen that n5 (loaded into the instruction latch from the backup instruction latch) is decoded and executed as well as n6 before the first interrupt vector.

Int ctl cyc1	i	†			i*						
Int ctl cyc2		i				i					
Fetch	n3	n4	n5			n6	ii1	ii2	n7	n8	n9
Decode	n2	REP	NOP	n4	n4	n5	n6	ii1	ii2	n7	n8
Execute	n1	n2	REP	NOP	n4	n4	n5	n6	ii1	ii2	n7

```
  i = interrupt
 ii = interrupt instruction word
  n = normal instruction word
 n3 = REP #2 instruction
 n4 = instruction being repeated twice
 n5 = instruction that waits in the backup instruction latch
  †   interrupt rejected at this time
  *   interrupt can be reenabled at this time
```

Figure 8-5.
Example Of Interrupt Service When Interrupt Is Presented To REP Instruction

8.4 INTERRUPT SOURCES

Exceptions may originate from a number of interrupt sources. The DSP96002 interrupt sources are given in Figure 8-6. The corresponding interrupt starting addresses for each interrupt source are shown. Interrupt starting addresses are internally-generated 32-bit addresses which point to the starting address of the fast interrupt service routine. The interrupt starting address for each interrupt is an address constant for minimum overhead. Motorola reserves 128 interrupt starting address locations, while 128 locations are reserved for user applications. These locations occupy the lowest 512 words of program memory space, except for Hardware Reset, which may also occupy a location in the upper range of the program memory address. If some of this space is not used, it may be used for program storage.

8.4.1 Internal Peripheral Interrupt Sources

The internal peripheral interrupt sources include all of the on-chip peripheral devices (Host and DMA). Each internal interrupt source is level sensitive; i.e., each is serviced any time it is present and the interrupt is not masked. Each internal hardware source has independent enable control.

8.4.2 Hardware RESET

The Hardware RESET interrupt is level sensitive and is the highest priority 3 interrupt. It is caused by asserting the $\overline{RESET}$ pin.

8.4.3 External Interrupt Requests IRQA, IRQB and IRQC

The IRQA, IRQB and IRQC interrupts can be programmed to be level-sensitive or edge-sensitive. Level-sensitive interrupts are not internally latched and are not automatically cleared when they are serviced; they must be cleared by other means to prevent multiple interrupts. The edge-sensitive interrupts are latched as pending on the high-to-low transition of the interrupt input and are automatically cleared when the interrupt is serviced. IRQA, IRQB and IRQC can be programmed to one of three priority levels: level 0, 1, or 2, all of

Interrupt Starting Address	interrupt Source
$FFFFFFFE	Hardware RESET
$00000000	Hardware RESET
$00000002	Stack Error
$00000004	Illegal Instruction
$00000006	(F)TRAPcc (default)
$00000008	IRQA
$0000000A	IRQB
$0000000C	IRQC
$0000000E	Reserved
$00000010	DMA Channel 1
$00000012	DMA Channel 2
$00000014	Reserved
$00000016	Reserved
$00000018	Reserved
$0000001A	Reserved
$0000001C	Host A Command (default)
$0000001E	Host B Command (default)
$00000020	Host A Receive Data
$00000022	Host A Transmit Data
$00000024	Host A Read X Memory
$00000026	Host A Read Y Memory
$00000028	Host A Read P Memory
$0000002A	Host A Write X Memory
$0000002C	Host A Write Y Memory
$0000002E	Host A Write P Memory
$00000030	Host B Receive Data
$00000032	Host B Transmit Data
$00000034	Host B Read X Memory
$00000036	Host B Read Y Memory
$00000038	Host B Read P Memory
$0000003A	Host B Write X Memory
$0000003C	Host B Write Y Memory
$0000003E	Host B Write P Memory
$00000040	Reserved
:	:
$000000FE	Reserved
$00000100	User interrupt vector
:	:
$000001FE	User interrupt vector

Note: User interrupt vector locations are available for host commands.

Figure 8-6. DSP96002 Interrupt Sources

which are maskable. Additionally, each of these interrupts has independent enable control. When the IRQA, IRQB or IRQC interrupts are disabled in the interrupt priority register, pending requests will be discarded, no new requests will be accepted, and the edge-detection latch will remain in the reset state. Also, if the interrupt is defined as level-sensitive, its edge-detection latch will remain in the reset state.

Interrupt service, which begins by fetching the instruction word in the first vector location, is considered finished when the instruction word in the second vector location is fetched. In the case of an edge-sensitive interrupt, the internal latch is automatically cleared when the second vector location is fetched. The fetch of the first vector location does not guarantee that the second location will be fetched. Figure 8.7 illustrates the one case where the second vector location is not fetched. In Figure 8.7, the (F)TRAPcc instruction "discards" the fetch of the first interrupt vector to ensure that the (F)TRAPcc vectors will be fetched. Instruction n4 is decoded as a (F)TRAPcc while ii1 is being fetched. Execution of the (F)TRAPcc requires that ii1 be discarded and the two (F)TRAPcc vectors (ii3 and ii4) be fetched instead.

8.4.4 (F)TRAPcc (Conditional Software Interrupt Instruction)

The (F)TRAPcc instruction causes a non-maskable interrupt which is serviced immediately following the (F)TRAPcc instruction if the specified condition is true. (F)TRAPcc is a priority 3 interrupt.

Int ctl cyc1	i				i*						
Int ctl cyc2		i				i					
Fetch	n3	n4	ii1				ii3	ii4	tr1	tr2	tr3
Decode	n2	n3	trap	--	--	--	--	JSR	--	tr1	tr2
Execute	n1	n2	n3	trap	--	--	--	--	JSR	--	tr1

```
  i  = interrupt request
  i* = interrupt request generated by (F)TRAPcc
 ii1 = first vector of interrupt i
 ii3 = first (F)TRAPcc vector (one word JSR)
 ii4 = second (F)TRAPcc vector
  n  = normal instruction word
 n4  = (F)TRAPcc, cc condition true
 tr  = instructions pertaining to the (F)TRAPcc long interrupt routine
```

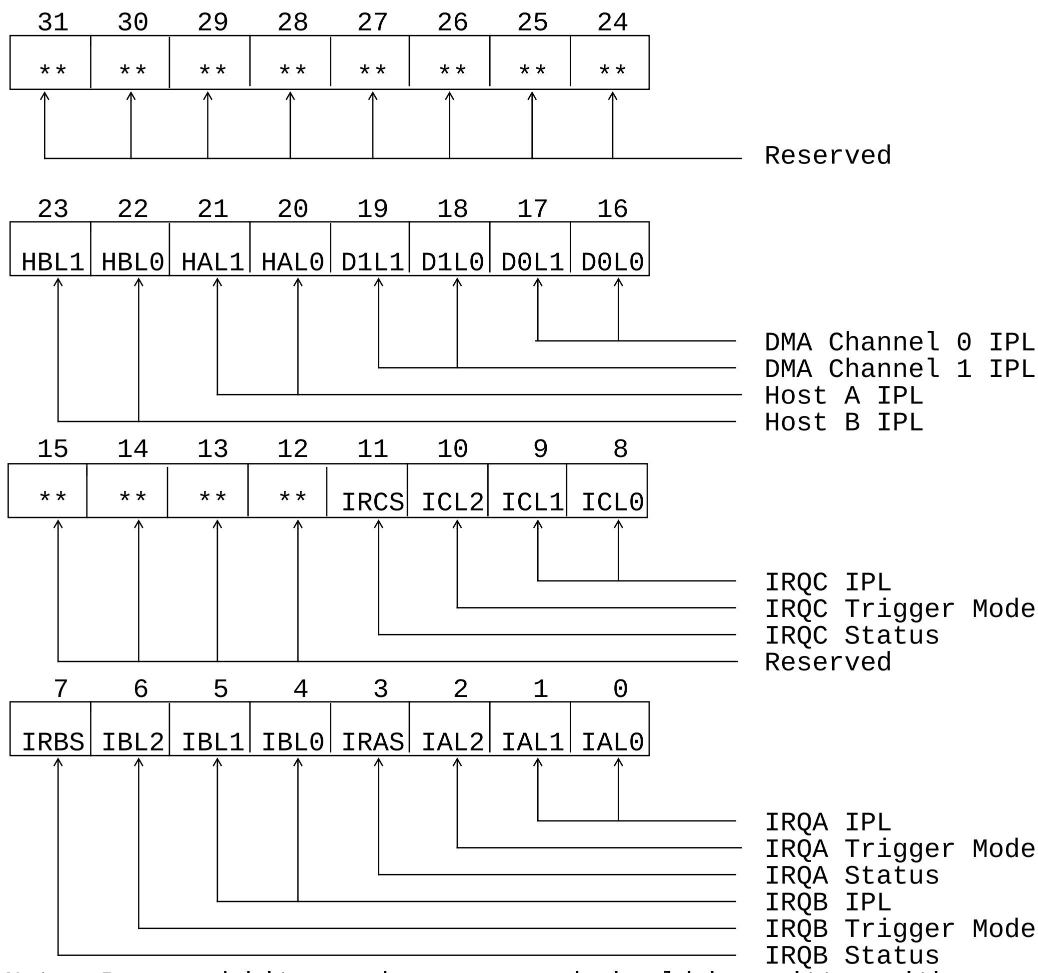
Figure 8-7. (F)TRAPcc Instruction Rejecting Another Interrupt

CAUTION

On all level-sensitive interrupts, the Interrupt must be externally released before interrupts are internally re-enabled or the processor will be interrupted repeatedly until the interrupt is released.

I1	I0	Exceptions Permitted	Exceptions Masked
0	0	IPL 0, 1, 2, 3	None
0	1	IPL 1, 2, 3	IPL 0
1	0	IPL 2, 3	IPL 0,1
1	1	IPL 3	IPL 0,1,2

Figure 8-8. Status Register Interrupt Mask Bits



Note: Reserved bits read as zero and should be written with zero for future compatibility.

Figure 8-9. Interrupt Priority Register IPR (Address X:$FFFFFFFF)

xxL1	xxL0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

Figure 8-10. Interrupt Priority Level Bits

IxL2	Trigger Mode
0	level
1	neg. edge

IRxS	Status
0	Serviced
1	Pending

Figure 8-11. External Interrupt Trigger Mode and Status

8.4.5 Illegal Instruction Interrupt

The illegal instruction interrupt is a non-maskable interrupt which is serviced immediately following the illegal instruction interrupt instruction (ILLEGAL) or upon loading an illegal instruction in the instruction latch. The illegal instruction interrupt is a priority 3 interrupt.

8.4.6 Stack Error Interrupt

The Stack Error interrupt is a priority 3 interrupt. It is generated by turning on the Stack Error flag in the Stack Pointer register, generally due to improper stack operation. The Stack Error flag will remain set until it is cleared by some instruction that explicitly writes into the SP register. Since the IPL level (3) of this interrupt does not mask other pending interrupts of this same level, it is recommended that the Stack Error flag be cleared by the first instruction of the Stack Error interrupt service routine in order not to get the same request again.

8.5 INTERRUPT PRIORITY STRUCTURE

Four levels of interrupt priority are provided. Interrupt priority levels (IPLs) numbered 0, 1, and 2, are maskable. Level 0 is the lowest level. Level 3 is the highest level, and is nonmaskable. The only level 3 interrupts are Stack Error, Reset, Illegal Instruction (ILLEGAL) and (F)TRAPcc. The interrupt mask bits (I1, I0) in the status register reflect the current processor priority level and indicate the interrupt minimum priority level needed for an interrupt source to interrupt the processor. Figure 8-8 gives a description of the interrupt mask bits. Interrupts are inhibited for all priority levels less than the current processor priority level. Level 3 interrupts can always interrupt the processor.

8.5.1 Interrupt Priority Levels (IPL)

The interrupt priority level for each on-chip peripheral device (Host, DMA) and for each external interrupt source (IRQA, IRQB, IRQC) can be programmed under software control. Each on-chip or external peripheral device can be programmed to one of the three maskable priority levels (IPL 0, 1, or 2). Interrupt priority levels are set by writing to the Interrupt Priority Register.

8.5.2 Interrupt Priority Register (IPR)

This read/write register specifies the interrupt priority level for each of the interrupting devices (Host, DMA, IRQA, IRQB, IRQC). In addition, this register specifies the trigger mode of each external interrupt source and shows the status of the external interrupt request. The register is cleared on Hardware reset or by the RESET instruction. The Interrupt Priority Register is shown in Figure 8-9. Figure 8-10 defines the interrupt priority level bits. Figure 8-11 defines the external interrupt trigger mode bits and status information.

8.5.2.1 IRQA Interrupt Priority Level - IAL1-IAL0 (Bits 0-1)

The IRQA Interrupt Priority Level (IAL1-IAL0) bits are used to enable and specify the priority level of the external interrupt input IRQA.

IAL1	IAL0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

8.5.2.2 IRQA Trigger Mode - IAL2 (Bit 2)

The IRQA Trigger Mode (IAL2) bit specifies the trigger method for the external interrupt input IRQA.

IAL2	Trigger Mode
0	level
1	negative edge

8.5.2.3 IRQA Status - IRAS (Bit 3)

The read-only IRQA Status (IRAS) bit indicates the status of the interrupt request for the external interrupt input IRQA. If the IRQA interrupt is defined as edge-sensitive and it is enabled, the IRAS bit indicates the state of the edge-detection latch. If the IRQA interrupt is defined as level-sensitive or is disabled, the IRAS bit indicates the state of the IRQA pin after internal synchronization.

IRAS	Status (edge and enabled)	IRQA pin (level or disabled)
0	Serviced	High
1	Pending	Low

8.5.2.4 IRQB Interrupt Priority Level - IBL1-IBL0 (Bits 4-5)

The IRQB Interrupt Priority Level (IBL1-IBL0) bits are used to enable and specify the priority level of the external interrupt input IRQB.

IBL1	IBL0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

8.5.2.5 IRQB Trigger Mode - IBL2 (Bit 6)

The IRQB Trigger Mode (IBL2) bit specifies the trigger method for the external interrupt input IRQB.

IBL2	Trigger Mode
0	level
1	negative edge

8.5.2.6 IRQB Status - IRBS (Bit 7)

The read-only IRQB Status (IRBS) bit indicates the status of the interrupt request for the external interrupt input IRQB. If the IRQB interrupt is defined as edge-sensitive and it is enabled, the IRBS bit indicates the state of the edge-detection latch. If the IRQB interrupt is defined as level-sensitive or is disabled, the IRBS bit indicates the state of the IRQB pin after internal synchronization.

IRBS	Status (edge and enabled)	IRQB pin (level or disabled)
0	Serviced	High
1	Pending	Low

8.5.2.7 IRQC Interrupt Priority Level - ICL1-ICL0 (Bits 8-9)

The IRQC Interrupt Priority Level (ICL1-ICL0) bits are used to enable and specify the priority level of the external interrupt input IRQC.

ICL1	ICL0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

8.5.2.8 IRQC Trigger Mode - ICL2 (Bit 10)

The IRQC Trigger Mode (ICL2) bit specifies the trigger method for the external interrupt input IRQC.

ICL2	Trigger Mode
0	level
1	negative edge

8.5.2.9 IRQC Status - IRCS (Bit 11)

The read-only IRQC Status (IRCS) bit indicates the status of the interrupt request for the external interrupt input IRQC. If the IRQC interrupt is defined as edge-sensitive and it is enabled, the IRCS bit indicates the state of the edge-detection latch. If the IRQC interrupt is defined as level-sensitive or is disabled, the IRCS bit indicates the state of the IRQC pin after internal synchronization.

IRCS	Status (edge and enabled)	IRQC pin (level or disabled)
0	Serviced	High
1	Pending	Low

8.5.2.10 Reserved bits (Bits 12-15, 24-31)

These reserved bits read as zero and should be written with zero for future compatibility.

8.5.2.11 DMA Channel 0 Interrupt Priority Level - D0L1-D0L0 (Bits 16-17)

The DMA Channel 0 Interrupt Priority Level (D0L1-D0L0) bits are used to enable and specify the priority level of the DMA Channel 0 interrupt.

D0L1	D0L0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

8.5.2.12 DMA Channel 1 Interrupt Priority Level - D1L1-D1L0 (Bits 18-19)

The DMA Channel 1 Interrupt Priority Level (D1L1-D1L0) bits are used to enable and specify the priority level of the DMA Channel 1 interrupt.

D1L1	D1L0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

8.5.2.13 Host A Interrupt Priority Level - HAL1-HAL0 (Bits 20-21)

The Host A Interrupt Priority Level (HAL1-HAL0) bits are used to enable and specify the priority level of all interrupt sources located in the Port A Host Interface.

HAL1	HAL0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

8.5.2.14 Host B Interrupt Priority Level - HBL1-HBL0 (Bits 22-23)

The Host B Interrupt Priority Level (HBL1-HBL0) bits are used to enable and specify the priority level of all interrupt sources located in the Port B Host Interface.

HBL1	HBL0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

8.5.3 Exception Priorities within an IPL

If more than one exception is pending when an instruction is executed, the interrupt with the highest priority level is serviced first. Within a given interrupt priority level, a second priority structure determines which interrupt is serviced when multiple interrupt requests with the same IPL are pending. The priority of equal IPL interrupts is given in Figure 8-12. Also given in Figure 8-12 are the interrupt enable bits for all interrupts.

Priority	Exception	Enabled by
highest	**Hardware RESET**	**-**
	Illegal Instruction	-
	Stack Error	-
	(F)TRAPcc	-
	IRQA (External Interrupt)	(IPR) IAL1-IAL0
	IRQB (External Interrupt)	(IPR) IBL1-IBL0
	IRQC (External Interrupt)	(IPR) ICL1-ICL0
	Host A Command Interrupt	(HCR) HCIE
	Host A Receive Data Interrupt	(HCR) HRIE
	Host A Read X Memory Interrupt	(HCR) HXRE
	Host A Read Y Memory Interrupt	(HCR) HYRE
	Host A Read P Memory Interrupt	(HCR) HPRE
	Host A Write X Memory Interrupt	(HCR) HXWE
	Host A Write Y Memory Interrupt	(HCR) HYWE
	Host A Write P Memory Interrupt	(HCR) HPWE
	Host A Transmit Data Interrupt	(HCR) HTIE
	Host B Command Interrupt	(HCR) HCIE
	Host B Receive Data Interrupt	(HCR) HRIE
	Host B Read X Memory Interrupt	(HCR) HXRE
	Host B Read Y Memory Interrupt	(HCR) HYRE
	Host B Read P Memory Interrupt	(HCR) HPRE
	Host B Write X Memory Interrupt	(HCR) HXWE
	Host B Write Y Memory Interrupt	(HCR) HYWE
	Host B Write P Memory Interrupt	(HCR) HPWE
	Host B Transmit Data Interrupt	(HCR) HTIE
	DMA Channel 0 Interrupt	(DCS0) DIE0
lowest	DMA Channel 1 Interrupt	(DCS1) DIE1

Figure 8-12. DSP96002 Exception Priorities within an IPL

SECTION 9
CHIP OPERATING MODES AND MEMORY MAPS

9.1 OPERATING MODES AND PROGRAM MEMORY MAPS

The operating mode bits MA, MB, and MC in the OMR register determine the bus expansion mode for program memory and the startup procedure when the DSP96002 leaves the RESET state. The Data ROM Enable bit DE in the OMR determines the bus expansion mode for the data memories.

The MODA, MODB, and MODC pins are used to load MA, MB and MC with the initial operating mode of the DSP96002. These pins are sampled as the DSP96002 leaves the RESET state. These pins do not affect the operating mode after that time and are available for other functions. Chip operating modes are programmable by writing the operating mode bits MA, MB and MC in the operating mode register. Refer to Section 4.12 for a description of the operating mode register OMR. Figure 9-1 shows the mode assignments.

Mode	MC	MB	MA	DSP96002 Initial Chip Operating Mode
0	0	0	0	PRAM enabled, Reset at $FFFFFFFE (Port A)
1	0	0	1	PRAM enabled, Reset at $FFFFFFFE (Port B)
2	0	1	0	PRAM disabled, Reset at $00000000 (Port A)
3	0	1	1	PRAM disabled, Reset at $00000000 (Port B)
4	1	0	0	Bootstrap from byte-wide (bits D7-D0) external memory at $FFFF0000 (Port A)
5	1	0	1	Bootstrap from byte-wide (bits D7-D0) external memory at $FFFF0000 (Port B)
6	1	1	0	Bootstrap thru the Host Interface (Port A)
7	1	1	1	Bootstrap thru the Host Interface (Port B)

Figure 9-1. DSP96002 Initial Chip Operating Mode Summary

There are eight chip operating modes divided in two groups:

- Non-bootstrap modes - these modes are used to access program memories that are already programmed.
- Bootstrap modes - these modes are used to load the internal program memory implemented in RAM. After loading the internal program memory, the DSP96002 switches to Mode 0 or 1 but begins program execution at the address located at the on-chip program memory address $00000000.

9.1.1 Mode 0 (Internal PRAM enabled, Reset at $FFFFFFFE, Port A)

In mode 0, the internal program memory occupies the lower portion of the program memory space. Addresses higher than the highest internal program memory location are directed to external program memory. The address of the hardware reset vector is $FFFFFFFE, located in the Port A external program memory space. The program memory map for this mode is shown in Figure 9-2.

9.1.2 Mode 1 (Internal PRAM enabled, Reset at $FFFFFFFE, Port B)

In Mode 1, the internal program memory occupies the lower portion of the program memory space. Addresses higher than the highest internal program memory location are directed to external program memory. The address of the hardware reset vector is $FFFFFFFE, located in the Port B external program memory space. The program memory map for this mode is shown in Figure 9-2.

9.1.3 Mode 2 (Internal PRAM disabled, Reset at $00000000, Port A)

In Mode 2 the internal program memory is disabled. All references to program memory space are directed external program memory. The address of the hardware reset vector is $00000000, located in the Port A external program memory space. The program memory map for this mode is shown in Figure 9-2.

9.1.4 Mode 3 (Internal PRAM disabled, Reset at $00000000, Port B)

In Mode 3 the internal program memory is disabled. All references to program memory space are directed external program memory. The address of the hardware reset vector is $00000000, located in the Port B external program memory space. The program memory map for this mode is shown in Figure 9-2.

9.1.5 Modes 4-7 (Bootstrap modes)

The bootstrap modes load the internal program memory from an external source. The type and location of the source is selected according to the values of the MA and MB bits in the OMR. After loading the internal program memory, the DSP96002 begins program execution at the address located at the on-chip program memory address $00000000.

The bootstrap is implemented by executing a bootstrap program located in an user invisible bootstrap program ROM which is mapped into the program memory space for the duration of the bootstrap operations.

When the chip exits the reset state in one of the bootstrap modes, the following actions occur:

1. On-chip hardware maps a 64 word by 32-bit, user invisible, ROM into the internal DSP96002 program memory space starting at location $00000000.
2. On-chip hardware makes the internal program RAM write-only for the duration of the bootstrap load.
3. Program execution begins at location $00000000 of the internal bootstrap ROM. See Figure 9-3 for a listing of the DSP96002 Bootstrap program.
4. The bootstrap program reads OMR bits MA and MB to determine the bootstrap mode selected.

In mode 4, the bootstrap program loads the internal program RAM from 4,096 consecutive byte-wide external program memory locations starting at $FFFF0000 through Port A.

In mode 5, the bootstrap program loads the internal program RAM from 4,096 consecutive byte-wide external program memory locations starting at $FFFF0000 through Port B.

In mode 6, the bootstrap program loads the internal program RAM from an external host processor through the Host Interface in Port A. If the Host Interface flag HF1 is cleared, the bootstrap program assumes that the external host processor is an 8-bit wide source which will supply up to 4,096 bytes. If the Host Interface flag HF1 is set, the bootstrap program assumes that the external host processor is a 32-bit wide source which will supply up to 1,024 32-bit words to load into the program RAM. The external host processor may terminate the bootstrap program by setting the Host Interface flag HF0.

In mode 7, the bootstrap program loads the internal program RAM from an external host processor through the Host Interface in Port B. If the Host Interface flag HF1 is cleared, the bootstrap program assumes that the external host processor is an 8-bit wide source which will supply up to 4,096 bytes.

If the Host Interface flag HF1 is set, the bootstrap program assumes that the external host processor is a 32-bit wide source which will supply up to 1,024 32-bit words to load into the program RAM. The external host processor may terminate the bootstrap program by setting the Host Interface flag HF0.

5. Enter Mode 0 or 1 by writing to the OMR. This action will begin a timed delay to remove the bootstrap ROM from the program memory map.
6. This timed delay is exactly timed to allow the boot program to execute a NOP then a JMP to location $00000000 and begin execution of the user's program.

The user may also select a bootstrap mode by writing into the OMR. This technique allows the DSP96002 programmer to re-boot his system. From any operating mode, the user may program the OMR to the required bootstrap mode. This begins a timed delay to map the bootstrap ROM into the program address space. This timed delay is exactly timed to allow the programmer to execute a NOP then a JMP to bootstrap ROM location $00000000 and begin the bootstrap process described above in steps 1 to 6.

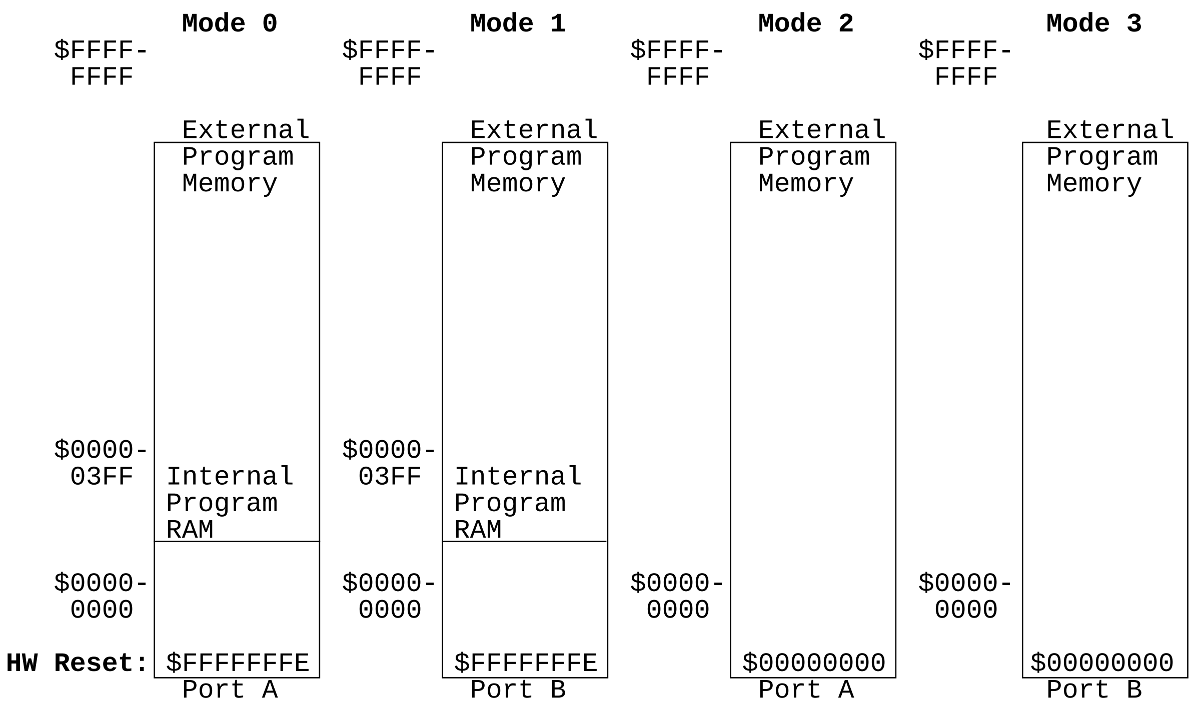


Figure 9-2. DSP96002 Program Memory Maps

```
                  PAGE 132,50,0,10
; BOOTSTRAP CODE FOR DSP96002 - © Copyright 1988 Motorola Inc.
;
; Host algorithm  / AND / external bus method.
;
; This is the Bootstrap program contained in the DSP96002. This program
; can load the internal program memory from one of 4 external sources.
; The program reads the OMR bits MA and MB to decide which external
; source to access.
; If MB:MA = 0X - load from 4,096 consecutive byte-wide P: memory
; locations (starting at P:$FFFF0000).
; If MB:MA = 10 - load internal PRAM thru Host Interface in Port A.
; If MB:MA = 11 - load internal PRAM thru Host Interface in Port B.   BOOT
EQU     $FFFF0000                      ; The location in P: memory
                                       ; where the external byte-wide
                                       ; EPROM is expected to be mapped
M_HCRA          EQU     $FFFFFFEC      ; Port A Host Control Register
M_HSRA          EQU     $FFFFFFED      ; Port A Host Status Register
M_HRXA          EQU     $FFFFFFEF      ; Port A Host Rec. Data Register
M_HCRB          EQU     $FFFFFFE4      ; Port B Host Control Register
M_HSRB          EQU     $FFFFFFE5      ; Port B Host Status Register
M_HRXB          EQU     $FFFFFFE7      ; Port B Host Rec. Data Register
                ORG     PL:$0          ; bootstrap code starts at P:$0
START           MOVE    #BOOT,R1       ; R1 = External P: address of
                                       ; bootstrap byte-wide ROM
                MOVEI   #0,R0          ; R0 = starting P: address of
                                       ; internal memory where program
                                       ; will begin loading.
; If this program is entered by changing the OMR to bootstrap mode,
; make certain that registers M0 and M1 have been set to $FFFFFFFF.
; Make sure the appropriate BCR register is set to $xxxxxxFx since
; EPROMs are slow.
; Make sure that the Port Selection Register is set to permit program
; memory accesses thru the required memory expansion port (Port A or B).
;
; The first routine will load 4,096 bytes from the external P memory
; space beginning at P:$FFFF0000 (bits 7-0).  These will be condensed
; into 1,024 32-bit words and stored in contiguous internal PRAM memory
; locations starting at P:$0.  Note that the first routine loads data
; starting with the least significant byte of P:$0 first.
; The Port Selection Register is not set by this program. It is set
; by HW Reset.
```

Figure 9-3. Assembler Source for DSP96002 Bootstrap Program (1 of 3)

```
; The second routine loads the internal PRAM using the Host
; Interface logic.
; If HF1=0, it will load 4,096 bytes from the external host processor.
; These will be condensed into 1,024 32-bit words and stored in
; contiguous internal PRAM memory locations starting at P:$0.  Note that
; the routine loads data starting with the least significant byte of
; P:$0 first.
; If HF1=1, it will load 1,024 32-bit words from the external host
; processor.
; If the host processor only wants to load a portion of the P memory,
; and start execution of the loaded program, the Host Interface
; bootstrap load program routine may be killed by setting HF0 = 0.
;
  INLOOP         DO      #1024,_LOOP1   ; Load 1,024 instruction words
; This is the context switch
                 JSET    #1,OMR,_HOSTLD ; Perform load from Host
                                        ; Interface if MB=1.
; This is the first routine. It loads from external P: memory.
                 DO      #4,_LOOP2      ; Get 4 bytes into D0.L
                 LSR     #8,D0          ; Shift previous byte down
                 MOVEM   P:(R1)+,D1.L   ; Get byte from ext. P mem.
                 LSL     #24,D1         ; Shift into upper byte
                 OR      D1,D0          ; concatenate
  _LOOP2         JMP     <_STORE        ; Then put the word in P memory
;
; This is the second routine. It loads thru the Host Interface.
  _HOSTLD        JSET    #0,OMR,_HOSTB  ; Port A or Port B?
; Boot thru Host Interface in Port A
  _HOSTA         BCLR    #5,X:M_HCRA    ; Enable Port A Host Interface
                 MOVE    #M_HSRA,R2     ; R2 points to HSRA
                 MOVE    #M_HRXA,R3     ; R3 points to HRXA
                 JMP     <_HOSTR        ; go to host routine
; Boot thru Host Interface in Port B
  _HOSTB         BCLR    #5,X:M_HCRB    ; Enable Port B Host Interface
                 MOVE    #M_HSRB,R2     ; R2 points to HSRB
                 MOVE    #M_HRXB,R3     ; R3 points to HRXB
```

Figure 9-3. Assembler Source for DSP96002 Bootstrap Program (2 of 3)

```
; Host load routine
_HOSTR
_LBL11          JCLR    #3,X:(R2),_LBL22    ; if HF0=1, stop loading data.
                ENDDO                       ; Must terminate the do loops
                JMP     <_BOOTEND

_LBL22          JCLR    #0,X:(R2),_LBL11    ; Wait for HRDF to go high
                                            ; (meaning data is present).
                JCLR    #4,X:(R2),_LBL33    ; 8-bit source?
                MOVE    X:(R3),D0.L         ; Get 32-bit word from host
                JMP     <_STORE

_LBL33          DO      #4,_LOOP4           ; Get 4 bytes into D0.L
                LSR     #8,D0               ; Shift previous byte down

_LBL1           JCLR    #3,X:(R2),_LBL2     ; if HF0=1, stop loading data.
                ENDDO                       ; Must terminate the do loops
                ENDDO
                JMP     <_BOOTEND
_LBL2           JCLR    #0,X:(R2),_LBL1     ; Wait for HRDF to go high
                                            ; (meaning data is present).
                MOVE    X:(R3),D1.L         ; Get byte from host
                LSL     #24,D1              ; Shift into upper byte
                OR      D1,D0               ; concatenate
_LOOP4

_STORE          MOVEM   D0.L,P:(R0)+        ; Store 32-bit result in P mem.

_LOOP1                                      ; and go get another 32-bit word

; This is the exit handler that returns execution to internal PRAM
_BOOTEND        ANDI    #$F9,OMR            ; Set the operating mode to 00x
                                            ; (and trigger an exit from
                                            ; bootstrap mode).
                ANDI    #$0,CCR             ; Clear CCR as if HW RESET.
                                            ; Also delay needed for
                                            ; Op. Mode change.
                JMP     <$0                 ; Start fetching from PRAM.
; DSP96002 bootstrap program size = 50 words
```

Figure 9-3. Assembler Source for DSP96002 Bootstrap Program (3 of 3)

9.2 DATA MEMORY MAPS

The data memory maps are shown in Figure 9-4 and Figure 9-5.

9.2.1 Internal Data RAMs

The on-chip X and Y Data RAMs occupy locations $00000000 to $000001FF in X and Y Data Memory maps, respectively, and they are always enabled.

9.2.2 Internal Data ROMs

The X and Y Data Memory expansion mode is affected by the DE bit located in the OMR. The on-chip X and Y Data ROMs occupy locations $00000400 to $000007FF in X and Y Data Memory maps, respectively, when enabled by setting DE=1 in the Operating Mode Register. If DE=0, the on-chip Data ROMs are disabled and the address range they previously occupied is now in external data memory.

The X and Y Data ROMs each occupy 1,024 locations. The X Data ROM contains a full cycle of cosine values while the Y Data ROM contains a full cycle of sine values. The sine and cosine values were generated using the MC68881 IEEE floating-point coprocessor rounded to IEEE single precision floating-point using the round to nearest mode.

When the internal Data ROMs are enabled (DE=1), the X and Y Data Memory locations in the address range $00000200 to $000003FF are defined as internal. This address range is unpopulated and is reserved for future expansion. When the internal Data ROMs are disabled (DE=0), the address range $00000200 to $000003FF is defined as external.

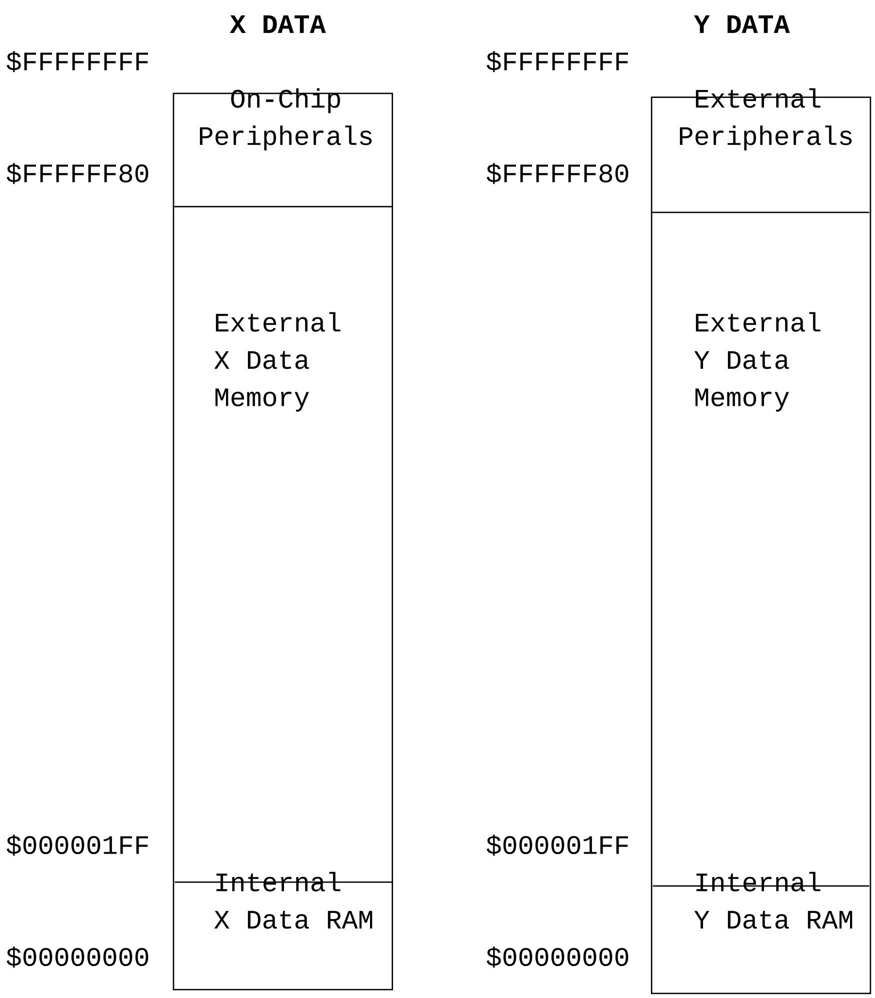


Figure 9-4. DSP96002 Data Memory Maps for DE=0

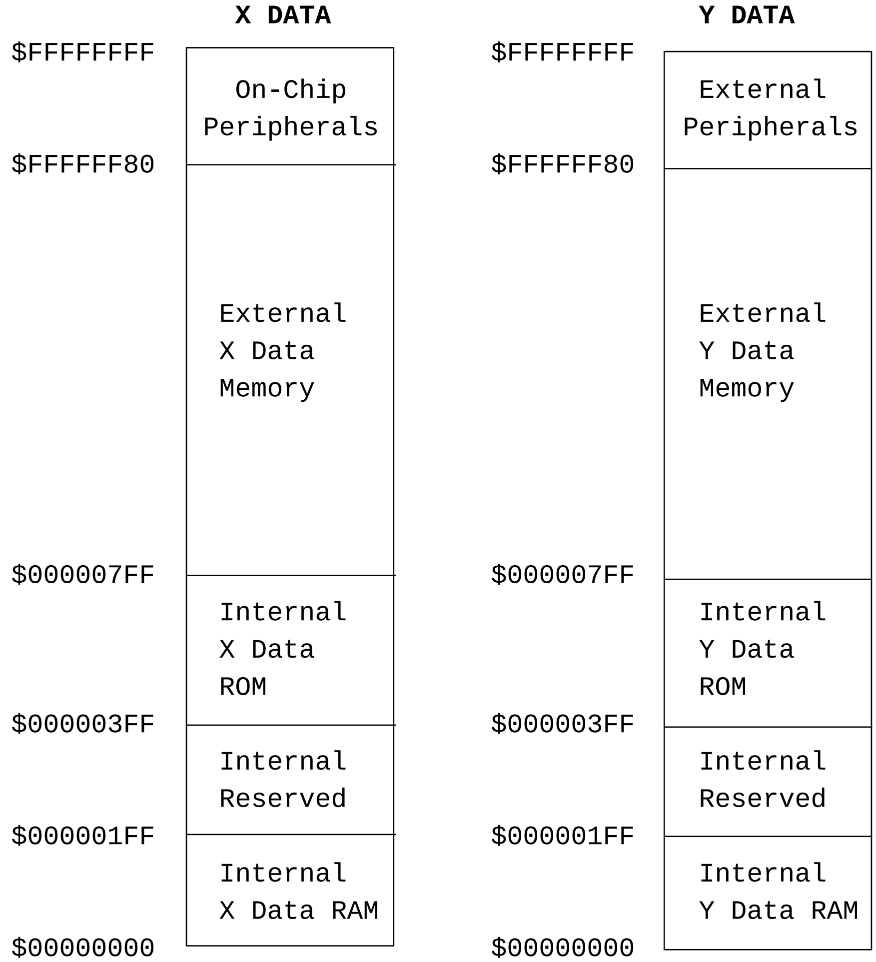


Figure 9-5. DSP96002 Data Memory Maps for DE=1

PROGRAM MEMORY

MMM CBA	HW RESET VECTOR	MODE	INTERNAL PROGRAM SPACE	EXTERNAL PROGRAM SPACE	PORT
000	$FFFFFFFE	0	$00000000-$000003FF	$00000400-$FFFFFFFF	A
001	$FFFFFFFE	1	$00000000-$000003FF	$00000400-$FFFFFFFF	B
010	$00000000	2	none	$00000000-$FFFFFFFF	A
011	$00000000	3	none	$00000000-$FFFFFFFF	B
1X0	$00000000 in Bootstrap ROM	4,6	For reading (Boot ROM): $00000000-$0000003F For writing (Prog RAM): $00000000-$000003FF	$00000400-$FFFFFFFF	A
1X1	$00000000 in Bootstrap ROM	5,7	For reading (Boot ROM): $00000000-$0000003F For writing (Prog RAM): $00000000-$000003FF	$00000400-$FFFFFFFF	B

Note: Bootstrap ROM is at $00000000-$0000003F, PRAM becomes write-only in Bootstrap modes.
After the bootstrap program executes, the chip reverts to Mode 0 (from Bootstrap Modes 4 or 6) or to Mode 1 (from Bootstrap Modes 5 or 7), and program execution begins at location $00000000 in internal PRAM.

DATA MEMORIES

D E	INTERNAL X AND Y DATA SPACE	EXTERNAL Y DATA SPACE	EXTERNAL X DATA SPACE
0	$00000000-$000001FF	$00000200-$FFFFFFFF	$00000200-$FFFFFF7F
1	$00000000-$000007FF	$00000800-$FFFFFFFF	$00000800-$FFFFFF7F

Note: Internal X I/O space is located in the range $FFFFFF80-$FFFFFFFF.

Figure 9-6. DSP96002 Memory Maps - Summary

SECTION 10
ON-CHIP EMULATOR

10.1 INTRODUCTION

Conventional methods of system development (for example the DSP56001) consist of a program which resides in the DSP program memory (monitor). An interface circuit which either uses on-chip resources or an additional program memory address communicates with a host computer or terminal. This technique is not transparent, loads the DSP bus and sometimes interferes with the user system configuration. To emulate the DSP in a user's target system an expensive cable must be used to bring out the DSP pins onto the system under development.

The DSP96002's on-chip emulation (OnCE™) circuitry provides a means of interacting with the DSP96002 and its peripherals non-intrusively so that a user may examine registers, memory or on-chip peripherals. This will facilitate hardware/software development on the DSP96002 processor. To achieve this, special circuits and dedicated pins on the DSP96002 die are defined to avoid sacrificing any user accessible on-chip resource.

A key feature of the OnCE™ dedicated pins is to allow the user to insert the DSP96002 into his target system yet retaining debug control. The need for a costly cable which brings out the DSP96002 footprint on an emulator system is eliminated because of the easy access to the dedicated OnCE™ debug serial port. Figure 10-1illustrates the block diagram of the OnCE™ serial interface.

10.2 ON-CHIP EMULATION (OnCE™) PINOUT

10.2.1 Debug Serial Input/Chip Status 0 (DSI/OS0)

Serial data or commands are provided to the OnCE™ controller through the DSI/OS0 pin when it is an input. The data received on the DSI pin will be recognized only when the DSP96002 has entered the debug mode of operation. Data must have valid TTL logic levels before being latched on the falling edge of the serial clock. Data is always shifted into the OnCE™ serial port most significant bit (MSB) first. When an output, this pin in conjunction with the OS1 pin, provides information about the chip status indicating why the debug mode cannot be entered in response to an external request. The DSI/OS0 pin is an output when not in Debug Mode (until the acknowledge signal is issued to the Command Controller). When switching from output to input, the pin is three-stated. In order to avoid any possible glitches, an external pull-down resistor should be attached to this pin. During hardware reset, this pin is defined as an output and it is driven low.

OnCE™ is a trademark of Motorola Inc.

Figure 10-1. OnCE™ Block Diagram

10.2.2 Debug Serial Clock/Chip Status 1 (DSCK/OS1)

The serial clock is supplied to the OnCE™ through the DSCK/OS1 pin when it is an input. The serial clock provides pulses required to shift data into and out of the OnCE™ serial port. Data is clocked into the OnCE™ on the falling edge and is clocked out of the OnCE™ serial port on the rising edge. When an output, this pin in conjunction with the OS0 pin, provides information about the chip status describing why the debug mode cannot be entered in response to an external request. The DSCK/OS1 pin is output when not in the Debug Mode (until the acknowledge signal is issued to the Command Controller). When switching from output to input, the pin is three-stated. In order to avoid any possible glitches, an external pull-down resistor should be attached to this pin. During hardware reset, this pin is defined as an output and it is driven low. The maximum SCK frequency is one third of the system clock frequency.

OS1	OS0	Status
0	0	Normal state
0	1	STOP or WAIT state
1	0	Core busy state
1	1	Core or DMA busy state

10.2.3 Debug Serial Output (DSO)

The debug serial output provides the data contained in one of the OnCE™ controller registers as specified by the last command received from the external command controller. When idle, this pin is held high. When the requested data is available, the DSO line will be asserted (negative true logic) for two T cycles (2T = period of DSP96002 master clock) to indicate that the serial shift registers are ready to receive clocks in order to deliver the data. When a trace or breakpoint occurs this line will be asserted for one T cycle to indicate (acknowledge) that the chip has entered the debug mode and is waiting for commands. Data is always shifted out the OnCE™ serial port most significant bit (MSB) first. During hardware reset, this pin is held high.

10.2.4 Debug Enable Input ($\overline{DR}$)

The debug request input provides a means of entering the debug mode of operation from the external command controller. This pin, when asserted, causes the DSP96002 to finish the current instruction being executed, save the instruction pipeline information, enter the debug mode, and wait for commands to be entered from the debug serial input line.

10.3 OnCE™ CONTROLLER AND SERIAL INTERFACE

The OnCE™ Controller and Serial Interface contains the following blocks: input shift register, bit counter, OnCE™ decoder, and the status/control register. Figure 10-2 illustrates a block diagram of the OnCE™ serial interface.

10.3.1 OnCE™ Input Shift Register (OISR)

The OnCE™ Input Shift Register is an 8-bit shift register that receives the serial data from the DSI line. The data is clocked into the register on the falling edge of the clock applied to the DSCK pin. After the 8th bit is received, the OISR will stop shifting in new data. The latched data will be used as input for the OnCE™ Decoder. The data is always shifted into the OISR most significant bit (MSB) first.

10.3.2 OnCE™ Bit Counter (OBC)

The OnCE™ Bit Counter is a 5-bit counter associated with shifting in and out the data bits. The OBC is incremented by the falling edges of the DSCK. The OBC is cleared during hardware reset and whenever the DSP96002 acknowledges that the Debug Mode has been entered. The OBC supplies two signals to the OnCE™ Decoder: one indicating that the first 8 bits were shifted-in (so a new command is available) and the second indicating that 32 bits were shifted-in (the data associated with that command is available) or that 32 bits were shifted-out (the data required by a read command was shifted out).

10.3.3 OnCE™ Decoder (ODEC)

The OnCE™ Decoder supervises the entire OnCE™ activity. It receives as input the 8-bit command from the OISR, two signals from OBC (one indicating that 8 bits have been received and the other that 32 bits have

Figure 10-2. OnCE™ Controller and Serial Interface

been received), and two signals indicating that the core was halted and the DMA was halted. The ODEC generates all the strobes required for reading and writing the selected OnCE™ registers.

10.3.4 OnCE™ Status and Control Register (OSCR)

The Status and Control Register is a 32-bit register used to select the events that will put the chip in Debug Mode (see Figure 10-3). Breakpoints may be disabled or enabled on one or more memory spaces. The Trace Mode of operation is also selected from OSCR. The control bits are read/write while the status bits are read only.

10.3.4.1 Program Memory Breakpoint Enable (PBE0-PBE1) Bits 0-1

These control bits unmask program memory breakpoints allowing break-point interrupts to occur when a program memory address is within the low and high program memory address registers and will select whether the breakpoint will be recognized for read or write accesses. These bits are cleared on hardware reset.

PBE1	PBE0	Selection
0	0	Breakpoint disabled
0	1	Breakpoint on write accesses
1	0	Breakpoint on read accesses
1	1	Breakpoint on both read and write accesses

10.3.4.2 Program Memory Breakpoint Selection (PBS0-PBS1) Bits 2-3

These control bits select whether the program memory breakpoints will be recognized on core program memory fetches, core program memory accesses (MOVEM or MOVEP) or DMA program memory accesses. These bits are cleared on hardware reset.

31 19	18	17	16	15 9	8	7	6	5	4	3	2	1	0
*	TO	DBO	PBO	*	TME	DBS1	DBS0	DBE1	DBE0	PBS1	PBS0	PBE1	PBE0

* Read as zeroes, should be written with zero for future compatibility.

Figure 10-3. OnCE™ Programming Model

PBS1	PBS0	Selection
0	0	Breakpoint on Core fetch accesses
0	1	Breakpoint on Core P move accesses
1	0	Breakpoint on Core and P move accesses
1	1	Breakpoint on DMA accesses

When PBS1=0 and PBS0=0, program memory breakpoints are enabled only for fetches of the first instruction word of instructions that are actually executed (not the killed instructions and not the second word of jump instructions that are not taken). Program memory address breakpoints occur after the fetched instruction is executed and the breakpoint counter has been decremented to zero.

When PBS1=0 and PBS0=1, program memory breakpoints are enabled only for **explicit** program memory access resulting from MOVEP and MOVEM instructions to/from P: memory space (MOVEP P:..,.. or MOVE ..,P:..).

When PBS1=1 and PBS0=0, program memory breakpoints are enabled for **any** access to the Program space (any kind of PMOVE, true and false fetches, fetches of 2nd word, etc.).

When PBS1=1 and PBS0=1, program memory breakpoints are enabled only for **DMA** accesses to program memory space.

10.3.4.3 Data Memory Breakpoint Enable (DBE0-DBE1) Bit 4-5

These control bits enable data memory breakpoints to occur when a data memory address is within the low and high data memory address registers and will select whether the breakpoint will be recognized for read or write accesses. These bits are cleared on hardware reset.

DBE1	DBE0	Selection
0	0	Breakpoint disabled
0	1	Breakpoint on write accesses
1	0	Breakpoint on read accesses
1	1	Breakpoint on both read and write accesses

10.3.4.4 Data Memory Breakpoint Selection (DBS0-DBS1) Bits 6-7

These control bits select whether the data memory breakpoints will be recognized on core or DMA data memory accesses for X or Y data spaces. These bits are cleared on hardware reset.

DBS1	DBS0	Selection
0	0	Breakpoint on X Core fetch addresses
0	1	Breakpoint on Y Core fetch addresses
1	0	Breakpoint on X DMA fetch addresses
1	1	Breakpoint on Y DMA fetch addresses

10.3.4.5 Trace Mode Enable (TME) Bit 8

This control bit, when set, enables the Trace Mode causing the chip to enter the Debug Mode whenever the execution of an instruction is completed and the Trace Counter is zero. This bit is cleared on hardware reset.

10.3.4.6 Reserved (Bits 9-15, 20-31)

These bits are reserved for future use. They are read as zero and should be written as zero for future compatibility.

10.3.4.7 Program Memory Breakpoint Occurrence (PBO) Bit 16

This read only status bit is set when a program memory breakpoint occurs. It is used by the external command controller to determine how the Debug Mode was entered. This bit is cleared on hardware reset and when the OSCR is read.

10.3.4.8 Data Memory Breakpoint Occurrence (DBO) Bit 17

This read only status bit is set when a data memory breakpoint occurs. It is used by the external command controller to determine how the debug mode was entered. This bit is cleared on hardware reset and when the OSCR is read.

10.3.4.9 Trace Occurrence (TO) Bit 18

This read only status bit is set when the debug mode of operation is entered from a decrement to zero of the trace counter and the trace mode has been armed. This bit is cleared on hardware reset and when the OSCR is read.

10.3.4.10 Software Debug Occurrence (SWO) Bit 19

This status bit is set when the debug mode of operation is entered due to the execution of the (F)DEBUGcc instruction with condition true. This bit is cleared on hardware reset and when the OSCR is read.

10.4 OnCE™ HARDWARE BREAKPOINT LOGIC

Hardware breakpoints may be set on program memory or data memory locations. Also, the breakpoint does not have to be in the program flow but within an approximate address range of where the program may be executing. This significantly increases the programmer’s ability to monitor what the program is doing real-time (see **Section 10-3.4** for programming details).

The breakpoint logic has two identical sections: one for program memory breakpoints and one for data memory breakpoints. Each section contains latches for core or DMA addresses, registers that store the upper and lower address limit, comparators and a counter. Figure 10-4 illustrates a block diagram of the OnCE™ Program Memory Breakpoint Logic and Figure 10-5 illustrates a block diagram of the OnCE™ Data Memory Breakpoint Logic.

10.4.1 Address Comparator Breakpoint Registers

Address comparators are useful in determining where a program may be getting lost or when data is being written to areas that should not be written to in real-time. They are also useful in halting a program at a spe-

Figure 10-4. Program Memory Breakpoint Logic

cific point to examine/change registers or memory. Using address comparators to set breakpoints enables the user to set breakpoints in RAM or ROM and while in any operating mode.

The low address comparator will cause a logic true signal when the address on the bus is greater than or equal to the low boundary. The high address comparator will cause a logic true signal when the address on the bus is less than or equal to the high boundary. If the low address comparator and high address comparator both issue a logic true signal, the address is within the address range and the breakpoint counter is decremented if the contents are greater than zero. If zero, the counter is not decremented and the breakpoint exception occurs.

Conditional jump addresses produced by the instruction pipeline that are within a program address block being monitored are only valid if the conditional jump instruction occurs, otherwise the conditional jump ad-

Figure 10-5. Data Memory Breakpoint Logic

dress is ignored. Program memory address breakpoints occur after the opcode or operand is executed and the breakpoint counter has been decremented to zero.

Data memory address breakpoints also occur after the execution of the instruction which formed the data memory address and the breakpoint counter has decremented to zero.

All breakpoint registers are controlled by the debug status and control register, OSCR.

10.4.2 Breakpoint Counter

The breakpoint counter is useful for stopping at the nth iteration of a program loop or when the nth occurrence of a data memory access occurs. This information significantly decreases algorithm debug time and

provides a means of checking hot spots in program segments as well as peripheral or data memory accesses.

Program hot spots may be statistically evaluated by setting the breakpoint counter to a value, setting a program address in the program address comparator registers, passing control of the DSP96002 back to the user program and checking to see if a breakpoint occurs after n iterations of the program memory access.

The breakpoint counter becomes a powerful tool when debugging real-time fast interrupt sequences such as servicing an A/D or D/A converter or stopping after a specific number of host transfers have occurred.

The breakpoint counters are cleared by hardware reset.

10.4.3 Program Memory Address Latch (OPAL)

The Program Memory Address Latch is a 32-bit register that latches the PAB on every cycle during the core slot or during the DMA slot according to the PBS1-PBS0 bits in OSCR.

10.4.4 Program Memory Upper Limit Register (OPULR)

The Program Memory Upper Limit Register is a 32-bit register that stores the program memory breakpoint upper limit. OPULR can only be read or written through the serial interface. Before enabling breakpoints, OPULR must be loaded by the command controller.

10.4.5 Program Memory Lower Limit Register (OPLLR)

The Program Memory Lower Limit Register is a 32-bit register that stores the program memory breakpoint lower limit. OPLLR can only be read or written through the serial interface. Before enabling breakpoints, OPLLR must be loaded by the command controller.

10.4.6 Program Memory High Address Comparator (OPHC)

The Program Memory High Address Comparator compares the current program memory address (stored by OPAL) with the OPULR contents. If OPULR is higher or equal than OPAL then the comparator delivers a signal indicating that the address is lower than or equal to the high limit.

10.4.7 Program Memory Low Address Comparator (OPLC)

The Program Memory Low Address Comparator compares the current program memory address (stored by OPAL) with the OPLLR contents. If OPLLR is lower or equal than OPAL then the comparator delivers a signal indicating that the address is higher than or equal to the low limit.

10.4.8 Program Memory Breakpoint Counter (OPBC)

The Program Memory Breakpoint Counter is a 32-bit counter which is loaded with a value equal to the number of times minus one that a program memory address should be accessed before a breakpoint is acknowledged. On each occurrence of the program memory address access, the counter is decremented. When the counter has reached the value of zero and a new occurrence takes place a signal is generated and if PBE is set the chip will enter the Debug Mode. The OPBC can only be read or written through the serial interface. Before enabling Program Memory Breakpoints, OPBC must be loaded by the command controller. Figure 10-5 illustrates a block diagram of the Program Memory Breakpoint Counter logic.

10.4.9 Data Memory Address Latch (ODAL)

The Data Memory Address Latch is a 32-bit register that latches the XAB or YAB on every cycle during the core or DMA slot according to the DBS1-DBS0 bits in OSCR.

10.4.10 Data Memory Upper Limit Register (ODULR)

The Data Memory Upper Limit Register is a 32-bit register that stores the program memory breakpoint upper limit. ODULR can only be read or written through the serial interface. Before enabling breakpoints, ODULR must be loaded by the command controller.

10.4.11 Data Memory Lower Limit Register (ODLLR)

The Data Memory Lower Limit Register is a 32-bit register that stores the program memory breakpoint lower limit. ODLLR can only be read or written through the serial interface. Before enabling breakpoints, ODLLR must be loaded by the command controller.

10.4.12 Data Memory High Address Comparator (ODHC)

The Data Memory High Address Comparator compares the current data memory address (stored by ODAL) with the ODULR contents. If ODULR is higher than or equal to ODAL then the comparator delivers a signal indicating that the address is lower than or equal to the high limit.

10.4.13 Data Memory Low Address Comparator (ODLC)

The Data Memory Low Address Comparator compares the current data memory address (stored by ODAL) with the ODLLR contents. If ODLLR is lower than or equal to ODAL then the comparator delivers a signal indicating that address is higher than or equal to the low limit.

10.4.14 Data Memory Breakpoint Counter (ODBC)

The Data Memory Breakpoint Counter is a 32-bit counter which is loaded with a value equal to the number of times minus one that a data memory address should be accessed before a breakpoint is acknowledged. On each data memory access, the counter is decremented. When the counter has reached the value of zero and a new occurrence takes place, a signal is generated and if the DBE bit is set, the chip will enter the Debug Mode. ODBC can only be read or written through the serial interface. Before enabling Data Memory Breakpoints, ODBC must be loaded by the command controller. Figure 10-5 illustrates a block diagram of the Program Memory Breakpoint Counter logic.

10.5 TRACE/STEP MODE

To execute DSP96002 instructions in single or multiple steps, a special mode similar to the trace mode of operation on the DSP56001 is necessary. The DSP96002 does not cause an interrupt exception as is the case with the DSP56001 but enters the debug mode of operation instead and waits for further instructions from the debug serial port after each instruction or group of instructions.

10.5.1 Trace Counter (OTC)

The trace mode has a 32-bit counter associated with it so that more than one instruction may be executed before returning back to the debug mode of operation. The objective of the counter is to allow the user to take multiple instruction steps real-time with no interference from the debug mode. This feature helps the

software developer debug sections of code which do not have a normal flow or are getting hung up in infinite loops. The trace counter also enables the user to debug areas of code which are time critical.

To enable the trace mode of operation the counter is loaded with a value, the program counter is set to the start location of the instruction(s) to be executed real-time, the trace mode is selected in the OSCR and the DSP96002 exits the debug mode by executing the appropriate command issued by the external command controller.

Upon exiting the debug mode the counter is decremented after each execution of an instruction. Interrupts are serviceable and all instructions executed (including fast interrupt services) will decrement the trace counter. Upon decrementing to zero, the DSP96002 will re-enter the debug mode (interrupt service breakpoint signal, ISBKPT, set), the trace occurrence bit in the OSCR will be set and the DSO pin will be toggled to indicate that the DSP96002 has entered debug mode and is requesting service.

The Trace Counter is cleared by hardware reset or whenever the debug mode of operation is entered. Figure 10-6 illustrates a block diagram of the Trace Counter logic.

10.6 OnCE™ SERIAL PORT TIMING

External data is fed into the serial input line by clocking each bit at a variable rate. The minimum clock rate should be 1 MHZ and the maximum clock rate should be 10 MHZ. The serial input bit must be stable at least 10 ns before the falling edge of the serial clock (set up time) and must remain stable for at least 10 ns after the falling edge of the clock (hold time).

The serial output line will clock out data from selected register as specified by the last command entered from the command controller. The data bit value will be valid on the rising edge of the clock and will remain valid for at least 10 ns after the rising edge of the clock.

After entering the debug mode of operation the serial output line will go low for at least one T cycle to flag the command controller that the DSP96002 is requesting a breakpoint or trace service.

10.7 METHODS OF ENTERING THE DEBUG MODE

Entering the Debug Mode is acknowledged by the chip by toggling the DSO line for 1 T cycle. This informs the external command controller that the chip has entered the Debug Mode and is waiting for commands. There are seven ways in which the Debug Mode may be entered.

10.7.1 External request during $\overline{\text{RESET}}$

Holding the $\overline{\text{DR}}$ line asserted during the assertion of $\overline{\text{RESET}}$ causes the chip to enter the Debug Mode. After receiving the acknowledge, the command controller must deassert the $\overline{\text{DR}}$ line. Note that in this case the chip does not perform any fetch or memory access before entering the Debug Mode.

10.7.2 External request during normal activity

Holding the $\overline{\text{DR}}$ line asserted during normal chip activity causes the chip to finish the execution of the current instruction and then enter the Debug Mode. After receiving the acknowledge, the command controller must deassert the $\overline{\text{DR}}$ line. Note that in this case the chip completes the execution of the current instruction and stops after the newly fetched instruction enters the instruction latch. This process is the same

Figure 10-6. Breakpoint and Trace Counter Logic

for any newly fetched instruction including instructions fetched by the interrupt processing or instructions that will be killed by the interrupt processing.

10.7.3 External Request During STOP

Asserting $\overline{DR}$ when the chip is in the STOP state (i. e., has executed a STOP instruction) causes the chip to exit the STOP state and enter the Debug Mode. After receiving the acknowledge, the command controller must negate $\overline{DR}$. Note that in this case, the chip completes the execution of the STOP instruction and halts after the next instruction enters the instruction latch.

10.7.4 External Request During WAIT

Asserting $\overline{DR}$ when the chip is in the WAIT state (i. e., has executed a WAIT instruction) causes the chip to exit the WAIT state and enter the Debug Mode. After receiving the acknowledge, the command controller must negate $\overline{DR}$. Note that in this case, the chip completes the execution of the WAIT instruction and halts after the next instruction enters the instruction latch.

10.7.5 Software request during normal activity

Upon executing the (F)DEBUGcc instruction when the specified condition is true, the chip enters the Debug Mode after the instruction following the (F)DEBUGcc instruction has entered the instruction latch (see the DEBUGcc and FDEBUGcc instruction descriptions in Appendix A).

10.7.6 Enabling Trace Mode

When operating in Trace Mode and the Trace Counter has reached a value of zero, the chip enters the Debug Mode after completing the execution of the instruction that caused the last Trace Counter decrement. Only instructions actually executed cause the Trace Counter to decrement i.e. a killed instruction will not decrement the Trace Counter and will not cause the chip to enter the Debug Mode.

10.7.7 Enabling breakpoints

When operating in Trace Mode or in Normal Mode, and the breakpoint mechanism is enabled with a Breakpoint Counter value of zero, the chip enters the Debug Mode after completing the execution of the instruction that caused the Breakpoint Counter decrement. In case of breakpoints on Program memory addresses, the breakpoint will be acknowledged immediately after the execution of the instruction that has caused the occurrence of the specified address. In case of breakpoints on Data memory addresses, the breakpoint will be acknowledged after the completion of the instruction following the instruction that caused the occurrence of the specified address.

10.8 PIPELINE INFORMATION

In order restore the pipeline to resume normal chip activity upon returning from the Debug Mode, a number of on-chip registers store the chip pipeline status. Figure 10-7 illustrates a block diagram of Pipeline Information Registers with the exception of the PAB registers which are shown in Figure 10-7.

10.8.1 PAB Registers (OPABF, OPABD)

There are two read only PAB registers which give pipeline information when the debug mode is entered. The OPABF register tells which opcode address is in the fetch stage of the pipeline and OPABD tells which opcode is in the decode stage. Under normal program flow conditions, the program address saved will be that of the instruction preceding the last instruction fetched and decoded before the debug mode was entered. The PAB registers can only be read or written through the serial interface.

10.8.2 PDB Register (OPDBR)

The PDB Register is a 32-bit latch that stores the value of the Program Data Bus generated by the last Program Memory access of the core before the Debug Mode is entered. OPDBR can only be read or written

through the serial interface. This register is affected by the operations performed during the Debug Mode and must be restored by the command controller when returning to normal mode.

10.8.3 PIL Register (OPILR)

The PIL Register is a 32-bit latch that stores the value of the Instruction Latch before the Debug Mode is entered. OPILR can only be read through the serial interface. This register is affected by the operations performed during the Debug Mode and must be restored by the command controller when returning to normal mode. Since there is no direct access to this register, this task is accomplished by writing the OPDBR first and then the data from OPDBR is latched in OPILR.

10.8.4 GDB Register (OGDBR)

The GDB Register is a 32-bit latch that can only be read through the serial interface. OGDBR is not actually required from a pipeline status restore point of view but is required as a means of passing information between the chip and the command controller. OGDBR is mapped on the X internal I/O space at address $FFFFFFF0. Whenever the command controller needs a data word such as a register or memory value, it will force the chip to execute an instruction that brings that information to OGDBR. Then, the contents of OGDBR will be delivered serially to the command controller by the command "READ GDB REGISTER".

10.9 PAB HISTORY BUFFER

To ease the debugging activity and keep track of the program flow, a First-In-First-Out buffer is provided which stores the addresses of the last five instructions that were executed as well as the addresses of the last fetched instruction and of the instruction currently in the Instruction Latch.

Figure 10-7. Pipeline Information Registers

Figure 10-8. Program Address Bus FIFO

10.9.1 PAB Register for Fetch (OPABFR)

The PAB Register for Fetch is a 32-bit register that stores the address of the last instruction that was fetched before the Debug Mode was entered. OPABFR can only be read through the serial interface. This register is not affected by the operations performed during the Debug Mode.

10.9.2 PAB Register for Decode (OPABDR)

The PAB Register for Decode is a 32-bit register that stores the address of the instruction currently in the Instruction Latch. This is the instruction that would have been decoded if the chip would not have entered the Debug Mode. OPABDR can only be read through the serial interface. This register is not affected by the operations performed during the Debug Mode.

10.9.3 PAB FIFO

To ease the debugging activity and keep track of the program flow, a First-In-First-Out buffer is provided which stores the addresses of the last five instructions that were executed. The FIFO is implemented as a circular buffer containing five 32-bit registers and one 3-bit counter. All the registers have the same address but any read access to the FIFO address will cause the counter to increment thus pointing to the next FIFO register. The registers are serially available to the command controller through their common FIFO address. Figure 10-8 illustrates a block diagram of the Program Address Bus FIFO. The FIFO is not affected by the operations performed during the Debug Mode except for the FIFO pointer increment when reading the FIFO. The last instruction executed before entering debug mode will be on the bottom of the FIFO.

Caution

To ensure FIFO coherence, a complete set of five reads of the FIFO must be performed. This is necessary due to the fact that each read increments the FIFO pointer thus pointing to the next location. After five reads the pointer will point to the same location as before starting the read procedure.

10.10 SERIAL PROTOCOL DESCRIPTION

In order to permit an efficient means of communication between the command controller and the DSP96002 chip, the following protocol is adopted. Before starting any debugging activity the command controller has to wait for an acknowledge that the chip has entered the Debug Mode. Note that in case of a breakpoint, trace, or software (F)DEBUGcc instruction, the acknowledge itself is the initiates the debug session. The command controller communicates with the chip by sending 8-bit commands that may be accompanied by 32-bit data. After sending a command the command processor waits for the chip to acknowledge execution of the command. The command processor may send a new command only after the chip has acknowledged execution of the previous command.

10.10.1 OnCE™ Commands

There are two types of commands: read commands (when the chip will deliver required data) and write commands (when the chip will receive data and will write the data in one of the on-chip resources). The commands are 8 bits long and have the format shown in Figure 10-9.

10.10.1.1 Register Select (RS4-RS0) Bits 0-4

The Register Select bits define which register is source(destination) for the read(write) operation.

RS4-RS0	Register Selected
00000	Debug Status/Control (OSCR)
00001	Breakpoint Counter Program (OPBC)
00010	Breakpoint Counter Data (ODBC)
00011	Trace Counter (OTC)
00100	Breakpoint Data Memory Higher-Equal (ODULR)
00101	Breakpoint Data Memory Lower-Equal (ODLLR)
00110	Breakpoint Program Memory Higher-Equal (OPULR)
00111	Breakpoint Program Memory Lower-Equal (OPLLR)
01000	Transfer Register (OGDBR)
01001	Program Data Bus Latch (OPDBR)
01010	Program Address Bus Latch for Fetch (OPABF)
01011	Program Instruction Latch (OPILR)
01100	Clear Program Breakpoint Counter
01101	Clear Data Breakpoint Counter
01110	Clear Trace Counter
01111	Reserved
10000	Reserved
10001	Program Address Bus FIFO and Increment Counter
10010	Program Address Bus Latch for Decode (OPABD)
10011	Reserved
101xx	Reserved
11xx0	Reserved
11x0x	Reserved
110xx	Reserved
11111	No Register Selected

7	6	5	4	3	2	1	0
R/W	GO	EX	RS4	RS3	RS2	RS1	RS0

Figure 10-9. OnCE™ Command Format

10.10.1.2 Exit Command (EX) Bit 5

If EX is set, leave Debug Mode and resume normal operation.

EX	Action
0	remain in Debug Mode
1	leave Debug Mode

10.10.1.3 Go Command (GO) Bit 6

If GO is set, execute instruction.

GO	Action
0	inactive (no action taken)
1	execute instruction

10.10.1.4 Read/Write Command (R/W) Bit 7

R/W Action

```
0   write the data associated with the command into the
    register specified by RS4-RS0
1   read the data contained in the register specified by
    RS4-RS0
```

10.11 DSP96002 TARGET SITE DEBUG SYSTEM REQUIREMENTS

A typical DSP96002 debug environment consists of a target system where the DSP96002 resides in the user defined hardware. The debug serial port interfaces to the command convertor over a 6 wire link consisting of the 4 OnCE™ wires, a ground and reset wire. The reset wire is optional and is only used to reset the DSP96002 and its associated circuitry.

The command controller acts as the medium between the DSP96002 target system and a host computer. The host computer interfaces to the controller using a standard RS232 three wire cable or the DSP96002 Application Development System parallel bus. A jumper option on the command controller board selects which method of communications will be used. This allows a variety of different host computers to communicate with the controller circuit.

The controller circuit provides several important functions. It acts as a DSP96002 serial debug port driver, host computer command interpreter, and DSP96002 controller. The DSP96002 acts as a slave when in the debug mode and provides data only upon request. The controller issues commands based on the host computer inputs from a user interface program which communicates with the user.

10.12 USING THE OnCE™

The following notations are used:

ACK = Wait for acknowledge on the DSO pin

CLK = issue 32 clocks to read out data from selected register

10.12.1 Begin Debug Activity

Most of the Debug activities will have the following beginning:

1. ACK
2. Save pipeline information:
   1. Send command READ PDB REGISTER
   2. ACK
   3. CLK
   4. Send command READ PIL REGISTER (instruction latch).
   5. ACK
   6. CLK
3. Read PAB FIFO and fetch/decode info (this step is optional):
   1. Send command READ PAB address for fetch
   2. ACK
   3. CLK
   4. Send command READ PAB address for decode
   5. ACK

6. CLK
7. Send command READ FIFO REGISTER (and increment pointer).
8. ACK
9. CLK
10. Send command READ FIFO REGISTER (and increment pointer).
11. ACK
12. CLK
13. Send command READ FIFO REGISTER (and increment pointer).
14. ACK
15. CLK
16. Send command READ FIFO REGISTER (and increment pointer).
17. ACK
18. CLK
19. Send command READ FIFO REGISTER (and increment pointer).
20. ACK
21. CLK

10.12.2 Displaying a specified register

1. Send command WRITE PDB REGISTER and GO (no EX).

   (ODEC selects PDB as destination for serial data.)
2. ACK
3. Send the 32-bit opcode: "MOVE reg,x:OGDB"

   (After 32 bits have been received, the PDB register drives the PDB. ODEC releases the chip from the "halt" state and the contents of the register specified in the instruction are loaded in the GDB REGISTER. The signal that marks the end of the instruction returns the chip to the "halt" state and an acknowledge is issued to the command controller.)
4. ACK
5. Send command READ GDB REGISTER
   (ODEC selects GDB as source for serial data and an acknowledge is issued to the command controller.)
6. ACK
7. CLK

10.12.3 Displaying X memory area starting from address xxxx

This command uses Rn to minimize serial traffic.

1. Send command WRITE PDB REGISTER and GO (no EX).
   (ODEC selects PDB as destination for serial data.)
2. ACK
3. Send the 32-bit opcode: "MOVE R0,x:OGDB"

   (After 32 bits have been received the PDB register drives the PDB. ODEC releases the chip from the "halt" state and the contents of R0 are loaded in the GDB REGISTER. The signal that marks the end of the instruction returns the chip to the "halt" state and an acknowledge is issued to the command controller.)

4. ACK
5. Send command READ GDB REGISTER

   (ODEC selects GDB as source for serial data and an acknowledge is issued to the command controller.
6. ACK
7. CLK

   (The command controller generates 32 clocks that shift out the contents of the GDB register. The value of R0 is thus saved and will be restored before exiting the Debug Mode.)
8. Send command WRITE PDB REGISTER (no GO, no EX).

   (ODEC selects PDB as destination for serial data.)
9. ACK
10. Send the 32-bit opcode: "MOVE #$xxxx,R0"

    (After 32 bits have been received, the PDB register drives the PDB. ODEC causes the core to load the opcode. An acknowledge is issued to the command controller.)
11. ACK
12. Send command WRITE PDB REGISTER and GO (no EX).

    (ODEC selects PDB as destination for serial data.)
13. ACK
14. Send the 32-bit 2nd word of: "MOVE #$xxxx,R0" (the xxxx field).

    (After 32 bits have been received, the PDB register drives the PDB. ODEC releases the chip from the "halt" state and the instruction starts execution. The signal that marks the end of the instruction returns the chip to the "halt" state and an acknowledge is issued to the command controller.)
15. ACK
16. Send command WRITE PDB REGISTER and GO (no EX).

    (ODEC selects PDB as destination for serial data.)
17. ACK
18. Send the 32-bit opcode: "MOVE X:(R0)+,x:OGDB"

    (After 32 bits have been received, the PDB register drives the PDB. ODEC releases the chip from the "halt" state and the contents of X:(R0) are loaded in the GDB REGISTER. The signal that marks the end of the instruction returns the chip to the "halt" state and an acknowledge is issued to the command controller.)
19. ACK
20. Send command READ GDB REGISTER

    (ODEC selects GDB as source for serial data and an acknowledge is issued to the command controller.)
21. ACK
22. CLK
23. Send command NO SELECTION and GO (no EX).

(ODEC releases the chip from the "halt" state and the instruction is executed again (in a "REPEAT-like" fashion. The signal that marks the end of the instruction returns the chip to the "halt" state and an acknowledge is issued to the command controller.)

24. ACK
25. Send command READ GDB REGISTER

    (ODEC selects GDB as source for serial data and an acknowledge is issued to the command controller.)

26. ACK
27. CLK
28. Repeat from step 23 until the entire memory area is examined. At the end of the process R0 has to be restored.

10.12.4 Returning from Debug Mode to Normal Mode

There are two cases for returning from the debug mode. Either control will be returned to the program that was running before debug was initiated or the registers will be changed to jump to a different program.

10.12.4.1 Case 1: Return to the previous program (Return to normal mode).

1. Send command WRITE PDB REGISTER (no GO, no EX).
   (ODEC selects the PDB as the destination for serial data – also, ODEC selects the on-chip PAB register as the source for the PAB bus. After the PAB was driven, an acknowledge is issued to the command controller.)
2. ACK
3. Send the 32 bits of the saved PIL (instruction latch) value.

   (After all the 32-bits have been received the PDB register drives the PDB. ODEC causes the core to load the opcode. An acknowledge is issued to the command controller.)

4. ACK
5. Send command WRITE PDB REGISTER (GO, EX).

   (ODEC selects PDB as destination for serial data.)

6. ACK
7. Send the 32-bit of the saved PDB value.

   (After 32 bits have been received, the PDB register drives the PDB. ODEC releases the chip from the "halt" state and the Debug Mode bit in OSCR is cleared. The chip continues to execute instructions until a Debug Mode condition occurs.)

10.12.4.2 Case 2: Jump to a new program (Go from address $xxxxxxxx).

1. Send command WRITE PDB REGISTER (no GO, no EX).

   (ODEC selects PDB as destination for serial data.)

2. ACK
3. Send 32 bits of the opcode of a two word jump instruction ($030c3f80) instead of the saved PIL (instruction latch) value.

   (After all the 32-bits have been received the PDB register drives the PDB. ODEC causes the core to load the opcode. An acknowledge is issued to the command controller.)

4. ACK
5. Send command WRITE PDB REGISTER (GO, EX).

   (ODEC selects PDB as destination for serial data.)
6. ACK
7. Send 32 bits of the target absolute address ($xxxxxxxx). The chip will resume fetching from the target address (you do not have to worry about the pipeline). Note that the trace counter will count this instruction so the current trace counter may need to be corrected if the trace mode enable bit in the OSCR has been set.

   (e. g., After 32 bits have been received, the PDB register drives the PDB. ODEC releases the chip from the "halt" state and the Debug Mode bit in OSCR is cleared. The chip executes first the jump instruction and will then fetch the instruction from the target address. The chip continues to execute instructions from that address until a Debug Mode condition occurs.)

APPENDIX A
INSTRUCTION SET DETAILS

A.1 INTRODUCTION

This appendix contains detailed information about each instruction defined in the DSP96002 instruction set. They are arranged in alphabetical order.

A.2 ADDRESSING MODES

Addressing modes are categorized by the ways in which they may be used. The following classifications will be used in the instruction definitions. Figure A-1 shows the various categories to which each addressing mode belongs.

Update (U) The addressing mode may be used to modify address registers without an associated data move.

Parallel (P) The addressing mode may be used in instructions where two effective addresses are required.

Memory (M) The addressing mode uses the effective addressing field and refers to operands in memory.

Alterable (A) The addressing mode refers to alterable (writable) registers or memory.

These addressing mode categories may be combined so that additional, more restrictive classifications may be defined. For example, the instruction descriptions may use a memory alterable classification. This refers to addressing modes which are both memory addressing modes and alterable addressing modes. Memory alterable addressing modes use the effective address to address memory and exclude the immediate addressing mode and the long displacement addressing mode.

The address register indirect addressing modes require that the offset register number be the same as the address register number. The assembler syntax "Nn" supports this future feature. The assembler syntax "N" may be used instead of "Nn" in the address register indirect memory addressing modes. If "N" is specified, the offset register number is the same as the address register number.

A.2.1 Addressing Mode Modifiers

The addressing mode selected in the instruction word is further specified by the contents of the address modifier register Mn. The addressing mode update modifiers are shown in Figure A-2. There are no restrictions on the use of modifier types with any memory addressing mode.

Addressing Mode	Mode	Reg	Addressing Categories U	P	M	A	Assembler Syntax
Register Direct							
Data or Control Register	–	–				X	Note 1
Address Register	–	–				X	Rn
Address Offset Register	–	–				X	Nn
Address Modifier Register	–	–				X	Mn
Address Register Indirect							
No Update	100	Rn	X	X	X	X	(Rn)
Postincrement by 1	011	Rn	X	X	X	X	(Rn)+
Postdecrement by 1	010	Rn	X	X	X	X	(Rn)-
Postincrement by Offset Nn	001	Rn	X	X	X	X	(Rn)+Nn
Postdecrement by Offset Nn	000	Rn	X		X	X	(Rn)- Nn
Indexed by Offset Nn	101	Rn			X	X	(Rn+Nn)
Predecrement by 1	111	Rn			X	X	-(Rn)
Long Displacement	–	Rn				X	(Rn+displacement)
PC Relative							
Long Displacement	–	–					(PC+displacement)
Short Displacement	–	–					(PC+xx)
Address Register	–	Rn					(PC+Rn)
Special							
Immediate Data	110	100			X		#Data
Absolute Address	110	000			X	X	label
Absolute Short Address	–	–				X	aa
I/O Short Address	–	–				X	pp
Immediate Short Data	–	–					#xx
Short Jump Address	–	–				X	xx
Implicit	–	–				X	

Note 1: Refer to Figure A-6 for the assembler syntax.

Figure A-1. Addressing Mode Summary

A.3 CONDITION CODE COMPUTATION

The CCR contains the condition code bits Carry (C), Overflow (V), Zero (Z), Negative (N), Infinity (I), Local Reject (LR), Reject ($\overline{R}$), and Accept (A).

The C, V, Z, N, I, LR, $\overline{R}$, and A bits are true condition code bits that reflect the condition of the result of a Data ALU operation. The C, V, Z and N bits are also affected by Address Generation Unit calculations during MOVETA instruction execution. The CCR bits are not affected by data transfers over the X, Y or global data buses.

The **standard definition** of the CCR bits is given below. Exceptions to these are given in Figure A-4.

C(Carry) Set if a carry is generated in an integer addition. Also set if a borrow is generated in an integer subtraction. The carry or borrow is generated out of the most significant bit (MSB) of the result. The carry bit is also modified by bit manipulation, rotate, and shift integer instructions as well as by the Address Generation Unit operation when executing MOVETA instructions. Cleared otherwise. The carry bit is not affected by floating-point instructions. The C bit is cleared during processor reset.

V(Overflow) Set if an arithmetic overflow occurs in a fixed point operation. This indicates that the result is not representable in the destination size. The V bit is not affected by floating-point operations unless they have a fixed point result. The overflow bit is also modified

Modifier MMMMMM M M	Address Calculation Arithmetic
0 0 0 0 0 0 0 0	Reverse Carry (Bit Reversed Update)
0 0 0 0 0 0 0 1	Modulo 2
0 0 0 0 0 0 0 2	Modulo 3
.	. .
.	. .
.	. .
0 0 F F F F F E	Modulo 16,777,215 ((2**24)-1)
0 0 F F F F F F	Modulo 16,777,216 (2**24)
0 1 x x x x x x	reserved
0 2 x x x x x x	reserved
.	. .
.	. .
.	. .
F D x x x x x x	reserved
F E x x x x x x	reserved
F F 0 0 0 0 0 0	reserved
F F 0 0 0 0 0 1	Multiple Wrap-Around Modulo 2
F F 0 0 0 0 0 3	Multiple Wrap-Around Modulo 4
F F 0 0 0 0 0 7	Multiple Wrap-Around Modulo 8
F F 3 F F F F F	Multiple Wrap-Around Modulo 2**22
F F 7 F F F F F	Multiple Wrap-Around Modulo 2**23
F F F F F F F F	Linear (Modulo 2**32)

where MMMMMMMM = Modifier Register Contents in Hex

Figure A-2. Address Modifier Summary

by Address Generation Unit operation when executing MOVETA instructions. Cleared otherwise. The V bit is cleared during processor reset.

Z(Zero) Set if the result equals zero. The Z bit is also set for floating-point -zero as well as +zero. The zero bit is also modified by Address Generation Unit operation when executing MOVETA instructions. Cleared otherwise. The Z bit is cleared during processor reset.

N(Negative) Set if the MSB of the result is set for integer operations or if the sign bit of the result is set for floating-point operations. The negative bit is also modified by Address Generation Unit operation when executing MOVETA instructions. Cleared otherwise. The N bit is cleared during processor reset.

I(Infinity) Set if the result of a floating-point operation is a signed infinity. Cleared otherwise. The I bit is not affected by fixed point operations but is affected by some conversion instructions. For example, if D is infinity, then executing FABS.S D will set the I bit. The I bit is cleared during processor reset.

LR(Local Reject) The LR bit is only affected by the compare instructions CMP, CMPG, FCMP and FCMPG. The LR bit is cleared during processor reset. See the example for the FCMPG instruction for additional information.

$\overline{R}$(Reject) The $\overline{R}$ bit is only affected by the compare instructions CMP, CMPG, FCMP and FCMPG. The $\overline{R}$ bit is calculated based on its previous value and the results of the current

compare instruction. The $\overline{R}$ bit is cleared during processor reset. See the example for the FCMPG instruction for additional information.

A(Accept) The A bit is only affected by the compare instructions CMP, CMPG, FCMP and FCMPG. The A bit is calculated based on its previous value and the results of the current compare instruction. The A bit is cleared during processor reset. See the example for the FCMPG instruction for additional information.

There are 16 theoretical combinations of N, Z, I and NAN for floating point results, but only eight combinations are possible in practice due to the exclusive nature of the data types described by the condition codes. The eight possible combinations are shown in Figure A-3.

Figure A-4 details how each instruction affects the condition codes. Figure A-4 gives the chip implementation viewpoint while the opcode descriptions in Section A-3 give the user viewpoint. For example, the Z bit computation for the CLR instruction is shown in the figure as the standard definition while the opcode description indicates that Z is always set.

N	Z	I	NAN	Result Data Type
0	0	0	0	+Normalized/Denormalized
1	0	0	0	- Normalized/Denormalized
0	1	0	0	+0
1	1	0	0	-0
0	0	1	0	+Infinity
1	0	1	0	-Infinity
0	0	0	1	+NaN
1	0	0	1	- NaN

Figure A-3.
Possible Combinations of the N, Z, I and NAN Bits for Floating-Point Results

Mnemonic	A	$\overline{R}$	LR	I	N	Z	V	C	Special Definitions
ABS	–	–	–	–	*	*	*	–	
ADD	–	–	–	–	–	*	*	*	
ADDC	–	–	–	–	*	?	*	*	Note 1
AND	–	–	–	–	*	*	0	–	
ANDC	–	–	–	–	*	*	0	–	
ANDI	?	?	?	?	?	?	?	?	Note 2
ASL	–	–	–	–	*	*	?	?	Note 3,4
ASR	–	–	–	–	*	*	0	?	Note 3
Bcc	–	–	–	–	–	–	–	–	
BCHG	?	?	?	?	?	?	?	?	Note 29
BCLR	?	?	?	?	?	?	?	?	Note 30
BFIND	–	–	–	–	?	?	0	–	Note 15,24
BRA	–	–	–	–	–	–	–	–	
BRCLR	–	–	–	–	–	–	–	–	
BRSET	–	–	–	–	–	–	–	–	
BScc	–	–	–	–	–	–	–	–	
BSCLR	–	–	–	–	–	–	–	–	
BSET	?	?	?	?	?	?	?	?	Note 31
BSR	–	–	–	–	–	–	–	–	
BSSET	–	–	–	–	–	–	–	–	
BTST	–	–	–	–	–	–	–	?	Note 5
CLR	–	–	–	–	*	*	0	–	
CMP	?	–	?	–	*	*	*	*	Note 32,33
CMPG	?	?	1	–	*	*	*	?	Note 23,32,34
DEBUGcc	–	–	–	–	–	–	–	–	
DEC	–	–	–	–	*	*	*	*	
DO	–	–	–	–	–	–	–	–	
DOR	–	–	–	–	–	–	–	–	
ENDDO	–	–	–	–	–	–	–	–	
EOR	–	–	–	–	*	*	0	–	
EXT	–	–	–	–	*	*	0	–	
EXTB	–	–	–	–	*	*	0	–	
FABS.S	–	–	–	*	*	*	–	–	
FABS.X	–	–	–	*	*	*	–	–	
FADD.S	–	–	–	*	*	*	–	–	
FADD.X	–	–	–	*	*	*	–	–	
FADDSUB.S	–	–	–	?	?	?	–	–	Note 9,10,11
FADDSUB.X	–	–	–	?	?	?	–	–	Note 9,10,11
FBcc	–	–	–	–	–	–	–	–	
FBScc	–	–	–	–	–	–	–	–	
FCLR	–	–	–	*	*	*	–	–	
FCMP	?	?	?	?	*	?	–	–	Note 27,35,36,37,40
FCMPG	?	?	1	?	*	?	–	?	Note 27,35,38,39,40
FCMPM	–	–	–	?	*	?	–	–	Note 27,40
FCOPYS.S	–	–	–	*	*	*	–	–	

Symbols: * Set according to the standard definition by the result
- Not affected by the operation
0 Cleared
1 Set
? Set according to the special computation definition by the result of the operation

Figure A-4. Condition Codes Computation

Mnemonic	A	$\overline{R}$	LR	I	N	Z	V	C	Special Definitions
FCOPYS.X	–	–	–	*	*	*	–	–	
FDEBUGcc	–	–	–	–	–	–	–	–	
FFcc	–	–	–	–	–	–	–	–	
FFcc.U	?	?	?	?	?	?	?	?	Note 21
FGETMAN	–	–	–	*	*	*	–	–	
FINT	–	–	–	*	*	*	–	–	
FJcc	–	–	–	–	–	–	–	–	
FJScc	–	–	–	–	–	–	–	–	
FLOAT.S	–	–	–	*	*	*	–	–	
FLOAT.X	–	–	–	*	*	*	–	–	
FLOATU.S	–	–	–	*	*	*	–	–	
FLOATU.X	–	–	–	*	*	*	–	–	
FLOOR	–	–	–	*	*	*	–	–	
FMPY//FADD.S	–	–	–	?	?	?	–	–	Note 9,10,11
FMPY//FADD.X	–	–	–	?	?	?	–	–	Note 9,10,11
FMPY//FADDSUB.S	–	–	–	?	?	?	–	–	Note 9,10,11
FMPY//FADDSUB.X	–	–	–	?	?	?	–	–	Note 9,10,11
FMPY//FSUB.S	–	–	–	?	?	?	–	–	Note 12,13,14
FMPY//FSUB.X	–	–	–	?	?	?	–	–	Note 12,13,14
FMPY.S	–	–	–	*	*	*	–	–	
FMPY.X	–	–	–	*	*	*	–	–	
FNEG.S	–	–	–	*	*	*	–	–	
FNEG.X	–	–	–	*	*	*	–	–	
FSCALE.S	–	–	–	*	*	*	–	–	
FSCALE.X	–	–	–	*	*	*	–	–	
FSEEDD	–	–	–	*	*	*	–	–	
FSEEDR	–	–	–	*	*	*	–	–	
FSUB.S	–	–	–	*	*	*	–	–	
FSUB.X	–	–	–	*	*	*	–	–	
FTFR.S	–	–	–	*	*	*	–	–	
FTFR.X	–	–	–	*	*	*	–	–	
FTRAPcc	–	–	–	–	–	–	–	–	
FTST	–	–	–	*	*	*	–	–	
GETEXP	–	–	–	?	*	*	–	–	Note 16
IFcc	–	–	–	–	–	–	–	–	
IFcc.U	?	?	?	?	?	?	?	?	Note 21
ILLEGAL	–	–	–	–	–	–	–	–	
INC	–	–	–	–	*	*	*	*	
INT	–	–	–	?	?	*	?	–	Note 16,17,24
INTRZ	–	–	–	?	?	?	–		Note 16,17,24
NTU	–	–	–	?	?	*	?	–	Note 16,24,41
INTURZ	–	–	–	?	?	*	?	–	Note 16,24,41
Jcc	–	–	–	–	–	–	–	–	
JCLR	–	–	–	–	–	–	–	–	
JMP	–	–	–	–	–	–	–	–	

Symbols: * Set according to the standard definition by the result
- Not affected by the operation
0 Cleared
1 Set
? Set according to the special computation definition by the result of the operation

Figure A-4. Condition Codes Computation (continued)

Mnemonic	A	$\overline{R}$	LR	I	N	Z	V	C	Special Definitions
JOIN	–	–	–	–	*	*	0	–	
JOINB	–	–	–	–	*	*	0	–	
JScc	–	–	–	–	–	–	–	–	
JSCLR	–	–	–	–	–	–	–	–	
JSET	–	–	–	–	–	–	–	–	
JSR	–	–	–	–	–	–	–	–	
JSSET	–	–	–	–	–	–	–	–	
LEA	?	?	?	?	?	?	?	?	Note 28
LRA	?	?	?	?	?	?	?	?	Note 28
LSL	–	–	–	–	*	*	0	?	Note 3
LSR	–	–	–	–	*	*	0	?	Note 3
MOVE	–	–	–	–	–	–	–	–	
MOVEC	?	?	?	?	?	?	?	?	Note 28
MOVEI	?	?	?	?	?	?	?	?	Note 28
MOVEM	?	?	?	?	?	?	?	?	Note 28
MOVEP	?	?	?	?	?	?	?	?	Note 28
MOVES	?	?	?	?	?	?	?	?	Note 28
MOVETA	–	–	–	–	?	?	?	?	Note 6,7,8,22
MPYS	–	–	–	–	*	*	?	–	Note 18
MPYU	–	–	–	–	0	*	?	–	Note 25
NEG	–	–	–	–	*	*	*	*	
NEGC	–	–	–	–	*	?	*	*	Note 1
NOP	–	–	–	–	–	–	–	–	
NOT	–	–	–	–	*	*	0	–	
OR	–	–	–	–	*	*	0	–	
ORC	–	–	–	–	*	*	0	–	
ORI	?	?	?	?	?	?	?	?	Note 19
REP	–	–	–	–	–	–	–	–	
RESET	–	–	–	–	–	–	–	–	
ROL	–	–	–	–	*	*	0	?	Note 26
ROR	–	–	–	–	*	*	0	?	Note 26
RTI	?	?	?	?	?	?	?	?	Note 20
RTR	?	?	?	?	?	?	?	?	Note 20
RTS	–	–	–	–	–	–	–	–	
SETW	–	–	–	–	*	*	0	–	
SPLIT	–	–	–	–	*	*	0	–	
SPLITB	–	–	–	–	*	*	0	–	
STOP	–	–	–	–	–	–	–	–	
SUB	–	–	–	–	*	*	*	*	
SUBC	–	–	–	–	*	?	*	*	Note 1
TFR	–	–	–	–	–	–	–	–	
TRAPcc	–	–	–	–	–	–	–	–	
TST	–	–	–	–	*	*	0	–	
WAIT	–	–	–	–	–	–	–	–	

Symbols: * Set according to the standard definition by the result
- Not affected by the operation
0 Cleared
1 Set
? Set according to the special computation definition by the result of the operation

Figure A-4. Condition Codes Computation (continued)

Note 1	Z - Cleared if the result is not zero. Unchanged otherwise.
Note 2	All ? Bits - Cleared if corresponding bit in immediate data is cleared and the operand is CCR. Not affected otherwise.
Note 3	C - Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero.
Note 4	V - Set if the MSB is changed any time during the shift operation. Cleared otherwise.
Note 5	C - Set if bit #n of the source operand is set. Cleared otherwise.
Note 6	C - For increment addressing modes: Set if carry occurred out of the MSB during address calculation with linear modifier or carry occurred out of the LSB during address calculation with reverse carry modifier. Cleared otherwise. For decrement addressing modes: Set if borrow occurred out of the MSB during address calculation with linear modifier or borrow occurred out of the LSB during address calculation with reverse carry modifier. Cleared otherwise.
Note 7	V - Set if overflow occurred out of the MSB during the address calculation with a linear modifier. Set if overflow occurred out of the least significant bit (LSB) during the address calculation with a reverse carry modifier. Set if wraparound occurred during the address calculation with a modulo modifier. Set if at least one wrap-around occurred during address calculation with a multiple wrap-around modulo modifier. Cleared otherwise.
Note 8	Z - Set if the result of the address calculation is zero. Cleared otherwise.
Note 9	I - Set if the result of the addition is infinity. Cleared otherwise.
Note 10	N - Set if the result of the addition is negative. Cleared otherwise.
Note 11	Z - Set if the result of the addition is zero. Cleared otherwise.
Note 12	I - Set if the result of the subtraction is infinity. Cleared otherwise.
Note 13	N - Set if the result of the subtraction is negative. Cleared otherwise.
Note 14	Z - Set if the result of the subtraction is zero. Cleared otherwise.
Note 15	Z - Set if the source operand is zero. Cleared otherwise.
Note 16	I - Set if the source operand is infinity. Cleared otherwise.
Note 17	V - Set if source operand is a NaN, infinity, or its magnitude is too big to be representable in the integer number range. Cleared otherwise.
Note 18	V - Cleared if the most significant 32 bits of the 64-bit result are the sign extension of the least significant 32 bits. Set otherwise.
Note 19	All ? Bits - Set if corresponding bit in immediate data is set and the operand is CCR. Not affected otherwise.
Note 20	All ? Bits - Set according to the value pulled from the stack.
Note 21	All ? Bits - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.
Note 22	N - Set if the MSB of the result of the address calculation with linear modifier is set. Set if the LSB of the result of the address calculation with reverse carry modifier is set. Set if the MSB of the result of the address calculation with modulo modifier is set. Cleared otherwise.
Note 23	C - Set if result is negative without overflow. Set if result is positive with overflow. Cleared otherwise.
Note 24	N - Set if the source operand is negative. Cleared otherwise.
Note 25	V - Cleared if the most significant 32 bits of the 64-bit result are zero. Set otherwise.
Note 26	C - Set if the last bit shifted out of the operand is set. Cleared otherwise.
Note 27	I - Set if any one of the source operands is infinity. Cleared otherwise.

Note 28 All ? bits - If SR is specified as a destination operand, set according to the corresponding bit of the source operand. Not affected otherwise.

Note 29 All ? bits - If SR is specified as destination operand, and A, $\overline{R}$, LR, I, N, Z, V or C is selected, then the selected bit will be changed. If SR is not specified, then C will be set if bit #n of the source operand is set and cleared if bit #n of the source operand is set. Not affected otherwise.

Note 30 All ? bits - If SR is specified as destination operand, and A, $\overline{R}$, LR, I, N, Z, V or C is selected, then the selected bit will be cleared. If SR is not specified, then C will be set if bit #n of the source operand is set set and cleared if bit #n of the source operand is set. Not affected otherwise.

Note 31 All ? bits - If SR is specified as destination operand, and A, $\overline{R}$, LR, I, N, Z, V or C is selected, then the selected bit will be set. If SR is not specified, then C will be set if bit #n of the source operand is set set and cleared if bit #n of the source operand is set. Not affected otherwise.

Note 32 A - Cleared if result is negative without overflow. Cleared if result is positive with overflow. Not affected otherwise.

Note 33 LR - Cleared if result is positive without overflow. Cleared if result is negative with overflow. Not affected otherwise.

Note 34 R - Cleared if LR was set and result is negative without overflow. Cleared if LR was set and result is positive with overflow. Not affected otherwise.

Note 35 A - Cleared if result is a NaN. Cleared if result is negative and not zero. Not affected otherwise.

Note 36 LR - Cleared if result is positive, zero or NaN. Not affected otherwise.

Note 37 R - Cleared if result is a NaN. Not affected otherwise.

Note 38 R - Cleared if result is a NaN. Cleared if result is negative and not zero and LR was set. Not affected otherwise.

Note 39 C - Set if result is a NaN. Set if result is negative and not zero. Cleared otherwise.

Note 40 Z - Set if source operands are equal. Cleared otherwise.

Note 41 V - Set if source operand is a NaN, infinity or negative non-zero. Set if positive source operand is too big to be representable in the integer number range. Cleared otherwise.

A.4 EXCEPTION STATUS BITS COMPUTATION

Floating-point operations affect the seven status bits located in the IER register. The **standard definitions** of the ER bits is given below. These definitions are based on the ANSI/IEEE Standard 754-1985 which can be ordered from:

IEEE

345 East 47th Street

New York, N.Y. 10017

Additional information (particularly relating to test cases) can be found in J. T. Coonen, An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic, Computer, 1980, pages 68-79. Examples of the use of these bits are given in **Section 4.6**.

INX (Inexact) - Set if a floating-point mantissa, considered as having infinite precision, has too many significant bits to be represented exactly in the current rounding precision. That is, a result is inexact if there was a loss of accuracy due to rounding. Cleared otherwise. The INX bit is not affected by fixed point operations. The INX bit is cleared during processor reset.

DZ (Division by Zero) - Set if the dividend is a finite nonzero number and the divisor is zero. The result will be a correctly signed infinity (generated by the exclusive OR of the signs of the source operands). Cleared otherwise. The DZ bit is not affected by fixed point operations. The DZ bit is cleared during processor reset.

UNF (Underflow) - Set if tininess is detected, that is, set if an intermediate unrounded result of a floating-point operation is too small to be represented in a floating-point data register with the selected rounding precision as a normalized result. The UNF bit is not affected by fixed point operations. The UNF bit is cleared during processor reset.

OVF (Overflow) - Set if a rounded floating-point intermediate result is too large to be represented in a floating-point data register with the selected rounding precision. If the result is greater than or equal to $|\pm 2^{E_{max}-1}|$ the OVF bit will be set; otherwise it will be cleared. The largest single precision IEEE floating-point number representable in memory is $7F7FFFFF. It is possible to set OVF and have INX cleared if the overflow is exact. The OVF bit is not affected by fixed point operations. The OVF bit is cleared during processor reset.

OPERR (Operand Error) - Set if an operation has no mathematical interpretation for the given operands. Cleared otherwise. The result will be a quiet NaN if the destination has a floating-point format. Examples of operations which generate quiet NaNs and set the OPERR bit are $(+\infty)+(-\infty)$, $0\times\infty$, and $\sqrt{-n}$. The OPERR bit is not affected by fixed point operations. The OPERR bit is cleared during processor reset.

SNAN (Signaling NaN) - Set when a signaling NaN is involved in an arithmetic floating-point operation. Cleared otherwise. The result will be a non-signaling NaN obtained by setting the most significant fraction bit of the significand. The SNAN bit is not affected by fixed point operations. The SNAN bit is cleared during processor reset.

NaN (Not-a-Number) - Set if the result of a floating-point operation is a NaN. Cleared otherwise. The NAN bit is not affected by fixed point operations but is affected by some conversion instructions. The NAN bit is cleared during processor reset.

UNCC (Unordered Condition) - Set if a non-aware floating-point conditional instruction (FBcc, FJcc, FFcc, etc.) is executed when the NAN bit is set (the unordered condition). Not affected otherwise. The UNCC bit is cleared during processor reset.

The IEEE Standard 754-1985 for Binary Floating-Point Arithmetic (754-standard) explicitly specifies how to handle comparison operations when one or more of the operands is a NaN (which the 754-standard created). However, a great deal of software has been written and some is still being written, in an environment which is not aware of the NaN data type. In order to port such software to an IEEE 754-1985 standard environment, a special bit, the unordered condition code (UNCC) bit, was created in the DSP96002. This bit can be used when porting the software to ensure that the intended branch is taken, or an exception is generated, when the ported program processes a NaN.

Typically, branches are taken on predicates and their compliments assuming the operands can be ordered (i. e., placed on the real number line). However, NaNs, by definition, do not have any order relationship to numbers on the real number line. For example, when a FCMP is executed the NaN bit in the ER will be set if either operand is a NaN. When a subsequent conditional instruction (e. g. FBGT) is executed, the UNCC bit in the ER will be set if the NaN bit in the ER was set when the conditional branch instruction was executed. Because one of the operands was a NaN, the branch will be failed. If the original author was "aware" of NaNs, then the decision may be wrong. In this case, it may be prudent to insert a FBERR instruction following the conditional branch instruction in the failed path. This is because one of the error conditions that the FBERR instruction detects is that the UNCC bit was set. The error handler will be aware of NaNs and take corrective action.

It could be argued that the same result would be achieved by executing a Floating-Point Branch on an unordered (FBUN) instruction instead of the FBERR instruction thereby eliminating the

need for the UNCC bit. This would be true except for the way in which the 754-standard treats the equal and "not equal" predicates. From the condition code tables associated with the floating-point conditional instructions, it can be seen that the UNCC bit will not be set if one or both of the operands is a NaN. This is because the 754-standard recognizes that operands do not have to be ordered to be tested for equality (i. e., UNCC will not be affected when executing FBEQ or FBNE). That is, the same branch should be taken in a programming environment which was aware of the IEEE binary floating-point number system as in one which was not aware. This is not the case for inequality predicates.

In summary, conditional predicates whose outcome may depend upon "NaN awareness" by the original author of the program are those involving inequalities. The UNCC bit has been provided on the DSP96002 to aid in porting programs written in an IEEE non-aware environment to the DSP96002 (IEEE aware environment). FBERR instructions which branch on UNCC set can be inserted in branches which could have been incorrectly taken due to NaN operands being involved in the FCMP. When executing programs whose author was "NaN aware", the UNCC bit can be ignored. When executing programs whose author was "NaN unaware", the UNCC bits status should be tested since the original author’s intentions are unclear.

Figure A-5 details how each floating-point instruction affects the ER register bits.

Mnemonic	UNCC	NAN	SNAN	OPERR	OVF	UNF	DZ	INX	Special Definitions
ABS	–	–	–	–	–	–	–	–	
ADD	–	–	–	–	–	–	–	–	
ADDC1	–	–	–	–	–	–	–	–	
AND	–	–	–	–	–	–	–	–	
ANDC	–	–	–	–	–	–	–	–	
ANDI	?	?	?	?	?	?	?	?	Note 9
ASL	–	–	–	–	–	–	–	–	
ASR	–	–	–	–	–	–	–	–	
Bcc	–	–	–	–	–	–	–	–	
BCHG	?	?	?	?	?	?	?	?	Note 13
BCLR	?	?	?	?	?	?	?	?	Note 14
BFIND	–	–	–	–	–	–	–	–	
BRA	–	–	–	–	–	–	–	–	
BRCLR	–	–	–	–	–	–	–	–	
BRSET	–	–	–	–	–	–	–	–	
BScc	–	–	–	–	–	–	–	–	
BSCLR	–	–	–	–	–	–	–	–	
BSET	?	?	?	?	?	?	?	?	Note 15
BSR	–	–	–	–	–	–	–	–	
BSSET	–	–	–	–	–	–	–	–	
BTST	–	–	–	–	–	–	–	–	
CLR	–	–	–	–	–	–	–	–	
CMP	–	–	–	–	–	–	–	–	
CMPG	–	–	–	–	–	–	–	–	
DEBUGcc	–	–	–	–	–	–	–	–	
DEC	–	–	–	–	–	–	–	–	
DO	–	–	–	–	–	–	–	–	
DOR	–	–	–	–	–	–	–	–	
ENDDO	–	–	–	–	–	–	–	–	
EOR	–	–	–	–	–	–	–	–	
EXT	–	–	–	–	–	–	–	–	
EXTB	–	–	–	–	–	–	–	–	
FABS.S	0	*	*	0	*	*	0	*	
FABS.X	0	*	*	0	*	*	0	*	
FADD.S	0	*	*	?	*	*	0	*	Note 18
FADD.X	0	*	*	?	*	*	0	*	Note 18
FADDSUB.S	0	?	*	?	?	?	0	?	Note 2, 3, 4, 5, 7
FADDSUB.X	0	?	*	?	?	?	0	?	Note 2, 3, 4, 5, 7
FBcc	*	–	–	–	–	–	–	–	
FBScc	*	–	–	–	–	–	–	–	
FCLR	0	*	0	0	0	0	0	0	
FCMP	0	*	*	0	0	0	0	0	
FCMPG	0	*	*	0	0	0	0	0	
FCMPM	0	*	*	0	0	0	0	0	
FCOPYS.S	0	*	*	0	*	*	0	*	

SYMBOLS:
* set according to the standard definition by the result
- not affected by the operation
0 cleared
1 set
? set according to the special computation definition by the result of the operation

Figure A-5. ER Exception Bits Computation

Mnemonic	UNCC	NAN	SNAN	OPERR	OVF	UNF	DZ	INX	Special Definitions
FCOPYS.X	0	*	*	0	*	*	0	*	
FDEBUGcc	*	–	–	–	–	–	–	–	
FFcc	*	–	–	–	–	–	–	–	
FFcc.U	*	?	?	?	?	?	?	?	Note 26
FGETMAN	0	*	*	?	0	0	0	0	Note 27
FINT	0	*	*	0	0	0	0	*	
FJcc	*	–	–	–	–	–	–	–	
FJScc	*	–	–	–	–	–	–	–	
FLOAT.S	0	0	0	0	0	0	0	*	
FLOAT.X	0	0	0	0	0	0	0	*	
FLOATU.S	0	0	0	0	0	0	0	*	
FLOATU.X	0	0	0	0	0	0	0	*	
FLOOR	0	*	*	0	0	0	0	*	
FMPY//FADD.S	0	?	?	?	?	?	0	?	Notes 1,7,19,22,23,24
FMPY//FADD.X	0	?	?	?	?	?	0	?	Notes 1,7,19,22,23,24
FMPY//FADDSUB.S	0	?	?	?	?	?	0	?	Notes 1,7,21,22,23,24
FMPY//FADDSUB.X	0	?	?	?	?	?	0	?	Notes 1,7,21,22,23,24
FMPY//FSUB.S	0	?	?	?	?	?	0	?	Notes 1,8,20,22,23,24
FMPY//FSUB.X	0	?	?	?	?	?	0	?	Notes 1,8,20,22,23,24
FMPY.S	0	*	*	?	*	*	0	*	Note 6
FMPY.X	0	*	*	?	*	*	0	*	Note 6
FNEG.S	0	*	*	0	*	*	0	*	
FNEG.X	0	*	*	0	*	*	0	*	
FSCALE.S	0	*	*	0	*	*	0	*	
FSCALE.X	0	*	*	0	*	*	0	*	
FSEEDD	0	*	*	0	*	*	0	0	
FSEEDR	0	*	*	?	0	0	0	0	Note 31
FSUB.S	0	*	*	?	*	*	0	*	Note 28
FSUB.X	0	*	*	?	*	*	0	*	Note 28
FTFR.S	0	*	*	0	*	*	0	*	
FTFR.X	0	*	*	0	*	*	0	*	
FTRAPcc	*	–	–	–	–	–	–	–	
FTST	0	*	*	0	0	0	0	0	
GETEXP	0	?	*	?	0	0	0	0	Notes 29,30
IFcc	–	–	–	–	–	–	–	–	
IFcc.U	–	?	?	?	?	?	?	?	Note 26
ILLEGAL	–	–	–	–	–	–	–	–	
INC	–	–	–	–	–	–	–	–	
INT	0	?	*	?	0	0	0	?	Notes 12,17,29
INTRZ	0	?	*	?	0	0	0	?	Notes 12,17,29
INTU	0	?	*	?	0	0	0	?	Notes 12,25,29
INTURZ	0	?	*	?	0	0	0	?	Notes 12,25,29
Jcc	–	–	–	–	–	–	–	–	
JCLR	–	–	–	–	–	–	–	–	
JMP	–	–	–	–	–	–	–	–	

SYMBOLS:
- * set according to the standard definition by the result
- \- not affected by the operation
- 0 cleared
- 1 set
- ? set according to the special computation definition by the result of the operation

Figure A-5. ER Exception Bits Computation (Continued)

Mnemonic	A	$\overline{R}$	LR	I	N	Z	V	C	Special Definitions
JOIN	–	–	–	–	–	–	–	–	
JOINB	–	–	–	–	–	–	–	–	
JScc	–	–	–	–	–	–	–	–	
JSCLR	–	–	–	–	–	–	–	–	
JSET	–	–	–	–	–	–	–	–	
JSR	–	–	–	–	–	–	–	–	
JSSET	–	–	–	–	–	–	–	–	
LEA	?	?	?	?	?	?	?	?	Note 16
LRA	?	?	?	?	?	?	?	?	Note 16
LSL	–	–	–	–	–	–	–	–	
LSR	–	–	–	–	–	–	–	–	
MOVE	–	–	–	–	–	–	–	–	
MOVEC	?	?	?	?	?	?	?	?	Note 16
MOVEI	?	?	?	?	?	?	?	?	Note 16
MOVEM	?	?	?	?	?	?	?	?	Note 16
MOVEP	?	?	?	?	?	?	?	?	Note 16
MOVES	?	?	?	?	?	?	?	?	Note 16
MOVETA	–	–	–	–	–	–	–	–	
MPYS	–	–	–	–	–	–	–	–	
MPYU	–	–	–	–	–	–	–	–	
NEG	–	–	–	–	–	–	–	–	
NEGC	–	–	–	–	–	–	–	–	
NOP	–	–	–	–	–	–	–	–	
NOTB	–	–	–	–	–	–	–	–	
OR	–	–	–	–	–	–	–	–	
ORC	–	–	–	–	–	–	–	–	
ORI	?	?	?	?	?	?	?	?	Note 10
REP	–	–	–	–	–	–	–	–	
RESET	–	–	–	–	–	–	–	–	
ROL	–	–	–	–	–	–	–	–	
ROR	–	–	–	–	–	–	–	–	
RTI	?	?	?	?	?	?	?	?	Note 11
RTR	?	?	?	?	?	?	?	?	Note 11
RTS	–	–	–	–	–	–	–	–	
SETW	–	–	–	–	–	–	–	–	
SPLIT	–	–	–	–	–	–	–	–	
SPLITB	–	–	–	–	–	–	–	–	
STOP	–	–	–	–	–	–	–	–	
SUB	–	–	–	–	–	–	–	–	
SUBC	–	–	–	–	–	–	–	–	
TFR	–	–	–	–	–	–	–	–	
TRAPcc	–	–	–	–	–	–	–	–	
TST	–	–	–	–	–	–	–	–	
WAIT	–	–	–	–	–	–	–	–	

SYMBOLS:
* set according to the standard definition by the result
- not affected by the operation
0 cleared
1 set
? set according to the special computation definition by the result of the operation

Figure A- 5. ER Exception Bits Computation (Continued)

Note 1 SNAN - Set if anyone of the source operands is a signaling NaN. Cleared otherwise.

Note 2 OPERR - Set if the operands of the floating-point addition are opposite-signed infinities or if the operands of the floating-point subtraction are like-signed infinities. Cleared otherwise.

Note 3 UNF - Set if the addition or subtraction operation underflows. Cleared otherwise.

Note 4 INX - Set if the addition or subtraction result is inexact. Cleared otherwise.

Note 5 OVF - Set if the addition or subtraction overflows. Cleared otherwise.

Note 6 OPERR -Set if one operand is infinity and the other is zero. Cleared otherwise.

Note 7 NAN - Set if the result of the addition is a NaN. Cleared otherwise.

Note 8 NAN - Set if the result of the subtraction is a NaN. Cleared otherwise.

Note 9 All ? bits - Cleared if corresponding bit in immediate data is cleared and the operand is ER. Not affected otherwise.

Note 10 All ? bits - Set if corresponding bit in immediate data is set and the operand is ER. Not affected otherwise.

Note 11 All ? bits - Set according to the value pulled from the stack.

Note 12 INX - Set if the floating-point number has no exact integer representation. Cleared otherwise.

Note 13 All ? bits - If SR is specified as destination operand, and INX, DZ, UNF, OVF, OPERR, SNAN, NAN or UNCC is selected, then the selected bit will be changed. Not affected otherwise.

Note 14 All ? bits - If SR is specified as destination operand, and INX, DZ, UNF, OVF, OPERR, SNAN, NAN or UNCC is selected, then the selected bit will be cleared. Not affected otherwise.

Note 15 All ? bits - If SR is specified as destination operand, and INX, DZ, UNF, OVF, OPERR, SNAN, NAN or UNCC is selected, then the selected bit will be set. Not affected otherwise.

Note 16 All ? bits - If SR is specified as a destination operand, set according to the corresponding bit of the source operand. Not affected otherwise.

Note 17 OPERR - Set if the source operand is a NaN or infinity. Also set if overflow occurred. Cleared otherwise.

Note 18 OPERR - Set if the operands are opposite-signed infinities. Cleared otherwise.

Note 19 OPERR - Set if one of the multiply operands is infinity and the other is zero. Set if the addition operands are opposite-signed infinities. Cleared otherwise.

Note 20 OPERR - Set if one of the multiply operands is infinity and the other is zero. Set if the subtraction operands are like-signed infinities. Cleared otherwise.

Note 21 OPERR - Set if one of the multiply operands is infinity and the other is zero. Set if the subtraction operands are like-signed infinities. Set if the addition operands are opposite-signed infinities. Cleared otherwise.

Note 22 OVF - Set if anyone of the operations overflows. Cleared otherwise.

Note 23 UNF - Set if anyone of the operations underflows. Cleared otherwise.

Note 24 INX - Set if the result of one or more operations is inexact. Cleared otherwise.

Note 25 OPERR - Set if the source operand is a NaN, infinity or negative non-zero. Also set if overflow occurred. Cleared otherwise.

Note 26 All ? bits - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

Note 27 OPERR - Set if the source operand is infinity. Cleared otherwise.

Note 28 OPERR - Set if the operands are like-signed infinities. Cleared otherwise.

Note 29 NAN - Set if the source operand is a NaN. Cleared otherwise.

Note 30 OPERR - Set if the source operand is infinity, zero or NaN. Cleared otherwise.

Note 31 OPERR - Set if the source operand is less than zero. Cleared otherwise.

A.5 IEEE EXCEPTION BITS COMPUTATION

The IEEE Exception bits are the five exception bits required by the IEEE standard for trap disabled operations. They actually record a history of all floating-point exceptions which have occurred since the user last cleared the IER register. At the end of each floating-point operation, the bits of the ER are logically combined and then are logically ORed into the existing IER bits creating "sticky" floating-point exception bits which can be polled at the end of a series of floating-point operations. The standard definition of the IER bits and the complete IER exception flag computation rules are given below.

SINX (IEEE Inexact) - signaled when the rounded result of an operation is not exact or if it overflows without an overflow trap.

SINX = SINX v (OVF v INX)

SDZ (IEEE Division by Zero) - signaled if the dividend is a finite nonzero number and the divisor is zero.

SDZ = SDZ v DZ

SUNF (IEEE Underflow) - signaled when both tininess and loss of accuracy have been detected. Tininess is detected before round (see definition of UNF in the ER register). Loss of accuracy is detected as an inexact result (see definition of INX in the ER register).

SUNF = SUNF v (UNF & INX)

SOVF (IEEE Overflow) - signaled when the destination format largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded.

SOVF = SOVF v OVF

SIOP (IEEE Invalid Operation) - signaled if an operand is invalid for the operation to be performed.

SIOP = SIOP v (UNCC v SNAN v OPERR)

A.6 NOTATION

Symbols are used to abbreviate operands and operations in each instruction description. Figure A-6 lists the symbols used and their respective meanings.

Operands

Data ALU

Dn	Data ALU Registers, n= 0-9, SP/SEP/Integer reference as specified by the Data ALU operation.
Dn.S	Floating-Point Registers, n= 0-9 (96 bits) SP reference
Dn.D	Floating-Point Registers, n= 0-9 (96 bits) DP reference
Dn.L	Integer Registers, n= 0-9 (32 bits, Low part of Dn)
Dn.M	Integer Registers, n= 0-9 (32 bits, Middle part of Dn)
Dn.H	Integer Registers, n= 0-9 (32 bits, High part of Dn)
Dn.ML	Long Integer Register, n=0-9 (Dn.M:Dn.L, 64 bits)

Address Generation Unit

Rn	Address registers R0 through R7 (32 bits)
Nn	Address offset registers N0 through N7 (32 bits)
Mn	Address modifier registers M0 through M7 (32 bits)

Program Controller

PC	Program counter (32 bits)
MR	Mode register (8 bits)
ER	Exception register (8 bits)
IER	IEEE Exception register (8 bits)
CCR	Condition code register (8 bits)
SR	Status register (32 bits)
OMR	Operating mode register (32 bits)
LA	Hardware loop address register (32 bits)
LC	Hardware loop counter (32 bits)
SP	System stack pointer (32 bits)
SS	System stack RAM (15 x 64 bits)
SSH	Upper 32 bits of the contents of the current top of stack.
SSL	Lower 32 bits of the contents of the current top of stack.

Addresses

ea	Effective address
xxxx	Absolute address (32 bits)
xx	Short jump address (15 bits sign extended)
pp	I/O short address (7 bits one extended)
aa	Absolute short address (7 bits zero extended)
<...>	The contents of the specified address
X:	X memory reference (32 bits)
Y:	Y memory reference (32 bits)
L:	Long memory reference - X concatenated with Y (64 bits)
P:	Program memory reference (32 bits)

Figure A-6. Instruction Description Notation

Operators

Miscellaneous

#xx	Immediate short data (16 bits sign extended)
#xxx	Immediate short data (19 bits zero extended)
#Data	Immediate data (32 bits)
#shift, #bit, or #bits	Immediate short data (5 or 6 bits)
#byte	Immediate short data (8 bits)
S,Sn	Source operand register
D,Dn	Destination operand register
D{n}	Bit n of D affected
D(8,9)	Destination Operand Register D8 or D9 only
D(MS)	Most significant word of double precision or long integer destination
D(LS)	Least significand word of double precision or long integer destination
S(MS)	Most significant word of double precision or long integer source
S(LS)	Least significant word of double precision or long integer source
R	Round optional rounding precision
I1,I0	Interrupt priority level in SR
LF	Loop flag in SR

Unary

-	Negation
~	Logical NOT
PUSH	Push onto SS
PULL	Pull from SS
READ	Read top of SS
PURGE	Delete top of SS
| |	Absolute Value

Binary

+	Addition
-	Subtraction
*	Multiplication
/	Division
v	Logical Inclusive OR
&	Logical AND
&&	Logical Exclusive OR
$\rightarrow$	Is transferred to
:	Concatenation

Miscellaneous

(..)	Indicates an optional operand or operation
Sign Ext	Sign Extension
Zero	Zero a register

Figure A-6. Instruction Description Notation (Continued)

A.7 OPCODE DESCRIPTIONS

The following pages define each opcode in the instruction set and its associated operands. Instructions which may use a parallel move operation are indicated by the notation **"(parallel data bus move)"** in the Operation portion of the description. Detailed information on each parallel move operation is given in the MOVE instruction description.

Operation:

|-D.L| $\rightarrow$ D.L (parallel data bus move)

Assembler Syntax:

ABS D (move syntax - see the MOVE instruction description.)

Description:

Take the absolute value of the destination operand low portion and store the result in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Set if result overflows. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ABS D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0100	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

D.L + S.L → D.L (parallel data bus move)

Assembler Syntax:

ADD S,D (move syntax - see the MOVE instruction description.)

Description:

Add the low portion of the two specified operands and store the result in the low portion of the destination operand D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if carry is generated from MSB of the result. Cleared otherwise.
V	- Set if result overflows. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ADD S,D (move syntax - see the MOVE instruction description.)

31 … 14	13 … 0			
DATA BUS MOVE FIELD	00	1sss	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

ADDC — Add with Carry

Operation:

D.L + S.L + C → D.L (parallel data bus move)

Assembler Syntax:

ADDC S,D (move syntax - see the MOVE instruction description.)

Description:

Add the low portion of the two specified operands along with the C bit of the condition code register and store the result in the low portion of destination operand D. When doing multiple precision addition, the higher precision long words of the input variables must be moved to the low portion of the Dn register.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

- C - Set if carry is generated from the MSB of the result. Cleared otherwise.
- V - Set if result overflows. Cleared otherwise.
- Z - Cleared if the result is not zero. Unchanged otherwise.
- N - Set if result is negative. Cleared otherwise.
- I - Not affected.
- LR - Not affected.
- $\overline{R}$ - Not affected.
- A - Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ADDC S,D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0
DATA BUS MOVE FIELD	00 1sss uu01 1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Logical AND

Operation:

D.L & S.L → D.L (parallel data bus move)

Assembler Syntax:

AND S,D (move syntax - see the MOVE instruction description.)

Description:

Logically AND the low portion of the two specified operands and store the result in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C - Not affected.
V - Always cleared.
Z - Set if result is zero. Cleared otherwise.
N - Set if result is negative. Cleared otherwise.
I - Not affected.
LR - Not affected.
$\overline{R}$ - Not affected.
A - Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: AND S,D (move syntax - see the MOVE instruction description.)

Instruction Fields:

31 — 14	13 — 0
DATA BUS MOVE FIELD	00 0sss uu00 1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

ANDC Logical AND with Complement ANDC

Operation:

D.L & ~S.L $\rightarrow$ D.L (parallel data bus move)

Assembler Syntax:

ANDC S,D (move syntax - see the MOVE instruction description.)

Description:

Logically AND the low portion of D with the logical complement of the low portion of S, and store the result in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C - Not affected.
V - Always cleared.
Z - Set if result is zero. Cleared otherwise.
N - Set if result is negative. Cleared otherwise.
I - Not affected.
LR - Not affected.
$\overline{R}$ - Not affected.
A - Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: AND S,D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0
DATA BUS MOVE FIELD	11 0sss 1000 1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

ANDI — AND Immediate to Control Register

Operation:

D & #xx $\rightarrow$ D

Assembler Syntax:

AND(I) #Byte,D

Description:

Logically AND the contents of the control register with an 8-bit immediate operand. The result is stored back into the specified control register. See **Section A.10** for restrictions.

CCR Condition Codes:

For CCR operand:

C	- Cleared if bit 0 of the immediate operand is cleared. Not affected otherwise.
V	- Cleared if bit 1 of the immediate operand is cleared. Not affected otherwise.
Z	- Cleared if bit 2 of the immediate operand is cleared. Not affected otherwise.
N	- Cleared if bit 3 of the immediate operand is cleared. Not affected otherwise.
I	- Cleared if bit 4 of the immediate operand is cleared. Not affected otherwise.
LR	- Cleared if bit 5 of the immediate operand is cleared. Not affected otherwise.
$\overline{R}$	- Cleared if bit 6 of the immediate operand is cleared. Not affected otherwise.
A	- Cleared if bit 7 of the immediate operand is cleared. Not affected otherwise.

For OMR, MR, IER, ER operands:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

For ER operand:

INX	- Cleared if bit 0 of the immediate operand is cleared. Not affected otherwise.
DZ	- Cleared if bit 1 of the immediate operand is cleared. Not affected otherwise.
UNF	- Cleared if bit 2 of the immediate operand is cleared. Not affected otherwise.
OVF	- Cleared if bit 3 of the immediate operand is cleared. Not affected otherwise.
OPERR	- Cleared if bit 4 of the immediate operand is cleared. Not affected otherwise.
SNAN	- Cleared if bit 5 of the immediate operand is cleared. Not affected otherwise.
NAN	- Cleared if bit 6 of the immediate operand is cleared. Not affected otherwise.
UNCC	- Cleared if bit 7 of the immediate operand is cleared. Not affected otherwise.

For OMR, MR, IER, CCR operands:

INX	- Not affected.
DZ	- Not affected.
UNF	- Not affected.
OVF	- Not affected.
OPERR	- Not affected.
SNAN	- Not affected.
NAN	- Not affected.
UNCC	- Not affected.

IER Flags:

For IER operand:

SINX	- Cleared if bit 0 of the immediate operand is cleared. Not affected otherwise.
SDZ	- Cleared if bit 1 of the immediate operand is cleared. Not affected otherwise.
SUNF	- Cleared if bit 2 of the immediate operand is cleared. Not affected otherwise.
SOVF	- Cleared if bit 3 of the immediate operand is cleared. Not affected otherwise.
SIOP	- Cleared if bit 4 of the immediate operand is cleared. Not affected otherwise.

For OMR, MR, ER, CCR operands:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	- Not affected.

Instruction Format: AND(I) #Byte,D

31				14	13			0
0000	0001	0001	i i i i	i i	i i	00ff	0111	10EE

Instruction Fields:

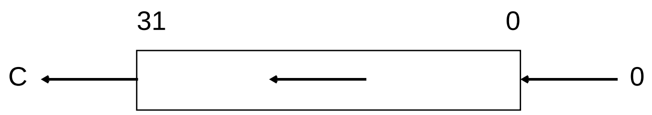
Immediate Short Data - iiiiiiii (8 bits)

D	E E f f
CCR	0 1 0 0
ER	0 1 0 1
IER	0 1 1 0
MR	0 1 1 1
OMR	1 0 0 0

Timing: 2 oscillator clock cycles

Memory: 1 program words

Operation:



(parallel data bus move)

Assembler Syntax:

ASL D (move syntax - see the MOVE instruction description.)

ASL S,D (move syntax - see the MOVE instruction description.)

ASL #shift,D

Description:

Single-bit shift: Arithmetically shift the low portion of the specified operand one bit to the left. The carry bit receives the MSB shifted out of the low portion of the source operand. A zero is shifted into the least significant bit of the destination operand. The result is stored in the low portion of D.

Multi-bit shift: Arithmetically shift the low portion of the specified operand N bits (up to 63 bits) to the left. The number of bits to shift is determined by the 11-bit unsigned integer located in the 11 LSBs of the high portion of S or by a 6-bit immediate field in the instruction. The carry bit receives the Nth bit shifted out of the low portion of the source operand; it is cleared for a shift count of zero. N zeros are shifted into the LSBs of the destination operand. If more than 32 bits are shifted, zeros will be stored in D and the carry bit. The result is stored in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C - Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero.

V - Set if the MSB is changed any time during the shift operation. Cleared otherwise.

Z - Set if result is zero. Cleared otherwise.

N - Set if result is negative. Cleared otherwise.

I - Not affected.

LR - Not affected.

$\overline{R}$ - Not affected.

A - Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ASL D(move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	0101	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: ASL S,D(move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	11	0sss	0011	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: ASL #shift,D

31				14	13			0
0000	0000	0000	0000	10	01	001n	nnnn	nddd

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

S	**s s s**	
Dn.H	n n n	where nnn = 0-7

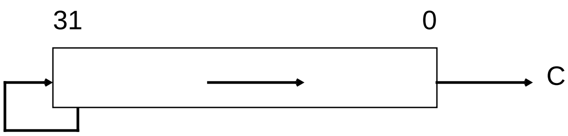
N	**n n n n n n**
0	0 0 0 0 0 0
1	0 0 0 0 0 1
2	0 0 0 0 1 0
.	.
.	. .
.	. .
62	1 1 1 1 1 0
63	1 1 1 1 1 1

Timing: 2 + mv oscillator clock cycles (2 oscillator clock cycles for ASL #shift)

Memory: 1 + mv program words (1 program word for ASL #shift)

ASR Arithmetic Shift Right ASR

Operation:



(parallel data bus move)

Assembler Syntax:

ASR	D	(move syntax - see the MOVE instruction description.)
ASR	S,D	(move syntax - see the MOVE instruction description.)
ASR	#shift,D	

Description:

Single-bit shift: Arithmetically shift the low portion of the specified operand one bit to the right. The carry bit receives the LSB shifted out of the low portion of the source operand. The MSB of the operand is held constant. The result is stored in the low portion of D.

Multi-bit shift: Arithmetically shift the low portion of the specified operand N bits (up to 63 bits) to the right. The number of bits to shift is determined by the 11-bit unsigned integer located in the 11 LSBs of the high portion of S or by a 6-bit immediate field in the instruction. The carry bit receives the Nth bit shifted out of the low portion of the source operand; it is cleared for a shift count of zero. N copies of the MSB of the operand are shifted into the N MSBs of the destination operand. If more than 32 bits are shifted, copies of the MSB will be stored in D and the carry bit. The result is stored in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ASR D(move syntax - see the MOVE instruction description.)

31 … 14	13 … 0			
DATA BUS MOVE FIELD	10	0000	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: ASR S,D(move syntax - see the MOVE instruction description.)

31 … 14	13 … 0			
DATA BUS MOVE FIELD	11	0sss	0011	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: ASR #shift,D

31				14	13			0
0000	0000	0000	0000	10	01	000n	nnnn	nddd

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

S	**s s s**	
Dn.H	n n n	where nnn = 0-7

N	**n n n n n n**
0	0 0 0 0 0 0
1	0 0 0 0 0 1
2	0 0 0 0 1 0
.	.
.	. .
.	. .
62	1 1 1 1 1 0
63	1 1 1 1 1 1

Timing: 2 + mv oscillator clock cycles (2 oscillator clock cycles for ASR #shift)

Memory: 1 + mv program words (1 program word for ASR #shift)

Branch Conditionally

Operation:

If cc,	then	PC+xx	$\rightarrow$ PC
	else	PC+1	$\rightarrow$ PC
If cc,	then	PC+xxxx	$\rightarrow$ PC
	else	PC+1	$\rightarrow$ PC
If cc,	then	PC+Rn	$\rightarrow$ PC
	else	PC+1	$\rightarrow$ PC

Assembler Syntax:

Bcc	label (short)
Bcc	label
Bcc	Rn

Description:

If the specified condition is true, program execution continues at location PC+displacement. The PC contains the address of the next instruction. If the specified condition is false, the PC is incremented and program execution continues sequentially. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the PC relative displacement. See **Section A.10** for restrictions.

"cc" may specify the following conditions:

Mnemonic		**Condition**
CC (HS)	- carry clear (higher or same)	C = 0
CS (LO)	- carry set (lower)	C = 1
EQ	- equal	Z = 1
GE	- greater or equal	N && V = 0
GT	- greater than	Z v (N && V) = 0
HI	- higher	Z v C = 0
LE	- less or equal	Z v (N && V) = 1
LS	- lower or same	Z v C = 1
LT	- less than	N && V = 1
MI	- minus	N = 1
NE(Q)	- not equal	Z = 0
PL	- plus	N = 0
VC	- overflow clear	V = 0
VS	- overflow set	V = 1

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: Bcc label (short)

31				14	13			0
0000	0011	10aa	aaaa	aa	1c	cccc	0aaa	aaaa

Instruction Format: Bcc label

31				14	13			0
0000	0011	0000	0000	00	1c	cccc	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: Bcc Rn

31				14	13			0
0000	0011	0000	001R		1c	cccc	0000	0000

Instruction Fields:

Rn - R0-R7

Long Displacement - 32 bits

Short Displacement - aaaaaaaaaaaaaaa (15 bits)

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
CC(HS)	0 1 0 1 0	CS(LO)	1 1 0 1 0
GE	0 1 0 1 1	LT	1 1 0 1 1
GT	0 1 1 0 0	LE	1 1 1 0 0
VC	0 1 1 0 1	VS	1 1 1 0 1
HI	0 1 1 1 0	LS	1 1 1 1 0

Timing: 6 + jx oscillator clock cycles

Memory: 1 + ea program words

BCHG Bit Test and Change BCHG

Operation:	Assembler Syntax:
D{n} → C; ~D{n} → D{n}	BCHG #bit,X: ea
D{n} → C; ~D{n} → D{n}	BCHG #bit,X: aa
D{n} → C; ~D{n} → D{n}	BCHG #bit,X: pp
D{n} → C; ~D{n} → D{n}	BCHG #bit,Y: ea
D{n} → C; ~D{n} → D{n}	BCHG #bit,Y: aa
D{n} → C; ~D{n} → D{n}	BCHG #bit,Y: pp
D{n} → C; ~D{n} → D{n}	BCHG #bit,D

Description:

The nth bit of the destination operand is tested and the state of the nth bit is reflected in the C condition code bit. After the test, the state of the nth bit is changed in the destination. All memory alterable addressing modes may be used. Register, Absolute Short and I/O Short addressing may also be used.

The bit to be tested is selected by an immediate bit number 0-31. This instruction performs a read-modify-write operation on the destination operand and requires two destination accesses. This instruction provides a test-and-change capability which is useful for synchronizing multiple processors using a shared memory. See **Section A.10** for restrictions.

CCR Condition Codes:

For destination operand SR:

C	- Changed if bit 0 is specified. Not affected otherwise.
V	- Changed if bit 1 is specified. Not affected otherwise.
Z	- Changed if bit 2 is specified. Not affected otherwise.
N	- Changed if bit 3 is specified. Not affected otherwise.
I	- Changed if bit 4 is specified. Not affected otherwise.
LR	- Changed if bit 5 is specified. Not affected otherwise.
$\overline{R}$	- Changed if bit 6 is specified. Not affected otherwise.
A	- Changed if bit 7 is specified. Not affected otherwise.

For other destination operands:

C - Set if bit tested is set. Cleared otherwise.

V - Not affected.

Z - Not affected.

N - Not affected.

I - Not affected.

LR - Not affected.

$\overline{R}$ - Not affected.

A - Not affected.

ER Status Bits:

For destination operand SR:

INX - Changed if bit 8 is specified. Not affected otherwise.

DZ - Changed if bit 9 is specified. Not affected otherwise.

UNF - Changed if bit 10 is specified. Not affected otherwise.

OVF - Changed if bit 11 is specified. Not affected otherwise.

OPERR - Changed if bit 12 is specified. Not affected otherwise.

SNAN - Changed if bit 13 is specified. Not affected otherwise.

NAN - Changed if bit 14 is specified. Not affected otherwise.

UNCC - Changed if bit 15 is specified. Not affected otherwise.

For other destination operands:

INX - Not affected.

DZ - Not affected.

UNF - Not affected.

OVF - Not affected.

OPERR - Not affected.

SNAN - Not affected.

NAN - Not affected.

UNCC - Not affected.

IER Flags:

For destination operand SR:

SINX - Changed if bit 16 is specified. Not affected otherwise.

SDZ - Changed if bit 17 is specified. Not affected otherwise.

SUNF - Changed if bit 18 is specified. Not affected otherwise.

SOVF - Changed if bit 19 is specified. Not affected otherwise.

SIOP - Changed if bit 20 is specified. Not affected otherwise.

For other destination operands:

SINX - Not affected.
SDZ - Not affected.
SUNF - Not affected.
SOVF - Not affected.
SIOP - Not affected.

Instruction Format: BCHG #bit,D

31				14	13			0
0000	0010	0111	dddd	dd	d0	0100	000b	bbbb

Instruction Format: BCHG #bit,X: pp
BCHG #bit,Y: pp

31				14	13			0
0000	0010	0110	1ppp	pp	pp	010S	000b	bbbb

Instruction Format: BCHG #bit,X: aa
BCHG #bit,Y: aa

31				14	13			0
0000	0010	0110	0aaa	aa	aa	010S	000b	bbbb

Instruction Format: BCHG #bit,X: ea
BCHG #bit,Y: ea

31				14	13			0
0000	0010	0101	MMMR		00	010S	000b	bbbb
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

<ea> Rn - R0-R7 (Memory alterable addressing modes only)
Immediate Short Data - bbbbb (5 bits)
Absolute Short Address - aaaaaaa (7 bits)
I/O Short Address - ppppppp (7 bits)

Memory Space	**S**	**Bit Number**	**b b b b b**	
X Memory	0	Bit 0-31	n n n n n	where nnnnn = 0-31
Y Memory	1			

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 4 + mvb oscillator clock cycles

Memory: 1 + ea program words

Operation:	Assembler Syntax:
D{n} → C; 0 → D{n}	BCLR #bit,X: ea
D{n} → C; 0 → D{n}	BCLR #bit,X: aa
D{n} → C; 0 → D{n}	BCLR #bit,X: pp
D{n} → C; 0 → D{n}	BCLR #bit,Y: ea
D{n} → C; 0 → D{n}	BCLR #bit,Y: aa
D{n} → C; 0 → D{n}	BCLR #bit,Y: pp
D{n} → C; 0 → D{n}	BCLR #bit,D

Description:

The nth bit of the destination operand is tested and the state of the nth bit is reflected in the C condition code bit. After the test, the nth bit is cleared in the destination. All memory alterable addressing modes may be used. Register, Absolute Short and I/O Short addressing may also be used.

The bit to be tested is selected by an immediate bit number 0-31. This instruction performs a read-modify-write operation on the destination operand and requires two destination accesses. This instruction provides a test-and-clear capability which is useful for synchronizing multiple processors using a shared memory. See **Section A.10** for restrictions.

CCR Condition Codes:

For destination operand SR:

C - Cleared if bit 0 is specified. Not affected otherwise.
V - Cleared if bit 1 is specified. Not affected otherwise.
Z - Cleared if bit 2 is specified. Not affected otherwise.
N - Cleared if bit 3 is specified. Not affected otherwise.
I - Cleared if bit 4 is specified. Not affected otherwise.
LR - Cleared if bit 5 is specified. Not affected otherwise.
$\overline{R}$ - Cleared if bit 6 is specified. Not affected otherwise.
A - Cleared if bit 7 is specified. Not affected otherwise.

For other destination operands:

C - Set if bit tested is set. Cleared otherwise.

V - Not affected.

Z - Not affected.

N - Not affected.

I - Not affected.

LR - Not affected.

$\overline{R}$ - Not affected.

A - Not affected.

ER Status Bits:

For destination operand SR:

INX - Cleared if bit 8 is specified. Not affected otherwise.

DZ - Cleared if bit 9 is specified. Not affected otherwise.

UNF - Cleared if bit 10 is specified. Not affected otherwise.

OVF - Cleared if bit 11 is specified. Not affected otherwise.

OPERR - Cleared if bit 12 is specified. Not affected otherwise.

SNAN - Cleared if bit 13 is specified. Not affected otherwise.

NAN - Cleared if bit 14 is specified. Not affected otherwise.

UNCC - Cleared if bit 15 is specified. Not affected otherwise.

For other destination operands:

INX - Not affected.

DZ - Not affected.

UNF - Not affected.

OVF - Not affected.

OPERR - Not affected.

SNAN - Not affected.

NAN - Not affected.

UNCC - Not affected.

IER Flags:

For destination operand SR:

SINX - Cleared if bit 16 is specified. Not affected otherwise.

SDZ - Cleared if bit 17 is specified. Not affected otherwise.

SUNF - Cleared if bit 18 is specified. Not affected otherwise.

SOVF - Cleared if bit 19 is specified. Not affected otherwise.

SIOP - Cleared if bit 20 is specified. Not affected otherwise.

For other destination operands:

SINX - Not affected.

SDZ - Not affected.

SUNF - Not affected.

SOVF - Not affected.

SIOP - Not affected.

Instruction Format: BCLR #bit,D

31				14	13			0
0000	0010	0011	dddd	dd	d0	0100	000b	bbbb

Instruction Format: BCLR #bit,X: pp
BCLR #bit,Y: pp

31				14	13			0
0000	0010	0010	1ppp	pp	pp	010S	000b	bbbb

Instruction Format: BCLR #bit,X: aa
BCLR #bit,Y: aa

31				14	13			0
0000	0010	0010	0aaa	aa	aa	010S	000b	bbbb

Instruction Format: BCLR #bit,X: ea
BCLR #bit,Y: ea

31				14	13			0
0000	0010	0001	MMMR		00	010S	000b	bbbb
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

<ea> Rn - R0-R7 (Memory alterable addressing modes only)

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

Bit Number	**b b b b b**	
Bit 0-31	n n n n n	where nnnnn = 0-31

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 4 + mvb oscillator clock cycles

Memory: 1 + ea program words

BFIND Find Leading One BFIND

Operation:

Leading One(S.L) $\rightarrow$ D.H (Parallel data bus move)

Assembler Syntax:

BFIND S,D (move syntax - see the MOVE instruction description.)

Description:

Return the position of the source operand S leading one, considered from left to right, as a 2's complement integer in the high portion of destination operand D. If the source operand is zero then return $80000000.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if source operand is zero. Cleared otherwise.
N	- Set if source operand is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\bar{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: BFIND S,D (move syntax - see the MOVE instruction description.)

31 14	13			0
DATA BUS MOVE FIELD	11	0sss	0111	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D	**d d d**	
Dn.H	n n n	where nnn = 0-7

S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

PC+xx → PC
PC+xxxx → PC
PC+Rn → PC

Assembler Syntax:

BRA label (short)
BRA label
BRA Rn

Description:

Program execution continues at location PC+displacement. The PC contains the address of the next instruction. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the PC relative displacement. See **Section A.10** for restrictions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: BRA label (short)

31				14	13			0
0000	0011	10aa	aaaa	aa	11	1111	0aaa	aaaa

Instruction Format: BRA label

31				14	13			0
0000	0011	0000	0000	00	11	1111	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: BRA Rn

31				14	13			0
0000	0011	0000	001R		11	1111	0000	0000

Instruction Fields:

Rn - R0-R7
Long PC Relative Displacement - 32 bits
Short PC Relative Displacement - aaaaaaaaaaaaaaa (15 bits)

Timing: 6 + jx oscillator clock cycles
Memory: 1 + ea program words

Operation:			Assembler Syntax:
If S{n} = 0, then PC + xxxx else PC + 1	→ →	PC PC	BRCLR #bit,X: ea, label
If S{n} = 0, then PC + xxxx else PC + 1	→ →	PC PC	BRCLR #bit,X: aa, label
If S{n} = 0, then PC + xxxx else PC + 1	→ →	PC PC	BRCLR #bit,X: pp, label
If S{n} = 0, then PC + xxxx else PC + 1	→ →	PC PC	BRCLR #bit,Y: ea, label
If S{n} = 0, then PC + xxxx else PC + 1	→ →	PC PC	BRCLR #bit,Y: aa, label
If S{n} = 0, then PC + xxxx else PC + 1	→ →	PC PC	BRCLR #bit,Y: pp, label
If S{n} = 0, then PC + xxxx else PC + 1	→ →	PC PC	BRCLR #bit,S,label

Description:

The nth bit in the source operand is tested. If the tested bit is cleared, program execution continues at location PC+displacement. The PC contains the address of the next instruction. If the tested bit is set, the PC is incremented and program execution continues sequentially. However, the address register specified in the effective address field is always updated independently of the condition. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. The 32-bit displacement is contained in the extension word of the instruction. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one. The bit to be tested is selected by an immediate bit number 0-31. See **Section A.10** for restrictions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: BRCLR #bit,S,label

31				14	13			0
0000	0010	1011	dddd	dd	d0	0100	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BRCLR #bit,X: pp, label
BRCLR #bit,Y: pp, label

31				14	13			0
0000	0010	1010	1ppp	pp	pp	010S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BRCLR #bit,X: aa, label
BRCLR #bit,Y: aa, label

31				14	13			0
0000	0010	1010	0aaa	aa	aa	010S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BRCLR #bit,X: ea, label
BRCLR #bit,Y: ea, label

31				14	13			0
0000	0010	1000	MMMR		00	010S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Fields:

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxxx))

PC Relative Displacement - 32 bits

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

Bit Number	**b b b b b**	
Bit 0-31	n n n n n	where nnnnn = 0-31

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 8 + jx oscillator clock cycles

Memory: 2 program words

BRSET — Branch if Bit Set

Operation:			Assembler Syntax:
If S{n} = 1, then PC + xxxx	→	PC	BRSET #bit,X: ea, label
else PC + 1	→	PC	
If S{n} = 1, then PC + xxxx	→	PC	BRSET #bit,X: aa, label
else PC + 1	→	PC	
If S{n} = 1, then PC + xxxx	→	PC	BRSET #bit,X: pp, label
else PC + 1	→	PC	
If S{n} = 1, then PC + xxxx	→	PC	BRSET #bit,Y: ea, label
else PC + 1	→	PC	
If S{n} = 1, then PC + xxxx	→	PC	BRSET #bit,Y: aa, label
else PC + 1	→	PC	
If S{n} = 1, then PC + xxxx	→	PC	BRSET #bit,Y: pp, label
else PC + 1	→	PC	
If S{n} = 1, then PC + xxxx	→	PC	BRSET #bit,S,label
else PC + 1	→	PC	

Description:

The nth bit in the source operand is tested. If the tested bit is set, program execution continues at location PC+displacement. The PC contains the address of the next instruction. If the tested bit is cleared, the PC is incremented and program execution continues sequentially. However, the address register specified in the effective address field is always updated independently of the condition. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. The 32-bit displacement is contained in the extension word of the instruction. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one. The bit to be tested is selected by an immediate bit number 0-31. See **Section A.10** for restrictions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: BRSET #bit,S,label

31				14	13			0
0000	0010	1011	dddd	dd	d0	1100	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BRSET #bit,X: pp, label
BRSET #bit,Y: pp, label

31				14	13			0
0000	0010	1010	1ppp	pp	pp	110S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BRSET #bit,X: aa, label
BRSET #bit,Y: aa, label

31				14	13			0
0000	0010	1010	0aaa	aa	aa	110S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BRSET #bit,X: ea, label
BRSET #bit,X: ea, label

31				14	13			0
0000	0010	1000	MMMR		00	110S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Fields:

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxxx))

PC Relative Displacement - 32 bits

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**	**Bit Number**	**b b b b b**	
X Memory	0	Bit 0-31	n n n n n	where nnnnn = 0-31
Y Memory	1			

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 8 + jx oscillator clock cycles
Memory: 2 program words

BScc Branch to Subroutine Conditionally BScc

Operation:			Assembler Syntax:
If cc, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC+xx	$\rightarrow$ PC		BScc label (short)
else PC + 1	$\rightarrow$ PC		
If cc, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC+xxxx	$\rightarrow$ PC		BScc label
else PC + 1	$\rightarrow$ PC		
If cc, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC+Rn	$\rightarrow$ PC		BScc Rn
else PC + 1	$\rightarrow$ PC		

Description:

If the specified condition is true, the address of the instruction immediately following the BScc instruction and the status register are pushed onto the stack. Program execution then continues at location PC+displacement. The PC contains the address of the next instruction. If the specified condition is false, the PC is incremented and program execution continues sequentially. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the PC relative displacement. See **Section A.10** for restrictions.

"cc" may specify the following conditions:

Mnemonic		**Condition**
CC (HS)	- carry clear (higher or same)	C = 0
CS (LO)	- carry set (lower)	C = 1
EQ	- equal	Z = 1
GE	- greater or equal	N && V = 0
GT	- greater than	Z v (N && V) = 0
HI	- higher	Z v C = 0
LE	- less or equal	Z v (N && V) = 1
LS	- lower or same	Z v C = 1
LT	- less than	N && V = 1
MI	- minus	N = 1
NE(Q)	- not equal	Z = 0
PL	- plus	N = 0
VC	- overflow clear	V = 0
VS	- overflow set	V = 1

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: BScc label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	1c	cccc	0aaa	aaaa

Instruction Format: BScc label

31				14	13			0
0000	0011	0100	0000	00	1c	cccc	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: BScc Rn

31				14	13			0
0000	0011	0100	001R		1c	cccc	0000	0000

Instruction Fields:

Rn - R0-R7

Long Displacement - 32 bits

Short Displacement - aaaaaaaaaaaaaaa (15 bits)

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
CC(HS)	0 1 0 1 0	CS(LO)	1 1 0 1 0
GE	0 1 0 1 1	LT	1 1 0 1 1
GT	0 1 1 0 0	LE	1 1 1 0 0
VC	0 1 1 0 1	VS	1 1 1 0 1
HI	0 1 1 1 0	LS	1 1 1 1 0

Timing: 6 + jx oscillator clock cycles
Memory: 1 + ea program words

BSCLR Branch to Subroutine if Bit Clear BSCLR

Operation:		Assembler Syntax:
If S{n} = 0,then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC + xxxx$\rightarrow$ PC		BSCLR #bit,X: ea, label
else	PC + 1 $\rightarrow$ PC	
If S{n} = 0,then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC + xxxx$\rightarrow$ PC		BSCLR #bit,X: aa, label
else	PC + 1 $\rightarrow$ PC	
If S{n} = 0,then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC + xxxx$\rightarrow$ PC		BSCLR #bit,X: pp, label
else	PC + 1 $\rightarrow$ PC	
If S{n} = 0,then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC + xxxx$\rightarrow$ PC		BSCLR #bit,Y: ea, label
else	PC + 1 $\rightarrow$ PC	
If S{n} = 0,then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC + xxxx$\rightarrow$ PC		BSCLR #bit,Y: aa, label
else	PC + 1 $\rightarrow$ PC	
If S{n} = 0,then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC + xxxx$\rightarrow$ PC		BSCLR #bit,Y: pp, label
else	PC + 1 $\rightarrow$ PC	
If S{n} = 0,then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; PC + xxxx$\rightarrow$ PC		BSCLR #bit,S,label
else	PC + 1 $\rightarrow$ PC	

Description:

The nth bit in the source operand is tested. If the tested bit is cleared, the address of the instruction immediately following the BSCLR instruction and the status register are pushed onto the stack. Program execution then continues at location PC+displacement. The PC contains the address of the next instruction. If the tested bit is set, the PC is incremented and program execution continues sequentially. However, the address register specified in the effective address field is always updated independently of the condition. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. The 32-bit displacement is contained in the extension word of the instruction. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one; if the condition is true, the push operation will write over the stack level where the SSH value was taken. The bit to be tested is selected by an immediate bit number 0-31. See **Section A.10** for restrictions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: BSCLR #bit,S,label

31				14	13			0
0000	0010	1111	dddd	dd	d0	0100	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BSCLR #bit,X: pp, label
BSCLR #bit,Y: pp, label

31				14	13			0
0000	0010	1110	1ppp	pp	pp	010S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BSCLR #bit,X: aa, label
BSCLR #bit,Y: aa, label

31				14	13			0
0000	0010	1110	0aaa	aa	aa	010S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BSCLR #bit,X: ea, label
BSCLR #bit,Y: ea, label

31				14	13			0
0000	0010	1100	MMMR		00	010S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Fields:

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxx))

PC Relative Displacement - 32 bits

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

D **d d d d d d d**

Memory Space	**S**	**Bit Number**	**b b b b b**	
X Memory	0	Bit 0-31	n n n n n	where nnnnn = 0-31
Y Memory	1			

D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 8 + jx oscillator clock cycles
Memory: 2 program words

Bit Test and Set

Operation:	Assembler Syntax:
D{n} → C; 1 → D{n}	BSET #bit,X: ea
D{n} → C; 1 → D{n}	BSET #bit,X: aa
D{n} → C; 1 → D{n}	BSET #bit,X: pp
D{n} → C; 1 → D{n}	BSET #bit,Y: ea
D{n} → C; 1 → D{n}	BSET #bit,Y: aa
D{n} → C; 1 → D{n}	BSET #bit,Y: pp
D{n} → C; 1 → D{n}	BSET #bit,D

Description:

The nth bit of the destination operand is tested and the state of the nth bit is reflected in the C condition code bit. After the test, the nth bit is set in the destination. All memory alterable addressing modes may be used. Register, Absolute Short and I/O Short addressing may also be used.

The bit to be tested is selected by an immediate bit number 0-31. This instruction performs a read-modify-write operation on the destination operand and requires two destination accesses. This instruction provides a test-and-set capability which is useful for synchronizing multiple processors using a shared memory. See **Section A.10** for restrictions.

CCR Condition Codes:

For destination operand SR:

C	- Set if bit 0 is specified. Not affected otherwise.
V	- Set if bit 1 is specified. Not affected otherwise.
Z	- Set if bit 2 is specified. Not affected otherwise.
N	- Set if bit 3 is specified. Not affected otherwise.
I	- Set if bit 4 is specified. Not affected otherwise.
LR	- Set if bit 5 is specified. Not affected otherwise.
$\overline{R}$	- Set if bit 6 is specified. Not affected otherwise.
A	- Set if bit 7 is specified. Not affected otherwise.

For other destination operands:

C	- Set if bit tested is set. Cleared otherwise.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

For destination operand SR:

INX	-Set if bit 8 is specified. Not affected otherwise.
DZ	-Set if bit 9 is specified. Not affected otherwise.
UNF	-Set if bit 10 is specified. Not affected otherwise.
OVF	-Set if bit 11 is specified. Not affected otherwise.
OPERR	-Set if bit 12 is specified. Not affected otherwise.
SNAN	-Set if bit 13 is specified. Not affected otherwise.
NAN	-Set if bit 14 is specified. Not affected otherwise.
UNCC	-Set if bit 15 is specified. Not affected otherwise.

For other destination operands:

INX	- Not affected.
DZ	- Not affected.
UNF	- Not affected.
OVF	- Not affected.
OPERR	- Not affected.
SNAN	- Not affected.
NAN	- Not affected.
UNCC	- Not affected.

IER Flags:

For destination operand SR:

SINX	-Set if bit 16 is specified. Not affected otherwise.
SDZ	-Set if bit 17 is specified. Not affected otherwise.
SUNF	-Set if bit 18 is specified. Not affected otherwise.
SOVF	-Set if bit 19 is specified. Not affected otherwise.
SIOP	-Set if bit 20 is specified. Not affected otherwise.

For other destination operands:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	- Not affected.

Instruction Format: BSET #bit,D

31				14	13			0
0000	0010	0011	dddd	dd	d0	1100	000b	bbbb

Instruction Format: BSET #bit,X: pp
BSET #bit,Y: pp

31				14	13			0
0000	0010	0010	1ppp	pp	pp	110S	000b	bbbb

Instruction Format: BSET #bit,X: aa
BSET #bit,Y: aa

31				14	13			0
0000	0010	0010	0aaa	aa	aa	110S	000b	bbbb

Instruction Format: BSET #bit,X: ea
BSET #bit,Y: ea

31				14	13			0
0000	0010	0000	MMMR		00	110S	000b	bbbb
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

<ea> Rn - R0-R7 (Memory alterable addressing modes only)

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

Bit Number	**b b b b b**	
Bit 0-31	n n n n n	where nnnnn = 0-31

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 4 + mvb oscillator clock cycles

Memory: 1 + ea program words

BSR — Branch to Subroutine

Operation:

PC → SSH; SR → SSL; PC+xx→ PC

PC → SSH; SR → SSL; PC+xxxx→ PC

PC → SSH; SR → SSL; PC+Rn→ PC

Assembler Syntax:

BSR label (short)

BSR label

BSR Rn

Description:

The address of the instruction immediately following the BSR instruction and the status register are pushed onto the stack. Program execution then continues at location PC+displacement. The PC contains the address of the next instruction. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the PC relative displacement. See **Section A.10** for restrictions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: BSR label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	11	1111	0aaa	aaaa

Instruction Format: BSR label

31				14	13			0
0000	0011	0100	001R		11	1111	0000	0000

Instruction Format: BSR Rn

31				14	13			0
0000	0011	0100	0000	00	11	1111	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Fields:

Rn - R0-R7

Long PC Relative Displacement - 32 bits

Short PC Relative Displacement - aaaaaaaaaaaaaaa (15 bits)

Timing: 6 + jx oscillator clock cycles

Memory: 1 + ea program words

BSSET Branch to Subroutine if Bit Set BSSET

Operation:	Assembler Syntax:
If S{n} = 1, then PC → SSH; SR → SSL; PC + xxxx → PC else PC + 1 → PC	BSSET #bit,X: ea, label
If S{n} = 1, then PC → SSH; SR → SSL; PC + xxxx → PC else PC + 1 → PC	BSSET #bit,X: aa, label
If S{n} = 1, then PC → SSH; SR → SSL; PC + xxxx → PC else PC + 1 → PC	BSSET #bit,X: pp, label
If S{n} = 1, then PC → SSH; SR → SSL; PC + xxxx → PC else PC + 1 → PC	BSSET #bit,Y: ea, label
If S{n} = 1, then PC → SSH; SR → SSL; PC + xxxx → PC else PC + 1 → PC	BSSET #bit,Y: aa, label
If S{n} = 1, then PC → SSH; SR → SSL; PC + xxxx → PC else PC + 1 → PC	BSSET #bit,Y: pp, label
If S{n} = 1, then PC → SSH; SR → SSL; PC + xxxx → PC else PC + 1 → PC	BSSET #bit,S,label

Description:

The nth bit in the source operand is tested. If the tested bit is set, the address of the instruction immediately following the BSSET instruction and the status register are pushed onto the stack. Program execution then continues at location PC+displacement. The PC contains the address of the next instruction. If the tested bit is cleared, the PC is incremented and program execution continues sequentially. However, the address register specified in the effective address field is always updated independently of the condition. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. The 32-bit displacement is contained in the extension word of the instruction. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one; if the condition is true, the push operation will write over the stack level where the SSH value was taken. The bit to be tested is selected by an immediate bit number 0-31. See **Section A.10** for restrictions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Fields:

Instruction Format: BSSET #bit,S,label

31				14	13			0
0000	0010	1111	dddd	dd	d0	1100	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BSSET #bit,X: pp, label
BSSET #bit,Y: pp, label

31				14	13			0
0000	0010	1110	1ppp	pp	pp	110S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BSSET #bit,X: aa, label
BSSET #bit,Y: aa, label

31				14	13			0
0000	0010	1110	0aaa	aa	aa	110S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BSSET #bit,X: ea, label
BSSET #bit,Y: ea, label

31				14	13			0
0000	0010	1100	MMMR		00	110S	000b	bbbb
PC RELATIVE DISPLACEMENT								

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxx))

PC Relative Displacement - 32 bits

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

Bit Number	**b b b b b**	
Bit 0-31	n n n n n	where nnnnn = 0-31

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 8 + jx oscillator clock cycles
Memory: 2 program words

BTST — Bit Test

Operation:	Assembler Syntax:
S{n} → C	BTST #bit,X: ea
S{n} → C	BTST #bit,X: aa
S{n} → C	BTST #bit,X: pp
S{n} → C	BTST #bit,Y: ea
S{n} → C	BTST #bit,Y: aa
S{n} → C	BTST #bit,Y: pp
S{n} → C	BTST #bit,S

Description:

The nth bit of the source operand is tested and the state of the nth bit is reflected in the C condition code bit. All memory alterable addressing modes may be used. Register Direct, Absolute Short and I/O Short addressing may also be used.

The bit to be tested is selected by an immediate bit number 0-31. When used with the appropriate rotate instructions, this instruction is useful for serial to parallel conversions.

If the system stack register SSH is specified as a source operand, the system stack pointer SP is postdecremented by 1 after SSH is read.

CCR Condition Codes:

C	- Set if bit tested is set. Cleared otherwise.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: BTST #bit,S

31				14	13			0
0000	0010	0111	dddd	dd	d0	1100	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BTST #bit,X: pp
BTST #bit,Y: pp

31				14	13			0
0000	0010	0110	1ppp	pp	pp	110S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BTST #bit,X: aa
BTST #bit,Y: aa

31				14	13			0
0000	0010	0110	0aaa	aa	aa	110S	000b	bbbb
PC RELATIVE DISPLACEMENT								

Instruction Format: BTST #bit,X: ea
BTST #bit,Y: ea

31				14	13			0
0000	0010	0100	MMMR		00	110S	000b	bbbb
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

<ea> Rn - R0-R7 (Memory alterable addressing modes only)

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

Bit Number	**b b b b b**	
Bit 0-31	n n n n n	where nnnnn = 0-31

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 4 + mvb oscillator clock cycles
Memory: 1 + ea program words

Operation:

0 $\rightarrow$ D.L (parallel data bus move)

Assembler Syntax:

CLR D (move syntax - see the MOVE instruction description.)

Description:

The low portion of the destination operand is cleared to zero. This instruction is implemented by executing ANDC D,D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Always set.
N	- Always cleared.
I	- Not affected.
LR	- Not affected.
$\bar{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: CLR D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	0uuu	1000	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D	**(u u u)** **d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

CMP Compare CMP

Operation:

S2.L - S1.L (parallel data bus move)

Assembler Syntax:

CMP S1,S2 (move syntax - see the MOVE instruction description.)

Description:

Subtract the low portion of the two operands as specified in the operation column above. No result is stored; however, the condition codes are affected as described below.

CMPG and CMP differ primarily in the definition of the CCR condition code bits LR and R. These differences are particularly useful in performing clipping operations in graphics applications. In the code segment, the CMP instruction tests the first **point** of a line, X0, against X_{min} and sets LR accordingly; the FCMPG instruction tests the second point of a line, X1, against X_{min} and sets $\overline{R}$ depending on the condition of LR. Note that the **line** segment will be trivially accepted if A is set (and R=1), whereas the line will be trivially rejected if $\overline{R}$ is cleared (and A=0). This choice of accept/reject conditions was selected to permit the CCR to be initialized by a single ORI instruction.

ORI	#$E0,CCR			;SET A, $\overline{R}$, LR – i. e.,
				;assume line is initially
				;accepted and not rejected.
MOVE		X:(R0)+N0,D0.L	Y:(R4)+,D1.S	;get X0, X_{min}
CMP	D1, D0	X:(R0)-N0, D0.L		;X0-X_{min}, get X1
CMPG	D1, D0			;X1=X_{min}

Input Operand(s) Precision: 32-bit 2’s complement integer.

Output Operand Precision: n.a.

CCR Condition Codes:

- C - Set if a borrow is generated from the MSB of the result. Cleared otherwise.
- V - Set if result overflows. Cleared otherwise.
- Z - Set if result is zero. Cleared otherwise.
- N - Set if result is negative. Cleared otherwise.
- I - Not affected.
- LR - Cleared if result is positive without overflow or zero. Cleared if result is negative with overflow. Not affected otherwise. See the example for the FCMPG instruction.
- $\overline{R}$ - Not affected. See the example for the FCMPG instruction.

A - Cleared if result is negative without overflow. Cleared if result is positive with overflow. Not affected otherwise. See the example for the FCMPG instruction.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: CMP S1,S2 (move syntax - see the MOVE instruction description.)

31 … 14	13 … 0			
DATA BUS MOVE FIELD	00	0sss	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

S1	**s s s**	
Dn.L	n n n	where nnn = 0-7

S2 **(u u)**

D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles
Memory: 1 + mv program words

CMPG Graphics Compare with Trivial Accept/Reject Flags CMPG

Operation:

S2.L - S1.L (parallel data bus move)

Assembler Syntax:

CMPG S1,S2

(move syntax - see the MOVE instruction description.)

Description:

Subtract the low portion of the two operands as specified in the operation column above. No result is stored; however, the condition codes are affected as described below.

CMPG and CMP differ primarily in the definition of the CCR condition code bits LR and R. These differences are particularly useful in performing clipping operations in graphics applications. In the code segment, the CMP instruction tests the first **point** of a line, X0, against X_{min} and sets LR accordingly; the FCMPG instruction tests the second point of a line, X1, against X_{min} and sets $\bar{R}$ depending on the condition of LR. Note that the **line** segment will be trivially accepted if A is set (and R=1), whereas the line will be trivially rejected if $\bar{R}$ is cleared (and A=0). This choice of accept/reject conditions was selected to permit the CCR to be initialized by a single ORI instruction.

```
ORI     #$E0,CCR                                    ;SET A, ¯R, LR – i. e.,
                                                    ;assume line is initially
                                                    ;accepted and not rejected.
MOVE              X:(R0)+N0,D0.L    Y:(R4)+,D1.S    ;get X0, Xmin
CMP     D1, D0    X:(R0)-N0, D0.L                   ;X0-Xmin, get X1
CMPG    D1, D0                                      ;X1=Xmin
```

Input Operand(s) Precision: 32-bit 2's complement integer.

Output Operand Precision: n.a.

CCR Condition Codes:

- C - Set if result is negative without overflow. Set if result is positive with overflow. Cleared otherwise.
- V - Set if result overflows. Cleared otherwise.
- Z - Set if result is zero. Cleared otherwise.
- N - Set if result is negative. Cleared otherwise.
- I - Not affected.
- LR - Always set (initialize for the next CMP, CMPG combination; see the example for the FCMPG instruction.
- $\bar{R}$ - Cleared if LR was set and result is negative without overflow. Cleared if LR was set and result is positive with overflow. Not affected otherwise. See the example for the FCMPG instruction.

A - Cleared if result is negative without overflow. Cleared if result is positive with overflow. Not affected otherwise. See the example for the FCMPG instruction.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: CMPG S1,S2 (move syntax - see the MOVE instruction description.)

31 14	13 0			
DATA BUS MOVE FIELD	11	0sss	0110	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

S1	**s s s**	
Dn.L	n n n	where nnn = 0-7

S2	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles
Memory: 1 + mv program words

DEBUGcc Enter Debug Mode Conditionally DEBUGcc

Operation:

If cc, then enter debug mode.

Assembler Syntax:

DEBUGcc

Description:

If the specified condition is true, enter Debug mode and wait for OnCE™ commands. If the specified condition is false, continue with the next instruction.

"cc" may specify the following conditions:

Mnemonic		Condition
CC (HS)	- carry clear (higher or same)	C = 0
CS (LO)	- carry set (lower)	C = 1
EQ	- equal	Z = 1
GE	- greater or equal	N && V = 0
GT	- greater than	Z v (N && V) = 0
HI	- higher	Z v C = 0
LE	- less or equal	Z v (N && V) = 1
LS	- lower or same	Z v C = 1
LT	- less than	N && V = 1
MI	- minus	N = 1
NE(Q)	- not equal	Z = 0
PL	- plus	N = 0
VC	- overflow clear	V = 0
VS	- overflow set	V = 1
AL	- always true	n.a.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: DEBUGcc

31				14	13			0
0000	0000	0000	0000	01	0c	cccc	1111	1111

OnCE™ is a trademark of Motorola Inc.

Instruction Fields:

Mnemonic	c c c c c	Mnemonic	c c c c c
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
CC(HS)	0 1 0 1 0	CS(LO)	1 1 0 1 0
GE	0 1 0 1 1	LT	1 1 0 1 1
GT	0 1 1 0 0	LE	1 1 1 0 0
VC	0 1 1 0 1	VS	1 1 1 0 1
HI	0 1 1 1 0	LS	1 1 1 1 0
AL	1 1 1 1 1		

Timing: 4 oscillator clock cycles
Memory: 1 program words

Decrement by One

Operation:

D.L - 1 → D.L (parallel data bus move)

Assembler Syntax:

DEC D (move syntax - see the MOVE instruction description.)

Description:

Decrement by one the low portion of the specified operand. The result is stored in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if a borrow is generated from the MSB of the result. Cleared otherwise.
V	- Set if result overflows. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: DEC D (move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	0111	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D	**(u u) d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles
Memory: 1 + mv program words

Operation:	Assembler Syntax:
LA → SSH; LC → SSL; X:<ea> → LC PC → SSH; SR → SSL; expr → LA; 1 → LF	DO X: ea, label
LA → SSH; LC → SSL; Y:<ea> → LC PC → SSH; SR → SSL; expr → LA; 1 → LF	DO Y: ea, label
LA → SSH; LC → SSL; S → LC PC → SSH; SR → SSL; expr → LA; 1 → LF	DO S,label
LA → SSH; LC → SSL; #xxx → LC PC → SSH; SR → SSL; expr → LA; 1 → LF	DO #count,label

Description:

Begin a hardware DO loop that is to be repeated the number of times specified in the instruction's source operand and whose range of execution is terminated by the destination operand (previously shown as "expr"). No overhead other than the execution of this DO instruction is required to set up this loop. DO loops can be nested and the loop count can be passed as a parameter.

During the first instruction cycle, the current contents of the loop address (LA) and the loop counter (LC) registers are pushed onto the system stack. The DO instruction's source operand is then loaded into the loop counter (LC) register. The LC register contains the remaining number of times the DO loop will be executed and can be accessed from inside the DO loop subject to certain restrictions. If LC equals zero, the DO loop is executed 2^{32} times. All address register indirect addressing modes (less long displacement) may be used to generate the effective address of the source operand. Register Direct addressing mode may also be used. If immediate short data is specified, the LC is loaded with the zero extended 19-bit immediate data.

During the second instruction cycle, the current contents of the program counter (PC) register and the status register (SR) are pushed onto the system stack. The stacking of the LA, LC, PC, and SR registers is the mechanism which permits nesting DO loops. The DO instruction's 32-bit absolute address extension word (which is the destination operand and shown as "expr") is then loaded into the loop address (LA) register. The value in the program counter (PC) register pushed onto the system stack is the address of the first instruction following the DO instruction (i.e., the first actual instruction in the DO loop). This value is read (i.e., copied but not pulled) from the top of the system stack to return to the top of the loop for another pass through the loop.

During the third instruction cycle, the loop flag (LF) is set. This results in the PC being repeatedly compared with LA to determine if the last instruction in the loop has been fetched. If LA equals PC, the last instruction in the loop has been fetched and the loop counter (LC) is tested. If LC is not equal to one, it is decremented by one and SSH is loaded into the PC to fetch the first instruction in the loop again. If LC equals one, the

"end-of-loop" processing begins.

When executing a DO loop, the instructions are actually fetched each time through the loop. Therefore, a DO loop can be interrupted. DO loops can also be nested. When DO loops are nested, the end-of-loop addresses must also be nested and are not allowed to be equal. The assembler generates an error message when DO loops are improperly nested. Nested DO loops are illustrated in the example.

NOTE: The assembler calculates the end-of-loop address to be loaded into LA (the absolute address extension word) by evaluating the end-of-loop expression "expr" and subtracting one. This is done to accommodate the case where the last word in the DO loop is a two-word instruction. Thus, the end-of-loop expression "expr" in the source code must represent the address of the instruction **after** the last instruction in the loop as shown in the example. This is in contrast to locating labels for instructions other than DO and DOR. In this case the labels are located on the same line as the target.

During the "end-of-loop" processing, the loop flag (LF) from the lower portion (SSL) of SP is written into the status register (SR), the contents of the loop address (LA) register are restored from the upper portion (SSH) of (SP-1), the contents of the loop counter (LC) are restored from the lower portion (SSL) of (SP-1), and the stack pointer (SP) is decremented by two. Instruction fetches now continue at the address of the instruction following the last instruction in the DO loop. Note that LF is the only bit in the status register (SR) that is restored after a hardware DO loop has been exited.
Note: The loop flag (LF) is cleared by a hardware reset.

Restrictions: The "end-of-loop" comparison previously described actually occurs at instruction fetch time. That is, LA is being compared with PC when the instruction at LA-2 is being executed. Therefore, instructions which access the program controller registers and/or change program flow cannot be used in locations LA-2, LA-1, or LA.

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, (implicitly) PC as a **destination** register. Similarly, the SSH program controller 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). A DO instruction cannot be repeated using the REP instruction.

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/BCLR/BSET LA, LC, SR, SP, SSH, or SSL
BTST SSH
JCLR/JSET/JSCLR/JSSET SSH
LEA to LA, LC, SR, SP, SSH, or SSL
LRA to LA, LC, SR, SP, SSH, or SSL
MOVEC/M/P/S from SSH
MOVEC/I/M/P/S to LA, LC, SR, SP, SSH, or SSL
ANDI MR
ORI MR

At LA:

any two word instruction
(F)Jcc, JMP, (F)JScc, JSR, (F)Bcc, BRA, (F)BScc, BSR,
LRA, REP, RESET, RTI, RTR, RTS, STOP, WAIT

Other restrictions:

BSR	to (LA), if Loop Flag is set
(F)BScc	to (LA), if Loop Flag is set
JSR	to (LA), if Loop Flag is set
(F)JScc	to (LA), if Loop Flag is set
JSCLR	to (LA), if Loop Flag is set
JSSET	to (LA), if Loop Flag is set
BSCLR	to (LA), if Loop Flag is set
BSSET	to (LA), if Loop Flag is set

A DO instruction cannot be repeated using the REP instruction.

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.

Similarly, since the DO instruction accesses the program controller registers, the DO instruction must not be immediately proceeded by any of the following instructions:

Immediately before DO:

BCHG/BCLR/BSET	LA, LC, SSH, SSL or SP
LEA	to LA, LC, SSH, SSL or SP
LRA	to LA, LC, SSH, SSL or SP
MOVEC/I/M/S	to LA, LC, SSH, SSL or SP
MOVEC/M/S	from SSH

During hardware loop operation, each instruction is fetched each time through the program loop. Therefore, instructions being executed in a hardware loop are interruptible and may be nested. The value of the PC pushed onto the system stack is the location of the first instruction after the DO instruction. This value is read from the top of the system stack to return to the start of the program loop. When DO instructions are nested, the end of loop addresses must also be nested and are not allowed to be equal. An example is shown:

Example:

```
        DO      #n1,END1
        DO      #n2,END2
        MOVE    D0,X:(R0)+
END2
        ADD     D1,D2         X:(R1)+,D3
END1
```

The assembler calculates the end of loop address (LA) (absolute address extension word xxxx) by evaluating the end of loop expression and subtracting one. Thus the end of loop expression in the source code represents the "next address" after the end of the loop. If a simple end of loop address label is used, it should be placed after the last instruction in the loop.

The LA register is compared to the PC to determine when the end of loop is reached. If the end of loop is reached, the loop counter (LC) is tested for one. If LC is not equal to one then it is decremented by one. If LC is equal to one, the system stack is purged and instruction fetches continue at the incremented PC address. Otherwise, the PC value on the top of the stack is read to fetch the start of the loop again.

Since the end of loop comparison is at fetch time and ahead of the end of loop execution, instructions which change program flow or change the system stack may not be used near the end of the loop without some restrictions. Proper DO loop operation is guaranteed if no instruction starting at address LA-2, LA-1 or LA specifies the program controller registers SR, SP, SSL, LA, LC or (implicitly) PC as a destination register; or specifies SSH as a source or destination register. Also, SSH cannot be specified as a source register in the DO instruction itself. The assembler will generate a warning if the restricted instructions are found within their restricted boundaries. See **Section A.10** for the complete list of restrictions.

Implementation Notes:

DO SP,label The actual value that will be loaded in the LC is the value of the SP before the DO instruction incremented by one.

DO SSL,label The LC will be loaded with its previous value that was saved in the stack by the DO instruction itself.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: DO #count,label

31				14	13			0
0000	0001	1011	i i i i	i i	i i	i i i i	i i i i	i i i i
ABSOLUTE ADDRESS								

Instruction Format: DO S,label

31				14	13			0
0000	0001	1010	0000	00	00	0000	1ddd	dddd
ABSOLUTE ADDRESS								

Instruction Format: DO X: ea, label
DO X: ea, label

31			14	13			0
0000	0001	100S	MMMR	00	0000	1000	0000
ABSOLUTE ADDRESS							

Instruction Fields:

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxx))

Absolute Address - 32 bits

Immediate Short Data - iiiiiiiiiiiiiiiiiii (19 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

D	**d d d d d d d**	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 6 + mv oscillator clock cycles
Memory: 2 program words

Operation:	Assembler Syntax:
LA → SSH; LC → SSL; X:<ea> → LC PC → SSH; SR → SSL; PC+xxxx → LA; 1 → LF	DOR X: ea, label
LA → SSH; LC → SSL; Y:<ea> → LC PC → SSH; SR → SSL; PC+xxxx → LA; 1 → LF	DOR Y: ea, label
LA → SSH; LC → SSL; S → LC PC → SSH; SR → SSL; PC+xxxx → LA; 1 → LF	DOR S,label
LA → SSH; LC → SSL; #xxx → LC PC → SSH; SR → SSL; PC+xxxx → LA; 1 → LF	DOR #count,label

Description:

This instruction initiates the beginning of a PC relative hardware program loop. The current loop address (LA) and loop counter (LC) values are pushed onto the system stack. With proper system stack management, this allows unlimited nested hardware DO loops. The PC and SR are pushed onto the system stack. The PC is added to the 32-bit address displacement extension word and the resulting address is loaded into the loop address register (LA). The PC points to the next instruction when it is added to the displacement. The effective address specifies the address of the loop count which is loaded into the loop counter (LC). The DO loop is executed LC times. If LC=0, the loop is executed 2**32 times. All address register indirect addressing modes (less Long Displacement) may be used. Register Direct addressing mode may also be used. If immediate short data is specified, the LC is loaded with the zero extended 19-bit immediate data.

During hardware loop operation, each instruction is fetched each time through the program loop. Therefore, instructions being executed in a hardware loop are interruptible and may be nested. The value of the PC pushed onto the system stack is the location of the first instruction after the DOR instruction. This value is read from the top of the system stack to return to the start of the program loop. When DOR instructions are nested, the end of loop addresses must also be nested and are not allowed to be equal. An example is shown below.

```
            DOR     #n1,END1
            DOR     #n2,END2
            MOVE    D0,X:(R0)+
END2
            ADD     D1,D2          X:(R1)+,D3
END1
```

The assembler calculates the end of loop address LA (PC relative address extension word xxxx) by evaluating the end of loop expression and subtracting one. Thus the end of loop expression in the source code

represents the "next address" after the end of the loop. If a simple end of loop address label is used, it should be placed after the last instruction in the loop.

The LA register is compared to the PC to determine when the end of loop is reached. If the end of loop is reached, the loop counter (LC) is tested for one. If LC is not equal to one then it is decremented by one. If LC is equal to one, the system stack is purged and instruction fetches continue at the incremented PC address. Otherwise, the PC value on the top of the stack is read to fetch the start of the loop again.

Since the end of loop comparison is at fetch time and ahead of the end of loop execution, instructions which change program flow or change the system stack may not be used near the end of the loop without some restrictions. Proper hardware loop operation is guaranteed if no instruction starting at address LA-2, LA-1 or LA specifies the program controller registers SR, SP, SSL, LA, LC or (implicitly) PC as a destination register; or specifies SSH as a source or destination register. Also, SSH cannot be specified as a source register in the DOR instruction itself. The assembler will generate a warning if the restricted instructions are found within their restricted boundaries. See **Section A.10** for the complete list of restrictions.

Implementation Notes:

DOR SP,label The actual value that will be loaded in the LC is the value of the SP before the DOR instruction incremented by one.

DOR SSL,label The LC will be loaded with its previous value that was saved in the stack by the DOR instruction itself.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Fields:

<ea> Rn - R0-R7 (All address register indirect addressing modes except (Rn+xxx))

Instruction Format: DOR #count,label

31 ... 14 13 ... 0

0000	0001	1011	i i i i	i i	i i	i i i i	0 i i i	i i i i
PC RELATIVE REPLACEMENT								

Instruction Format: DOR S,label

31 ... 14 13 ... 0

0000	0001	1010	0000	00	00	0000	0ddd	dddd
PC RELATIVE REPLACEMENT								

Instruction Format: DOR X: ea, label
DOR Y: ea, label

31 ... 14 13 ... 0

0000	0001	100S	MMMR	00	0000	0000	0000
PC RELATIVE REPLACEMENT							

PC displacement - 32 bits

Immediate Short Data - iiiiiiiiiiiiiiiiiii (19 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

D	**d d d d d d d**	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 8 + mv oscillator clock cycles

Memory: 2 program words

ENDDO End Current DO Loop ENDDO

Operation:

SSL(LF) $\rightarrow$ SR; SP-1 $\rightarrow$ SP
SSH $\rightarrow$ LA; SSL $\rightarrow$ LC; SP-1 $\rightarrow$ SP

Assembler Syntax:

ENDDO

Description:

This instruction will cause the termination of the current hardware DO loop before the current loop counter (LC) equals one. If the value of the current DO loop counter is needed, it must be read before the execution of the ENDDO instruction. Initially, the loop flag (LF) is restored from the system stack and the remaining portion of the status register (SR) and the program counter (PC) are purged from the system stack. The loop address (LA) and the loop counter (LC) registers are them restored from the system stack.

Restrictions:

Due to pipelining, and the fact that the ENDDO instruction accesses the program controller registers, the ENDDO instruction must not be immediately preceded by any of the following instructions:

Immediately before ENDDO	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
	ORI MR
	ANDI MR

Also, the ENDDO instruction cannot be the next to last instruction in a DO loop (at LA-1).

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Fields:

Instruction Format: ENDDO

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0111

None

Timing: 2 oscillator clock cycles

Memory: 1 program words

Operation:

D.L && S.L $\rightarrow$ D.L (parallel data bus move)

Assembler Syntax:

EOR S,D (move syntax - see the MOVE instruction description.)

Description:

Logically exclusive OR the low portion of the two specified operands and store the result in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: EOR S,D (move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	00	0sss	uu10	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

EXT — Sign Extend Half Word — EXT

Operation:

D.L {15:0} $\rightarrow$ D.L {15:0} (parallel data bus move)
D.L {15} $\rightarrow$ D.L {31:16}

Assembler Syntax:

EXT D (move syntax - see the MOVE instruction description.)

Description:

Sign extend the lower 16 bits of D.L into the upper 16 bits of D.L.

Input Operand(s) Precision: 16-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: EXT D (move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0
DATA BUS MOVE FIELD	10 0001 uu00 1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

D.L {7:0} $\rightarrow$ D.L {7:0} (parallel data bus move)
D.L {7} $\rightarrow$ D.L {31:8}

Assembler Syntax:

EXTB D (move syntax - see the MOVE instruction description.)

Description:

Sign extend the lower byte of D.L into the upper 24 bits of D.L.

Input Operand(s) Precision: 8-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: EXTB D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0
DATA BUS MOVE FIELD	10 0001 uu01 1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

FABS.S Absolute Value

Operation:

D → ROUND(SP) → D (parallel data bus move)

Assembler Syntax:

FABS.S D (move syntax - see the MOVE instruction description.)

Description:

Take the absolute value of the destination operand, round to single precision and store the result in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Always cleared.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FABS.S D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0001	uu11	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

Operation:

D $\rightarrow$ D (parallel data bus move)

Assembler Syntax:

FABS.X D (move syntax - see the MOVE instruction description.)

Description:

Take the absolute value of the destination operand and store the result in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Always cleared.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{\text{R}}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FABS.X D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0001	uu10	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FADD.S Floating-Point Add FADD.S

Operation:

D + S $\rightarrow$ ROUND(SP) $\rightarrow$ D
(parallel data bus move)

Assembler Syntax:

FADD.S S,D (move syntax - see the MOVE instruction description.)

Description:

Add the two specified operands, round to single precision and store the result in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\bar{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Set if operands are opposite-signed infinities. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FADD.S S,D (move syntax - see the MOVE instruction description.)

Instruction Fields:

31	14	13			0
DATA BUS MOVE FIELD		01	0sss	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA					

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FADD.X Floating-Point Add FADD.X

Operation:

D + S $\rightarrow$ ROUND(SEP) $\rightarrow$ D
(parallel data bus move)

Assembler Syntax:

FADD.X S,D (move syntax - see the MOVE instruction description.)

Description:

Add the two specified operands, round to single extended precision and store the result in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Set if operands are opposite-signed infinities. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FADD.X S,D (move syntax - see the MOVE instruction description.)

31	14	13			0
DATA BUS MOVE FIELD		01	0sss	uu00	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA					

Instruction Fields:

D	**d d d**	
Dn	n n n	where nnn = 0-7

S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FADDSUB.S Add and Subtract FADDSUB.S

Operation:

D1 + D2 → ROUND(SP) → D2 (parallel data bus move)

D1 - D2 → ROUND(SP) → D1

Assembler Syntax:

FADDSUB.S D1,D2

(move syntax - see the MOVE instruction description.)

Description:

Add and subtract the two specified operands and round to single precision. Store the rounded result of the addition in D2 and of the subtraction in D1.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand(s) Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Set if result of the addition is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the addition or subtraction result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if the addition or subtraction result underflows. Cleared otherwise.
OVF	-Set if the addition or subtraction result overflows. Cleared otherwise.
OPERR	-Set if operands of the addition are opposite-signed infinities or if the operands of the subtraction are like-signed infinities. Cleared otherwise.
SNAN	-Set if any operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result of the addition is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FADDSUB.S D1,D2 (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	0sss	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**s s s**	
Dn	n n n	where nnn = 0-7

D2	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FADDSUB.X Add and Subtract FADDSUB.X

Operation:

D1 + D2 → ROUND(SEP) → D2

(parallel data bus move)

D1 - D2 → ROUND(SEP) → D1

Assembler Syntax:

FADDSUB.X D1,D2 (move syntax - see the MOVE instruction description.)

Description:

Add and subtract the two specified operands and round to single extended precision. Store the result of the addition in D2 and of the subtraction in D1.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand(s) Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Set if result of the addition is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the addition or subtraction result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if the addition or subtraction result underflows. Cleared otherwise.
OVF	-Set if the addition or subtraction result overflows. Cleared otherwise.
OPERR	-Set if operands of the addition are opposite-signed infinities or if the operands of the subtraction are like-signed infinities. Cleared otherwise.
SNAN	-Set if any operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result of the addition is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FADDSUB.X D1,D2 (move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	01	0sss	uu01	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**s s s**	
Dn	n n n	where nnn = 0-7

D2	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FBcc Floating-Point Branch Conditionally

Operation:

If cc,	then	PC+xx	→ PC
	else	PC+1	→ PC
If cc,	then	PC+xxxx	→ PC
	else	PC+1	→ PC
If cc,	then	PC+Rn	→ PC
	else	PC+1	→ PC

Assembler Syntax:

FBcc	label (short)
FBcc	label
FBcc	Rn

Description:

If the specified floating-point condition is true, the address of the instruction immediately following the FBScc instruction and the status register are pushed onto the stack. Program execution then continues at location PC+displacement. The PC contains the address of the next instruction. If the specified condition is false, the PC is incremented and program execution continues sequentially. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the PC relative displacement. See **Section A.10** for restrictions. Non-aware floating-point conditions set the SIOP flag in the IER register and the UNCC bit in the ER register if the NAN bit is set.

"cc" may specify the following conditions:

Mnemonic		Condition	Non-aware Set UNCC*
EQ	- equal	Z = 1	No
ERR	- error	UNCC v SNAN v OPERR v OVF v UNF v DZ = 1	No
GE	- greater than or equal	NAN v (N & ~Z) = 0	Yes
GL	- greater or less than	NAN v Z = 0	Yes
GLE	- greater, less or equal	NAN = 0	Yes
GT	- greater than	NAN v Z v N = 0	Yes
INF	- infinity	I = 1	Yes
LE	- less than or equal	NAN v ~(N v Z) = 0	Yes
LT	- less than	NAN v Z v ~N = 0	Yes
MI	- minus	N = 1	No
NE(Q)	- not equal	Z = 0	No
NGE	- not(greater than or equal)	NAN v (N & ~Z) = 1	Yes
NGL	- not(greater or less than)	NAN v Z = 1	Yes
NGLE	- not(greater, less or equal)	NAN = 1	Yes
NGT	- not greater than	NAN v Z v N = 1	Yes
NINF	- not infinity	I = 0	Yes
NLE	- not(less than or equal)	NAN v ~(N v Z) = 1	Yes
NLT	- not less than	NAN v Z v ~N = 1	Yes
OR	- ordered	NAN = 0	No
PL	- plus	N = 0	No
UN	- unordered	NAN = 1	No

Note: The operands for the ERR condition are taken from the ER register.
See the description of UNcc in **Section A.4**.

CCR Condition Codes: Not affected.

ER Status Bits:

INX - Not affected.
DZ - Not affected.
UNF - Not affected.
OVF - Not affected.
OPERR- Not affected.
SNAN - Not affected.
NAN - Not affected.
UNCC - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

IER Flags:

SINX - Not affected.
SDZ - Not affected.
SUNF - Not affected.
SOVF - Not affected.
SIOP - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

Instruction Format: FBcc label (short)

Instruction Fields:

31				14	13			0
0000	0011	10aa	aaaa	aa	1c	cccc	0aaa	aaaa
PC RELATIVE DISPLACEMENT								

Instruction Format: FBcc Rn

31				14	13			0
0000	0011	0000	001R		1c	cccc	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: FBcc label

31				14	13			0
0000	0011	0000	0000	00	1c	cccc	0000	0000
PC RELATIVE DISPLACEMENT								

Rn - R0-R7

Long Displacement - 32 bits

Short Displacement - aaaaaaaaaaaaaaa (15 bits)

Mnemonic	c c c c c	Mnemonic	c c c c c
GT	0 0 0 0 0	NGT	1 0 0 0 0
LT	0 0 0 0 1	NLT	1 0 0 0 1
GE	0 0 0 1 0	NGE	1 0 0 1 0
LE	0 0 0 1 1	NLE	1 0 0 1 1
GL	0 0 1 0 0	NGL	1 0 1 0 0
INF	0 0 1 0 1	NINF	1 0 1 0 1
GLE	0 0 1 1 0	NGLE	1 0 1 1 0
OR	0 0 1 1 1	UN	1 0 1 1 1
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
ERR	0 1 1 1 1		

Timing: 6 + jx oscillator clock cycles
Memory: 1 + ea program words

Floating-Point Branch To Subroutine Conditionally

FBScc

Operation:	Assembler Syntax:
If cc, then PC → SSH; SR → SSL; PC+xx → PC else PC+1 → PC	FBScc label (short)
If cc, then PC → SSH; SR → SSL; PC+xxxx → PC else PC+1 → PC	FBScc label
If cc, then PC → SSH; SR → SSL; PC+Rn → PC else PC+1 → PC	FBScc Rn

Description:

If the specified floating-point condition is true, the address of the instruction immediately following the FB-Scc instruction and the status register are pushed onto the stack. Program execution then continues at a location specified by a PC relative address in the instruction. If the specified condition is false, the PC is incremented and the PC relative address is ignored. Short Displacement, Long Displacement, and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the PC relative displacement. The PC points to the next instruction when it is added to the displacement. See **Section A.10** for restrictions. Non-aware floating-point conditions set the SIOP flag in the IER and the UNCC bit in the ER if the NAN bit is set. This action occurs before stacking the status register when the specified non-aware floating-point condition is true.

"cc" may specify the following conditions:

Mnemonic		Condition	Non-aware* Set UNCC
EQ	- equal	Z = 1	No
ERR	- error	UNCC v SNAN v OPERR v OVF v UNF v DZ = 1	No
GE	- greater than or equal	NAN v (N & ~Z) = 0	Yes
GL	- greater or less than	NAN v Z = 0	Yes
GLE	- greater, less or equal	NAN = 0	Yes
GT	- greater than	NAN v Z v N = 0	Yes
INF	- infinity	I = 1	Yes
LE	- less than or equal	NAN v ~(N v Z) = 0	Yes
LT	- less than	NAN v Z v ~N = 0	Yes
MI	- minus	N = 1	No
NE(Q)	- not equal	Z = 0	No
NGE	- not(greater than or equal)	NAN v (N & ~Z) = 1	Yes
NGL	- not(greater or less than)	NAN v Z = 1	Yes
NGLE	- not(greater, less or equal)	NAN = 1	Yes
NGT	- not greater than	NAN v Z v N = 1	Yes
NINF	- not infinity	I = 0	Yes
NLE	- not(less than or equal)	NAN v ~(N v Z) = 1	Yes
NLT	- not less than	NAN v Z v ~N = 1	Yes
OR	- ordered	NAN = 0	No
PL	- plus	N = 0	No
UN	- unordered	NAN = 1	No

Note: The operands for the ERR condition are taken from the ER register.
* See description of UNcc bit in **Section A.4**.

CCR Condition Codes: Not affected.

ER Status Bits:

- INX - Not affected.
- DZ - Not affected.
- UNF - Not affected.
- OVF - Not affected.
- OPERR - Not affected.
- SNAN - Not affected.
- NAN - Not affected.
- UNCC - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

IER Flags:

- SINX - Not affected.
- SDZ - Not affected.
- SUNF - Not affected.
- SOVF - Not affected.
- SIOP - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

Instruction Format: FBScc label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	1c	cccc	0aaa	aaaa
OPTIONAL LONG DISPLACEMENT EXTENSION								

Instruction Format: FBScc Rn

31				14	13			0
0000	0011	0100	001R		1c	cccc	0000	0000
OPTIONAL LONG DISPLACEMENT EXTENSION								

Instruction Format: FBScc label

31				14	13			0
0000	0011	0100	0000	00	1c	cccc	0000	0000
OPTIONAL LONG DISPLACEMENT EXTENSION								

Instruction Fields:

Rn - R0-R7

Long Displacement - 32 bits

Short Displacement - aaaaaaaaaaaaaaa (15 bits)

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
GT	0 0 0 0 0	NGT	1 0 0 0 0
LT	0 0 0 0 1	NLT	1 0 0 0 1
GE	0 0 0 1 0	NGE	1 0 0 1 0
LE	0 0 0 1 1	NLE	1 0 0 1 1
GL	0 0 1 0 0	NGL	1 0 1 0 0
INF	0 0 1 0 1	NINF	1 0 1 0 1
GLE	0 0 1 1 0	NGLE	1 0 1 1 0
OR	0 0 1 1 1	UN	1 0 1 1 1
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
ERR	0 1 1 1 1		

Timing: 6 + jx oscillator clock cycles

Memory: 1 + ea program words

FCLR Clear Floating-Point Register FCLR

Operation:

+0 $\rightarrow$ D (parallel data bus move)

Assembler Syntax:

FCLR D (move syntax - see the MOVE instruction description.)

Description:

All 96 bits of the destination operand are cleared to zero.

Input Operand(s) Precision: DEP Floating-Point.

Output Operand Precision: DEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Always set.
N	- Always cleared.
I	- Always cleared.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Not affected.
NAN	-Always cleared.
UNCC	-Always cleared.

IER Flags: Not affected.

Instruction Format: FCLR D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0000	uu11	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

FCMP — Compare Two Floating-Point Operands — FCMP

Operation:

S2 - S1 (parallel data bus move)

Assembler Syntax:

FCMP S1,S2 (move syntax - see the MOVE instruction description.)

Description:

Subtract the two operands as specified in the operation column above. No result is stored; however, the condition codes are affected as described. This instruction differs from FSUB when S1=S2; in this case, the result is always +0 and therefore, N is cleared. Note that this is true even if S1, S2 are infinity.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: n.a.

CCR Condition Codes:

(Note: Since there is no destination, there is no rounding and therefore the condition code bits are set assuming an infinite precision result**)**

C	- Not affected.
V	- Not affected.
Z	- Set if source operands are equal. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if anyone of the operands is infinity. Cleared otherwise.
LR	- Cleared if result is positive, zero or NaN (if cleared first, print accepted; see the FCMPG example). Not affected otherwise.
$\overline{\text{R}}$	- Cleared if result is a NaN. Not affected otherwise.
A	- Cleared if result is a NaN. Cleared if result is negative and not zero. Not affected otherwise.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FCMP S1,S2 (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	1sss	uu01	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

S1	**s s s**	
Dn	n n n	where nnn = 0-7
	(u u)	
S2	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

FCMPG Graphics Compare with Trivial Accept/Reject Flags FCMPG

Operation:

S2 - S1 (parallel data bus move)

Assembler Syntax:

FCMPG S1,S2 (move syntax - see the MOVE instruction description.)

Description:

Subtract the two operands as specified in the operation column above. No result is stored; however, the condition codes are affected as described. This instruction differs from FSUB when S1=S2; in this case, the result is always +0 and therefore, N is cleared. Note that this is true even if S1, S2 are infinity.

FCMPG and FCMP differ primarily in the definition of the CCR condition code bits LR and R. These differences are particularly useful in performing clipping operations in graphics applications. In the code segment, the FCMP instruction tests the first **point** of a line, X0, against X_{min} and sets LR accordingly; the FCMPG instruction tests the second point of a line, X1, against X_{min} and sets $\overline{R}$ depending on the condition of LR. Note that the **line** segment will be trivially accepted if A is set (and R=1), whereas the line will be trivially rejected if $\overline{R}$ is cleared (and A=0). This choice of accept/reject conditions was selected to permit the CCR to be initialized by a single ORI instruction.

```
ORI     #$E0,CCR                                    ;SET A, ¯R, LR – i. e.,
                                                    ;assume line is initially
                                                    ;accepted and not rejected.
MOVE              X:(R0)+N0,D0.S  Y:(R4)+,D1.S      ;get X0, Xmin
FCMP    D1, D0    X:(R0)-N0, D0.S                   ;X0-Xmin, get X1
FCMPG   D1, D0                                      ;X1=Xmin
```

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: n.a.

CCR Condition Codes:

(Note: Since there is no destination, there is no rounding and therefore the condition codes are set assuming an infinite precision result**)**

C	- Set if result is a NaN. Set if result is negative and not zero. Cleared otherwise.
V	- Not affected.
Z	- Set if source operands are equal. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if anyone of the operands is infinity. Cleared otherwise.
LR	- Always set (initialize for the next FCMP, FCMPG combination).
$\overline{R}$	- Cleared if result is a NaN. Cleared if result is negative and not zero and LR was set (i. e., first point was rejected). Not affected otherwise.
A	- Cleared if result is a NaN. Cleared if result is negative and not zero. Not affected otherwise.

ER Status Bits:

IER Flags: Flags changed according to standard definition.

Instruction Format: FCMPG S1,S2 (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	1sss	uu10	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

S1	**s s s**	
Dn	n n n	where nnn = 0-7

	(u u)	
S2	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FCMPM Compare Magnitude of Two Floating-Point Operands FCMPM

Operation:

S2 - S1 (parallel data bus move)

Assembler Syntax:

FCMPM S1,S2 (move syntax - see the MOVE instruction description.)

Description:

Subtract the absolute value (magnitude) of the two operands as specified in the operation column above. No result is stored; however, the condition codes are affected as described. This instruction differs from FSUB when S1=S2; in this case, the result is always +0 and therefore, N is cleared. Note that this is true even if S1, S2 are infinity.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: n.a.

CCR Condition Codes:

(Note: Since there is no destination, there is no rounding and therefore the condition code bits are set assuming an infiite precision result**)**

C	- Not affected.
V	- Not affected.
Z	- Set if source operands are equal. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if anyone of the operands is infinity. Cleared otherwise.
LR	- Not affected.
¯R	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FCMPM S1,S2 (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	1sss	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

S1	**s s s**	
Dn	n n n	where nnn = 0-7

S2	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

FCOPYS.S Copy Sign FCOPYS.S

Operation:

Sign of S → D → ROUND(SP) → D
(parallel data bus move)

Assembler Syntax:

FCOPYS.S S,D (move syntax - see the MOVE instruction description.)

Description:

Copy the sign of the floating-point operand S to the floating-point operand D, round the resulting operand D to single precision and store the result in the specified destination D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FCOPYS.S S,D (move syntax - see the MOVE instruction description.)

31 … 14	13 … 0			
DATA BUS MOVE FIELD	01	1sss	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

S	**s s s**	
Dn	n n n	where nnn = 0-7

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

Operation:

Sign of S $\rightarrow$ D (parallel data bus move)

Assembler Syntax:

FCOPYS.X S,D (move syntax - see the MOVE instruction description.)

Description:

Copy the sign of the floating-point operand S to the floating-point operand D. Since both S and D are single extended precision operands, rounding is not performed.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FCOPYS.X S,D (move syntax - see the MOVE instruction description.)

31	14 13			0
DATA BUS MOVE FIELD	01	0sss	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

Enter Debug Mode Conditionally

Operation:

If cc, then enter debug mode.

Assembler Syntax:

FDEBUGcc

Description:

If the specified floating-point condition is true, enter Debug mode and wait for OnCE™ commands. If the specified condition is false, continue with the next instruction. Non-aware floating-point conditions set the SIOP flag in the IER register and the UNCC bit in the ER register if the NAN bit is set.

"cc" may specify the following conditions:

Mnemonic		Condition	Non-aware Set UNCC*
EQ	- equal	Z = 1	No
ERR	- error	UNCC v SNAN v OPERR v OVF v UNF v DZ = 1	No
GE	- greater than or equal	NAN v (N & ~Z) = 0	Yes
GL	- greater or less than	NAN v Z = 0	Yes
GLE	- greater, less or equal	NAN = 0	Yes
GT	- greater than	NAN v Z v N = 0	Yes
INF	- infinity	I = 1	Yes
LE	- less than or equal	NAN v ~(N v Z) = 0	Yes
LT	- less than	NAN v Z v ~N = 0	Yes
MI	- minus	N = 1	No
NE(Q)	- not equal	Z = 0	No
NGE	- not(greater than or equal)	NAN v (N & ~Z) = 1	Yes
NGL	- not(greater or less than)	NAN v Z = 1	Yes
NGLE	- not(greater, less or equal)	NAN = 1	Yes
NGT	- not greater than	NAN v Z v N = 1	Yes
NINF	- not infinity	I = 0	Yes
NLE	- not(less than or equal)	NAN v ~(N v Z) = 1	Yes
NLT	- not less than	NAN v Z v ~N = 1	Yes
OR	- ordered	NAN = 0	No
PL	- plus	N = 0	No
UN	- unordered	NAN = 1	No

Note: The operands for the ERR condition are taken from the ER register.
* See description of the UNcc bit in **Section A.4**.

CCR Condition Codes: Not affected.

Once™ is a trademark of Motorola Inc.

ER Status Bits:

INX - Not affected.

DZ - Not affected.

UNF - Not affected.

OVF - Not affected.

OPERR - Not affected.

SNAN - Not affected.

NAN - Not affected.

UNCC - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

IER Flags:

SINX - Not affected.

SDZ - Not affected.

SUNF - Not affected.

SOVF - Not affected.

SIOP - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise. Instruction Format:

Instruction Fields:

Instruction Format: FDEBUGcc

31				14	13			0
0000	0000	0000	0000	01	0c	cccc	1111	1111

Mnemonic	c c c c c	Mnemonic	c c c c c
GT	0 0 0 0 0	NGT	1 0 0 0 0
LT	0 0 0 0 1	NLT	1 0 0 0 1
GE	0 0 0 1 0	NGE	1 0 0 1 0
LE	0 0 0 1 1	NLE	1 0 0 1 1
GL	0 0 1 0 0	NGL	1 0 1 0 0
INF	0 0 1 0 1	NINF	1 0 1 0 1
GLE	0 0 1 1 0	NGLE	1 0 1 1 0
OR	0 0 1 1 1	UN	1 0 1 1 1
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
ERR	0 1 1 1 1		

Timing: 4 oscillator clock cycles
Memory: 1 program word

FGETMAN Extract the Mantissa FGETMAN

Operation:

Normalized mantissa of S $\rightarrow$ D
(parallel data bus move)

Assembler Syntax:

FGETMAN S,D (move syntax - see the MOVE instruction description.)

Description:

Extract the mantissa and sign of the floating-point operand S, normalizes the mantissa, forces the exponent to "ebias" so the result is in the range 1-2, and stores the result as a floating-point value in the specified destination D regardless of whether the mantissa is denormalized or not.

As an example of the use of FGETMAN, GETEXP, and FSCALE; consider decomposing a floating-point number into its mantissa and unbiased exponent and then recreating the original floating-point number.

```
FGETMAN   D0, D1      ;extract normalized mantissa
GETEXP    D0,D2       ;extract unbiased exponent
MOVE      D2.L, D2.H  ;move unbiased exponent
FSCALE.X  D2.H, D1    ;scale original mantissa
```

Input (SEP)	**Output (SEP)**
-infinity	NaN, signals OPERR
negative, non-zero	signed mantissa
-0.0	-0.0
+0.0	+0.0
positive, non-zero	signed mantissa
+infinity	NaN, signals OPERR
NaN	NaN

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Always cleared.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Set if the source operand is infinity. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FGETMAN S,D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	1sss	uu10	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FINT Extract the Integer Part FINT

Operation:

S → ROUND TO INTEGER → D
(parallel data bus move)

Assembler Syntax:

FINT S,D (move syntax - see the MOVE instruction description.)

Description:

Round the floating-point source operand S to an integer value using the current rounding mode specified by bits R1-R0 in the IER register, and store the result as a floating-point number in the specified destination D. The rounding precision is always SEP. For example: if the rounding is to +∞, then 110.50 rounds to 111.00; however if the rounding is to 0, -∞, or even, then 110.50 rounds to 110.0.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Fields:

Instruction Format: FINT S,D (move syntax - see the MOVE instruction description.)

31 14	13			0
DATA BUS MOVE FIELD	01	1sss	uu11	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FJcc Floating-Point Jump Conditionally FJcc

Operation:	**Assembler Syntax:**
If cc, then xx → PC else PC+1 → PC	FJcc label (short)
If cc, then ea → PC else PC+1 → PC	FJcc ea

Description:

If the specified floating-point condition is true, program execution then continues at a location specified by an effective address in the instruction. If the specified condition is false, the PC is incremented and the effective address is ignored. However, the address register specified in the effective address field is always updated independently of the condition. All memory alterable addressing modes may be used for the effective address. A Fast Short Jump addressing mode may also be used. The 15-bit data is sign extended to form the effective address. See **Section A.10** for restrictions. Non-aware floating-point conditions set the SIOP flag in the IER register and the UNCC bit in the ER register if the NAN bit is set.

"cc" may specify the following conditions:

Mnemonic		**Condition**	**Non-aware Set UNCC***
EQ	- equal	Z = 1	No
ERR	- error	UNCC v SNAN v OPERR v OVF v UNF v DZ = 1	No
GE	- greater than or equal	NAN v (N & ~Z) = 0	Yes
GL	- greater or less than	NAN v Z = 0	Yes
GLE	- greater, less or equal	NAN = 0	Yes
GT	- greater than	NAN v Z v N = 0	Yes
INF	- infinity	I = 1	Yes
LE	- less than or equal	NAN v ~(N v Z) = 0	Yes
LT	- less than	NAN v Z v ~N = 0	Yes
MI	- minus	N = 1	No
NE(Q)	- not equal	Z = 0	No
NGE	- not(greater than or equal)	NAN v (N & ~Z) = 1	Yes
NGL	- not(greater or less than)	NAN v Z = 1	Yes
NGLE	- not(greater, less or equal)	NAN = 1	Yes
NGT	- not greater than	NAN v Z v N = 1	Yes
NINF	- not infinity	I = 0	Yes
NLE	- not(less than or equal)	NAN v ~(N v Z) = 1	Yes
NLT	- not less than	NAN v Z v ~N = 1	Yes
OR	- ordered	NAN = 0	No
PL	- plus	N = 0	No
UN	- unordered	NAN = 1	No

Note: The operands for the ERR condition are taken from the ER register.
* See the description of the UNcc bit in **Section A.4**.

CCR Condition Codes: Not affected.

ER Status Bits:

INX - Not affected.

DZ - Not affected.

UNF - Not affected.

OVF - Not affected.

OPERR- Not affected.

SNAN - Not affected.

NAN - Not affected.

UNCC - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

IER Flags:

SINX - Not affected.

SDZ - Not affected.

SUNF - Not affected.

SOVF - Not affected.

SIOP - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

Instruction Format: FJcc label (short)

31				14	13			0
0000	0011	10aa	aaaa	aa	1c	cccc	1aaa	aaaa

Instruction Format: FJcc ea

31				14	13			0
0000	0011	0000	MMMR		1c	cccc	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Short Jump Address - aaaaaaaaaaaaaaa (15 bits)

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
GT	0 0 0 0 0	NGT	1 0 0 0 0
LT	0 0 0 0 1	NLT	1 0 0 0 1
GE	0 0 0 1 0	NGE	1 0 0 1 0
LE	0 0 0 1 1	NLE	1 0 0 1 1
GL	0 0 1 0 0	NGL	1 0 1 0 0
INF	0 0 1 0 1	NINF	1 0 1 0 1
GLE	0 0 1 1 0	NGLE	1 0 1 1 0
OR	0 0 1 1 1	UN	1 0 1 1 1
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
ERR	0 1 1 1 1		

Timing: 6 + jx oscillator clock cycles
Memory: 1 + ea program words

FJScc Floating-Point Jump To Subroutine Conditionally FJScc

Operation:

If cc, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xx $\rightarrow$ PC
else PC+1 $\rightarrow$ PC

If cc, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; ea $\rightarrow$ PC
else PC+1 $\rightarrow$ PC

Assembler Syntax:

FJScc label (short)

Description:

If the specified floating-point condition is true, the address of the instruction immediately following the FJScc instruction and the status register are pushed onto the stack. Program execution then continues at the effective address in program memory. If the specified condition is false, the PC is incremented and any extension word is ignored. However, the address register specified in the effective address field is always updated independently of the condition. All memory alterable addressing modes may be used for the effective address. A fast Short Jump addressing mode may also be used. The 15-bit data is sign extended to form the effective address. See **Section A.10** for restrictions. Non-aware floating-point conditions set the SIOP flag in the IER and the UNCC bit in the ER if the NAN bit is set. This action occurs before stacking the status register when the specified non-aware floating-point condition is true.

"cc" may specify the following conditions:

Mnemonic		**Condition**	**Non-aware Set UNCC***
EQ	- equal	Z = 1	No
ERR	- error	UNCC v SNAN v OPERR v OVF v UNF v DZ = 1	No
GE	- greater than or equal	NAN v (N & ~Z) = 0	Yes
GL	- greater or less than	NAN v Z = 0	Yes
GLE	- greater, less or equal	NAN = 0	Yes
GT	- greater than	NAN v Z v N = 0	Yes
INF	- infinity	I = 1	Yes
LE	- less than or equal	NAN v ~(N v Z) = 0	Yes
LT	- less than	NAN v Z v ~N = 0	Yes
MI	- minus	N = 1	No
NE(Q)	- not equal	Z = 0	No
NGE	- not(greater than or equal)	NAN v (N & ~Z) = 1	Yes
NGL	- not(greater or less than)	NAN v Z = 1	Yes
NGLE	- not(greater, less or equal)	NAN = 1	Yes
NGT	- not greater than	NAN v Z v N = 1	Yes
NINF	- not infinity	I = 0	Yes
NLE	- not(less than or equal)	NAN v ~(N v Z) = 1	Yes
NLT	- not less than	NAN v Z v ~N = 1	Yes
OR	- ordered	NAN = 0	No
PL	- plus	N = 0	No
UN	- unordered	NAN = 1	No

Note: The operands for the ERR condition are taken from the ER register.
* See the description of the UNcc bit in **Section A.4**.

CCR Condition Codes: Not affected.

ER Status Bits:

INX	- Not affected.
DZ	- Not affected.
UNF	- Not affected.
OVF	- Not affected.
OPERR	- Not affected.
SNAN	- Not affected.
NAN	- Not affected.
UNCC	-Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

IER Flags:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	-Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

Instruction Fields:

Instruction Format: FJScc label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	1c	cccc	1aaa	aaaa

Instruction Format: FJScc ea

31			14	13			0
0000	0011	0100	MMMR	1c	cccc	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION							

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Short Jump Address - aaaaaaaaaaaaaaa (15 bits)

Mnemonic	c c c c c	Mnemonic	c c c c c
GT	0 0 0 0 0	NGT	1 0 0 0 0
LT	0 0 0 0 1	NLT	1 0 0 0 1
GE	0 0 0 1 0	NGE	1 0 0 1 0
LE	0 0 0 1 1	NLE	1 0 0 1 1
GL	0 0 1 0 0	NGL	1 0 1 0 0
INF	0 0 1 0 1	NINF	1 0 1 0 1
GLE	0 0 1 1 0	NGLE	1 0 1 1 0
OR	0 0 1 1 1	UN	1 0 1 1 1
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
ERR	0 1 1 1 1		

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

FLOAT.S Integer to Floating-Point Conversion FLOAT.S

Operation:

D.L → CONVERT TO FP → ROUND(SP) → D
(parallel data bus move)

Assembler Syntax:

FLOAT.S D (move syntax - see the MOVE instruction description.)

Description:

Convert the 2's complement 32-bit integer located in the low portion of the operand D into a floating-point operand, round to single precision and store the result in the operand D.

Input Operand(s) Precision: 32-bit 2's complement integer.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Always cleared.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Always cleared.
NAN	-Always cleared.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FLOAT.S D (move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	0100	uu11	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FLOAT.X Integer to Floating-Point Conversion FLOAT.X

Operation:

D.L → CONVERT TO FP → D (parallel data bus move)

Assembler Syntax:

FLOAT.X D (move syntax - see the MOVE instruction description.)

Description:

Convert the 2’s complement 32-bit integer located in the low portion of the operand D into a floating-point operand and store the result in the operand D. The rounding precision is SEP.

Input Operand(s) Precision: 32-bit 2’s complement integer.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Always cleared.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Always cleared.
NAN	-Always cleared.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FLOAT.X D (move syntax - see the MOVE instruction description.)

31 14	13 0
DATA BUS MOVE FIELD	10 0100 uu10 0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FLOATU.S Unsigned Integer to Floating-Point Conversion FLOATU.S

Operation:

D.L → CONVERT TO FP → ROUND(SP) → D

(parallel data bus move)

Assembler Syntax:

FLOATU.S D

(move syntax - see the MOVE instruction description.)

Description:

Convert the unsigned 32-bit integer located in the low portion of the operand D into a floating-point operand, round to single precision and store the result in the operand D.

Input Operand(s) Precision: 32-bit unsigned integer.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Always cleared.
I	- Always cleared.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Always cleared.
NAN	-Always cleared.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FLOATU.S D move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0
DATA BUS MOVE FIELD	10 0101 uu11 0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FLOATU.X Unsigned Integer to Floating-Point Conversion FLOATU.X

Operation:

D.L → CONVERT TO FP → D
(parallel data bus move)

Assembler Syntax:

FLOATU.X D (move syntax - see the MOVE instruction description.)

Description:

Convert the unsigned 32-bit integer located in the low portion of the operand D into a floating-point operand and store the result in the operand D. The rounding precision is SEP.

Input Operand(s) Precision: 32-bit unsigned integer.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Always cleared.
I	- Always cleared.
LR	- Not affected.
$\overline{\ }$R	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Always cleared.
NAN	-Always cleared.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FLOATU.X D (move syntax - see the MOVE instruction description.)

31 14	13 0			
DATA BUS MOVE FIELD	10	0101	uu10	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

(u u)

D ddd

Dnnnn where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FLOOR   Extract the Integer Part   FLOOR

Operation:

S→ ROUND TO INTEGER → D

(parallel data bus move)

Assembler Syntax:

FLOOR S,D (move syntax - see the MOVE instruction description.)

Description:

Round the floating-point source operand S to an integer value using the round to minus infinity mode and store the result as a floating-point number in the specified destination D. The rounding precision is always SEP. FLOOR is equivalent to FINT with R1, R0 in the IER set to select minus infinity; however, the rounding mode does not need to be saved, changed, and recalled. This is particularly useful when using C since FLOOR is a standard C function.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FLOOR S,D (move syntax - see the MOVE instruction description.)

31	14	13			0
DATA BUS MOVE FIELD		01	0sss	uu11	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA					

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FMPY//FADD.S Floating-Point Multiply and Add FMPY//FADD.S

Operation:

$S1 * S2 \rightarrow$ ROUND(MP) $\rightarrow$ D1

(parallel data bus move)

$S3 + D2 \rightarrow$ ROUND(SP) $\rightarrow$ D2

Assembler Syntax:

FMPY S1,S2,D1 FADD.S S3,D2

(move syntax - see the MOVE instruction description.)

FMPY S2,S1,D1 FADD.S S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two operands S1 and S2, round to the precision indicated by the MP mode bit and store the result in the specified destination register D1. Simultaneously, add the two operands S3 and D2, round to single precision and store the result in the destination operand D2. Typically, if the result of the multiplication will be used immediately following a data ALU instruction such as FADD (i.e., equivalent to an FMAC), the maximum precision (MP=1) will be programmed. However, if the product is to be stored, then single precision (MP=0) rounding will be used.

Input Operand(s) Precision: SEP Floating-Point.

Addition Output Operand Precision: SP Floating-Point.

Multiplication Output Operand Precision: as indicated by MP.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Set if result of the addition is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	- Set if the result of the addition or the multiplication is inexact. Cleared otherwise.
DZ	- Always cleared.
UNF	- Set if the result of the addition or the multiplication underflows. Cleared otherwise.
OVF	- Set if the result of the addition or the multiplication overflows. Cleared otherwise.
OPERR	- Set if one of the multiply operands is infinity and the other is zero. Set if the addition operands are opposite-signed infinities. Cleared otherwise.

SNAN -Set if anyone of the source operands is a signaling NaN. Cleared otherwise.

NAN -Set if result of the addition is a NaN. Cleared otherwise.

UNCC -Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FMPY S1,S2,D1 FADD.S S3,D2 (move syntax - see the MOVE instruction description.)

Instruction Fields:

31 — 14	13 — 0			
DATA BUS MOVE FIELD	00	1sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

Multiply and Add

Operation:

S1 * S2 → ROUND(SEP) → D1

(parallel data bus move)

S3 + D2 → ROUND(SEP) → D2

Assembler Syntax:

FMPY S1,S2,D1 FADD.X S3,D2

(move syntax - see the MOVE instruction description.)

FMPY S2,S1,D1 FADD.X S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two operands S1 and S2, round to single extended precision and store the result in the specified destination register D1. Simultaneously, add the two operands S3 and D2, round to single extended precision and store the result in the destination operand D2.

Input Operand(s) Precision: SEP Floating-Point.

Addition Output Operand Precision: SEP Floating-Point.

Multiplication Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Set if result of the addition is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the result of the addition or the multiplication is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if the result of the addition or the multiplication underflows. Cleared otherwise.
OVF	-Set if the result of the addition or the multiplication overflows. Cleared otherwise.
OPERR	-Set if one of the multiply operands is infinity and the other is zero. Set if the addition operands are opposite-signed infinities. Cleared otherwise.
SNAN	-Set if anyone of the source operands is a signaling NaN. Cleared otherwise.
NAN	-Set if result of the addition is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FMPY S1,S2,D1 FADD.X S3,D2 (move syntax - see the MOVE instruction description.)

FMPY S2,S1,D1 FADD.X S3,D2 (move syntax - see the MOVE instruction description.)

Instruction Fields:

31		14	13				0
DATA BUS MOVE FIELD			00	0sss	ddQQ	QQDD	
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA							

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

FMPY//FADDSUB.S FMPY//FADDSUB.S

Floating-Point Multiply, Add, and Subtract

Operation:

S1 * S2 → ROUND(MP) → D1

(parallel data bus move)

D3 + D2 → ROUND(SP) → D2

D3 - D2 → ROUND(SP) → D3

Assembler Syntax:

FMPY S1,S2,D1 FADDSUB.S D3,D2

(move syntax - see the MOVE instruction description.)

FMPY S2,S1,D1 FADDSUB.S D3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two operands S1 and S2, round to the precision indicated by the MP mode bit and store the result in the specified destination register D1. Simultaneously, add the two operands D2 and D3, subtract D2 from D3, round both results to single precision and store the result of the addition in register D2 and of the subtraction in register D3. Typically, if the result of the multiplication will be used immediately following a data ALU instruction such as FADD (i.e., equivalent to an FMAC), the maximum precision (MP=1) will be programmed. However, if the product is to be stored, then single precision (MP=0) rounding will be used. For the special case of |s|=|D|, the result can be +0 or -0; the sign of the resulting zero will be the sign of the input operand in D.

Input Operand(s) Precision: SEP Floating-Point.

Addition Output Operand Precision: SP Floating-Point.

Subtraction Output Operand Precision: SP Floating-Point.

Multiplication Output Operand Precision: as indicated by MP.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Set if result of the addition is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the result of the addition, subtraction or multiplication is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if the result of the addition, subtraction or multiplication underflows. Cleared otherwise.
OVF	-Set if the result of the addition, subtraction or multiplication overflows. Cleared otherwise.
OPERR	-Set if one of the multiply operands is infinity and the other is zero. Set if the addition operands are opposite-signed infinities. Set if the subtract operands are like-signed infinities. Cleared otherwise.
SNAN	-Set if anyone of the source operands is a signaling NaN. Cleared otherwise.
NAN	-Set if result of the addition is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	1sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format:

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

Floating-Point Multiply, Add, and Subtract

Operation:

$S1 * S2 \rightarrow$ ROUND(SEP) $\rightarrow$ D1

(parallel data bus move)

$D3 + D2 \rightarrow$ ROUND(SEP) $\rightarrow$ D2

$D3 - D2 \rightarrow$ ROUND(SEP) $\rightarrow$ D3

Assembler Syntax:

FMPY S1,S2,D1 FADDSUB.X D3,D2

(move syntax - see the MOVE instruction description.)

FMPY S2,S1,D1 FADDSUB.X D3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two operands S1 and S2, round to single extended precision and store the result in the specified destination register D1. Simultaneously, add the two operands D2 and D3, subtract D2 from D3, round both results to single extended precision and store the result of the addition in register D2 and of the subtraction in register D3. Typically, if the result of the multiplication will be used immediately following FADD (i.e., equivalent to an FMAC), the maximum precision (MP=1) will be programmed. For the special case of |s|=|D|, the result can be +0 or -0; the sign of the resulting zero will be the sign of the input operand in D.

Input Operand(s) Precision: SEP Floating-Point.

Addition Output Operand Precision: SEP Floating-Point.

Subtraction Output Operand Precision: SEP Floating-Point.

Multiplication Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Set if result of the addition is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

- INX - Set if the result of the addition, subtraction or multiplication is inexact. Cleared otherwise.
- DZ - Always cleared.
- UNF - Set if the result of the addition, subtraction or multiplication underflows. Cleared otherwise.
- OVF - Set if the result of the addition, subtraction or multiplication overflows. Cleared otherwise.
- OPERR - Set if one of the multiply operands is infinity and the other is zero. Set if the addition operands are opposite-signed infinities. Set if the subtract operands are like-signed infinities. Cleared otherwise.
- SNAN - Set if anyone of the source operands is a signaling NaN. Cleared otherwise.
- NAN - Set if result of the addition is a NaN. Cleared otherwise.
- UNCC - Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FMPY S1,S2,D1 FADDSUB.X D3,D2 (move syntax - see the MOVE instruction description.)

FMPY S2,S1,D1 FADDSUB.X D3,D2 (move syntax - see the MOVE instruction description.)

31 – 14	13 – 0			
DATA BUS MOVE FIELD	10	0sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

FMPY//FSUB.S Floating-Point Multiply and Subtract FMPY//FSUB.S

Operation:

$S1 * S2 \rightarrow ROUND(MP) \rightarrow D1$

(parallel data bus move)

$D2 - S3 \rightarrow ROUND(SP) \rightarrow D2$

Assembler Syntax:

FMPY S1,S2,D1 FSUB.S S3,D2

(move syntax - see the MOVE instruction description.)

FMPY S2,S1,D1 FSUB.S S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two operands S1 and S2, round to the precision indicated by the MP mode bit and store the result in the specified destination register D1. Simultaneously, subtract S3 from D2, round to single precision and store the result in the destination operand D2. Typically, if the result of the multiplication will be used immediately following FADD (i.e., equivalent to an FMAC), the maximum precision (MP=1) will be programmed. For the special case of |s|=|D|, the result can be +0 or -0; the sign of the resulting zero will be the sign of the input operand in D.

Input Operand(s) Precision: SEP Floating-Point.

Subtraction Output Operand Precision: SP Floating-Point.

Multiplication Output Operand Precision: as indicated by MP.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result of the subtraction is zero. Cleared otherwise.
N	- Set if result of the subtraction is negative. Cleared otherwise.
I	- Set if result of the subtraction is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the result of the subtraction or multiplication is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if the result of the subtraction or multiplication underflows. Cleared otherwise.
OVF	-Set if the result of the subtraction or multiplication overflows. Cleared otherwise.

OPERR -Set if one of the multiply operands is infinity and the other is zero. Set if the subtract operands are like-signed infinities. Cleared otherwise.

SNAN -Set if anyone of the source operands is a signaling NaN. Cleared otherwise.

NAN -Set if result of the subtraction is a NaN. Cleared otherwise.

UNCC -Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FMPY S1,S2,D1 FSUB.S S3,D2 (move syntax - see the MOVE instruction description.)

31 … 14	13 … 0			
DATA BUS MOVE FIELD	01	1sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

FMPY S2,S1,D1 FSUB.S S3,D2 (move syntax - see the MOVE instruction description.)

FMPY//FSUB.X Floating-Point Multiply and Subtract FMPY//FSUB.X

Operation:

S1 $*$ S2 $\rightarrow$ ROUND(SEP) $\rightarrow$ D1
(parallel data bus move)

D2 - S3 $\rightarrow$ ROUND(SEP) $\rightarrow$ D2

Assembler Syntax:

FMPY S1,S2,D1 FSUB.X S3,D2
(move syntax - see the MOVE instruction description.)

FMPY S2,S1,D1 FSUB.X S3,D2
(move syntax - see the MOVE instruction description.)

Description:

Multiply the two operands S1 and S2, round to single extended precision and store the result in the specified destination register D1. Simultaneously, subtract S3 from D2, round to single extended precision and store the result in the destination operand D2. For the special case of |s|=|D|, the result can be +0 or -0; the sign of the resulting zero will be the sign of the input operand in D.

Input Operand(s) Precision: SEP Floating-Point.

Subtraction Output Operand Precision: SEP Floating-Point.

Multiplication Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result of the subtraction is zero. Cleared otherwise.
N	- Set if result of the subtraction is negative. Cleared otherwise.
I	- Set if result of the subtraction is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{\text{R}}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the result of the subtraction or multiplication is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if the result of the subtraction or multiplication underflows. Cleared otherwise.
OVF	-Set if the result of the subtraction or multiplication overflows. Cleared otherwise.
OPERR	-Set if one of the multiply operands is infinity and the other is zero. Set if the subtract operands are like-signed infinities. Cleared otherwise.
SNAN	-Set if anyone of the source operands is a signaling NaN. Cleared otherwise.

NAN -Set if result of the subtraction is a NaN. Cleared otherwise.

UNCC -Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FMPY S1,S2,D1 FSUB.X S3,D2 (move syntax - see the MOVE instruction description.)
FMPY S2,S1,D1 FSUB.X S3,D2 (move syntax - see the MOVE instruction description.)

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	0sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

Operation:	**Assembler Syntax:**
S1 $*$ S2 $\rightarrow$ ROUND(SP) $\rightarrow$ D (parallel data bus move)	FMPY.S S1,S2,D (move syntax - see the MOVE instruction description.)
S1 $*$ S2 $\rightarrow$ ROUND(SP) $\rightarrow$ D (parallel data bus move)	FMPY.S S2,S1,D (move syntax - see the MOVE instruction description.)

Description:

Multiply the two specified operands S1 and S2, round to single precision and store the result in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Set if one operand is infinity and the other zero. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FMPY.S S1,S2,D (move syntax - see the MOVE instruction description.)

31	14	13			0
DATA BUS MOVE FIELD		11	1sss	SSS1	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA					

Instruction Format: FMPY.S S1,S2(8,9),D (move syntax - see the MOVE instruction description.)

31	14	13			0
DATA BUS MOVE FIELD		11	0sss	11S1	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA					

Instruction Fields:

S1	**s s s**	
Dn	n n n	where nnn = 0-7

S2	**S S S**	
Dn	n n n	where nnn = 0-7

S2	**S**
D8	0
D9	1

D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles
Memory: 1 + mv program words

Operation:	Assembler Syntax:
$S1 * S2 \rightarrow ROUND(SEP) \rightarrow D$ (parallel data bus move)	FMPY.X S1,S2,D (move syntax - see the MOVE instruction description.)
$S1 * S2 \rightarrow ROUND(SEP) \rightarrow D$ (parallel data bus move)	FMPY.X S2,S1,D (move syntax - see the MOVE instruction description.)

Description:

Multiply the two specified operands S1 and S2, round to single extended precision and store the result in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{\ }$R	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Set if one operand is infinity and the other zero. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FMPY.X S1,S2,D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	1sss	SSS0	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: FMPY.X S1,S2(8,9),D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	0sss	11s0	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

S1	**s s s**	
Dn	n n n	where nnn = 0-7

S2	**S S S**	
Dn	n n n	where nnn = 0-7

S2	**s**
D8	0
D9	1

D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FNEG.S Negate FNEG.S

Operation:

0 - D → ROUND(SP) → D
(parallel data bus move)

Assembler Syntax:

FNEG.S D
(move syntax - see the MOVE instruction description.)

Description:

Subtract the destination operand D from zero, round to single precision and store the result in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FNEG.S D (move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	0001	uu01	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

Operation:

0 - D → D (parallel data bus move)

Assembler Syntax:

FNEG.X D
(move syntax - see the MOVE instruction description.)

Description:

Subtract the destination operand D from zero and store the result in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FNEG.X D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0
DATA BUS MOVE FIELD	10 0001 uu00 0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FSCALE.S Scale a Floating-Point Operand FSCALE.S

Operation:

$2^{S.H} * D \rightarrow ROUND(SP) \rightarrow D$ (parallel data bus move)

$2^{nn} * D \rightarrow ROUND(SP) \rightarrow D$

Assembler Syntax:

FSCALE.S S,D
(move syntax - see the MOVE instruction description.)

FSCALE.S #byte,D

Description:

Scale the destination operand D according to the scale factor contained in the 11 LSBs of the high portion of the source register S, round to single precision and store the result in the destination operand D. An 8-bit Immediate Short scaling factor, sign-extended to 11 bits, may also be used. The scale factor is a signed 2's complement 11-bit integer.

As an example of the use of FGETMAN, GETEXP, and FSCALE; consider decomposing a floating-point number into its mantissa and unbiased exponent and then recreating the original floating-point number.

```
FGETMAN    D0, D1       ;extract normalized mantissa
GETEXP     D0,D2        ;extract unbiased exponent
MOVE       D2.L, D2.H   ;move unbiased exponent
FSCALE.S   D2.H, D1     ;scale original mantissa
```

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FSCALE.S S,D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	0sss	uu10	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: FSCALE.S #byte,D

31				14	13			0
0000	0000	0000	0000	10	11	1nnn	nnnn	nddd

Instruction Fields:

Immediate Short Data - nnnnnnnn (8 bits)

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn.H	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles (2 + da oscillator clock cycles for FSCALE.S #byte,D)

Memory: 1 + mv program words (1 program word for FSCALE.S #byte,D)

FSCALE.X Scale a Floating-Point Operand FSCALE.X

Operation:

$2^{S.H} * D \rightarrow ROUND(SEP) \rightarrow D$

(parallel data bus move)

$2^{nn} * D \rightarrow ROUND(SEP) \rightarrow D$

Assembler Syntax:

FSCALE.X S,D

(move syntax - see the MOVE instruction description.)

FSCALE.X #byte,D

Description:

Scale the destination operand D according to the scale factor contained in the 11 LSBs of the high portion of the source register S, round to single extended precision and store the result in the destination operand D. An 8-bit Immediate Short scaling factor, sign-extended to 11 bits, may also be used. The scale factor is a signed 2's complement 11-bit integer.

As an example of the use of FGETMAN, GETEXP, and FSCALE; consider decomposing a floating-point number into its mantissa and unbiased exponent and then recreating the original floating-point number.

```
FGETMAN    D0, D1      ;extract normalized mantissa
GETEXP     D0,D2       ;extract unbiased exponent
MOVE       D2.L, D2.H  ;move unbiased exponent
FSCALE.S   D2.H, D1    ;scale original mantissa
```

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

Instruction Format: FSCALE.X S,D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	0sss	uu10	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format:FSCALE.X #byte,D

31				14	13			0
0000	0000	0000	0000	10	11	0nnn	nnnn	nddd

IER Flags: Flags changed according to standard definition.

Instruction Fields:

Immediate Short Data - nnnnnnnn (8 bits)

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn.H	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles (2 + da oscillator clock cycles for FSCALE.X #byte,D)
Memory: 1 + mv program words (1 program word for FSCALE.X #byte,D)

FSEEDD Reciprocal Approximation FSEEDD

Operation:

Approximation(1/S) → D

Assembler Syntax:

FSEEDD S,D

Description:

Take the contents of the specified source operand S, determine an approximation to 1.0/S, and store the result in the destination operand D. The 9 MSBs of the destination significand are determined by using a lookup ROM. The remaining bits of the significand are zeroed. This instruction is useful for initializing floating-point divide algorithms.

The table below describes the operation of the FSEEDD instruction:

Source Operand	Result
SNaN or QNaN	QNaN
+/- zero	+/- infinity
+/- denormalized	normalized, then FSEEDD approximation
+/- normalized	FSEEDD approximation
+/- infinity	+/- zero

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{}$R	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Always cleared.
SNAN	-Set if the source operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FSEEDD S,D

31				14	13			0
0000	0000	0000	0000	10	00	0sss	1111	1ddd

Instruction Fields:

D	**d d d**	
Dn	n n n	where nnn = 0-7

S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + da oscillator clock cycles

Memory: 1 program words

FSEEDR Square Root Reciprocal Approximation FSEEDR

Operation:

Approximation(1/SQRT(S)) → D

Assembler Syntax:

FSEEDR S,D

Description:

Take the contents of the specified source operand S, determine an approximation to sqrt(1.0/S), and store the result in the destination operand D. The 9 MSBs of the destination significand are determined by using a lookup ROM. The remaining bits of the significand are zeroed. This instruction is useful for initializing floating-point square root algorithms.

The table below describes the operation of the FSEEDR instruction:

Source Operand	Result
SNaN or QNaN	QNaN
less than zero	QNaN
+/- zero	+/- zero
+ denormalized	normalized, then FSEEDR approximation
+ normalized	FSEEDR approximation
+ infinity	+ infinity

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Set if the source operand is less than zero. Cleared otherwise.
SNAN	-Set if the source operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FSEEDR S,D

31				14	13			0
0000	0000	0000	0000	10	00	0sss	1111	0ddd

Instruction Fields:

D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + da oscillator clock cycles

Memory: 1 program words

Operation:

D - S → ROUND(SP) → D
(parallel data bus move)

Assembler Syntax:

FSUB.S S,D
(move syntax - see the MOVE instruction description.)

Description:

Subtract the two specified operands, round to single precision and store the result in the destination operand D. For the special case of |S| = |D|, the result can be +0 or -0; the sign of the resulting zero will be the sign of the input operand in D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{\ }$R	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Set if operands are like-signed infinities. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FSUB.S S,D (move syntax - see the MOVE instruction description.)

31 DATA BUS MOVE FIELD 14	13			0
DATA BUS MOVE FIELD	01	1sss	uu00	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FSUB.X Floating-Point Subtract FSUB.X

Operation:

D - S $\rightarrow$ ROUND(SEP) $\rightarrow$ D
(parallel data bus move)

Assembler Syntax:

FSUB.X S,D
(move syntax - see the MOVE instruction description.)

Description:

Subtract the two specified operands, round to single extended precision and store the result in the destination operand D. For the special case of |S| = |D|, the result can be +0 or -0; the sign of the resulting zero will be the sign of the input operand in D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Set if operands are like-signed infinities. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FSUB.X S,D (move syntax - see the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	01	1sss	uu00	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FTFR.S Transfer Floating-Point Data ALU Register FTFR.S

Operation:

$S \rightarrow ROUND(SP) \rightarrow D$
(parallel data bus move)

Assembler Syntax:

FTFR.S S,D
(move syntax - see the MOVE instruction description.)

Description:

Take the contents of the specified source operand S, round to single precision and store the result in the destination operand D. If S and D are the same register, this is equivalent to “Round to Single Precision” instruction.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if result is inexact. Cleared otherwise.
DZ	-Always cleared.
UNF	-Set if result underflows. Cleared otherwise.
OVF	-Set if result overflows. Cleared otherwise.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FTFR.S S,D (move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	1sss	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FTFR.X Transfer Floating-Point Data ALU Register FTFR.X

Operation:

S → D (parallel data bus move)

Assembler Syntax:

FTFR.X S,D
(move syntax - see the MOVE instruction description.)

Description:

Take the contents of the specified source operand S and store in the destination operand D.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: SEP Floating-Point.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FTFR.X S,D (move syntax - see the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	1sss	uu11	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7
S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

FTRAPcc Conditional Software Interrupt FTRAPcc

Operation:	**Assembler Syntax:**
If cc, then begin software exception processing.	FTRAPcc

Description:

If the specified floating-point condition is true, normal instruction execution is suspended and software exception processing is initiated. The interrupt priority level (I1,I0) is set to 3 in the status register if a long interrupt service routine is used. If the specified condition is false, continue with the next instruction. See **Section A.10** for restrictions. Non-aware floating-point conditions set the SIOP flag in the IER register and the UNCC bit in the ER register if the NAN bit is set. This action occurs before stacking the status register when the specified non-aware floating-point condition is true.

"cc" may specify the following conditions:

Mnemonic		**Condition**	**Non-aware Set UNCC***
EQ	- equal	Z = 1	No
ERR	- error	UNCC v SNAN v OPERR v OVF v UNF v DZ = 1	No
GE	- greater than or equal	NAN v (N & ~Z) = 0	Yes
GL	- greater or less than	NAN v Z = 0	Yes
GLE	- greater, less or equal	NAN = 0	Yes
GT	- greater than	NAN v Z v N = 0	Yes
INF	- infinity	I = 1	Yes
LE	- less than or equal	NAN v ~(N v Z) = 0	Yes
LT	- less than	NAN v Z v ~N = 0	Yes
MI	- minus	N = 1	No
NE(Q)	- not equal	Z = 0	No
NGE	- not(greater than or equal)	NAN v (N & ~Z) = 1	Yes
NGL	- not(greater or less than)	NAN v Z = 1	Yes
NGLE	- not(greater, less or equal)	NAN = 1	Yes
NGT	- not greater than	NAN v Z v N = 1	Yes
NINF	- not infinity	I = 0	Yes
NLE	- not(less than or equal)	NAN v ~(N v Z) = 1	Yes
NLT	- not less than	NAN v Z v ~N = 1	Yes
OR	- ordered	NAN = 0	No
PL	- plus	N = 0	No
UN	- unordered	NAN = 1	No

Note: The operands for the ERR condition are taken from the ER register.
* See the description of the UNcc bit in **Section A.4**.

CCR Condition Codes: Not affected.

ER Status Bits:

INX - Not affected.

DZ - Not affected.

UNF - Not affected.

OVF - Not affected.

OPERR - Not affected.

SNAN - Not affected.

UNCC - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

IER Flags:

SINX - Not affected.

SDZ - Not affected.

SUNF - Not affected.

SOVF - Not affected.

SIOP - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

Instruction Format: FTRAPcc

Instruction Fields:

31				14	13			0
0000	0000	0000	0000	00	1c	cccc	0000	0011

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
GT	0 0 0 0 0	NGT	1 0 0 0 0
LT	0 0 0 0 1	NLT	1 0 0 0 1
GE	0 0 0 1 0	NGE	1 0 0 1 0
LE	0 0 0 1 1	NLE	1 0 0 1 1
GL	0 0 1 0 0	NGL	1 0 1 0 0
INF	0 0 1 0 1	NINF	1 0 1 0 1
GLE	0 0 1 1 0	NGLE	1 0 1 1 0
OR	0 0 1 1 1	UN	1 0 1 1 1
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
ERR	0 1 1 1 1		

Timing: 10 oscillator clock cycles

Memory: 1 program words

FTST Test a Floating-Point Operand FTST

Operation:		**Assembler Syntax:**
S - 0	(parallel data bus move)	FTST S (move syntax - see the Move instruction description.)

Description:

Compare the specified operand with zero. No result is stored, however, the condition codes are affected as described.

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: n.a.

CCR Condition Codes:

Note: Since there is no destination, there is no rounding and therefore the condition code bits are set assuming an infinite precision result.

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if result is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Always cleared.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if result is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: FTST S (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0110	uu00	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
S	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

GETEXP Extract Exponent GETEXP

Operation:

Exponent(S) → D.L (parallel data bus move)

Assembler Syntax:

GETEXP S,D
(move syntax - see the Move instruction description.)

Description:

Extract the exponent of the single extended precision floating-point operand S and store it as an unbiased, 2's complement, 32-bit integer in the low portion of D . The exponent value is decremented by the number of shifts needed to normalize the mantissa if the floating-point number was denormalized.

As an example of the use of FGETMAN, GETEXP, and FSCALE; consider decomposing a floating-point number into its mantissa and unbiased exponent and then recreating the original floating-point number.

```
FGETMAN     D0, D1        ;extract normalized mantissa
GETEXP      D0,D2         ;extract unbiased exponent
MOVE        D2.L, D2.H    ;move unbiased exponent
FSCALE.S    D2.H, D1      ;scale original mantissa
```

The following table lists the results for some special cases:

Source operand	**Result**
+/- infinity	$7FFFFFFF
+/- zero	$80000000
SNaN or QNaN	$FFFFFFFF

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Set if the source operand is infinity. Cleared otherwise.
LR	- Not affected.
R	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Always cleared.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Set if the source operand is infinity, zero or NaN. Cleared otherwise.
SNAN	-Set if the source operand is a signaling NaN. Cleared otherwise.
NAN	-Set if the source operand is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

31 14	13			0
DATA BUS MOVE FIELD	11	0sss	0110	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: GETEXP S,D (move syntax - see the Move instruction description.)

Instruction Fields:

D	**d d d**	
Dn.L	n n n	where nnn = 0-7

S	**s s s**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

Illegal Instruction Interrupt

ILLEGAL

Operation:

Begin Illegal Instruction exception processing.

Assembler Syntax:

ILLEGAL

Description:

Normal instruction execution is suspended and Illegal Instruction exception processing is initiated. The interrupt priority level (I1,I0) is set to 3 in the status register if a long interrupt service routine is used.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ILLEGAL

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0101

Instruction Fields:

None

Timing: 8 oscillator clock cycles

Memory: 1 program words

INC Increment by One INC

Operation:

D.L + 1 $\rightarrow$ D.L (parallel data bus move)

Assembler Syntax:

INC D
(move syntax - see the Move instruction description.)

Description:

Increment by one the low portion of the specified operand. The result is stored in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if carry is generated from the MSB of the result. Cleared otherwise.
V	- Set if result overflows. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: INC D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0110	uu11	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

INT Floating-Point to Integer Conversion INT

Operation:	**Assembler Syntax:**
Integer(D) $\rightarrow$ D.L (parallel data bus move)	INT D (move syntax - see the Move instruction description.)

Description:

Convert the specified floating-point operand to 32-bit, 2's complement integer. The rounding mode is that programmed in the SR. The result is stored in the low portion of D. The high and middle portions of D remain unchanged.

The following table lists the results for some special cases:

Source operand	**Result**
Greater than 2^{31}- 1	\$7FFFFFFF
Less than -2^{31}	\$80000000
+infinity	\$7FFFFFFF
-infinity	\$80000000
NaN	\$FFFFFFFF

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Set if source operand is a NaN, infinity, or its magnitude is too big to be representable in the integer number range. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if source operand is negative. Cleared otherwise.
I	- Set if source operand is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the floating-point operand has no exact integer representation. Cleared otherwise.
DZ	-Always cleared.
UNF	-Always cleared.

OVF -Always cleared.

OPERR -Set if source operand is a NaN or infinity. Set if overflow occurred. Cleared otherwise.

SNAN -Set if operand is a signaling NaN. Cleared otherwise.

NAN -Set if source operand is a NaN. Cleared otherwise.

UNCC -Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: INT D (move syntax - see the Move instruction description.)

Instruction Fields:

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0011	uu00	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

INTRZ — Floating-Point to Integer Conversion with Round to Zero

Operation:

Integer(D) → D.L (parallel data bus move)

Assembler Syntax:

INTRZ D
(move syntax - see the Move instruction description.)

Description:

Convert the specified floating-point operand to 32-bit, 2's complement integer rounding towards zero. The result is stored in the low portion of D. The high and middle portions of D remain unchanged. Since this operation is frequently required (e. g., truncation assignment), this instruction has been implemented to eliminate the need to change the rounding mode associated with INT.

The following table lists the results for some special cases:

Source operand	Result
Greater than 2^{31}- 1	$7FFFFFFF
Less than -2^{31}	$80000000
+infinity	$7FFFFFFF
-infinity	$80000000
NaN	$FFFFFFFF

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Set if source operand is a NaN, infinity, or its magnitude is too big to be representable in the integer number range. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if source operand is negative. Cleared otherwise.
I	- Set if source operand is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the floating-point operand has no exact integer representation. Cleared otherwise.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Set if source operand is a NaN or infinity. Set if overflow occurred. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if source operand is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: INTRZ D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0011	uu10	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D	(u u) d d d	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

INTU — Floating-Point to Unsigned Integer Conversion

Operation:

Unsigned Integer(D) → D.L
(parallel data bus move)

Assembler Syntax:

INTU D
(move syntax - see the Move instruction description.)

Description:

Convert the specified floating-point operand to 32-bit, unsigned integer. The rounding mode is that specified in the SR. The result is stored in the low portion of D. The high and middle portions of D remain unchanged.

The following table lists the results for some special cases:

Source operand	**Result**
Greater than 2^{31}- 1	$7FFFFFFF
Less than -2^{31}	$80000000
+infinity	$7FFFFFFF
-infinity	$80000000
NaN	$FFFFFFFF
+/- Zero	$00000000

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Set if source operand is a NaN, infinity or negative non-zero. Set if positive source operand is too big to be representable in the integer number range. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if source operand is negative. Cleared otherwise.
I	- Set if source operand is infinity. Cleared otherwise.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	-Set if the floating-point operand has no exact integer representation. Cleared otherwise.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Set if source operand is a NaN, infinity or negative non-zero. Also set if overflow occurred. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if source operand is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: INTU D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0010	uu00	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D	**(u u) d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

to Unsigned Integer with Round to Zero

Operation:

Unsigned Integer(D) → D.L
(parallel data bus move)

Assembler Syntax:

INTURZ D (move syntax - see the Move instruction description.)

Description:

Convert the specified floating-point operand to 32-bit, unsigned integer rounding towards zero. The result is stored in the low portion of D. The high and middle portions of D remain unchanged. Since this operation is frequently required (e. g., truncation assignment), this instruction has been implemented to eliminate the need to change the rounding mode associated with INTU.

The following table lists the results for some special cases:

Source operand	Result
Greater than 2^{31}- 1	\$7FFFFFFF
Less than -2^{31}	\$80000000
+infinity	\$7FFFFFFF
-infinity	\$80000000
NaN	\$FFFFFFFF

Input Operand(s) Precision: SEP Floating-Point.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

- C - Not affected.
- V - Set if source operand is a NaN, infinity or negative non-zero. Set if positive source operand is too big to be representable in the integer number range. Cleared otherwise.
- Z - Set if result is zero. Cleared otherwise.
- N - Set if source operand is negative. Cleared otherwise.
- I - Set if source operand is infinity. Cleared otherwise.
- LR - Not affected.
- $\overline{R}$ - Not affected.
- A - Not affected.

ER Status Bits:

INX	-Set if the floating-point operand has no exact integer representation. Cleared otherwise.
DZ	-Always cleared.
UNF	-Always cleared.
OVF	-Always cleared.
OPERR	-Set if source operand is a NaN, infinity or negative non-zero. Also set if overflow occurred. Cleared otherwise.
SNAN	-Set if operand is a signaling NaN. Cleared otherwise.
NAN	-Set if source operand is a NaN. Cleared otherwise.
UNCC	-Always cleared.

IER Flags: Flags changed according to standard definition.

Instruction Format: INTURZ D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0010	uu10	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv + da oscillator clock cycles

Memory: 1 + mv program words

Jump Conditionally

Operation:

If cc, then xx $\rightarrow$ PC
else PC + 1 $\rightarrow$ PC

If cc, then ea $\rightarrow$ PC
else PC + 1 $\rightarrow$ PC

Assembler Syntax:

Jcc label (short)

Jcc ea

Description:

If the specified condition is true, program execution continues at a location specified by an effective address in the instruction. If the specified condition is false, the PC is incremented and the effective address is ignored. However, the address register specified in the effective address field is always updated independently of the condition. All memory alterable addressing modes may be used for the effective address. A Fast Short Jump addressing mode may also be used. The 15-bit data is sign extended to form the effective address. See **Section A.10** for restrictions.

"cc" may specify the following conditions:

Mnemonic		Condition
CC (HS)	- carry clear (higher or same)	C = 0
CS (LO)	- carry set (lower)	C = 1
EQ	- equal	Z = 1
GE	- greater or equal	N && V = 0
GT	- greater than	Z v (N && V) = 0
HI	- higher	Z v C = 0
LE	- less or equal	Z v (N && V) = 1
LS	- lower or same	Z v C = 1
LT	- less than	N && V = 1
MI	- minus	N = 1
NE(Q)	- not equal	Z = 0
PL	- plus	N = 0
VC	- overflow clear	V = 0
VS	- overflow set	V = 1

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: Jcc label (short)

31				14	13			0
0000	0011	10aa	aaaa	aa	1c	cccc	1aaa	aaaa

Instruction Format: Jcc ea

31				14	13			0
0000	0011	0000	MMMR		1c	cccc	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Short Jump Address - aaaaaaaaaaaaaaa (15 bits)

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
CC(HS)	0 1 0 1 0	CS(LO)	1 1 0 1 0
GE	0 1 0 1 1	LT	1 1 0 1 1
GT	0 1 1 0 0	LE	1 1 1 0 0
VC	0 1 1 0 1	VS	1 1 1 0 1
HI	0 1 1 1 0	LS	1 1 1 1 0

Timing: 4 + jx oscillator clock cycles

Memory: 1 + ea program words

Jump if Bit Clear

Operation:	Assembler Syntax:
If S{n} = 0, then xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JCLR #bit,X: ea, label
If S{n} = 0, then xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JCLR #bit,X: aa, label
If S{n} = 0, then xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JCLR #bit,X: pp, label
If S{n} = 0, then xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JCLR #bit,Y: ea, label
If S{n} = 0, then xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JCLR #bit,Y: aa, label
If S{n} = 0, then xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JCLR #bit,Y: pp, label
If S{n} = 0, then xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JCLR #bit,S,label

Description:

The nth bit in the source operand is tested. If the tested bit is zero, program execution continues at a location specified by a 32-bit absolute address in the extension word of the instruction. Otherwise, the PC is incremented and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the condition. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. The bit to be tested is selected by an immediate bit number 0-31. See **Section A.10** for restrictions. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: JCLR #bit,S,label

31				14	13			0
0000	0010	1011	dddd	dd	d0	0100	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JCLR #bit,X: pp, label
JCLR #bit,Y: pp, label

31				14	13			0
0000	0010	1010	1ppp	pp	pp	010S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JCLR #bit,X: aa, label
JCLR #bit,Y: aa, label

31				14	13			0
0000	0010	1010	0aaa	aa	aa	010S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JCLR #bit,X: ea, label
JCLR #bit,Y: ea, label

31				14	13			0
0000	0010	1000	MMMR		00	010S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Fields:

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxx))

Absolute Address - 32 bits

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

Bit Number	**b b b b b**	
Bit 0-31	n n n n n	where nnnnn = 0-31

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 6 + jx oscillator clock cycles

Memory: 2 program words

JMP — Jump

Operation:

xx → PC

ea → PC

Assembler Syntax:

JMP label (short)

JMP ea

Description:

Program execution continues at the effective address in program memory. All memory alterable addressing modes may be used for the effective address. A fast Short Jump addressing mode may also be used. The 15-bit data is sign extended to form the effective address. See **Section A.10** for restrictions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: JMP label (short)

31				14	13			0
0000	0011	10aa	aaaa	aa	11	1111	1aaa	aaaa

Instruction Format: JMP ea

31				14	13			0
0000	0011	0000	MMMR		11	1111	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Short Jump Address - aaaaaaaaaaaaaaa (15 bits)

Timing: 4 + jx oscillator clock cycles

Memory: 1 + ea program words

JOIN Join Two 16-bit Integers JOIN

Operation:

S.L {15:0} → D.L {31:16}
(parallel data bus move)

D.L {15:0} → D.L {15:0}

Assembler Syntax:

JOIN S,D
(move syntax - see the Move instruction description.)

Description:

Transfer the 16 LSBs of the lower portion of source operand S into the 16 MSBs of the lower portion of destination D. The 16 LSBs of the lower portion of D remain unchanged.

Input Operand(s) Precision: 16-bit integer**.**

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: JOIN S,D (move syntax - see the Move instruction description.)

31 14	13 0			
DATA BUS MOVE FIELD	11	0sss	1010	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

D.L {7:0} → D.L {7:0} (parallel data bus move)

S.L {7:0} → D.L {15:8}

0 → D.L {31:16}

Assembler Syntax:

JOINB S,D
(move syntax - see the Move instruction description.)

Description:

Transfer the 8 LSBs of the lower portion of source operand S into bits 15-8 of the lower portion of destination D. The 8 LSBs of the lower portion of D remain unchanged. The 16 MSBs of the lower portion of D are zeroed.

Input Operand(s) Precision: 8-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Always cleared.
I	- Not affected.
LR	- Not affected.
$\overline{\text{R}}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: JOINB S,D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	0sss	1010	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

JScc Jump to Subroutine Conditionally JScc

Operation:	**Assembler Syntax:**
If cc, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JScc label (short)
If cc, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; ea $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JScc ea

Description:

If the specified condition is true, the address of the instruction immediately following the JScc instruction and the status register are pushed onto the stack. Program execution then continues at the effective address in program memory. If the specified condition is false, the PC is incremented and any extension word is ignored. However, the address register specified in the effective address field is always updated independently of the condition. All memory alterable addressing modes may be used for the effective address. A fast Short Jump addressing mode may also be used. The 15-bit data is sign extended to form the effective address. See **Section A.10** for restrictions.

"cc" may specify the following conditions:

Mnemonic		**Condition**
CC (HS)	- carry clear (higher or same)	C = 0
CS (LO)	- carry set (lower)	C = 1
EQ	- equal	Z = 1
GE	- greater or equal	N && V = 0
GT	- greater than	Z v (N && V) = 0
HI	- higher	Z v C = 0
LE	- less or equal	Z v (N && V) = 1
LS	- lower or same	Z v C = 1
LT	- less than	N && V = 1
MI	- minus	N = 1
NE(Q)	- not equal	Z = 0
PL	- plus	N = 0
VC	- overflow clear	V = 0
VS	- overflow set	V = 1

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: JScc label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	1c	cccc	1aaa	aaaa

Instruction Format: JScc ea

31			14	13			0
0000	0011	0100	MMMR	1c	cccc	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION							

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Short Jump Address - aaaaaaaaaaaaaaa (15 bits)

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
CC(HS)	0 1 0 1 0	CS(LO)	1 1 0 1 0
GE	0 1 0 1 1	LT	1 1 0 1 1
GT	0 1 1 0 0	LE	1 1 1 0 0
VC	0 1 1 0 1	VS	1 1 1 0 1
HI	0 1 1 1 0	LS	1 1 1 1 0
AL	1 1 1 1 1		

Timing: 4 + jx oscillator clock cycles
Memory: 1 + ea program words

Operation:	Assembler Syntax:
If S{n} = 0, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JSCLR #bit,X: ea, label
If S{n} = 0, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JSCLR #bit,X: aa, label
If S{n} = 0, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JSCLR #bit,X: pp, label
If S{n} = 0, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JSCLR #bit,Y: ea, label
If S{n} = 0, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JSCLR #bit,Y: aa, label
If S{n} = 0, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JSCLR #bit,Y: pp, label
If S{n} = 0, then PC $\rightarrow$ SSH; SR $\rightarrow$ SSL; xxxx $\rightarrow$ PC else PC + 1 $\rightarrow$ PC	JSCLR #bit,S,label

Description:

The nth bit in the source operand is tested. If the tested bit is zero, the address of the instruction immediately following the JSCLR instruction and the status register are pushed onto the stack. Program execution then continues at a location specified by a 32-bit absolute address in the extension word of the instruction. Otherwise, the PC is incremented and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the condition. All memory alterable addressing modes may be used for the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. The bit to be tested is selected by an immediate bit number 0-31. See **Section A.10** for restrictions. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one; if the condition is true, the push operation will write over the stack level where the SSH value was taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Fields:

Instruction Format: JSCLR #bit,S,label

31				14	13			0
0000	0010	1111	dddd	dd	d0	0100	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSCLR #bit,X: pp, label
JSCLR #bit,Y: pp, label

31				14	13			0
0000	0010	1110	1ppp	pp	pp	010S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSCLR #bit,X: aa, label
JSCLR #bit,Y: aa, label

31				14	13			0
0000	0010	1110	0aaa	aa	aa	010S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSCLR #bit,X: ea, label
JSCLR #bit,Y: ea, label

31				14	13			0
0000	0010	1100	MMMR		00	010S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxx))
Absolute Address - 32 bits
Immediate Short Data - bbbbb (5 bits)
Absolute Short Address - aaaaaaa (7 bits)
I/O Short Address - ppppppp (7 bits)

D	**d d d d d d d**	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 6 + jx oscillator clock cycles
Memory: 2 program words

JSET — Jump if Bit Set

Operation:

If S{n} = 1, then xxxx → PC
else PC + 1 → PC

If S{n} = 1, then xxxx → PC
else PC + 1 → PC

If S{n} = 1, then xxxx → PC
else PC + 1 → PC

If S{n} = 1, then xxxx → PC
else PC + 1 → PC

If S{n} = 1, then xxxx → PC
else PC + 1 → PC

If S{n} = 1, then xxxx → PC
else PC + 1 → PC

If S{n} = 1, then xxxx → PC
else PC + 1 → PC

Assembler Syntax:

JSET #bit,X: ea, label

JSET #bit,X: aa, label

JSET #bit,X: pp, label

JSET #bit,Y: ea, label

JSET #bit,Y: aa, label

JSET #bit,Y: pp, label

JSET #bit,S,label

Description:

The nth bit in the source operand is tested. If the tested bit is set, program execution continues at a location specified by a 32-bit absolute address in the extension word of the instruction. Otherwise, the PC is incremented and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the condition. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. The bit to be tested is selected by an immediate bit number 0-31. See **Section A.10** for restrictions. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: JSET #bit,S,label

31				14	13			0
0000	0010	1011	dddd	dd	d0	1100	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSET #bit,X: pp, label
JSET #bit,Y: pp, label

31				14	13			0
0000	0010	1010	1ppp	pp	pp	110S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSET #bit,X: aa, label
JSET #bit,Y: aa, label

31				14	13			0
0000	0010	1010	0aaa	aa	aa	110S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSET #bit,X: ea, label
JSET #bit,X: ea, label

31				14	13			0
0000	0010	1000	MMMR		00	110S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Fields:

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxx))

Absolute Address - 32 bits

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

Bit Number	**b b b b b**	
Bit 0-31	n n n n n	where nnnnn = 0-31

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 6 + jx oscillator clock cycles
Memory: 2 program words

Jump to Subroutine

Operation:

PC → SSH; SR → SSL; xx → PC

PC → SSH; SR → SSL; ea → PC

Assembler Syntax:

JSR label (short)

JSR ea

Description:

The address of the instruction immediately following the JSR instruction and the status register are pushed onto the stack. Program execution then continues at the effective address in program memory. All memory alterable addressing modes may be used for the effective address. A fast Short Jump addressing mode may also be used. The 15-bit data is sign extended to form the effective address. See **Section A.10** for restrictions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: JSR label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	11	1111	1aaa	aaaa

Instruction Format: JSR ea

31				14	13			0
0000	0011	0100	MMMR		11	1111	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Short Jump Address - aaaaaaaaaaaaaaa (15 bits)

Timing: 4 + jx oscillator clock cycles
Memory: 1 + ea program words

JSSET Jump to Subroutine if Bit Set JSSET

Operation:	Assembler Syntax:
If S{n} = 1, then PC → SSH; SR → SSL; xxxx → PC else PC + 1 → PC	JSSET #bit,X: ea, label
If S{n} = 1, then PC → SSH; SR → SSL; xxxx → PC else PC + 1 → PC	JSSET #bit,X: aa, label
If S{n} = 1, then PC → SSH; SR → SSL; xxxx → PC else PC + 1 → PC	JSSET #bit,X: pp, label
If S{n} = 1, then PC → SSH; SR → SSL; xxxx → PC else PC + 1 → PC	JSSET #bit,Y: ea, label
If S{n} = 1, then PC → SSH; SR → SSL; xxxx → PC else PC + 1 → PC	JSSET #bit,Y: aa, label
If S{n} = 1, then PC → SSH; SR → SSL; xxxx → PC else PC + 1 → PC	JSSET #bit,Y: pp, label
If S{n} = 1, then PC → SSH; SR → SSL; xxxx → PC else PC + 1 → PC	JSSET #bit,S,label

Description:

The nth bit in the source operand is tested. If the tested bit is set, the address of the instruction immediately following the JSSET instruction and the status register are pushed onto the stack. Program execution then continues at a location specified by a 32-bit absolute address in the extension word of the instruction. Otherwise, the PC is incremented and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the condition. All memory alterable addressing modes may be used for the source operand. Register Direct, Absolute Short and I/O Short addressing modes may also be used. The bit to be tested is selected by an immediate bit number 0-31. See **Section A.10** for restrictions. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one; if the condition is true, the push operation will write over the stack level where the SSH value was read.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Fields:

Instruction Format:JSSET #bit,S,label

31				14	13			0
0000	0010	1011	dddd	dd	d0	1100	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSSET #bit,X: pp, label
JSSET #bit,Y: pp, label

31				14	13			0
0000	0010	1010	1ppp	pp	pp	110S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSSET #bit,X: aa, label
JSSET #bit,Y: aa, label

31				14	13			0
0000	0010	1010	0aaa	aa	aa	110S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

Instruction Format: JSSET #bit,X: ea, label
JSSET #bit,Y: ea, label

31				14	13			0
0000	0010	1000	MMMR		00	110S	100b	bbbb
ABSOLUTE ADDRESS EXTENSION								

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxx))

Absolute Address - 32 bits

Immediate Short Data - bbbbb (5 bits)

Absolute Short Address - aaaaaaa (7 bits)

I/O Short Address - ppppppp (7 bits)

Memory Space	**S**
X Memory	0
Y Memory	1

Bit Number	**b b b b b**	
Bit 0-31	n n n n n	where nnnnn = 0-31

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 6 + jx oscillator clock cycles

Memory: 2 program words

LEA Load Effective Address LEA

Operation:

ea → D

Rn+xxxx → D

Assembler Syntax:

LEA ea,D

LEA (Rn+displacement),D

Description:

The address calculation specified is executed and the resulting effective address is stored in the destination register. The source address registers are not affected. Post-update and Long Displacement address register indirect addressing modes may be used. Note that if D is SSH, the SP will be preincremented by one.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

For destination operand SR:

C	- Set according to bit 0 of the source operand.
V	- Set according to bit 1 of the source operand.
Z	- Set according to bit 2 of the source operand.
N	- Set according to bit 3 of the source operand.
I	- Set according to bit 4 of the source operand.
LR	- Set according to bit 5 of the source operand.
$\overline{R}$	- Set according to bit 6 of the source operand.
A	- Set according to bit 7 of the source operand.

For destination operands other than SR:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

For destination operand SR:

INX	-Set according to bit 8 of the source operand.
DZ	-Set according to bit 9 of the source operand.
UNF	-Set according to bit 10 of the source operand.
OVF	-Set according to bit 11 of the source operand.
OPERR	-Set according to bit 12 of the source operand.
SNAN	-Set according to bit 13 of the source operand.
NAN	-Set according to bit 14 of the source operand.
UNCC	-Set according to bit 15 of the source operand.

For destination operands other than SR:

INX	- Not affected.
DZ	- Not affected.
UNF	- Not affected.
OVF	- Not affected.
OPERR	- Not affected.
SNAN	- Not affected.
NAN	- Not affected.
UNCC	- Not affected.

IER Flags:

For destination operand SR:

SINX	-Set according to bit 16 of the source operand.
SDZ	-Set according to bit 17 of the source operand.
SUNF	-Set according to bit 18 of the source operand.
SOVF	-Set according to bit 19 of the source operand.
SIOP	-Set according to bit 20 of the source operand.

For destination operands other than SR:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	- Not affected.

Instruction Format: LEA ea,D

31			14	13			0
0000	0000	0100	0MMR	10	0000	1ddd	dddd

Instruction Format: LEA (Rn+displacement),D

31			14	13			0
0000	0000	0100	000R	00	0000	1ddd	dddd
LONG DISPLACEMENT							

Instruction Fields:

ea Rn - R0-R7 (Post-update addressing modes only)

Long Displacement - 32 bits

D	**d d d d d d d**	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 4 + le oscillator clock cycles
Memory: 1 + ea program words

LRA Load PC Relative Address LRA

Assembler Syntax:

Operation:

PC+Rn $\rightarrow$ D

LRA Rn,D

LRA label,D

Description:

The PC is added to the specified displacement and the result is stored in destination D. The PC contains the address of the next instruction. The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the destination PC. Long Displacement and Address Register PC Relative addressing modes may be used. See **Section A.10** for restrictions. Note that if D is SSH, the SP will be preincremented by one.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

For destination operand SR:

C	- Set according to bit 0 of the source operand.
V	- Set according to bit 1 of the source operand.
Z	- Set according to bit 2 of the source operand.
N	- Set according to bit 3 of the source operand.
I	- Set according to bit 4 of the source operand.
LR	- Set according to bit 5 of the source operand.
$\overline{R}$	- Set according to bit 6 of the source operand.
A	- Set according to bit 7 of the source operand.

For destination operands other than SR:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

PC+xxxx $\rightarrow$ D

ER Status Bits:

For destination operand SR:

INX -Set according to bit 8 of the source operand.
DZ -Set according to bit 9 of the source operand.
UNF -Set according to bit 10 of the source operand.
OVF -Set according to bit 11 of the source operand.
OPERR-Set according to bit 12 of the source operand.
SNAN -Set according to bit 13 of the source operand.
NAN -Set according to bit 14 of the source operand.
UNCC -Set according to bit 15 of the source operand.

For destination operands other than SR:

INX - Not affected.
DZ - Not affected.
UNF - Not affected.
OVF - Not affected.
OPERR- Not affected.
SNAN - Not affected.
NAN - Not affected.
UNCC - Not affected.

IER Flags:

For destination operand SR:

SINX -Set according to bit 16 of the source operand.
SDZ -Set according to bit 17 of the source operand.
SUNF -Set according to bit 18 of the source operand.
SOVF -Set according to bit 19 of the source operand.
SIOP -Set according to bit 20 of the source operand.

For destination operands other than SR:

SINX - Not affected.
SDZ - Not affected.
SUNF - Not affected.
SOVF - Not affected.
SIOP - Not affected.

Instruction Format: LRA Rn,D

31				14	13			0
0000	0000	0100	001R		00	0000	0ddd	dddd

Instruction Format: LRA label,D

31				14	13			0
0000	0000	0100	000R	00	00	0000	0ddd	dddd
OPTIONAL LONG DISPLACEMENT EXTENSION								

Instruction Fields:

Rn - R0-R7

Long Displacement - 32 bits

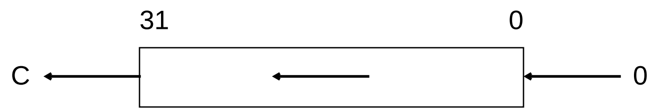
D	**d d d d d d d**	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 4 + lr oscillator clock cycles

Memory: 1 + lr program words

LSL Logical Shift Left LSL

Operation:



(parallel data bus move)

Assembler Syntax:

LSL D (move syntax - see the Move instruction description.)

LSL S,D (move syntax - see the Move instruction description.)

LSL #bits,D

Description:

Single-bit shift:

Logically shift the low portion of the specified operand one bit to the left. The carry bit receives the MSB shifted out of the low portion of the source operand. A zero is shifted into the least significant bit of the destination operand. The result is stored in the low portion of D.

Multi-bit shift:

Logically shift the low portion of the specified operand N bits (up to 63 bits) to the left. The number of bits to shift is determined by the 11-bit unsigned integer located in the 11 LSBs of the high portion of S, or by a a 6-bit immediate field in the instruction. The carry bit receives the Nth bit shifted out of the low portion of the source operand; it is cleared for a shift count of zero. N zeros are shifted into the LSBs of the destination operand. If more than 32 bits are shifted, zeros will be stored in D and the carry bit. The result is stored in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: LSL D (move syntax - see the Move instruction description.)

31	14	13			0
DATA BUS MOVE FIELD		10	0100	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA					

Instruction Format: LSL S.H,D (move syntax - see the Move instruction description.)

31	14	13			0
DATA BUS MOVE FIELD		11	0sss	0010	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA					

Instruction Format: LSL #bits,D

31				14	13			0
0000	0000	0000	0000	10	01	011n	nnnn	nddd

31				14	13			0
0000	0000	0000	0000	10	01	011n	nnnn	nddd

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

S	**s s s**	
Dn.H	n n n	where nnn = 0-7

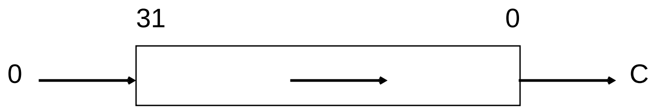
N	**n n n n n n**
0	0 0 0 0 0 0
1	0 0 0 0 0 1
2	0 0 0 0 1 0
.	.
.	. .
.	. .
62	1 1 1 1 1 0
63	1 1 1 1 1 1

Timing: 2 + mv oscillator clock cycles (2 oscillator clock cycles for LSL #shift)

Memory: 1 + mv program words (1 program word for LSL #shift)

LSR Logical Shift Right LSR

Operation:



(parallel data bus move)

Assembler Syntax:

LSR D (move syntax - see the Move instruction description.)

LSR S,D (move syntax - see the Move instruction description.)

LSR #shift,D

Description:

Single-bit shift:

Logically shift the low portion of the specified operand one bit to the right. The carry bit receives the LSB shifted out of the low portion of the source operand. A zero is shifted into bit 31 of the operand. The result is stored in the low portion of D.

Multi-bit shift:

Logically shift the low portion of the specified operand N bits (up to 63 bits) to the right. The number of bits to shift is determined by the 11-bit unsigned integer located in the 11 LSBs of the high portion of S or by a 6-bit immediate field in the instruction. The carry bit receives the Nth bit shifted out of the low portion of the source operand; it is cleared for a shift count of zero. N zeros are shifted into the MSBs of the destination operand. If more than 32 bits are shifted, zeros will be stored in D and the carry bit. The result is stored in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: LSR D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0000	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: LSR S.H,D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	0sss	0010	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: LSR #shift,D

31				14	13			0
0000	0000	0000	0000	10	01	010n	nnnn	nddd

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.H	n n n	where nnn = 0-7

N	**n n n n n n**
0	0 0 0 0 0 0
1	0 0 0 0 0 1
2	0 0 0 0 1 0
.	.
.	. .
.	. .
62	1 1 1 1 1 0
63	1 1 1 1 1 1

Timing: 2 + mv oscillator clock cycles (2 oscillator clock cycles for LSR #shift)

Memory: 1 + mv program words (1 program word for LSR #shift)

MOVE — Move Data Registers

Operation:

Parallel data bus move

Assembler Syntax:

MOVE (See the MOVE instruction description.)

Description:

Move the contents of the specified source to the specified destination. This instruction is a Data ALU NOP instruction with the parallel data move operations described in the following pages. Some parallel data move operations differentiate between integer or floating-point operands according to the kind of Data ALU operation specified. For this purpose, two Data ALU NOP opcodes are used: an "integer NOP" and a "floating-point NOP". For example, if a XY parallel move is specified with integer operands, the assembler will produce a 32-bit instruction word with the "integer NOP" in the Data ALU opcode field. If floating point XY parallel move operands are specified, the "floating-point NOP" is used instead.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: Fixed point NOP

Instruction Fields:

See the following pages for Data Bus Move Field encoding.

31 — 14	13 — 0
DATA BUS MOVE FIELD	10 0000 0000 0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Format: Floating-Point NOP

31 — 14	13 — 0
DATA BUS MOVE FIELD	10 0000 0000 0100
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

A.7.1 PARALLEL MOVE OPERATION DESCRIPTIONS

Many instructions provide the capability to specify an optional data bus movement over the X and Y Data Bus. This allows a Data ALU operation to be executed in parallel with up to two data bus moves in the same instruction cycle. Register to register, register to memory and memory to register data moves are provided. However, not all addressing modes are allowed for each memory reference type. Addressing mode restrictions which apply to specific move types are noted in the individual move operation descriptions. The following pages contain detailed information about each parallel move operation.

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

No Parallel Data Move

Operation:

Opcode Operation – none

Assembler Syntax:

Opcode-Operands

Description:

No data bus move activity.

Instruction Format: Opcode-operands

31				14	13			0
0000	0000	0110	0000	01	uu	uuuu	uuuu	uuuu

Instruction Fields:

None.

Timing: 0 oscillator clock cycles

Memory: 0 program words

Register To Register Parallel Move

Operation:

Opcode Operation	S1	$\rightarrow$	D1
Opcode Operation	S2	$\rightarrow$	D2

Assembler Syntax:

Opcode-Operands	S1,D1
Opcode-Operands	S2,D2

Description:

Move the source register to the destination register. Single precision to single precision moves (S1,D1) or double precision to double precision moves (S2,D2) may be specified.

If the opcode-operand portion of the instruction specifies as the destination a portion of the register Dn, the same register portion may not be specified as a destination D in the data bus move operation. That is, duplicate destinations are not allowed in the same instruction. For example, both a Data ALU operation and a data move operation cannot write into the same register in the same instruction.

If the opcode-operand portion of the instruction specifies as the source or destination a portion of the register Dn, the same register portion may be specified as a source S in the data bus move operation. That is, duplicate sources are allowed in the same instruction. For example, a data move operation can read the same register which is being used as a source or destination by a Data ALU operation in the same instruction.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

Single Precision Instruction Format - Opcode-operands: S1, D1

31				14	13			0
0000	10DD	DDDD	dddd	dd	uu	uuuu	uuuu	uuuu

Double Precision Instruction Format - Opcode-operands: S2, D2

31				14	13			0
0001	011D	DDDD	0ddd	dd	uu	uuuu	uuuu	uuuu

Instruction Fields:

S1 or D1	D D D D D D d d d d d d	
D0.S-D7.S	0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 1 n n n	
D0.M-D7.M	0 1 0 n n n	
D0.H-D7.H	0 1 1 n n n	
D8.S	1 0 0 0 0 0	
D9.S	1 0 0 0 0 1	
D8.L	1 0 0 0 1 0	
D9.L	1 0 0 0 1 1	
D8.M	1 0 0 1 0 0	
D9.M	1 0 0 1 0 1	
D8.H	1 0 0 1 1 0	
D9.H	1 0 0 1 11	
R0-R7	1 0 1 n n n	
N0-N7	1 1 0 n n n	
M0-M7	1 1 1 n n n	

S2 or D2	D D D D D d d d d d	
D0.ML-D7.ML	1 1 n n n	where nnn = 0-7
D0.D-D7.D	1 0 n n n	
reserved	0 1 x x x	
D9.ML	0 0 1 1 1	
D8.ML	0 0 1 1 0	
D9.D	0 0 1 0 1	
D8.D	0 0 1 0 0	

Timing: 0 oscillator clock cycles
Memory: 0 program words

Move Update (Effective Address Calculation)

Operation:

Opcode Operation ea

Assembler Syntax:

Opcode-Operands ea

Description:

The specified effective address calculation is executed. The specified address register is updated according to the addressing mode. All update addressing modes may be used. The No Update mode (Rn) is useful, in conjunction with the MOVETA instruction, to test address registers.

Instruction Format - Opcode-operands: ea

31			14	13			0
0001	0101	1011	MMMR	uu	uuuu	uuuu	uuuu

Instruction Fields:

ea Rn - R0-R7 (Update addressing modes only)

Timing: 0 oscillator clock cycles
Memory: 0 program words

X Memory Move

Operation:

X:<ea> → D
X:<Rn+xxxx> → D
S → X:<ea>
S → X:<Rn+xxxx>
#xxxx → D

Assembler Syntax:

X: ea, D
X:(Rn+displacement),D
S,X: ea
S,X:(Rn+displacement)
#Data,D

Description:

Move one word operand to/from X memory. One effective address is specified. All memory addressing modes, including absolute address and immediate data, may be used. Long displacement addressing may also be used. A memory to register or register to memory direction may be specified.

If the opcode-operand portion of the instruction specifies as the destination a portion of the register Dn, the same register portion may not be specified as a destination D in the data bus move operation. That is, duplicate destinations are not allowed in the same instruction. For example, both a Data ALU operation and a data move operation cannot write into the same register in the same instruction.

If the opcode-operand portion of the instruction specifies as the source or destination a portion of the register Dn, the same register portion may be specified as a source S in the data bus move operation. That is, duplicate sources are allowed in the same instruction. For example, a data move operation can read the same register which is being used as a source or destination by a Data ALU operation in the same instruction.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

Instruction Format - Opcode-operands: S,X: ea
X: ea, D
#Data,D

31			14	13			0
0011	W0DD	DDDD	MMMR	uu	uuuu	uuuu	uuuu
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA							

Instruction Format - Opcode-operands: S,X:(Rn+displacement)
X:(Rn+displacement),D

31			14	13			0
0000	11DD	DDDD	0W1R	uu	uuuu	uuuu	uuuu
LONG DISPLACEMENT							

Instruction Fields: <ea> Rn - R0-R7 (Memory addressing modes only)

Register	W
Read S	0
Write D	1

S1 or D1	D D D D D D d d d d d d	
D0.S-D7.S	0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 1 n n n	
D0.M-D7.M	0 1 0 n n n	
D0.H-D7.H	0 1 1 n n n	
D8.S	1 0 0 0 0 0	
D9.S	1 0 0 0 0 1	
D8.L	1 0 0 0 1 0	
D9.L	1 0 0 0 1 1	
D8.M	1 0 0 1 0 0	
D9.M	1 0 0 1 0 1	
D8.H	1 0 0 1 1 0	
D9.H	1 0 0 1 1 1	
R0-R7	1 0 1 n n n	
N0-N7	1 1 0 n n n	
M0-M7	1 1 1 n n n	

Timing: ea + ax oscillator clock cycles
Memory: ea program words

X Memory and Register Move

Move X: R

Operation:		**Assembler Syntax:**	
X:<ea> → D1	S2 → D2	X: ea, D1	S2,D2
S1 → X:<ea>	S2 → D2	S1,X: ea	S2,D2
#xxxx → D1	S2 → D2	#Data,D1	S2,D2

Description:

Move one word operand to/from X memory and one word operand from register to register. One effective address is specified. A memory to register or register to memory direction may be specified in the effective address.

When two parallel data move operations are specified in the same instruction, certain restrictions apply. If the instruction has an integer opcode, both data moves must be integer moves and specify integer operands. If the instruction has a floating-point opcode, both data moves must be floating-point moves and specify floating-point operands.

If the opcode-operand portion of the instruction specifies as the destination a portion of the register Dn, the same register portion may not be specified as a destination D in the data bus move operation. That is, duplicate destinations are not allowed in the same instruction. For example, both a Data ALU operation and a data move operation cannot write into the same register in the same instruction.

If the opcode-operand portion of the instruction specifies as the source or destination a portion of the register Dn, the same register portion may be specified as a source S in the data bus move operation. That is, duplicate sources are allowed in the same instruction. For example, a data move operation can read the same register which is being used as a source or destination by a Data ALU operation in the same instruction.

Instruction Format - Opcode-operands: X: ea, D1S2,D2

S1,X: ea S2,D2

#Data,D1S2,D2

31			14	13			0
010d	WdYY	YXXX	MMMR	uu	uuuu	uuuu	uuuu
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA							

Instruction Fields: <ea> Rn - R0-R7 (Memory addressing modes only)

Register	**W**
Read S1	0
Write D1	1

Integer Opcodes | **Floating-Point Opcodes**

S1,D1	**X X X**	**S1,D1**	**X X X**	
D0.L-D7.L	n n n	D0.S-D7.S	n n n	where nnn = 0-7

S2	**d d**	**D2**	**Y Y Y**	**S2**	**d d**	**D2**	**Y Y Y**
D4.L	0 0	D0.L	0 0 0	D4.S	0 0	D0.S	0 0 0
D5.L	0 1	D1.L	0 0 1	D5.S	0 1	D1.S	0 0 1
D6.L	1 0	D2.L	0 1 0	D6.S	1 0	D2.S	0 1 0
D7.L	1 1	D3.L	0 1 1	D7.S	1 1	D3.S	0 1 1
D0.L	0 0	D4.L	1 0 0	D0.S	0 0	D4.S	1 0 0
D1.L	0 1	D5.L	1 0 1	D1.S	0 1	D5.S	1 0 1
D2.L	1 0	D6.L	1 1 0	D2.S	1 0	D6.S	1 1 0
D3.L	1 1	D7.L	1 1 1	D3.S	1 1	D7.S	1 1 1

Timing: ea + ax oscillator clock cycles
Memory: ea program words

Move Y:

Y Memory Move

Move Y:

Operation:

Opcode Operation	Y:<ea> → D
Opcode Operation	Y:<Rn+xxxx> → D
Opcode Operation	S → Y:<ea>
Opcode Operation	S → Y:<Rn+xxxx>
Opcode Operation	#xxxx → D

Assembler Syntax:

Opcode-Operands	Y: ea, D
Opcode-Operands	Y:(Rn+displacement),D
Opcode-Operands	S,Y: ea
Opcode-Operands	S,Y:(Rn+displacement)
Opcode-Operands	#Data,D

Description:

Move one word operand to/from Y memory. One effective address is specified. All memory addressing modes, including absolute address and immediate data, may be used. Long displacement addressing may also be used. A memory to register or register to memory direction may be specified.

If the opcode-operand portion of the instruction specifies as the destination a portion of the register Dn, the same register portion may not be specified as a destination D in the data bus move operation. That is, duplicate destinations are not allowed in the same instruction. For example, both a Data ALU operation and a data move operation cannot write into the same register in the same instruction.

If the opcode-operand portion of the instruction specifies as the source or destination a portion of the register Dn, the same register portion may be specified as a source S in the data bus move operation. That is, duplicate sources are allowed in the same instruction. For example, a data move operation can read the same register which is being used as a source or destination by a Data ALU operation in the same instruction.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

Instruction Fields: <ea> Rn - R0-R7 (Memory addressing modes only)

Instruction Format - Opcode-operands: S,Y: ea
Y: ea, D
#Data,D

31			14	13			0
0011	W1DD	DDDD	MMMR	uu	uuuu	uuuu	uuuu
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA							

Instruction Format - Opcode-operands: S,Y:(Rn+displacement)
Y:(Rn+displacement),D

31			14	13			0
0000	11DD	DDDD	1W1R	uu	uuuu	uuuu	uuuu
LONG DISPLACEMENT							

Register	**W**
Read S	0
Write D	1

S1 or	**D D D D D D**	
D1	d d d d d d	
D0.S-D7.S	0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 1 n n n	
D0.M-D7.M	0 1 0 n n n	
D0.H-D7.H	0 1 1 n n n	
D8.S	1 0 0 0 0 0	
D9.S	1 0 0 0 0 1	
D8.L	1 0 0 0 1 0	
D9.L	1 0 0 0 1 1	
D8.M	1 0 0 1 0 0	
D9.M	1 0 0 1 0 1	
D8.H	1 0 0 1 1 0	
D9.H	1 0 0 1 11	
R0-R7	1 0 1 n n n	
N0-N7	1 1 0 n n n	
M0-M7	1 1 1 n n n	

Timing: ea + ay oscillator clock cycles
Memory: ea program words

Y Memory and Register Move

Operation:

Opcode Operation S1 → D1 Y:<ea> → D2

Opcode Operation S1 → D1 S2 → Y:<ea>

Opcode Operation S1 → D1 #xxxx → D2

Assembler Syntax:

Opcode-Operands S1,D1 Y: ea, D2

Opcode-Operands S1,D1 S2,Y: ea

Opcode-Operands S1,D1 #Data,D2

Description:

Move one word operand to/from Y memory and one word operand from register to register. One effective address is specified. A memory to register or register to memory direction may be specified in the effective address.

When two parallel data move operations are specified in the same instruction, certain restrictions apply. If the instruction has an integer opcode, both data moves must be integer moves and specify integer operands. If the instruction has a floating-point opcode, both data moves must be floating-point moves and specify floating-point operands.

If the opcode-operand portion of the instruction specifies as the destination a portion of the register Dn, the same register portion may not be specified as a destination D in the data bus move operation. That is, duplicate destinations are not allowed in the same instruction. For example, both a Data ALU operation and a data move operation cannot write into the same register in the same instruction.

If the opcode-operand portion of the instruction specifies as the source or destination a portion of the register Dn, the same register portion may be specified as a source S in the data bus move operation. That is, duplicate sources are allowed in the same instruction. For example, a data move operation can read the same register which is being used as a source or destination by a Data ALU operation in the same instruction.

Instruction Fields: <ea> Rn - R0-R7 (Memory addressing modes only)

Instruction Format - Opcode-operands: S1,D1 Y: ea, D2

S1,D1 S2,Y: ea

S1,D1 #Data,D2

31			14	13			0
011d	WdYY	YXXX	MMMR	uu	uuuu	uuuu	uuuu
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA							

Register	W
Read S2	0
Write D2	1

Integer Opcodes

S2,D2	Y Y Y
D0.L-D7.L	n n n

S1	d d	D1	X X X
D4.L	0 0	D0.L	0 0 0
D5.L	0 1	D1.L	0 0 1
D6.L	1 0	D2.L	0 1 0
D7.L	1 1	D3.L	0 1 1
D0.L	0 0	D4.L	1 0 0
D1.L	0 1	D5.L	1 0 1
D2.L	1 0	D6.L	1 1 0
D3.L	1 1	D7.L	1 1 1

Floating-Point Opcodes

S2,D2	Y Y Y	
D0.S-D7.S	n n n	where nnn = 0-7

S1	d d	D1	X X X
D4.S	0 0	D0.S	0 0 0
D5.S	0 1	D1.S	0 0 1
D6.S	1 0	D2.S	0 1 0
D7.S	1 1	D3.S	0 1 1
D0.S	0 0	D4.S	1 0 0
D1.S	0 1	D5.S	1 0 1
D2.S	1 0	D6.S	1 1 0
D3.S	1 1	D7.S	1 1 1

Timing: ea + ay oscillator clock cycles
Memory: ea program words

Move L:	Long Memory Move	Move L:

Operation:

X:<ea> → D(MS) Y:<ea> → D(LS)
X:<Rn+xxxx> → D(MS) Y:<Rn+xxxx> → D(LS)
S(MS) → X:<ea> S(LS) → Y:<ea>
S(MS) → X:<Rn+xxxx> S(LS) → Y:<Rn+xxxx>

Assembler Syntax:

L: ea, D
L:(Rn+displacement),D
S,L: ea
S,L:(Rn+displacement)

Description:

This instruction allows long word operand data moves to/from one effective address in L (X:Y) memory. The long word operand is a long integer for integer moves and a double precision IEEE data type for floating-point moves. One effective address is specified. All memory alterable addressing modes may be used. Long displacement addressing may also be used. A memory to register or register to memory direction may be specified.

If the opcode-operand portion of the instruction specifies as the destination a portion of the register Dn, the same register portion may not be specified as a destination D in the data bus move operation. That is, duplicate destinations are not allowed in the same instruction. For example, both a Data ALU operation and a data move operation cannot write into the same register in the same instruction.

If the opcode-operand portion of the instruction specifies as the source or destination a portion of the register Dn, the same register portion may be specified as a source S in the data bus move operation. That is, duplicate sources are allowed in the same instruction. For example, a data move operation can read the same register which is being used as a source or destination by a Data ALU operation in the same instruction.

Instruction Format - Opcode-operands: L: ea, D
S,L: ea

31			14	13			0
0010	01WD	DDDD	MMMR	uu	uuuu	uuuu	uuuu
OPTIONAL EFFECTIVE ADDRESS EXTENSION							

Instruction Format - Opcode-operands: L:(Rn+displacement),D
S,L:(Rn+displacement)

31			14	13			0
0000	111D	DDDD	0W0R	uu	uuuu	uuuu	uuuu
LONG DISPLACEMENT							

Instruction Fields: <ea>Rn - R0-R7 (Memory alterable addressing modes only)

Register	**W**
Read S	0
Write D	1

S2 or D2	**DDDDD ddddd**
D0.ML-D7.ML	1 1 n n n where nnn = 0-7
D0.D-D7.D	1 0 n n n
D9.ML	0 0 1 1 1
D8.ML	0 0 1 1 0
D9.D	0 0 1 0 1
D8.D	0 0 1 0 0

Timing: ea + axy oscillator clock cycles
Memory: ea program words

Move X: Y: — XY Memory — Move X: Y:

Operation:		Assembler Syntax:	
X:<ea> → D1	Y:<ea> → D2	X: ea, D1	Y: ea, D2
X:<ea> → D1	S2 → Y:<ea>	X: ea, D1	S2,Y: ea
S1 → X:<ea>	Y:<ea> → D2	S1,X: ea	Y: ea, D2
S1 → X:<ea>	S2 → Y:<ea>	S1,X: ea	S2,Y: ea
X:<ea> → D1	Y:<> → D2	X: ea, D1	Y:,D2
S1 → X:<ea>	S2 → Y:<>	S1,X: ea	S2,Y:
X:<Rn+xxxx> → D1	Y:<> → D2	X:(Rn+displacement),D1	Y:,D2
S1 → X:<Rn+xxxx>	S2 → Y:<>	S1,X:(Rn+displacement)	S2,Y:

Description:

Move two word operands to/from X and Y memory. All word operands are integer for integer moves and single precision IEEE data type for floating-point moves. They may represent a complex (real:imaginary) data pair, a data:coefficient data pair or two independent data words. One or two independent effective addresses may be specified. If one effective address is specified, all memory alterable addressing modes and long displacement may be used; both data moves have the same memory to register or register to memory direction. If two effective addresses are specified, all parallel addressing modes may be used and each data move may have a memory to register or register to memory direction.

When two parallel data move operations are specified in the same instruction, certain restrictions apply. If the instruction has an integer opcode, both data moves must be integer moves and specify integer operands. If the instruction has a floating-point opcode, both data moves must be floating-point moves and specify floating-point operands.

If the opcode-operand portion of the instruction specifies as the destination a portion of the register Dn, the same register portion may not be specified as a destination D in the data bus move operation. That is, duplicate destinations are not allowed in the same instruction. For example, both a Data ALU operation and a data move operation cannot write into the same register in the same instruction.

If the opcode-operand portion of the instruction specifies as the source or destination a portion of the register Dn, the same register portion may be specified as a source S in the data bus move operation. That is, duplicate sources are allowed in the same instruction. For example, a data move operation can read the same register which is being used as a source or destination by a Data ALU operation in the same instruction.

Instruction Format - Opcode-operands: X: ea, D1 Y: ea, D2
X: ea, D1 S2,Y: ea
S1,X: ea Y: ea, D2
S1,X: ea S2,Y: ea

31			14	13			0
1mmw	WrYY	YXXX	rMMR	uu	uuuu	uuuu	uuuu

Instruction Format - Opcode-operands: X: ea, D1 Y:,D2
S1,X: ea S2,Y:

31			14	13			0
0010	1WYY	YXXX	MMMR	uu	uuuu	uuuu	uuuu

Instruction Format - Opcode-operands: X: ea, D1(8,9) Y:,D2(8,9)

31			14	13			0
0001	010W	Y11X	MMMR	uu	uuuu	uuuu	uuuu

Instruction Format - Opcode-operands: X:(Rn+displacement),D1 Y:,D2
S1,X:(Rn+displacement) S2,Y:

31			14	13			0
0000	11YY	YXXX	1W0R	uu	uuuu	uuuu	uuuu
LONG DISPLACEMENT							

Instruction Format - Opcode-operands: X:(Rn+displacement),D1 Y:,D2(8,9)
S1,X:(Rn+displacement) S2(8,9),Y:

31			14	13			0
0000	1101	Y11X	0W0R	uu	uuuu	uuuu	uuuu
LONG DISPLACEMENT							

Instruction Fields:

For two independent effective addresses:
X: ea Rn - R0,R1,R2,R3 (Parallel addressing modes only)
Y: ea Rn - R4,R5,R6,R7
or
X: ea Rn - R4,R5,R6,R7 (Parallel addressing modes only)
Y: ea Rn - R0,R1,R2,R3

Register	**W**
Read S1	0
Write D1	1

Register	**w**
Read S2	0
Write D2	1

Effective Address		
X: ea	MM	R R R
Y: ea	mm	r r

Integer Opcodes

S1,D1	**X X X**
D0.L-D7.L	n n n

S2,D2	**Y Y Y**
D0.L-D7.L	n n n

Floating-Point Opcodes

S1,D1	**X X X**	
D0.S-D7.S	n n n	where nnn = 0-7

S2,D2	**Y Y Y**	
D0.S-D7.S	n n n	where nnn = 0-7

For a single effective address:

Register	**W**
Read S1,S2	0
Write D1,D2	1

Effective Address

X: ea = Y: ea	MMM RRR	(Memory alterable addressing modes only)
X: ea = Y: ea	RRR	(Long displacement addressing mode)

Integer Opcodes		**Floating-Point Opcodes**		
S1,D1	**X X X**	**S1,D1**	**X X X**	
D0.L-D7.L	n n n	D0.7.S	n n n	where nnn = 0-7
S2,D2	**Y Y Y**	**S2,D2**	**Y Y Y**	
D0.7-D7.L	n n n	D0.S-D7.S	n n n	
S1,D1	**X**	**S1,D1**	**X**	
D8.L	0	D8.S	0	
D9.L	1	D9.S	0	
S2,D2	**Y**	**S2,D2**	**Y**	
D8.L	0	D8.S	0	
D9.L	1	D9.S	0	

Timing: axy oscillator clock cycles
Memory: program words

Floating-Point iF Conditional Instruction without CCR, ER, IER update

Operation:

If cc, then

Opcode Operation   S $\rightarrow$ D

Assembler Syntax:

Opcode-Operands   S,D   FFcc

Opcode-Operands   FFcc

Description:

If the specified floating-point condition is true, transfer data from the specified source S to the specified destination D. Also, store result(s) of the specified Data ALU operation. If the specified condition is false, no destinations are altered. The CCR and ER registers are not updated with the condition codes generated by the Data ALU operation. The UNCC bit in the ER register and SIOP flag in the IER are set by the FFcc instruction if the NAN bit in the ER register was set and the specified condition is one of the conditions with a "Yes" entry in the "Set UNCC" column. If no register move is specified, this instruction is assembled with a R0 to R0 move.

"cc" may specify the following conditions:

Mnemonic		Condition	Non-aware Set UNCC*
EQ	- equal	Z = 1	No
ERR	- error	UNCC v SNAN v OPERR v OVF v UNF v DZ = 1	No
GE	- greater than or equal	NAN v (N & ~Z) = 0	Yes
GL	- greater or less than	NAN v Z = 0	Yes
GLE	- greater, less or equal	NAN = 0	Yes
GT	- greater than	NAN v Z v N = 0	Yes
INF	- infinity	I = 1	Yes
LE	- less than or equal	NAN v ~(N v Z) = 0	Yes
LT	- less than	NAN v Z v ~N = 0	Yes
MI	- minus	N = 1	No
NE(Q)	- not equal	Z = 0	No
NGE	- not(greater than or equal)	NAN v (N & ~Z) = 1	Yes
NGL	- not(greater or less than)	NAN v Z = 1	Yes
NGLE	- not(greater, less or equal)	NAN = 1	Yes
NGT	- not greater than	NAN v Z v N = 1	Yes
NINF	- not infinity	I = 0	Yes
NLE	- not(less than or equal)	NAN v ~(N v Z) = 1	Yes
NLT	- not less than	NAN v Z v ~N = 1	Yes
OR	- ordered	NAN = 0	No
PL	- plus	N = 0	No
UN	- unordered	NAN = 1	No

Note: The operands for the ERR condition are taken from the ER register.

* See description of UNcc bit in **Section A.4**.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	- Not affected.
DZ	- Not affected.
UNF	- Not affected.
OVFD	- Not affected.
OPERR	- Not affected.
SNAN	- Not affected.
NAN	- Not affected.
UNCC	- Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

IER Flags:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	- Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

Instruction Format - Opcode-operands: S,D FFcc
FFcc

31			14	13			0
0000	011c	cccc	tttT	uu	uuuu	uuuu	uuuu

Instruction Fields:

S	**t t t**	
Rn	n n n	where nnn = 0-7

D	**T T T**	
Rn	n n n	where nnn = 0-7

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
GT	0 0 0 0 0	NGT	1 0 0 0 0
LT	0 0 0 0 1	NLT	1 0 0 0 1
GE	0 0 0 1 0	NGE	1 0 0 1 0
LE	0 0 0 1 1	NLE	1 0 0 1 1
GL	0 0 1 0 0	NGL	1 0 1 0 0
INF	0 0 1 0 1	NINF	1 0 1 0 1
GLE	0 0 1 1 0	NGLE	1 0 1 1 0
OR	0 0 1 1 1	UN	1 0 1 1 1
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
ERR	0 1 1 1 1		

Timing: 2 + da oscillator clock cycles
Memory: 1 program words

Floating-Point iF Conditional Instruction with CCR, ER, IER Update

Operation:

If cc, then opcode operation

S $\rightarrow$ D

Assembler Syntax:

Opcode-Operands S,D FFcc.U

FFcc.U

Description:

If the specified floating-point condition is true, transfer data from the specified source S to the specified destination D. Also, store result(s) of the specified Data ALU operation and update the CCR, ER and IER registers with the status information generated by the Data ALU operation. If the specified condition is false, no destinations are altered and the status register is not affected by the Data ALU operation. The UNCC bit in the ER register and SIOP flag in the IER are set by the FFcc.U instruction if the NAN bit in the ER register was set and the specified condition is one of the conditions with a "Yes" entry in the "Set UN-CC" column. If no register move is specified, this instruction is assembled with a R0 to R0 move.

"cc" may specify the following conditions:

Mnemonic		**Condition**	**Non-aware Set UNCC***
EQ	- equal	Z = 1	No
ERR	- error	UNCC v SNAN v OPERR v OVF v UNF v DZ = 1	No
GE	- greater than or equal	NAN v (N & ~Z) = 0	Yes
GL	- greater or less than	NAN v Z = 0	Yes
GLE	- greater, less or equal	NAN = 0	Yes
GT	- greater than	NAN v Z v N = 0	Yes
INF	- infinity	I = 1	Yes
LE	- less than or equal	NAN v ~(N v Z) = 0	Yes
LT	- less than	NAN v Z v ~N = 0	Yes
MI	- minus	N = 1	No
NE(Q)	- not equal	Z = 0	No
NGE	- not(greater than or equal)	NAN v (N & ~Z) = 1	Yes
NGL	- not(greater or less than)	NAN v Z = 1	Yes
NGLE	- not(greater, less or equal)	NAN = 1	Yes
NGT	- not greater than	NAN v Z v N = 1	Yes
NINF	- not infinity	I = 0	Yes
NLE	- not(less than or equal)	NAN v ~(N v Z) = 1	Yes
NLT	- not less than	NAN v Z v ~N = 1	Yes
OR	- ordered	NAN = 0	No
PL	- plus	N = 0	No
UN	- unordered	NAN = 1	No

Note: The operands for the ERR condition are taken from the ER register.

* See description of UNcc bit in **Section A.4**.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

C - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

V - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

Z - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

N - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

I - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

LR - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

R - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

A - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

ER Status Bits:

INX - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

DZ - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

UNF - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

OVF - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

OPERR - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

SNAN - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

NAN - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

UNCC - Set if NAN is set and a non-aware floating-point condition is tested ("cc" conditions marked "YES" above). Not affected otherwise.

IER Flags: Flags changed according to standard definition.

Instruction Format - Opcode-operands: S,D FFcc.U

FFcc.U

31				14	13			0
0000	011c	cccc	tttT	TT	uu	uuuu	uuuu	uuuu

Instruction Fields:

S	**t t t**	
Rn	n n n	where nnn = 0-7

D	**T T T**	
Rn	nn n	where nnn = 0-7

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
GT	0 0 0 0 0	NGT	1 0 0 0 0
LT	0 0 0 0 1	NLT	1 0 0 0 1
GE	0 0 0 1 0	NGE	1 0 0 1 0
LE	0 0 0 1 1	NLE	1 0 0 1 1
GL	0 0 1 0 0	NGL	1 0 1 0 0
INF	0 0 1 0 1	NINF	1 0 1 0 1
GLE	0 0 1 1 0	NGLE	1 0 1 1 0
OR	0 0 1 1 1	UN	1 0 1 1 1
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
ERR	0 1 1 1 1		

Timing: 2 + da oscillator clock cycles
Memory: 1 program words

Integer iF Conditional Instruction without CCR Update

Move IFcc

Operation:

If cc, then opcode operation

S $\rightarrow$ D

Assembler Syntax:

Opcode-Operands	S,D	IFcc
		IFcc

Description:

If the specified integer condition is true, transfer data from the specified source S to the specified destination D. Also, store result(s) of the specified Data ALU operation. If the specified condition is false, no destinations are altered. The CCR, ER and IER registers are never updated with the condition codes generated by the Data ALU operation. If no register move is specified, this instruction is assembled with a R0 to R0 move.

"cc" may specify the following conditions:

Mnemonic		**Condition**
CC (HS)	- carry clear (higher or same)	C = 0
CS (LO)	- carry set (lower)	C = 1
EQ	- equal	Z = 1
GE	- greater or equal	N && V = 0
GT	- greater than	Z v (N && V) = 0
HI	- higher	Z v C = 0
LE	- less or equal	Z v (N && V) = 1
LS	- lower or same	Z v C = 1
LT	- less than	N && V = 1
MI	- minus	N = 1
NE(Q)	- not equal	Z = 0
PL	- plus	N = 0
VC	- overflow clear	V = 0
VS	- overflow set	V = 1
AL	- always true	n.a.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

INX	- Not affected.
DZ	- Not affected.
UNF	- Not affected.
OVFD	- Not affected.
OPERR	- Not affected.
SNAN	- Not affected.
NAN	- Not affected.
UNCC	- Not affected.

IER Flags:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	- Not affected.

Instruction Format - Opcode-operands: S,D IFcc

IFcc

31				14	13			0
0000	011c	cccc	tttT	TT	uu	uuuu	uuuu	uuuu

Instruction Fields:

S	**t t t**	
Rn	n n n	where nnn = 0-7

D	**T T T**	
Rn	n n n	where nnn = 0-7

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
CC(HS)	0 1 0 1 0	CS(LO)	1 1 0 1 0
GE	0 1 0 1 1	LT	1 1 0 1 1
GT	0 1 1 0 0	LE	1 1 1 0 0
VC	0 1 1 0 1	VS	1 1 1 0 1
HI	0 1 1 1 0	LS	1 1 1 1 0
AL	1 1 1 1 1		

Timing: 2 + da oscillator clock cycles
Memory: 1 program words

Integer iF Conditional Instruction with CCR, ER, and IER Update

Operation:

If cc, then opcode operation

S $\rightarrow$ D

Assembler Syntax:

Opcode-Operands	S,D	IFcc.U
		IFcc.U

Description:

If the specified integer condition is true, transfer data from the specified source S to the specified destination D. Also, store result(s) of the specified Data ALU operation and update the CCR, ER and IER registers with the status information generated by the Data ALU operation. If the specified condition is false, no destinations are altered and the status register is not affected. The UNCC bit in the ER register is never updated by the Data ALU operation. If no register move is specified, this instruction is assembled with a R0 to R0 move.

"cc" may specify the following conditions:

Mnemonic		Condition
CC (HS)	- carry clear (higher or same)	C = 0
CS (LO)	- carry set (lower)	C = 1
EQ	- equal	Z = 1
GE	- greater or equal	N && V = 0
GT	- greater than	Z v (N && V) = 0
HI	- higher	Z v C = 0
LE	- less or equal	Z v (N && V) = 1
LS	- lower or same	Z v C = 1
LT	- less than	N && V = 1
MI	- minus	N = 1
NE(Q)	- not equal	Z = 0
PL	- plus	N = 0
VC	- overflow clear	V = 0
VS	- overflow set	V = 1
AL	- always true	n.a.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

C - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

V - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

Z - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

N - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

I - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

LR - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

$\overline{R}$ - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

A - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

ER Status Bits:

INX - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

DZ - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

UNF - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

OVF - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

OPERR - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

SNAN - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

NAN - Affected by the accompanying Data ALU operation if the specified condition is true. Not affected otherwise.

UNCC - Not affected.

IER Flags: Flags changed according to standard definition.

Instruction Format - Opcode-operands: S,D IFcc.U

IFcc.U

31				14	13			0
0000	010c	cccc	tttT	TT	uu	uuuu	uuuu	uuuu

Instruction Fields:

S	**t t t**	
Rn	n n n	where nnn = 0-7

D	**T T T**	
Rn	n n n	where nnn = 0-7

Mnemonic	**c c c c c**	**Mnemonic**	**c c c c c**
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
CC(HS)	0 1 0 1 0	CS(LO)	1 1 0 1 0
GE	0 1 0 1 1	LT	1 1 0 1 1
GT	0 1 1 0 0	LE	1 1 1 0 0
VC	0 1 1 0 1	VS	1 1 1 0 1
HI	0 1 1 1 0	LS	1 1 1 1 0
AL	1 1 1 1 1		

Timing: 2 + da oscillator clock cycles
Memory: 1 program words

Operation:

S3 $\rightarrow$ D2	MOVE(C) S3,D2
S2 $\rightarrow$ D1	MOVE(C) S2,D1
#xxxx $\rightarrow$ D1	MOVE(C) #Data,D1
X:<ea> $\rightarrow$ D1	MOVE(C) X: ea, D1
X:<Rn+xxxx> $\rightarrow$ D1	MOVE(C) X:(Rn+displacement),D1
S1 $\rightarrow$ X:<ea>	MOVE(C) S1,X: ea
S1 $\rightarrow$ X:<Rn+xxxx>	MOVE(C) S1,X:(Rn+displacement)
Y:<ea> $\rightarrow$ D1	MOVE(C) Y: ea, D1
Y:<Rn+xxxx> $\rightarrow$ D1	MOVE(C) Y:(Rn+displacement),D1
S1 $\rightarrow$ Y:<ea>	MOVE(C) S1,Y: ea
S1 $\rightarrow$ Y:<Rn+xxxx>	MOVE(C) S1,Y:(Rn+displacement)

Description:

Assembler Syntax:

Move the contents of the specified control register to the specified destination or move the specified source to the specified control register. The control registers S1, S3, and D1 are the program controller registers and may be moved to or from any other register or memory space. All operands are word operands. All memory addressing modes plus Long Displacement addressing may be used.

If the system stack register SSH is specified as a source operand, the system stack pointer SP is postdecremented by 1 after SSH is read. If the system stack register SSH is specified as a destination operand, the system stack pointer SP is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

See **Section A.10** for restrictions that apply to this instruction.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

For destination operand SR:

C - Set according to bit 0 of the source operand.
V - Set according to bit 1 of the source operand.
Z - Set according to bit 2 of the source operand.
N - Set according to bit 3 of the source operand.
I - Set according to bit 4 of the source operand.
LR - Set according to bit 5 of the source operand.
$\overline{\text{R}}$ - Set according to bit 6 of the source operand.
A - Set according to bit 7 of the source operand.

For destination operands other than SR:

C - Not affected.
V - Not affected.
Z - Not affected.
N - Not affected.
I - Not affected.
LR - Not affected.
$\overline{\text{R}}$ - Not affected.
A - Not affected.

ER Status Bits:

For destination operand SR:

INX -Set according to bit 8 of the source operand.
DZ -Set according to bit 9 of the source operand.
UNF -Set according to bit 10 of the source operand.
OVF -Set according to bit 11 of the source operand.
OPERR-Set according to bit 12 of the source operand.
SNAN -Set according to bit 13 of the source operand.
NAN -Set according to bit 14 of the source operand.
UNCC -Set according to bit 15 of the source operand.

For destination operands other than SR:

INX - Not affected.
DZ - Not affected.
UNF - Not affected.
OVF - Not affected.
OPERR- Not affected.
SNAN - Not affected.
NAN - Not affected.
UNCC - Not affected.

IER Flags:

For destination operand SR:

SINX	-Set according to bit 16 of the source operand.
SDZ	-Set according to bit 17 of the source operand.
SUNF	-Set according to bit 18 of the source operand.
SOVF	-Set according to bit 19 of the source operand.
SIOP	-Set according to bit 20 of the source operand.

For destination operands other than SR:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	- Not affected.

Instruction Format: MOVE(C) X: ea, D1   MOVE(C) S1,X: ea
MOVE(C) Y: ea, D1   MOVE(C) S1,Y: ea

31				14	13			0
0000	0001	0011	MMMR	RR	1W	s001	0ddd	dddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Format: MOVE(C) X:(Rn+displacement),D1   MOVE(C) S1,X:(Rn+displacement)
MOVE(C) X:(Rn+displacement),D1   MOVE(C) S1,X:(Rn+displacement)

31				14	13			0
0000	0001	0011	xxxR	RR	0W	s001	0ddd	dddd
LONG DISPLACEMENT								

Instruction Format: MOVE(C) S1,D2
MOVE(C) S2,D1

31				14	13			0
0000	0001	0010	DDDD	DD	D0	0001	0ddd	dddd

Instruction Fields:

<ea> Rn - R0-R7 (Memory addressing modes only)

Immediate data - 32 bits

Absolute Address - 32 bits

Long Displacement - 32 bits

Memory Space	**S**
X Memory	0
Y Memory	1

Register	**W**
Read S	0
Write D	1

S3 **S1, D1**	**D D D D D D D** **d d d d d d d**
SR	1 1 1 1 0 0 1
OMR	1 1 1 1 0 1 0
SP	1 1 1 1 0 1 1
SSH	1 1 1 1 1 0 0
SSL	1 1 1 1 1 0 1
LA	1 1 1 1 1 1 0
LC	1 1 1 1 1 1 1

S2 **D2**	**D D D D D D D** **d d d d d d d**	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 2 + mvc oscillator clock cycles

Memory: 1 + ea program words

MOVE(I) Immediate Short Data Move MOVE(I)

Operation:

#xx → D

Assembler Syntax:

MOVE(I) #Data,D

Description:

The 16-bit immediate short operand is sign extended to a word operand and is stored in the destination register D. Care should be taken if the specified destination register is D0.S-D9.S, since there is no special formatting for short floating-point data and the sign extended immediate short operand may produce small positive denormalized numbers or a negative NANs. See **Section A.10** for restrictions that apply to this instruction. Note that if D is SSM, the SP will be preincremented by one.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

For destination operand SR:

C	- Set according to bit 0 of the source operand.
V	- Set according to bit 1 of the source operand.
Z	- Set according to bit 2 of the source operand.
N	- Set according to bit 3 of the source operand.
I	- Set according to bit 4 of the source operand.
LR	- Set according to bit 5 of the source operand.
$\overline{R}$	- Set according to bit 6 of the source operand.
A	- Set according to bit 7 of the source operand.

For destination operands other than SR:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

For destination operand SR:

INX	-Set according to bit 8 of the source operand.
DZ	-Set according to bit 9 of the source operand.
UNF	-Set according to bit 10 of the source operand.
OVF	-Set according to bit 11 of the source operand.
OPERR	-Set according to bit 12 of the source operand.
SNAN	-Set according to bit 13 of the source operand.
NAN	-Set according to bit 14 of the source operand.
UNCC	-Set according to bit 15 of the source operand.

For destination operands other than SR:

INX	- Not affected.
DZ	- Not affected.
UNF	- Not affected.
OVF	- Not affected.
OPERR	- Not affected.
SNAN	- Not affected.
NAN	- Not affected.
UNCC	- Not affected.

IER Flags:

For destination operand SR:

SINX	-Set according to bit 16 of the source operand.
SDZ	-Set according to bit 17 of the source operand.
SUNF	-Set according to bit 18 of the source operand.
SOVF	-Set according to bit 19 of the source operand.
SIOP	-Set according to bit 20 of the source operand.

For destination operands other than SR:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	- Not affected.

Instruction Format: MOVE(I) #Data,D

31				14	13			0
0000	0000	1 i i i	i i i i	i i	i i	i i i i	i ddd	dddd

Instruction Fields:

Immediate Short Data - iiiiiiiiiiiiiiii (16 bits)

D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 2 oscillator clock cycles

Memory: 1 program words

MOVE(M) Move Program Memory MOVE(M)

Operation:	Assembler Syntax:
P:<ea> → D	MOVE(M) P: ea, D
S → P:<ea>	MOVE(M) S,P: ea

Description:

Move the specified program memory word operand to the specified destination register or move the specified source register to the specified program memory location. The registers S and D may be any register. All memory alterable addressing modes may be used.

If the system stack register SSH is specified as a source operand, the system stack pointer SP is postdecremented by 1 after SSH is read. If the system stack register SSH is specified as a destination operand, the system stack pointer SP is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

See **Section A.10** for restrictions that apply to this instruction.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

For destination operand SR:

C	- Set according to bit 0 of the source operand.
V	- Set according to bit 1 of the source operand.
Z	- Set according to bit 2 of the source operand.
N	- Set according to bit 3 of the source operand.
I	- Set according to bit 4 of the source operand.
LR	- Set according to bit 5 of the source operand.
$\overline{R}$	- Set according to bit 6 of the source operand.
A	- Set according to bit 7 of the source operand.

For destination operands other than SR:

C - Not affected.
V - Not affected.
Z - Not affected.
N - Not affected.
I - Not affected.
LR - Not affected.
$\overline{R}$ - Not affected.
A - Not affected.

ER Status Bits:

For destination operand SR:

INX -Set according to bit 8 of the source operand.
DZ -Set according to bit 9 of the source operand.
UNF -Set according to bit 10 of the source operand.
OVF -Set according to bit 11 of the source operand.
OPERR-Set according to bit 12 of the source operand.
SNAN -Set according to bit 13 of the source operand.
NAN -Set according to bit 14 of the source operand.
UNCC -Set according to bit 15 of the source operand.

For destination operands other than SR:

INX - Not affected.
DZ - Not affected.
UNF - Not affected.
OVF - Not affected.
OPERR- Not affected.
SNAN - Not affected.
NAN - Not affected.
UNCC - Not affected.

IER Flags:

For destination operand SR:

SINX -Set according to bit 16 of the source operand.
SDZ -Set according to bit 17 of the source operand.
SUNF -Set according to bit 18 of the source operand.
SOVF -Set according to bit 19 of the source operand.
SIOP -Set according to bit 20 of the source operand.

For destination operands other than SR:

SINX - Not affected.
SDZ - Not affected.
SUNF - Not affected.
SOVF - Not affected.
SIOP - Not affected.

Instruction Format: MOVE(M) P: ea, D MOVE(M) S,P: ea

31				14	13			0
0000	0001	0110	MMMR	RR	1W	0001	0ddd	dddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

<ea> Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Register	**W**
Read S	0
Write D	1

D	**d d d d d d d**	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 6 + mvm oscillator clock cycles
Memory: 1 + ea program words

MOVE(P) Move Peripheral Data MOVE(P)

Operation:

X:<pp> → D
S → X:<pp>
#xxxx → X:<pp>
Y:<pp> → D
S → Y:<pp>
#xxxx → Y:<pp>
X:<pp> → X:<ea>
X:<ea> → X:<pp>
X:<pp> → Y:<ea>
Y:<ea> → X:<pp>
Y:<pp> → X:<ea>
X:<ea> → Y:<pp>
Y:<pp> → Y:<ea>
Y:<ea> → Y:<pp>
X:<pp> → X:<Rn+xxxx>
X:<Rn+xxxx> → X:<pp>
X:<pp> → Y:<Rn+xxxx>
Y:<Rn+xxxx> → X:<pp>
Y:<pp> → X:<Rn+xxxx>
X:<Rn+xxxx> → Y:<pp>
Y:<pp> → Y:<Rn+xxxx>
Y:<Rn+xxxx> → Y:<pp>
X:<pp> → P:<ea>
P:<ea> → X:<pp>
Y:<pp> → P:<ea>
P:<ea> → Y:<pp>

Assembler Syntax:

MOVE(P) X: pp, D
MOVE(P) S,X: pp
MOVE(P) #Data,X: pp
MOVE(P) Y: pp, D
MOVE(P) S,Y: pp
MOVE(P) #Data,Y: pp
MOVE(P) X: pp, X: ea
MOVE(P) X: ea, X: pp
MOVE(P) X: pp, Y: ea
MOVE(P) Y: ea, X: pp
MOVE(P) Y: pp, X: ea
MOVE(P) X: ea, Y: pp
MOVE(P) Y: pp, Y: ea
MOVE(P) Y: ea, Y: pp
MOVE(P) X: pp, X:(Rn+displacement)
MOVE(P) X:(Rn+displacement),X: pp
MOVE(P) X: pp, Y:(Rn+displacement)
MOVE(P) Y:(Rn+displacement),X: pp
MOVE(P) Y: pp, X:(Rn+displacement)
MOVE(P) X:(Rn+displacement),Y: pp
MOVE(P) Y: pp, Y:(Rn+displacement)
MOVE(P) Y:(Rn+displacement),Y: pp
MOVE(P) X: pp, P: ea
MOVE(P) P: ea, X: pp
MOVE(P) Y: pp, P: ea
MOVE(P) P: ea, Y: pp

Description:

Move the word operand to or from the X and Y I/O peripherals. The 7-bit I/O Short Address is one extended permitting access to the I/O peripheral addresses located in the highest 128 locations of the X and Y data memories. All memory addressing modes may be used for the memory effective address. The Long Displacement addressing mode may also be used.

If the system stack register SSH is specified as a source operand, the system stack pointer SP is postdecremented by 1 after SSH is read. If the system stack register SSH is specified as a destination operand,

the system stack pointer SP is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

See **Section A.10** for restrictions that apply to this instruction.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

For destination operand SR:

C	- Set according to bit 0 of the source operand.
V	- Set according to bit 1 of the source operand.
Z	- Set according to bit 2 of the source operand.
N	- Set according to bit 3 of the source operand.
I	- Set according to bit 4 of the source operand.
LR	- Set according to bit 5 of the source operand.
$\overline{R}$	- Set according to bit 6 of the source operand.
A	- Set according to bit 7 of the source operand.

For destination operands other than SR:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

For destination operand SR:

INX	-Set according to bit 8 of the source operand.
DZ	-Set according to bit 9 of the source operand.
UNF	-Set according to bit 10 of the source operand.
OVF	-Set according to bit 11 of the source operand.
OPERR	-Set according to bit 12 of the source operand.
SNAN	-Set according to bit 13 of the source operand.
NAN	-Set according to bit 14 of the source operand.
UNCC	-Set according to bit 15 of the source operand.

For destination operands other than SR:

- INX - Not affected.
- DZ - Not affected.
- UNF - Not affected.
- OVF - Not affected.
- OPERR - Not affected.
- SNAN - Not affected.
- NAN - Not affected.
- UNCC - Not affected.

IER Flags:

For destination operand SR:

- SINX -Set according to bit 16 of the source operand.
- SDZ -Set according to bit 17 of the source operand.
- SUNF -Set according to bit 18 of the source operand.
- SOVF -Set according to bit 19 of the source operand.
- SIOP -Set according to bit 20 of the source operand.

For destination operands other than SR:

- SINX - Not affected.
- SDZ - Not affected.
- SUNF - Not affected.
- SOVF - Not affected.
- SIOP - Not affected.

Instruction Format:

MOVE(P) X: pp, X: ea
MOVE(P) X: ea, X: pp
MOVE(P) X: pp, Y: ea
MOVE(P) Y: ea, X: pp
MOVE(P) Y: pp, X: ea
MOVE(P) X: ea, Y: pp
MOVE(P) Y: pp, Y: ea
MOVE(P) Y: ea, Y: pp
MOVE(P) #Data,X: pp
MOVE(P) #Data,Y: pp

31				14	13			0
0000	0000	0111	MMMR	RR	11	1sSW	1ppp	pppp
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA								

Instruction Format:

MOVE(P) X: pp, X:(Rn+displacement)
MOVE(P) X:(Rn+displacement),X: pp
MOVE(P) X: pp, Y:(Rn+displacement)
MOVE(P) Y:(Rn+displacement),X: pp
MOVE(P) Y: pp, X:(Rn+displacement)
MOVE(P) X:(Rn+displacement),Y: pp
MOVE(P) Y: pp, Y:(Rn+displacement)
MOVE(P) Y:(Rn+displacement),Y: pp

31				14	13			0
0000	0000	0111	000R	RR	10	1sSW	1ppp	pppp
LONG DISPLACEMENT								

Instruction Format: MOVE(P) X: pp, P: ea MOVE(P) Y: pp, P: ea
MOVE(P) P: ea, X: pp MOVE(P) P: ea, Y: pp

31				14	13			0
0000	0000	0111	MMMR	RR	11	01SW	1ppp	pppp
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Format: MOVE(P) X: pp, D MOVE(P) Y: pp, D
MOVE(P) S,X: pp MOVE(P) S,Y: pp

31				14	13			0
0000	0000	0111	dddd	dd	d0	00SW	1ppp	pppp

Instruction Fields:

<ea> Rn - R0-R7

X: or Y: reference (Memory addressing modes only)

P: reference (Memory Alterable addressing modes only)

Absolute Address - 32 bits

Long Displacement - 32 bits

I/O Short Address - ppppppp (7 bits)

Memory Space	**s**	**Periph Space**	**S**	**Peripheral**	**W**
X Memory	0	X Memory	0	Read	0
Y Memory	1	Y Memory	1	Write	1

S,D	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 2 + mvp oscillator clock cycles

Memory: 1 + mv program words

MOVE(S) Move Absolute Short

Operation:

X:<aa> → D1
S1 → X :<aa>
#xxxx → X:<aa>
Y:<aa> → D1
S1 → Y:<aa>
#xxxx → Y:<aa>
L:<aa> → D2
S2 → L:<aa>
X:<aa> → X:<ea>
X:<ea> → X:<aa>
X:<aa> → Y:<ea>
Y:<ea> → X:<aa>
Y:<aa> → X:<ea>
X:<ea> → Y:<aa>
Y:<aa> → Y:<ea>
Y:<ea> → Y:<aa>
X:<aa> → X:<Rn+xxxx>
X:<Rn+xxxx> → X:<aa>
X:<aa> → Y:<Rn+xxxx>
Y:<Rn+xxxx> → X:<aa>
Y:<aa> → X:<Rn+xxxx>
X:<Rn+xxxx> → Y:<aa>
Y:<aa> → Y:<Rn+xxxx>
Y:<Rn+xxxx> → Y:<aa>
X:<aa> → P:<ea>
P:<ea> → X:<aa>
Y:<aa> → P:<ea>
P:<ea> → Y:<aa>

Assembler Syntax:

MOVE(S) X: aa, D1
MOVE(S) S1,X: aa
MOVE(S) #Data,X: aa
MOVE(S) Y: aa, D1
MOVE(S) S1,Y: aa
MOVE(S) #Data,Y: aa
MOVE(S) L: aa, D2
MOVE(S) S2,L: aa
MOVE(S) X: aa, X: ea
MOVE(S) X: ea, X: aa
MOVE(S) X: aa, Y: ea
MOVE(S) Y: ea, X: aa
MOVE(S) Y: aa, X: ea
MOVE(S) X: ea, Y: aa
MOVE(S) Y: aa, Y: ea
MOVE(S) Y: ea, Y: aa
MOVE(S) X: aa, X:(Rn+displacement)
MOVE(S) X:(Rn+displacement),X: aa
MOVE(S) X: aa, Y:(Rn+displacement)
MOVE(S) Y:(Rn+displacement),X: aa
MOVE(S) Y: aa, X:(Rn+displacement)
MOVE(S) X:(Rn+displacement),Y: aa
MOVE(S) Y: aa, Y:(Rn+displacement)
MOVE(S) Y:(Rn+displacement),Y: aa
MOVE(S) X: aa, P: ea
MOVE(S) P: ea, X: aa
MOVE(S) Y: aa, P: ea
MOVE(S) P: ea, Y: aa

Description:

Move the word operand to or from the lower 128 memory locations in X and Y Data memories. The 7-bit Absolute Short Address is zero extended. All memory addressing modes may be used for the memory effective address. The Long Displacement addressing mode may also be used.

If the system stack register SSH is specified as a source operand, the system stack pointer SP is postdecremented by 1 after SSH is read. If the system stack register SSH is specified as a destination operand, the system stack pointer SP is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

See **Section A.10** for restrictions that apply to this instruction.

CAUTION

See restrictions in Section A.10.6 concerning Rn, Mn, and Nn registers as a destination.

CCR Condition Codes:

For destination operand SR:

C	- Set according to bit 0 of the source operand.
V	- Set according to bit 1 of the source operand.
Z	- Set according to bit 2 of the source operand.
N	- Set according to bit 3 of the source operand.
I	- Set according to bit 4 of the source operand.
LR	- Set according to bit 5 of the source operand.
$\overline{R}$	- Set according to bit 6 of the source operand.
A	- Set according to bit 7 of the source operand.

For destination operands other than SR:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

For destination operand SR:

INX	-Set according to bit 8 of the source operand.
DZ	-Set according to bit 9 of the source operand.
UNF	-Set according to bit 10 of the source operand.
OVF	-Set according to bit 11 of the source operand.
OPERR	-Set according to bit 12 of the source operand.
SNAN	-Set according to bit 13 of the source operand.
NAN	-Set according to bit 14 of the source operand.
UNCC	-Set according to bit 15 of the source operand.

For destination operands other than SR:

INX - Not affected.
DZ - Not affected.
UNF - Not affected.
OVF - Not affected.
OPERR- Not affected.
SNAN - Not affected.
NAN - Not affected.
UNCC - Not affected.

IER Flags:

For destination operand SR:

SINX -Set according to bit 16 of the source operand.
SDZ -Set according to bit 17 of the source operand.
SUNF -Set according to bit 18 of the source operand.
SOVF -Set according to bit 19 of the source operand.
SIOP -Set according to bit 20 of the source operand.

For destination operands other than SR:

SINX - Not affected.
SDZ - Not affected.
SUNF - Not affected.
SOVF - Not affected.

Instruction Format:

MOVE(S) X: aa, X: ea	MOVE(S) Y: aa, X: ea
MOVE(S) X: ea, X: aa	MOVE(S) X: ea, Y: aa
MOVE(S) X: aa, Y: ea	MOVE(S) Y: aa, Y: ea
MOVE(S) Y: ea, X: aa	MOVE(S) Y: ea, Y: aa
MOVE(S) #Data,X: aa	
MOVE(S) #Data,Y: aa	

31				14	13			0
0000	0000	0111	MMMR	RR	11	1sSW	0aaa	aaaa
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA								

Instruction Format:

MOVE(S) X: aa, X:(Rn+displacement)	MOVE(S) Y: aa, X:(Rn+displacement)
MOVE(S) X:(Rn+displacement),X: aa	MOVE(S) X:(Rn+displacement),Y: aa
MOVE(S) X: aa, Y:(Rn+displacement)	MOVE(S) Y: aa, Y:(Rn+displacement)
MOVE(S) Y:(Rn+displacement),X: aa	MOVE(S) Y:(Rn+displacement),Y: aa

31				14	13			0
0000	0000	0111	000R	RR	10	1sSW	0aaa	aaaa
LONG DISPLACEMENT								

SIOP - Not affected.

Instruction Format: MOVE(S) X: aa, P: ea MOVE(S) Y: aa, P: ea
MOVE(S) P: ea, X: aa MOVE(S) P: ea, Y: aa

31				14	13			0
0000	0000	0111	MMMR	RR	11	01SW	0aaa	aaaa
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Format: MOVE(S) X: aa, D1 MOVE(S) Y: aa, D1
MOVE(S) S1,X: aa MOVE(S) S1,Y: aa

31				14	13			0
0000	0000	0111	dddd	dd	d0	00SW	0aaa	aaaa

Instruction Format: MOVE(S) L: aa, D2
MOVE(S) S2,L: aa

31				14	13			0
0000	0000	0111	DDDD	DD	D1	000W	0aaa	aaaa

Instruction Fields:

<ea> Rn - R0-R7

X: or Y: reference (Memory addressing modes only)

P: reference (Memory Alterable addressing modes only)

Absolute Address - 32 bits

Long Displacement - 32 bits

Absolute Short Address - aaaaaaa (7 bits)

Memory Space	**s**	**Abs. Short Space**	**S**	**Abs. Short Location**	**W**
X Memory	0	X Memory	0	Read	0
Y Memory	1	Y Memory	1	Write	1

S1, D1	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

S2,D2	D D D D D D D	
D0.ML-D7.ML	1 0 1 1 n n n	where nnn = 0-7
D0.D-D7.D	1 0 1 0 n n n	
D9.ML	1 0 0 0 1 1 1	
D8.ML	1 0 0 0 1 1 0	
D9.D	1 0 0 0 1 0 1	
D8.D	1 0 0 0 1 0 0	

Timing: 2 + mvs oscillator clock cycles

Memory: 1 + mv program words

MOVETA Move Data Registers and Test Address MOVETA

Operation:

parallel data bus move

Assembler Syntax:

MOVETA
(move syntax - see the Move instruction description).

Description:

Move the contents of the specified source to the specified destination and update the C, V, N and Z flags in the CCR according to the result of the address calculation. Only Address Register Indirect addressing modes will give meaningful flag updates. For the No Update addressing mode, the address calculation is assumed to be Rn-0 with linear modifier while ignoring the contents of the Mn and Nn registers. For XY moves, update the CCR according to the result of the X address calculation. This instruction is a Data ALU NOP instruction with the parallel data move operations described in the MOVE instruction description.

Some parallel data move operations differentiate between integer or floating-point operands according to the kind of Data ALU operation specified. For this purpose, two Data ALU NOP opcodes are used: an "integer NOP" and a "floating-point NOP". For example, if a XY parallel move is specified with integer operands, the assembler will produce a 32 bit instruction word with the "integer NOP" in the Data ALU opcode field. If floating-point operands are specified, the "floating-point NOP" is used instead.

CCR Condition Codes:

C - For increment addressing modes: Set if carry occurred out of the MSB during address calculation with linear modifier or carry occurred out of the LSB during address calculation with reverse carry modifier. Cleared otherwise.

For decrement addressing modes: Set if borrow occurred out of the MSB during address calculation with linear modifier or borrow occurred out of the LSB during address calculation with reverse carry modifier. Cleared otherwise.

For modulo addressing modes: Always cleared.

V - Set if overflow occurred out the MSB during address calculation with a linear modifier. Set if overflow occurred out the LSB during address calculation with a reverse carry modifier. Set if wrap-around occurred during address calculation with a modulo modifier. Set if at least one wrap-around occurred during address calculation with a multiple wrap-around modulo modifier. Cleared otherwise.

Z - Set if result of the address calculation is zero. Cleared otherwise.

N - Set if the MSB of the result of the address calculation with linear or modulo modifier is set. Set if the LSB of the result of the address calculation with reverse carry modifier is set. Cleared otherwise.

I - Not affected.

LR - Not affected.

$\overline{\mathrm{R}}$ - Not affected.

A - Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: MOVETA (Integer NOP)

Instruction Fields:

See the MOVE instruction description for Data Bus Move Field encoding.

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	0000	1000	0010
OPTIONAL EFFECTIVE ADDRESS EXTENSION				

Instruction Format: MOVETA (Integer NOP)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	0000	1000	0110
OPTIONAL EFFECTIVE ADDRESS EXTENSION				

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYS Signed Multiply MPYS

Operation:

S1.L $*$ S2.L $\rightarrow$ D.M:D.L (parallel data bus move)

Assembler Syntax:

MPYS S1,S2,D
(See the MOVE instruction description.)

MPYS S2,S1,D
(See the MOVE instruction description.)

Description:

Multiply two signed operands and store the product in the specified destination register. The two source operands are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit signed integer stored in the middle and low portions of D. Registers D8 and D9 can be used as source registers.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Cleared if the most significant 32 bits of the 64-bit result are the sign extension of the least significant 32 bits. Set otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{\text{R}}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

Instruction Format: MPYS S1,S2,D (See the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	1sss	SSS0	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: MPYS S2(8,9),S1,D (See the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	0sss	11S0	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

IER Flags: Not affected.

Instruction Fields:

S1	**s s s**	
Dn	n n n	where nnn = 0-7

S2	**S S S**	
Dn	n n n	where nnn = 0-7

S2	**S**
D8	0
D9	1

D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

$S1.L * S2.L \rightarrow D.M:D.L$ (parallel data bus move)

Assembler Syntax:

MPYU S1,S2,D
(See the MOVE instruction description.)

MPYU S2,S1,D
(See the MOVE instruction description.)

Description:

Multiply two unsigned operands and store the product in the specified destination register. The two source operands are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit unsigned integer stored in the middle and low portions of D. Registers D8 and D9 can be used as source registers.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Cleared if the most significant 32 bits of the 64-bit result are zero. Set otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Always cleared.
I	- Not affected.
LR	- Not affected.
$\overline{\ }$R	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Fields:

Instruction Format: MPYU S1,S2,D (See the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	1sss	SSS1	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Format: MPYU S2(8,9),S1,D (See the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	0sss	11S1	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

S1	**s s s**	
Dn	n n n	where nnn = 0-7

S2	**S S S**	
Dn	n n n	where nnn = 0-7

S2	**s**
D8	0
D9	1

D	**d d d**	
Dn	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

$0 - D.L \rightarrow D.L$ (parallel data bus move)

Assembler Syntax:

NEG D
(See the MOVE instruction description.)

Description:

The low portion of the destination operand is subtracted from zero. The result is stored in the low portion of D. This instruction is preferable to using the SUB instruction since it is not necessary to zero an input operand.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if a borrow is generated from the MSB of the result. Cleared otherwise.
V	- Set if result overflows. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: NEG D (See the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0101	uu11	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

0 - D.L - C → D.L (parallel data bus move)

Assembler Syntax:

NEGC D
(See the MOVE instruction description.)

Description:

Subtract the low portion of the destination operand D from zero along with the C bit of the condition code register and store the result in the low portion of D. This instruction is useful when negating a multiple precision number since it is not necessary to first zero an input operand as would be the case if the SUB instruction were used. Note that the higher precision long words of the input variable must first be moved to the lower portion of the Dn.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if a borrow is generated from the MSB of the result. Cleared otherwise.
V	- Set if result overflows. Cleared otherwise.
Z	- Cleared if the result is not zero. Unchanged otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: NEGC D (See the MOVE instruction description.)

31 — 14	13 — 0
DATA BUS MOVE FIELD	10 0001 uu11 1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

No Operation

Operation:
None

Assembler Syntax:
NOP

Description:

No operation occurs. The processor state, other than the program counter, is not affected. Execution continues with the instruction following the NOP.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: NOP

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0000

Instruction Fields:

None

Timing: 2 oscillator clock cycles

Memory: 1 program words

Operation:

~D{31:0} → D{31:0} (parallel data bus move)

Assembler Syntax:

NOT D
(See the MOVE instruction description.)

Description:

The one's complement of the low portion of the destination operand is taken and the result is stored in D. This instruction is a 32-bit operation and is performed on bits 0-31 of D. The remaining bits of D are not affected.

Input Operand(s) Precision: 32-bit integer**.**

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: NOT D (See the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	10	0010	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Logical Inclusive OR

Operation:

D.L v S.L → D.L (parallel data bus move)

Assembler Syntax:

OR S,D
(See the MOVE instruction description.)

Description:

Logically inclusive OR the low portion of the two specified operands and store the result in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{\ }$R	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: OR S,D (See the MOVE instruction description.)

Instruction Fields:

31 — 14	13 — 0			
DATA BUS MOVE FIELD	00	0sss	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

ORC Logical Inclusive OR with Complement ORC

Operation:

D.L v ~S.L → D.L (parallel data bus move)

Assembler Syntax:

ORC S,D
(See the MOVE instruction description.)

Description:

Logically inclusive OR the low portion of D with the logical complement of the low portion of S, and store the result in the low portion of D. This instruction is useful for manipulating bit maps in graphic operations.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ORC S,D (See the MOVE instruction description.)

31 ... 14	13 ... 0			
DATA BUS MOVE FIELD	11	0sss	1001	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION				

Instruction Fields:

D	**d d d**	
Dn.L	n n n	where nnn = 0-7

S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

ORI OR Immediate to Control Register ORI

Operation:

D v #xx → D

Assembler Syntax:

OR(I) #Mask,D

Description:

Logically OR the contents of the control register with an 8-bit immediate operand. The result is stored back into the specified control register. See **Section A.10** for restrictions.

CCR Condition Codes:

For CCR operand:

C	- Set if bit 0 of the immediate operand is set. Not affected otherwise.
V	- Set if bit 1 of the immediate operand is set. Not affected otherwise.
Z	- Set if bit 2 of the immediate operand is set. Not affected otherwise.
N	- Set if bit 3 of the immediate operand is set. Not affected otherwise.
I	- Set if bit 4 of the immediate operand is set. Not affected otherwise.
LR	- Set if bit 5 of the immediate operand is set. Not affected otherwise.
$\overline{R}$	- Set if bit 6 of the immediate operand is set. Not affected otherwise.
A	- Set if bit 7 of the immediate operand is set. Not affected otherwise.

For OMR, MR, IER, ER operands:

C	- Not affected.
V	- Not affected.
Z	- Not affected.
N	- Not affected.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits:

For ER operand:

INX	-Set if bit 0 of the immediate operand is set. Not affected otherwise.
DZ	-Set if bit 1 of the immediate operand is set. Not affected otherwise.
UNF	-Set if bit 2 of the immediate operand is set. Not affected otherwise.
OVF	-Set if bit 3 of the immediate operand is set. Not affected otherwise.
OPERR	-Set if bit 4 of the immediate operand is set. Not affected otherwise.
SNAN	-Set if bit 5 of the immediate operand is set. Not affected otherwise.
NAN	-Set if bit 6 of the immediate operand is set. Not affected otherwise.
UNCC	-Set if bit 7 of the immediate operand is set. Not affected otherwise.

For OMR, MR, IER, CCR operands:

INX	- Not affected.
DZ	- Not affected.
UNF	- Not affected.
OVF	- Not affected.
OPERR	- Not affected.
SNAN	- Not affected.
NAN	- Not affected.
UNCC	- Not affected.

IER Flags:

For IER operand:

SINX	-Set if bit 0 of the immediate operand is set. Not affected otherwise.
SDZ	-Set if bit 1 of the immediate operand is set. Not affected otherwise.
SUNF	-Set if bit 2 of the immediate operand is set. Not affected otherwise.
SOVF	-Set if bit 3 of the immediate operand is set. Not affected otherwise.
SIOP	-Set if bit 4 of the immediate operand is set. Not affected otherwise.

For OMR, MR, ER, CCR operands:

SINX	- Not affected.
SDZ	- Not affected.
SUNF	- Not affected.
SOVF	- Not affected.
SIOP	- Not affected.

Instruction Format: OR(I) #Mask,D

31				14	13			0
0000	0001	0001	i i i i	i i	i i	00ff	1111	10EE

Instruction Fields:

Immediate Short Data - iiiiiiii (8 bits)

D	**E E f f**
CCR	0 1 0 0
ER	0 1 0 1
IER	0 1 1 0
MR	0 1 1 1
OMR	1 0 0 0

Timing: 2 + mv oscillator clock cycles

Memory: 1 program words

REP
REP
Repeat Next Instruction

Operation:	**Assembler Syntax:**
LC $\rightarrow$ TEMP; X:<ea> $\rightarrow$ LC Repeat next instruction until LC = 1. TEMP $\rightarrow$ LC	REP X: ea
LC $\rightarrow$ TEMP; Y:<ea> $\rightarrow$ LC Repeat next instruction until LC = 1. TEMP $\rightarrow$ LC	REP Y: ea
LC $\rightarrow$ TEMP; S $\rightarrow$ LC Repeat next instruction until LC = 1. TEMP $\rightarrow$ LC	REP S
LC $\rightarrow$ TEMP; #xxx $\rightarrow$ LC Repeat next instruction until LC = 1. TEMP $\rightarrow$ LC	REP #Count

Description:

The single word instruction following the REP instruction is executed LC times repetitively, where LC is the value in the loop counter. If LC=0, the instruction is repeated 2**32 times. The current loop counter (LC) value is stored in an internal temporary register. The effective address specifies the address of the repeat count which is loaded into LC. All address register indirect addressing modes except Long Displacement may be used. Immediate Short and Register Direct addressing modes may also be used. The 19-bit immediate data is zero extended to form the loop counter value.

When the REP instruction is in effect, the repeated instruction is fetched only once and remains in the instruction register for the duration of the repeat count.

REP is not interruptible and can repeat any single word instruction which does not change program flow. See **Section A.10** for the complete list of restricted instructions.

If the system stack register SSH is specified as a source operand, the system stack pointer SP is postdecremented by 1 after SSH is read.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: REP #Count

31				14	13			0
0000	0001	1111	i i i i	i i	i i	i i i i	1 i i i	i i i i

Instruction Format: REP S

31				14	13			0
0000	0001	1110	0000	00	00	0000	1ddd	dddd

Instruction Format: REP X: ea
REP Y: ea

31				14	13			0
0000	0001	110s	MMMR	RR	00	0000	1000	0000

Instruction Fields:

<ea> Rn - R0-R7 (Address Register Indirect Modes except (Rn+xxx))

Immediate Short Data - iiiiiiiiiiiiiiiiiii (19 bits)

Memory Space	**s**
X Memory	0
Y Memory	1

S	d d d d d d d	
D0.S-D7.S	0 0 0 0 n n n	where nnn = 0-7
D0.L-D7.L	0 0 0 1 n n n	
D0.M-D7.M	0 0 1 0 n n n	
D0.H-D7.H	0 0 1 1 n n n	
D8.L	0 1 0 0 0 0 0	
D9.L	0 1 0 0 0 0 1	
D8.M	0 1 0 0 0 1 0	
D9.M	0 1 0 0 0 1 1	
D8.H	0 1 0 0 1 0 0	
D9.H	0 1 0 0 1 0 1	
D8.S	0 1 0 0 1 1 0	
D9.S	0 1 0 0 1 1 1	
R0-R7	0 1 0 1 n n n	
N0-N7	0 1 1 0 n n n	
M0-M7	0 1 1 1 n n n	
SR	1 1 1 1 0 0 1	
OMR	1 1 1 1 0 1 0	
SP	1 1 1 1 0 1 1	
SSH	1 1 1 1 1 0 0	
SSL	1 1 1 1 1 0 1	
LA	1 1 1 1 1 1 0	
LC	1 1 1 1 1 1 1	

Timing: 4 + mv oscillator clock cycles

Memory: 1 program words

Reset Peripheral Devices

Operation:

Reset all on-chip peripherals and the Interrupt Priority Register.

Assembler Syntax:

RESET

Description:

All on-chip peripherals and the Interrupt Priority Register are reset. See Chapter 7 for a description of the effect of the RESET instruction on the peripherals. The processor state is not affected and execution continues with the next instruction, but all maskable interrupt sources are disabled. The only interrupts that can then occur are Stack Error, Hardware Reset, ILLEGAL, TRAPcc and FTRAPcc.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: RESET

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0100

Instruction Fields:

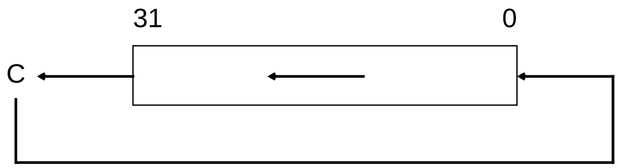
None

Timing: 4 oscillator clock cycles

Memory: 1 program words

ROL — Rotate Left

Operation:



(parallel data bus move)

Assembler Syntax:

ROL D (See the MOVE instruction description.)

Description:

Rotate the low portion of the specified operand one bit to the left. The carry bit receives the previous value of bit 31 of the operand. The previous value of the carry bit is shifted into bit 0 of the operand. The result is stored in the low portion of D. This instruction is a 32 bit operation and is performed on bits 0-31 of D. The remaining bits of D are not affected.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if the bit shifted out of the operand is set. Cleared otherwise.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ROL D (See the MOVE instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	10	0011	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

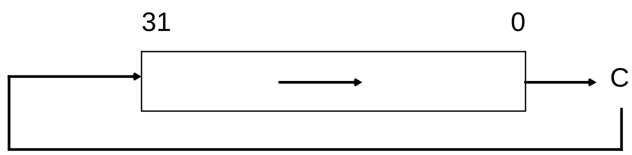
Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:



(parallel data bus move)

Assembler Syntax:

ROR D (See the MOVE instruction description.)

Description:

Rotate the low portion of the specified operand one bit to the right. The carry bit receives the previous value of bit 0 of the operand. The previous value of the carry bit is shifted into bit 31 of the operand. The result is stored in the low portion of D. This instruction is a 32 bit operation and is performed on bits 0-31 of D. The remaining bits of D are not affected.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if the bit shifted out of the operand is set. Cleared otherwise.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: ROR D (See the MOVE instruction description.)

31 14 13 0

DATA BUS MOVE FIELD	10	0011	uu01	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

RTI Return from Interrupt RTI

Operation:

Assembler Syntax:

RTI

Description:

The program counter and the status register are pulled from the system stack. The interrupt routine program counter and status register are lost. RTI if functionally identical to RTR but has been made a separate instruction to be upward compatible with future parts and to simplify porting software.

Due to pipelining, the RTI instruction must not be immediately preceded by some instructions. See **Section A.10** for the list of restricted instructions.

CCR Condition Codes:

C	- Set according to value pulled from stack.
V	- Set according to value pulled from stack.
Z	- Set according to value pulled from stack.
N	- Set according to value pulled from stack.
I	- Set according to value pulled from stack.
LR	- Set according to value pulled from stack.
$\overline{R}$	- Set according to value pulled from stack.
A	- Set according to value pulled from stack.

ER Status Bits:

INX	-Set according to value pulled from stack.
DZ	-Set according to value pulled from stack.
UNF	-Set according to value pulled from stack.
OVF	-Set according to value pulled from stack.
OPERR	-Set according to value pulled from stack.
SNAN	-Set according to value pulled from stack.
NAN	-Set according to value pulled from stack.
UNCC	-Set according to value pulled from stack.

IER Flags:

SINX	-Set according to value pulled from stack.
SDZ	-Set according to value pulled from stack.
SUNF	-Set according to value pulled from stack.
SOVF	-Set according to value pulled from stack.
SIOP	-Set according to value pulled from stack.

SSH $\rightarrow$ PC; SSL $\rightarrow$ SR; SP – 1 $\rightarrow$ SP

Instruction Format: RTI

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	1100

Instruction Fields:

None.

Timing: 4 + rx oscillator clock cycles

Memory: 1 program words

RTR Return from Subroutine with Restore RTR

Operation:

SSH → PC; SSL → SR; SP – 1 → SP

Assembler Syntax:

RTR

Description:

The program counter and the status register are pulled from the system stack. The subroutine program counter and status register are lost. RTR if functionally identical to RTI but has been made a separate instruction to be upward compatible with future parts and to simplify porting software.

Due to pipelining, the RTR instruction must not be immediately preceded by some instructions. See **Section A.10** for the list of restricted instructions.

CCR Condition Codes:

C	- Set according to value pulled from stack.
V	- Set according to value pulled from stack.
Z	- Set according to value pulled from stack.
N	- Set according to value pulled from stack.
I	- Set according to value pulled from stack.
LR	- Set according to value pulled from stack.
$\overline{R}$	- Set according to value pulled from stack.
A	- Set according to value pulled from stack.

ER Status Bits:

INX	-Set according to value pulled from stack.
DZ	-Set according to value pulled from stack.
UNF	-Set according to value pulled from stack.
OVF	-Set according to value pulled from stack.
OPERR	-Set according to value pulled from stack.
SNAN	-Set according to value pulled from stack.
NAN	-Set according to value pulled from stack.
UNCC	-Set according to value pulled from stack.

IER Flags:

SINX	-Set according to value pulled from stack.
SDZ	-Set according to value pulled from stack.
SUNF	-Set according to value pulled from stack.
SOVF	-Set according to value pulled from stack.
SIOP	-Set according to value pulled from stack.

Instruction Format: RTR

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	1000

Instruction Fields:

None.

Timing: 4 + rx oscillator clock cycles

Memory: 1 program words

Operation:

SSH → PC; SP – 1 → SP

Assembler Syntax:

RTS

Description:

The program counter is pulled from the system stack. The status register is not affected. The subroutine program counter is lost.

Due to pipelining, the RTS instruction must not be immediately preceded by some instructions. See **Section A.10** for the list of restricted instructions.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: RTS

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	1101

Instruction Fields:

None.

Timing: 4 + rx oscillator clock cycles

Memory: 1 program words

SETW — Set Long Word Operand

Operation:

$FFFFFFFF → D.L (parallel data bus move)

Assembler Syntax:

SETW D

(move syntax - see the Move instruction description.)

Description:

The low portion (long word) of the destination operand is set to all ones.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Always cleared.
N	- Always set.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: SETW D (move syntax - see the Move instruction description.)

31 14	13 0			
DATA BUS MOVE FIELD	11	0uuu	1001	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D	**(u u)** **d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

SPLIT Extract a 16-bit Integer SPLIT

Operation:

S.L {31:16} → D.L {15:0} (parallel data bus move)

S.L {31} → D.L {31:16}

Assembler Syntax:

SPLIT S,D
(move syntax - see the Move instruction description.)

Description:

Transfer the 16 MSBs of the lower portion of source operand S into the 16 LSBs of the lower portion of destination D and sign-extend to 32 bits.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: SPLIT S,D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	0sss	1011	0ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D	**d d d**	
Dn.L	n n n	where nnn = 0-7

S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

SPLITB Extract an 8-bit Integer SPLITB

Operation:

S.L {15:8} → D.L {7:0} (parallel data bus move)

S.L {15} → D.L {31:8}

Assembler Syntax:

SPLITB S,D

(move syntax - see the Move instruction description.)

Description:

Transfer bits 15-8 of the lower portion of source operand S into the 8 LSBs of the lower portion of destination D and sign-extend to 32 bits.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: SPLITB S,D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	11	0sss	1011	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D	**d d d**	
Dn.L	n n n	where nnn = 0-7

S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

STOP Stop Instruction Processing STOP

Operation:

Enter the STOP processing state and stop the clock oscillator.

Assembler Syntax:

STOP

Description:

When a STOP instruction is executed, the processor enters the STOP processing state. The clock oscillator is gated off. All activity in the processor is suspended until the $\overline{RESET}$ or $\overline{IRQA}$ pin is asserted. The STOP processing state is the lowest-power stand-by state.

During the STOP state, port A is in an idle state with the control signals held inactive (i.e., $\overline{RD}$ = $\overline{WR}$ = V_{cc} etc., the data pins (D0–D23) are high impedance, and the address pins (A1–A15) are unchanged from the previous instruction. If the bus grant was asserted when the STOP instruction was executed, port A will remain three-stated until the DSP exits the STOP state.

If the exit from the STOP state was caused by a low level on the $\overline{RESET}$ pin, then the processor will enter the reset processing state. Consult the DSP96002 Technical Data Sheet (DSP96002/D) for timing details.

If the exit from the STOP state was caused by a low level on the $\overline{IRQA}$ pin, then the processor will service the highest priority pending interrupt and will not service the $\overline{IRQA}$ interrupt unless it is highest priority. The interrupt will be serviced after an internal delay (see the DSP96002 Technical Data Sheet (DSP96002/D) for details). The processor will resume program execution at the instruction following the STOP instruction that caused the entry into the STOP state after the interrupt has been serviced or if no interrupt was pending immediately after the delay. If the $\overline{IRQA}$ pin is asserted when the STOP instruction is executed, the clock will not be gated off, and the internal delay counter will be started.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: STOP

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	1111

Instruction Fields:

None

Timing: n/a

Memory: 1 program words

SUB Subtract SUB

Operation:

D.L - S.L $\rightarrow$ D.L (parallel data bus move)

Assembler Syntax:

SUB S,D
(move syntax - see the Move instruction description.)

Description:

Subtract the low portion of the specified source operand S from the low portion of the destination operand D and store the result in the low portion of D.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if a borrow is generated from the MSB of the result. Cleared otherwise.
V	- Set if result overflows. Cleared otherwise.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: SUB S,D (move syntax - see the Move instruction description.)

31 14	13 0			
DATA BUS MOVE FIELD	00	1sss	uu00	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7

S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

D.L - S.L - C → D.L (parallel data bus move)

Assembler Syntax:

SUBC S,D
(move syntax - see the Move instruction description.)

Description:

Subtract the low portion of the specified source operand S from the low portion of the destination operand D along with the C bit of the condition code register and store the result in the low portion of D. This instruction is useful in multiple precision integer arithmetic routines. Note that the higher precision long words of the input variables must be moved to the low portion of the Dn.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes:

C	- Set if a borrow is generated from the MSB of the result. Cleared otherwise.
V	- Set if result overflows. Cleared otherwise.
Z	- Cleared if the result is not zero. Unchanged otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: SUBC S,D (move syntax - see the Move instruction description.)

31 — 14	13 — 0			
DATA BUS MOVE FIELD	00	1sss	uu10	1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

TFR Transfer Data ALU Register TFR

Operation:

S.L → D.L (parallel data bus move)

Assembler Syntax:

TFR S,D
(move syntax - see the Move instruction description.)

Description:

Transfer data from the low portion of the specified source Data ALU register to the low portion of the specified destination Data ALU register. TFR uses the internal Data ALU paths but does not affect the condition code bits. When the S and D registers are the same, this instruction is equivalent to an integer rounding operation.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: 32-bit integer.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: TFR S,D (move syntax - see the Move instruction description.)

31 — 14	13 — 0
DATA BUS MOVE FIELD	10 1sss uu01 1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

	(u u)	
D	**d d d**	
Dn.L	n n n	where nnn = 0-7
S	**s s s**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

TRAPcc Conditional Software Interrupt TRAPcc

Operation:

If cc, then
begin software exception processing.

Assembler Syntax:

TRAPcc

Description:

If the specified integer condition is true, normal instruction execution is suspended and software exception processing is initiated. The interrupt priority level (I1,I0) is set to 3 in the status register if a long interrupt service routine is used. If the specified condition is false, continue with the next instruction. See **Section A.10** for restrictions.

"cc" may specify the following conditions:

Mnemonic		**Condition**
CC (HS)	- carry clear (higher or same)	C = 0
CS (LO)	- carry set (lower)	C = 1
EQ	- equal	Z = 1
GE	- greater or equal	N && V = 0
GT	- greater than	Z v (N && V) = 0
HI	- higher	Z v C = 0
LE	- less or equal	Z v (N && V) = 1
LS	- lower or same	Z v C = 1
LT	- less than	N && V = 1
MI	- minus	N = 1
NE(Q)	- not equal	Z = 0
PL	- plus	N = 0
VC	- overflow clear	V = 0
VS	- overflow set	V = 1
AL	- always true	n.a.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: TRAPcc

31				14	13			0
0000	0000	0000	0000	00	1c	cccc	0000	0011

Instruction Fields:

Mnemonic	c c c c c	Mnemonic	c c c c c
EQ	0 1 0 0 0	NE(Q)	1 1 0 0 0
PL	0 1 0 0 1	MI	1 1 0 0 1
CC(HS)	0 1 0 1 0	CS(LO)	1 1 0 1 0
GE	0 1 0 1 1	LT	1 1 0 1 1
GT	0 1 1 0 0	LE	1 1 1 0 0
VC	0 1 1 0 1	VS	1 1 1 0 1
HI	0 1 1 1 0	LS	1 1 1 1 0
AL	1 1 1 1 1		

Timing: 10 oscillator clock cycles
Memory: 1 program words

Operation:

S - 0 (parallel data bus move)

Assembler Syntax:

TST S
(move syntax - see the Move instruction description.)

Description:

Compare the low portion of the specified operand with zero. No result is stored, however the condition codes are affected.

Input Operand(s) Precision: 32-bit integer.

Output Operand Precision: n.a.

CCR Condition Codes:

C	- Not affected.
V	- Always cleared.
Z	- Set if result is zero. Cleared otherwise.
N	- Set if result is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\bar{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: TST S (move syntax - see the Move instruction description.)

31 — 14	13 — 0
DATA BUS MOVE FIELD	10 0110 uu01 1ddd
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

S	**(u u) d d d**	
Dn.L	n n n	where nnn = 0-7

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:

Enter WAIT processing state and stop all internal processing.

Wait for an unmasked interrupt to occur.

Assembler Syntax:

WAIT

Description:

When a WAIT instruction is executed, the processor enters the WAIT state. The internal clocks to the processor core, memories, and DMA are gated off and all activity in the processor is suspended until an unmasked interrupt occurs. However the clock oscillator and the internal I/O peripheral clocks remain active. If WAIT is executed when an interrupt is pending, the interrupt will be processed; the effect will be the same as if the processor never entered the WAIT state and three NOPs followed the WAIT instruction. When an unmasked interrupt or external (hardware) processor RESET occurs, the processor leaves the WAIT state. The WAIT state is then cleared and exception processing of the unmasked interrupt or RESET condition begins. The $\overline{BR}/\overline{BG}$ circuits remain active during the WAIT state. The WAIT state is a low-power standby mode. The processor always leaves the WAIT state in the T2 clock phase (see the DSP96002 Advance Information Data Sheet (DSP96002/D)). Therefore, multiple processors may be synchronized by having them all enter the WAIT state and then interrupting them with a common interrupt.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: WAIT

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	1110

Instruction Fields:

None

Timing: n/a

Memory: 1 program words

A.8 INSTRUCTION ENCODING SUMMARY

The encoding for each instruction is provided with the instruction descriptions in subsection A.7. An instruction encoding summary is available upon request. Some instructions have legal operation codes but specify the same destination for two or more simultaneous operations. These instructions are called insane instructions. An example of an insane instruction is:

MOVE **X: ea, D3 Y: ea, D3**

Both parallel moves write to the same register (D3) which puts an indeterminate result in D3. These instructions are flagged as errors by the assembler. However, it is possible to produce an illegal or insane instruction with the assembler using the DC command.

The following parallel instructions produce insane instructions which will be flagged by the assembler and should not be used:

X: ea, XXX Y: ea, YYY – for YYY=XXX,

or for all combinations where YYY specifies the same destination as the Data ALU operation,

or for all combinations where XXX specifies the same destination as the Data ALU operation.

$\overline{X}$dd → XXX Y: ea, YYY – for YYY=XXX, where $\overline{X}$ is the inversion of the MSB of the XXX field.

or for all combinations where XXX specifies the same destination as the Data ALU operation,

or for all combinations where YYY specifies the same destination as the Data ALU operation.

X: ea, XXX $\overline{Y}$dd → YYY – for YYY=XXX, where $\overline{Y}$ is the inversion of the MSB of the YYY field.

or for all combinations where YYY specifies the same destination as the Data ALU operation.

or for all combinations where XXX specifies the same destination as the Data ALU operation.

S: ea, 0DDDDDD – for all combinations where DDDDDD specifies the same destination as the Data ALU operation.

X: ea, XXX Y:,YYY – for YYY=XXX,

or for all combinations where YYY or XXX specifies the same destination as the Data ALU operation.

L: ea, 10DDDDD – for all combinations where DDDDD specifies the same destination as the Data ALU operation.

10DDDDD → 10ddddd (DP) – for all combinations where ddddd specifies the same destination as the Data ALU operation.

X: ea, X Y:,Y – for Y=X.

S:(Rn+aaaa),0DDDDDD – for all combinations where DDDDDD specifies the same destination as the Data ALU operation.

X:(Rn+aaaa),XXX Y:,YYY – for YYY=XXX

or for all combinations where YYY or XXX specifies the same destination as the Data ALU operation.

L:(Rn+aaaa),10DDDDD (DP) – for all combinations where DDDDD specifies the same destination as the Data ALU operation.

X:(Rn+aaaa),X Y:,Y – for Y=X.

0DDDDDD → 0dddddd – for all combinations where dddddd specifies the same destination as the Data ALU operation.

A.9 INSTRUCTION TIMING

Figure A-7 shows the number of words and the number of clock cycles required for instruction execution. The symbols used reference other tables to complete the instruction word and cycle count. The number of words per instruction is dependent on the addressing mode and the type of parallel data bus move operation specified. The number of execution 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, whether the Data ALU is operating in the IEEE mode, the number of external bus accesses and the number of wait states inserted in each external access. The following tables assume:

1. All instruction cycles are counted in clock oscillator cycles.
2. The instruction fetch pipeline is full.
3. There is no contention for instruction fetches.
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 branch instructions (JMP, Jcc, RTI, etc.).

Mnemonic	Words	Cycles
ABS	1 + mv	2 + mv
ADD	1 + mv	2 + mv
ADDC	1 + mv	2 + mv
AND	1 + mv	2 + mv
ANDC	1 + mv	2 + mv
ANDI	1	2
ASL	1 + mv	2 + mv
ASL #shift	1	2
ASR	1 + mv	2 + mv
ASR #shift	1	2
Bcc	1 + ea	6 + jx
BCHG	1 + ea	4 + mvb
BCLR	1 + ea	4 + mvb
BFIND	1 + mv	2 + mv
BRA	1 + ea	6 + jx
BRCLR	2	8 + jx
BRSET	2	8 + jx
Mnemonic	**Words**	**Cycles**

Figure A-7 Instruction Timing Summary

Mnemonic	Words	Cycles
BScc	1 + ea	6 + jx
BSCLR	2	8 + jx
BSET	1 + ea	4 + mvb
BSR	1 + ea	6 + jx
BSSET	2	8 + jx
BTST	1 + ea	4 + mvb
CLR	1 + mv	2 + mv
CMP	1 + mv	2 + mv
CMPG	1 + mv	2 + mv
DEBUGcc	1	4
DEC	1 + mv	2 + mv
DO	2	6 + mv
DOR	2	8 + mv
ENDDO	1	2
EOR	1 + mv	2 + mv
EXT	1 + mv	2 + mv
EXTB	1 + mv	2 + mv
FABS.S	1 + mv	2+mv+da
FABS.X	1 + mv	2+mv+da
FADD.S	1 + mv	2+mv+da
FADD.X	1 + mv	2+mv+da
FADDSUB.S	1 + mv	2+mv+da
FADDSUB.X	1 + mv	2+mv+da
FBcc	1 + ea	6 + jx
FBScc	1 + ea	6 + jx
FCLR	1 + mv	2+mv+da
FCMP	1 + mv	2+mv+da
FCMPG	1 + mv	2+mv+da
FCMPM	1 + mv	2+mv+da
FCOPYS.S	1 + mv	2+mv+da
FCOPYS.X	1 + mv	2+mv+da
FDEBUGcc	1	4
FFcc	1	2 + da
FFcc.U	1	2 + da
FGETMAN	1 + mv	2+mv+da
FINT	1 + mv	2+mv+da
FJcc	1 + ea	6 + jx

Figure A-7 Instruction Timing Summary (Continued)

Mnemonic	Words	Cycles
FJScc	1 + ea	6 + jx
FLOAT.S	1 + mv	2+mv+da
FLOAT.X	1 + mv	2+mv+da
FLOATU.S	1 + mv	2+mv+da
FLOATU.X	1 + mv	2+mv+da
FLOOR	1 + mv	2+mv+da
FMPY//FADD.S	1 + mv	2+mv+da
FMPY//FADD.X	1 + mv	2+mv+da
FMPY//FADDSUB.S	1 + mv	2+mv+da
FMPY//FADDSUB.X	1 + mv	2+mv+da
FMPY//FSUB.S	1 + mv	2+mv+da
FMPY//FSUB.X	1 + mv	2+mv+da
FMPY.S	1 + mv	2+mv+da
FMPY.X	1 + mv	2+mv+da
FNEG.S	1 + mv	2+mv+da
FNEG.X	1 + mv	2+mv+da
FSCALE.S	1 + mv	2+mv+da
FSCALE.X	1 + mv	2+mv+da
FSCALE.S #byte	1	2 + da
FSCALE.X #byte	1	2 + da
FSEEDD	1	2 + da
FSEEDR	1	2 + da
FSUB.S	1 + mv	2+mv+da
FSUB.X	1 + mv	2+mv+da
FTFR.S	1 + mv	2+mv+da
FTFR.X	1 + mv	2+mv+da
FTRAPcc	1	10
FTST	1 + mv	2+mv+da
GETEXP	1 + mv	2+mv+da
IFcc	1	2 + da
IFcc.U	1	2 + da
ILLEGAL	1	8
INC	1 + mv	2 + mv
INT	1 + mv	2+mv+da
INTRZ	1 + mv	2+mv+da
INTU	1 + mv	2+mv+da
INTURZ	1 + mv	2+mv+da

Figure A-7 Instruction Timing Summary (Continued)

Jcc	1 + ea	4 + jx
JCLR	2	6 + jx
JMP	1 + ea	4 + jx
JOIN	1 + mv	2 + mv
JOINB	1 + mv	2 + mv
JScc	1 + ea	4 + jx
JSCLR	2	6 + jx
JSET	2	6 + jx
JSR	1 + ea	4 + jx
JSSET	2	6 + jx
LEA	1 + ea	4 + le
LRA	1 + lr	4 + lr
LSL	1 + mv	2 + mv
LSL #shift	1	2
LSR	1 + mv	2 + mv
LSR #shift	1	2
MOVE	1 + mv	2 + mv
MOVEC	1 + ea	2 + mvc
MOVEI	1	2
MOVEM	1 + ea	6 + mvm
MOVEP	1 + ea	2 + mvp
MOVES	1 + ea	2 + mvs
MOVETA	1 + mv	2 + mv
MPYS	1 + mv	2 + mv
MPYU	1 + mv	2 + mv
NEG	1 + mv	2 + mv
NEGC	1 + mv	2 + mv
NOP	1	2
NOT	1 + mv	2 + mv
OR	1 + mv	2 + mv
ORC	1 + mv	2 + mv
ORI	1	2
REP	1	4 + mv
RESET	1	4
ROL	1 + mv	2 + mv
ROR	1 + mv	2 + mv
RTI	1	4 + rx
Mnemonic	**Words**	**Cycles**

Figure A-7 Instruction Timing Summary (Continued)

RTR	1	4 + rx	
RTS	1	4 + rx	
SETW	1 + mv	2 + mv	
SPLIT	1 + mv	2 + mv	
SPLITB	1 + mv	2 + mv	
STOP	1	n/a	Note 1
SUB	1 + mv	2 + mv	
SUBC	1 + mv	2 + mv	
TFR	1 + mv	2 + mv	
TRAPcc	1	10	
TST	1 + mv	2 + mv	
WAIT	1	n/a	Note 2

Figure A-7 Instruction Timing Summary (Continued)

Note 1: The STOP instruction disables all internal clocks.

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

A.9.1 Data ALU Operation Timing Summary

All Data ALU operations require only one instruction word. The actual number of words may be more than one due to the parallel move specified with the Data ALU operation; this is indicated by the term "+mv" which can be obtained from Figure A-9 . The number of cycles required for execution is also affected by the parallel move operation, and the values for the term "+mv" are listed in Figure A-9. The values for the term "+da" are listed in Figure A-8 for Data ALU operations when the IEEE mode is selected. In the Flush-to-Zero mode, the term "+da" is always zero.

Data ALU Operation	+da Cycles

(IEEE Mode)	+ da Cycles	worst case	Comments
FABS.S	das	6	Worst case: res=1, den=1
FABS.X	dax	4	Worst case: den=1
FADD.S	das	8	Worst case: res=1, den=2
FADD.X	dax	6	Worst case: den=2
FADDSUB.S	das	10	Worst case: res=2, den=2
FADDSUB.X	dax	6	Worst case: den=2
FCLR	0	0	
FCMP	dax	6	Worst case: den=2
FCMPG	dax	6	Worst case: den=2
FCMPM	dax	6	Worst case: den=2
FCOPYS.S	das	8	Worst case: res=1, den=2
FCOPYS.X	dax	6	Worst case: den=2
FFcc	daff	n/a	
FFcc.U	daff	n/a	
FGETMAN	dax	4	Worst case: den=1
FINT	dax	4	Worst case: den=1
FLOAT.S	0	0	
FLOAT.X	0	0	
FLOATU.S	0	0	
FLOATU.X	0	0	
FLOOR	dax	4	Worst case: den=1
FMPY//FADD.S	dams	14	Worst case: res=2, den=4
FMPY//FADD.X	damx	12	Worst case: res=1, den=4
FMPY//FADDSUB.S	dams	16	Worst case: res=3, den=4
FMPY//FADDSUB.X	damx	12	Worst case: res=1, den=4
FMPY//FSUB.S	dams	14	Worst case: res=2, den=4
FMPY//FSUB.X	damx	12	Worst case: res=1, den=4
FMPY.S	dam	8	Worst case: res=1, den=2
FMPY.X	dam	8	Worst case: res=1, den=2
FNEG.S	das	6	Worst case: res=1, den=1

Figure A-8 Data ALU Operation Timing Summary

Data ALU Operation (IEEE Mode)	+ da Cycles	+da Cycles worst case	Comments
FNEG.X	dax	4	Worst case: den=1
FSCALE.S	dam	6	Worst case: res=1, den=1
FSCALE.X	dam	6	Worst case: res=1, den=1
FSEEDD	dam	6	Worst case: res=1, den=1
FSEEDR	dam	4	Worst case: res=0 den=1
FSUB.S	das	8	Worst case: res=1, den=2
FSUB.X	dax	6	Worst case: den=2
FTFR.S	das	6	Worst case: res=1, den=1
FTFR.X	dax	4	Worst case: den=1
FTST	dax	4	Worst case: den=1
GETEXP	dam	4	Worst case: den=1
IFcc	daff	n/a	
IFcc.U	daff	n/a	
INT	dax	4	Worst case: den=1
INTRZ	dax	4	Worst case: den=1
INTU	dax	4	Worst case: den=1
INTURZ	dax	4	Worst case: den=1

Figure A-8 Data ALU Operation Timing Summary (Continued)

where

dam = 2 * (res + i * (1 + den)) clock cycles

res= number of de/unnormalized results.
den= number of source operands with U-tag or V-tag set.
i = 0, if den=0; 1 otherwise.

dams= 2 * (res + i * (1 + den)) clock cycles

res= number of multiplier and add/sub de/unnormalized results.
den= number of multiplier source operands with U-tag or V-tag set +
number of add/sub source operands with U-tag set.
i = 0, if den=0; 1 otherwise.

damx= 2 * (res + i * (1 + den)) clock cycles

res = number of multiplier de/unnormalized results.
den = number of multiplier source operands with U-tag or V-tag set +
number of add/sub source operands with U-tag set.
i = 0, if den=0; 1 otherwise.

das = 2 * (res + i * (1 + den)) clock cycles

res = number of de/unnormalized results.
den = number of de/unnormalized source operands (U-tag set).
i = 0, if den=0; 1 otherwise.

dax = 2 * i * (1 + den) clock cycles

den = number of de/unnormalized source operands (U-tag set).
i = 0, if den=0; 1 otherwise.

daff If the accompanying Data ALU operation is a Data ALU NOP (MOVE or MOVETA) then the "+da" term will be zero. Otherwise the "+da" term will be determined by the Data ALU operation. If the specified condition is true, the "+da" term is as specified in Figure A-8 for the Data ALU operation. If the specified condition is false, the "+da" term is calculated as in the figure but always setting res=0.

A.9.2 Parallel Data Move Timing Summary

Parallel Move Operation		+ mv Words	+ mv Cycles	Comments
	No Parallel Data Move	0	0	
R	Register to Register	0	0	
U	Address Reg. Update	0	0	
X:	X Memory Move	ea	ea + ax	Note 1
X: R	X Memory and Register	ea	ea + ax	Note 1
Y:	Y Memory Move	ea	ea + ay	Note 1
R Y:	Y Memory and Register	ea	ea + ay	Note 1
L:	Long Memory Move	ea	ea + axy	
X: Y:	XY Memory Move	0	axy	

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

Figure A-9 Parallel Data Move Timing Summary

If there are wait states, (i.e., assumption 4 is not applicable) then to each 1-word instruction timing a "+ap" term should be added and to each 2-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.

A.9.3 MOVEC Timing Summary

MOVEC Operation			+ mvc Cycles	Comments
Register	↔	Register	0	
X Memory	↔	Register	ea + ax	Note 1
Y Memory	↔	Register	ea + ay	Note 1

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

Figure A-10 MOVEC Timing Summary

If there are wait states, (i.e., assumption 4 is not applicable) then to each 1-word instruction timing a "+ap" term should be added and to each 2-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.

A.9.4 MOVEM Timing Summary

MOVEM Operation			+ mvm Cycles	Comments
P Memory	↔	Register	ea + ap	

Figure A-11 MOVEM Timing Summary

If there are wait states, (i.e., assumption 4 is not applicable) then to each 1-word instruction timing a "+ap" term should be added and to each 2-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 Figure A-11 for the P Memory Move entry represents the wait states spent when accessing the program memory during DATA read or write and does not refer to instruction fetches.

A.9.5 MOVEP Timing Summary

MOVEC Operation			+ mvp Cycles	Comments
Register	↔	Peripheral	2 + aio	
X Memory	↔	Peripheral	2 + ea + ax + aio	Note 1
Y Memory	↔	Peripheral	2 + ea + ay + aio	Note 1
P Memory	↔	Peripheral	4 + ea + ap + aio	

Note: The ax(ay) term does not apply to MOVE IMMEDIATE DATA.

Figure A-12 MOVEP Timing Summary

If there are wait states, (i.e., assumption 4 is not applicable) then to each 1-word instruction timing a "+ap" term should be added and to each 2-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 Figure A-12 for the P Memory Move entry represents the wait states spent when accessing the program memory during DATA read or write and does not refer to instruction fetches.

A.9.6 MOVES Timing Summary

MOVEC Operation			+ mvs Cycles	Comments
Register	↔	Abs. Short Mem.	0	
X Memory	↔	Abs. Short Mem.	2 + ea + ax	Note 1
Y Memory	↔	Abs. Short Mem.	2 + ea + ay	Note 1
P Memory	↔	Abs. Short Mem.	4 + ea + ap	

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

Figure A-13 MOVES Timing Summary

If there are wait states, (i.e., assumption 4 is not applicable) then to each 1-word instruction timing a "+ap" term should be added and to each 2-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 Figure A-13 for the P Memory Move entry represents the wait states spent when accessing the program memory during DATA read or write and does not refer to instruction fetches.

A.9.7 LEA Timing Summary

MOVEC Operation	+ le Cycles	Comments
Update Addressing Modes	0	
Long Displacement	2	

Figure A-14 LEA Timing Summary

If there are wait states, (i.e., assumption 4 is not applicable) then to each 1-word instruction timing a "+ap" term should be added and to each 2-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.

A.9.8 LRA Timing Summary

LRA Operation	+ lr Words	+ lr Cycles
PC Relative Long Displacement	1	2
PC Relative Address Reg.	0	0

Figure A-15 LRA Timing Summary

If there are wait states, (i.e., assumption 4 is not applicable) then to each 1-word instruction timing a "+ap" term should be added and to each 2-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.

A.9.9 Bit Manipulation Timing Summary

Bit Manipulation Operation	+ mvb Cycles	
Bxxx I/O Short	2 * aio	where Bxxx = BCHG, BCLR or BSET
Bxxx Absolute Short	0	where Bxxx = BCHG, BCLR or BSET
Bxxx Register Direct	0	where Bxxx = BCHG, BCLR or BSET
Bxxx X Memory	ea + (2 * ax)	where Bxxx = BCHG, BCLR or BSET
Bxxx Y Memory	ea + (2 * ay)	where Bxxx = BCHG, BCLR or BSET
BTST I/O Short	aio	
BTST Absolute Short	0	
BTST Register Direct	0	
BTST X Memory	ea + ax	
BTST Y Memory	ea + ay	

Figure A-16 Bit Manipulation Timing Summary

If there are wait states, (i.e., assumption 4 is not applicable) then to each 1-word instruction timing a "+ap" term should be added and to each 2-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.

A.9.10 Jump Instructions Timing Summary

Jump Instruction Operation	+ jx Cycles
Jbit I/O Short	aio + (2 * ap)
Jbit Absolute Short	2 * ap
Jbit Register Direct	2 * ap
Jbit X Memory	ea + ax + (2 * ap)
Jbit Y Memory	ea + ay + (2 * ap)
Jxxx	ea + (2 * ap)

where Jbit = JCLR, JSCLR, JSET, JSSET, BRCLR, BSCLR, BRSET and BSSET

Jxxx = Jcc, JMP, JScc, JSR, Bcc, BRA, BScc and BSR

Figure A-17 Jump Instruction Timing Summary

The "ea" term in the Jbit equations refers only to the clock cycles spent in X and Y Data memory accesses to obtain the bit to be tested. The "ea" term in the Jxxx equation refers only to the clock cycles spent while calculating the jump target address.

All one-word jump instructions execute TWO program memory fetches to refill the pipeline and this 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),and 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.

A.9.11 RTI/RTR/RTS Timing Summary

Operation	+ rx cycles
RTI	2 * ap
RTR	2 * ap
RTS	2 * ap

Figure A-18 RTI/RTR/RTS Timing Summary

The term "2 * ap" comes from the two instruction fetches done by the RTS/RTR/RTI instruction to refill the pipeline.

A.9.12 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
Long Displacement	1	4
PC Relative		
Long Displacement	1	2
Short Displacement	0	0
Address Register	0	0
Special		
Immediate Data	1	2
Absolute Address	1	2
Immediate Short Data	0	0
Short Jump Address	0	0
Absolute Short Address	0	0
I/O Short Address	0	0
Implicit	0	0

Figure A-19 Addressing Mode Timing Summary

A.9.13 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	—	—
IO:	—	—	—	Int	—	—	—	0	—
IO:	—	—	—	Ext	—	—	—	wio	—
L: XY:	Int	Int	—	—	—	—	—	—	0
L: XY:	Int	Ext	—	—	—	—	—	—	wy
L: XY:	Ext	Int	—	—	—	—	—	—	wx
L: XY:	Ext	Ext	—	—	—	—	—	—	2+wx+wy

where 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

Figure A-20 Memory Access Timing Summary

A.10 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 sequences as an assembly error. These restrictions are listed below.

A.10.1 Restrictions Near the End of DO Loops

Proper loop operation is guaranteed if no instruction starting at address LA-2, LA-1 or LA specifies the program controller registers SR, SP, SSL, LA, LC or (implicitly) PC as a destination register; or specifies SSH as a source or destination register.

These restricted instructions include:

at LA-2, LA-1 and LA:

DO
BCHG/BCLR/BSET LA, LC, SR, SP, SSH, or SSL
BTST SSH
JCLR/JSET/JSCLR/JSSET SSH
LEA to LA, LC, SR, SP, SSH, or SSL
LRA to LA, LC, SR, SP, SSH, or SSL
MOVEC/M/P/S from SSH
MOVEC/I/M/P/S to LA, LC, SR, SP, SSH, or SSL
ANDI MR
ORI MR

at LA:

any two word instruction
(F)Jcc, JMP, (F)JScc, JSR, (F)Bcc, BRA, (F)BScc, BSR,
LRA, REP, RESET, RTI, RTR, RTS, STOP, WAIT

Other restrictions:

BSR to (LA), if Loop Flag is set
(F)BScc to (LA), if Loop Flag is set
JSR to (LA), if Loop Flag is set
(F)JScc to (LA), if Loop Flag is set
JSCLR to (LA), if Loop Flag is set
JSSET to (LA), if Loop Flag is set
BSCLR to (LA), if Loop Flag is set
BSSET to (LA), if Loop Flag is set

A.10.2 DO and DOR Restrictions

SSH can not be specified as a source register in the DO and DOR instructions:

DO SSH,label
DOR SSH,label

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

BCHG/BCLR/BSET LA, LC, SSH, SSL or SP
LEA to LA, LC, SSH, SSL or SP
LRA to LA, LC, SSH, SSL or SP
MOVEC/I/M/S to LA, LC, SSH, SSL or SP
MOVEC/M/S from SSH

A.10.3 ENDDO Restrictions

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

BCHG/BCLR/BSET LA, LC, SR, SSH, SSL or SP
LEA to LA, LC, SR, SSH, SSL or SP
LRA to LA, LC, SR, SSH, SSL or SP
MOVEC/I/M/S to LA, LC, SR, SSH, SSL or SP
MOVEC/M/S from SSH
ANDI MR
ORI MR

A.10.4 RTI, RTR and RTS Restrictions

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

BCHG/BCLR/BSET SR, SSH, SSL or SP
LEA to SR, SSH, SSL or SP
LRA to SR, SSH, SSL or SP
MOVEC/I/M/S to SR, SSH, SSL or SP
MOVEC/M/S from SSH
ANDI MR, ANDI IER, ANDI ER or ANDI CCR
ORI MR, ORI IER, ORI ER or ORI CCR

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

BCHG/BCLR/BSET SSH, SSL or SP
LEA to SSH, SSL or SP
LRA to SSH, SSL or SP
MOVEC/I/M/S to SSH, SSL or SP
MOVEC/M/S from SSH

A.10.5 SP and SSH/SSL Manipulation Restrictions

In addition to all the above restrictions concerning MOVEC, MOVEP, SP, SSH, and SSL, the following instruction sequences are illegal:

1. BCHG/BCLR/BSET SP
2. MOVEC/M/P/S from SSH or SSL

and

1. MOVEC/I/M/S to SP
2. MOVEC/M/P/S from SSH or SSL

and

1. LEA to SP
2. MOVEC/M/P/S from SSH or SSL

and

1. LRA to SP
2. MOVEC/M/P/S from SSH or SSL

and

1. BCHG/BCLR/BSET SP
2. JCLR/JSET/JSCLR/JSSET SSH or SSL

and

1. MOVEC/I/M/S to SP
2. JCLR/JSET/JSCLR/JSSET SSH or SSL

and

1. LEA to SP
2. JCLR/JSET/JSCLR/JSSET SSH or SSL

and

1. LRA to SP
2. JCLR/JSET/JSCLR/JSSET SSH or SSL

Also, the instruction MOVEC SSH, SSH is illegal.

A.10.6 R, N, and M Register Restrictions

If an address register Rn is the destination of a MOVE instruction, the new contents will not be available for use as an address pointer until the second following instruction.

If an offset register Nn or a modifier register Mn is the destination of a MOVE instruction, the new contents will not be available for use in address calculations until the second following instruction.

From the above definitions, it is clear that if Mn or Nn is the destination of a MOVE instruction, the next instruction may use the corresponding Rn register as an address pointer if using the No Update or the Address Register PC Relative addressing mode (Mn and Nn are ignored).

Also, a MOVE to Nn may be followed by an instruction using Rn as an address pointer if the Long Displacement, Postincrement by 1, Postdecrement by 1, or Predecrement by 1 addressing mode is employed (Nn is ignored).

A.10.7 Fast Interrupt Routines

DO, (F)TRAPcc, STOP, and WAIT may not be used in a fast interrupt routine. All PC Relative instructions (Bcc, BScc, FBcc, FBScc, BRA, BSR, BRCLR, BSCLR, BRSET, BSSET, LRA and DOR) should not be used in fast interrupt routines since the resulting PC Relative address cannot be predicted.

A.10.8 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 a REP instruction:

any two-word instruction
(F)Bcc
BRA
BRCLR
BRSET
(F)BScc
BSCLR
BSR
BSSET
(F)Jcc
JCLR
JMP
JSET
(F)JScc
JSCLR
JSR
JSSET
LRA
REP
RTI
RTS
STOP
(F)TRAPcc
WAIT

APPENDIX B
DSP BENCHMARKS

B.1 DSP96002 STANDARD DSP BENCHMARKS

Program size and instruction cycle counts for the DSP56000/1 are in parentheses on the line following the DSP96002 program size and instruction cycle count.

All floating-point data ALU operations are performed using single precision operations (".s" extension on opcode) rather than in extended precision (".x" extension on opcode). Using only single precision will yield the same exact answers on any other machine using IEEE standard single precision assuming the same operations are used and performed in the same sequence. Using a mixture of extended precision and single precision may produce higher precision results at the expense of not obtaining exact IEEE conformance.

B.1.1 Real Multiply

$c = a * b$

```
                                                          Program  ICycles
                                                          Words
   move                    x:(r0),d4.s   y:(r4),d6.s       1       1
   fmpy.s    d4,d6,d0                                      1       1
   move                    d0.s,x:(r1)                     1       1
                                                          ---     ---
                                              Totals:      3       3
                                                          (3       3)
```

B.1.2 N Real Multiplies

c(I) = a(I) * b(I) , I=1,...,N

```
                                                          Program   ICycles
                                                          Words
   move #aaddr,r0                                            1         1
     move #baddr,r4                                          1         1
     move #caddr,r1                                          1         1
     move             x:(r0)+,d4.s  y:(r4)+,d6.s             1         1
     do #n,end                                               2         3

     fmpy.s  d4,d6,d0   x:(r0)+,d4.s  y:(r4)+,d6.s           1         1
     move             d0.s,x:(r1)+                           1         1
end                                                         ---       ---
                                                  Totals:    8        2N+7
                                                            (8        2N+7)
```

B.1.3 Real Update

d = c + a * b

```
                                                          Program   ICycles
                                                          Words
   move                       x:(r0),d4.s y:(r4),d6.s       1         1
   fmpy.s  d4,d6,d1           x:(r1),d0.s                   1         1
                  fadd.s      d1.s,d0.s                     1         1
   move                       d0.s,x:(r2)                   1         1
                                                           ---       ---
                                                 Totals:    4         4
                                                           (4         4)
```

B.1.4 N Real Updates

d(I) = c(I) + a(I) * b(I), I=1,2,...,N

```
                                                             Program   ICycles
                                                              Words
   move   #aaddr,r0                                             1       1
   move   #baddr,r4                                             1       1
   move   #caddr,r1                                             1       1
   move   #daddr,r5                                             1       1
   move                       x:(r0)+,d4.s y:(r4)+,d6.s 1       1
   fmpy.s   d4,d6,d1          x:(r1)+,d0.s                      1       1
   do     #N,_end                                               2       3
                fadd.s d1,d0  x:(r0)+,d4.s y:(r4)+,d6.s 1       1
   fmpy.s d4,d6,d1            x:(r1)+,d0.s d0.s,y:(r5)+ 1       1
_end                                                           ---     ---
                                                     Totals: 10      2N+9
                                                             (10     2N+9)
```

B.1.5 FIR Filter with Data Shift

N-1

c(n) = SUM {a(I) * b(n-I)}

I=0

```
                                                               Program   ICycles
                                                               Words
    move     #data,r0                                              1       1
   move     #coef,r4                                               1       1
   move     #n-1,m0                                                1       1
   fclr     d1                   m0,m4                             1       1
   movep                         x:input,x:(r0)                    1       2
   fclr     d0                   x:(r0)-,d4.s y:(r4)+,d6.s         1       1
   rep #N                                                          1       2
   fmpy   d4,d6,d1 fadd.s d1,d0  x:(r0)-,d4.s y:(r4)+,d6.s   1       1
                      fadd.s d1,d0  (r0)+  1      1
   movep                           d0.s,x:output                   1       2
                                                                  ---     ---
                                                       Totals:  10      1N+12
                                                                (10     1N+12)
```

B.1.6 Real * Complex Correlation Or Convolution (FIR Filter)

cr(n) + jci(n) = SUM(I=0,...,N-1) {(ar(I) + jai(I)) * b(n-I)}

cr(n) = SUM(I=0,...,N-1) { ar(I) * b(n-I) }

ci(n) = SUM(I=0,...,N-1) { ai(I) * b(n-I) }

```
                                                          Program  ICycles
                                                          Words
        move          #aaddr,r0                              1       1
        fclr   d0     #baddr+n,r4                            1       1
        fclr   d1                    x:(r0),d4.s             1       1
        fclr   d2                    x:(r4)-,d5.s y:(r0)+,d6.s 1     1
        do     #n,end                                        2       3
        fmpy d4,d5,d2  fadd.s d2,d1  x:(r0),d4.s             1       1
        fmpy d6,d5,d2  fadd.s d2,d0  x:(r4)-,d5.s y:(r0)+,d6.s 1     1
end
                       fadd.s d2,d1                          1       1
                                                            ---     ---
                                                   Totals    9     2N+8
                                                           (10     2N+9)
```

B.1.7 Complex Multiply

cr + jci = (ar + jai) * (br + jbi)

cr = ar * br - ai * bi   R1 → cr,ci   R0 → ar,ai R4 → br,bi

ci = ar * bi + ai * br   D5 = ar D6 = bi D4 = br D7 = ai

```
                                                          Program   ICycles
                                                          Words
        move                          x:(r0),d5.s y:(r4),d6.s  1       1
        fmpy.s  d6,d5,d1              x:(r4),d4.s y:(r0),d7.s  1       1
        fmpy.s  d4,d7,d2                                       1       1
        fmpy.s  d4,d5,d0                                       1       1
        fmpy    d6,d7,d2  fadd.s d2,d1                         1       1
                          fsub.s d2,d0         d1.s,y:(r1)     1       1
        move                          d0.s,x:(r1)              1       1
                                                              ---     ---
                                                    Totals:    7       7
                                                              (6       6)
```

B.1.8 N Complex Multiplies

cr(I) + jci(I) = (ar(I) + jai(I)) * (br(I) + jbi(I)), I=1,...,N

cr(I) = ar(I) * br(I) - ai(I) * bi(I)

ci(I) = ar(I) * bi(I) + ai(I) * br(I)

R1 → cr,ci R0 → ar,ai R4 → br,bi

D5 = ar D6 = bi D4 = br D7 = ai

```
                                                            Program  ICycles
                                                            Words
 move   #aaddr,r0                                              1        1
 move   #baddr,r4                                              1        1
 move   #caddr-1,r1                                            1        1
 move                          x:(r0),d5.s  y:(r4),d6.s        1        1
 fmpy.s  d6,d5,d1              x:(r4)+,d4.s y:(r0)+,d7.s       1        1
 fmpy.s  d4,d7,d2                                              1        1
 do     #N,_end                                                2        3
 fmpy    d6,d7,d2 fadd.s d2,d1              y:(r0),d7.s        1        1
 fmpy.s  d4,d5,d0              x:(r0)+,d5.s y:(r4),d6.s        1        1
 fmpy    d6,d5,d1 fsub.s d2,d0  x:(r4)+,d4.s d1.s,y:(r1)       1        1
 fmpy.s  d4,d7,d2              d0.s,x:(r1)+                    1        1
_end                                                          ---      ---
                                                   Totals:    12      4N+9
                                                             (12      4N+9)
```

B.1.9 Complex Update

dr + jdi = (cr + jci) + (ar + jai) $*$ (br + jbi)

dr = cr + ar $*$ br - ai $*$ bi   R0 $\rightarrow$ a   R4 $\rightarrow$ b   R1 $\rightarrow$ c   R $\rightarrow$ d

di = ci + ar $*$ bi + ai $*$ br

```
                                                                    Program   ICycles
                                                                    Words
move                                             y:(r1),d1.s         1         1
move                            x:(r0),d5.s     y:(r4),d6.s         1         1
fmpy.s d6,d5,d2                 x:(r4),d4.s     y:(r0),d7.s         1         1
fmpy d4,d7,d2   fadd.s d2,d1    x:(r1),d0.s                         1         1
fmpy d4,d5,d2   fadd.s d2,d1                                        1         1
fmpy d6,d7,d2   fadd.s d2,d0                    d1.s,y:(r2)         1         1
                fsub.s d2,d0                                        1         1
move                            d0.s,x:(r2)                         1         1
                                                                   ---       ---
                                                         Totals:    8         8
                                                                   (7         7)
```

B.1.10 N Complex Updates

dr(I)+jdi(I) = {cr(I)+jci(I)}+{ar(I)+jai(I)}$*${br(I)+jbi(I)}, I=1,...,N

dr(I) = cr(I) + ar(I) $*$ br(I) - ai(I) $*$ bi(I)

di(I) = ci(I) + ar(I) $*$ bi(I) + ai(I) $*$ br(I)

D5 = ar   D4 = ai   D6 = br   D7 = bi

X Memory Organization			**Y Memory Organization**		
	.			.	
	ci2			di2	
	cr2			dr2	
	ci1			di1	
R1 $\rightarrow$	cr1	CADDR	R5 $\rightarrow$	dr1	DADDR
	.			.	
	.			.	
	ai2			bi2	
	ar2			br2	
R0 $\rightarrow$	ai1			bi1	
	ar1	AADDR	R4 $\rightarrow$	br1	BADDR

```
                                                          Program  ICycles
                                                          Words
    move   #aaddr+1,r0                                       1        1
    move   #3,n0                                             1        1
    move   #baddr,r4                                         1        1
    move   #caddr,r1                                         1        1
    move   #daddr-1,r5                                       1        1
    move                      x:(r0)-,d4.s   y:(r4)+,d6.s    1        1
    fclr   d2                 x:(r0)+n0,d5.s y:(r5),d0.s     1        1
    do #n,end                                                2        3
    fmpy d5,d6,d2 fadd.s d2,d0  x:(r1)+,d1.s   y:(r4)+,d7.s  1        1
    fmpy d4,d7,d2 fadd.s d2,d1  x:(r1)+,d0.s   d0.s,y:(r5)+  1        1
    fmpy d4,d6,d2 fsub.s d2,d1  x:(r0)-,d4.s   y:(r4)+,d6.s  1        1
    fmpy d5,d7,d2 fadd.s d2,d0  x:(r0)+n0,d5.s d1.s,y:(r5)+  1        1
end
                  fadd.s d2,d0                               1        1
    move                                     d0.s,y:(r5)+    1        1
                                                        -------------------
                                                  Totals:  15      4N+12
                                                          (13      4N+10)
```

or

d5 = ar   d4 = br   d6 = bi   d7 = ai

X Memory Organization			Y Memory Organization		
	.			.	
	dr2			di2	
R5 →	dr1	DADDR	R2 →	di1	DADDR
	.			.	
	.			.	
	cr2			ci2	
R1 →	cr1	CADDR	R6 →	ci1	CADDR
	.			.	
	.			.	
	br2			bi2	
R4 →	br1	BADDR	R4 →	bi1	BADDR
	.			.	
	.			.	
	ar2			ai2	
R0 →	ar1	AADDR	R0 →	ai1	AADDR

Program Words   **ICycles**

```
    move  #aaddr,r0                                              1      1
    move  #baddr,r4                                              1      1
    move  #caddr,r1                                              1      1
    move  r1,r6                                                  1      1
    move  #daddr,r5                                              1      1
    move  r5,r2                                                  1      1
    move                         x:(r4),d6.s                     1      1
    move                         x:(r0),d4.s                     1      1
    fmpy.s d4,d6,d2                           y:(r0)+,d5.s       1      1
    fmpy.s d5,d6,d3              x:(r1)+,d0.s y:(r4)+,d7.s       1      1
    fmpy   d5,d7,d2 fadd.s d2,d0  x:(r4),d6.s                    1      1
    do     #N,_end                                               2      3
    fmpy   d4,d7,d2 fsub.s d2,d0  x:(r0),d4.s  y:(r6)+,d1.s      1      1
    fmpy   d4,d6,d2 fadd.s d2,d1  d0.s,x:(r5)+ y:(r0)+,d5.s      1      1
    fmpy   d5,d6,d3 fadd.s d3,d1  x:(r1)+,d0.s y:(r4)+,d7.s      1      1
    fmpy   d5,d7,d3 fadd.s d2,d0  x:(r4),d6.s  d1.s,y:(r2)+      1      1
_end                                                            ---    ---
                                                   Totals:      17     4N+14
                                                               (13     5N+9)
```

B.1.11 Complex Correlation Or Convolution (FIR Filter)

cr(n) + jci(n) = SUM(I=0,...,N-1) { (ar(I) + jai(I)) *

(br(n-I) + jbi(n-I)) }

cr(n) = SUM(I=0,...,N-1) { ar(I) * br(n-I) - ai(I) * bi(n-I) }

ci(n) = SUM(I=0,...,N-1) { ar(I) * bi(n-I) + ai(I) * br(n-I) }

```
                                                          Program   ICycles
                                                          Words
       move    #aaddr,r0                                      1        1
       fclr    d2                 #baddr,r4                   1        1
       fclr    d0                                             1        1
       fclr    d1                 x:(r0),d5.s  y:(r4),d6.s    1        1
       do      #N,end                                         2        3
       fmpy d6,d5,d2 fsub.s d2,d0  x:(r4)+,d4.s y:(r0)+,d7.s  1        1
       fmpy d4,d7,d2 fadd.s d2,d1                             1        1
       fmpy d4,d5,d2 fadd.s d2,d1                             1        1
       fmpy d6,d7,d2 fadd.s d2,d0  x:(r0),d5.s  y:(r4),d6.s   1        1
end
                     fsub.s d2,d0                             1        1
                                                             ---      ---
                                                 Totals:      1 1     4N+8
                                                             (11     4N+8)
```

B.1.12 Nth Order Power Series (Real)

c = SUM (I=0,...,N) { a(I) * b^I } $c = a_N b^N + a_{N-1} b^{N-1} + \ldots + a_1 b^1 + a_0$

```
                                                         Program   ICycles
                                                         Words
    move    #baddr,r4                                       1         1
    move    #aaddr,r0                                       1         1
    move                                 y:(r4),d7.s        1         1
    fclr    d2                x:(r0)+,d0.s y:(r4),d6.s      1         1
    do      #N,end                                          2         3
    fmpy    d6,d7,d1 fadd.s d2,d0  x:(r0)+,d4.s             1         1
    fmpy.s  d6,d4,d2          d1.s,d6.s                     1         1
end
    fadd.s  d2,d0                                           1         1
                                                           ---       ---
                                                 Totals:    9       2N+8
                                                           (9       2N+8)
```

B.1.13 2nd Order Real Biquad IIR Filter

w(n) = x(n) - a1 * w(n-1) - a2 * w(n-2)

y(n) = w(n) + b1 * w(n-1) + b2 * w(n-2)

Input sample in d0.

X Memory Order - w(n-2), w(n-1)

Y Memory Order - a2, a1, b2, b1

```
                                                              Program   ICycles
                                                              Words
  move                              x:(r0)+,d4.s  y:(r4)+,d6.s  1         1
  fmpy.s d6,d4,d2                   x:(r0)-,d5.s  y:(r4)+,d6.s  1         1
  fmpy   d6,d5,d2 fsub.s d2,d0.s d5.s,x:(r0)+   y:(r4)+,d6.s  1         1
  fmpy   d6,d4,d2 fsub.s d2,d0                  y:(r4),d6.s   1         1
  fmpy   d6,d5,d2 fadd.s d2,d0   d0.s,x:(r0)                  1         1
                  fadd.s d2,d0                                1         1
  move                           d0.s,x:output                1         1
                                                             ---       ---
                                                   Totals:    7         7
                                                             (7         7)
```

B.1.14 N Cascaded Real Biquad IIR Filters

$$w(n) = x(n) - a1 * w(n-1) - a2 * w(n-2)$$

$$y(n) = w(n) + b1 * w(n-1) + b2 * w(n-2)$$

X Memory Organization			Y Memory Organization		
				b1N	Coef. + 4N-1
				b2N	
				a1N	
				a2N	
	wN(n-1)	Data + 2N-1		.	
	wN(n-2)			.	
	.			b11	
	.			b21	
	w1(n-1)			a11	
R1,R0 →	w1(n-2)	Data	R4 →	a21	Coef.

DSP56000 IMPLEMENTATION

```
                                                          Program   ICycles
                                                          Words
        move   #$ffffffff,m0                                 2         2
        move   m0,m4                                         1         1
        move   #data,r0                                      2         2
        move   #coef,r4                                      2         2
        movep              x:input,a                         1         2
        move               x:(r0)+,x0    y:(r4)+,y0          1         1
        do      #n,end                                       2         3
        mac     -x0,y0,a   x:(r0)-,x1    y:(r4)+,y0          1         1
        macr    -x1,y0,a   x1,x:(r0)+    y:(r4)+,y0          1         1
        mac     x0,y0,a    a,x:(r0)+     y:(r4)+,y0          1         1
        mac     x1,y0,a    x:(r0)+,x0    y:(r4)+,y0          1         1
end
        rnd     a                                            1         1
        movep              a,x:output                        1         2
                                                      -------------------
                                                      Totals 17     4N+16
```

DSP96002 IMPLEMENTATION

```
                                                          Program Cycles
                                                          Words

    move   #$ffffffff,m0                                   2     2
    move   m0,m4                                           1     1
    move   m0,m1                                           1     1
    move   #data,r0                                        2     2
    move   r0,r1                                           1     1
    move   #coef,r4                                        2     2
    movep                     x:input,d0.s                 1     2
    fclr   d1                 x:(r0)+,d4.s y:(r4)+,d6.s    1     1
    do     #n,end                                          2     3
    fmpy d4,d6,d1 fadd.s d1,d0  x:(r0)+,d5.s y:(r4)+,d6.s  1     1
    fmpy d5,d6,d1 fsub.s d1,d0  d5.s,x:(r1)+ y:(r4)+,d6.s  1     1
    fmpy d4,d6,d1 fsub.s d1,d0  x:(r0)+,d4.s y:(r4)+,d6.s  1     1
    fmpy d5,d6,d1 fadd.s d1,d0  d0.s,x:(r1)+ y:(r4)+,d6.s  1     1
end
                  fadd.s d1,d0                             1     1
    movep                     d0.s,x:output                1     2
                                                          ---   ---
                                                 Totals:  19    4N+18
                                                         (17    4N+16)
```

B.1.15 Fast Fourier Transforms

B.1.15.1 Radix 2 Decimation in Time FFT

```
metr2a      macro      points,data,coef,coefsize
metr2a      ident      1,4
;
;Radix 2 Decimation in Time In-Place Fast Fourier Transform Routine
;
;    Complex input and output data
;        Real data in X memory
;        Imaginary data in Y memory
;    Normally ordered input data
;    Bit reversed output data
;
;    Coefficient lookup table
;        +Cosine value (1/2 cycle) in X memory
```

```
;        +Sine value (1/2 cycle) in Y memory
;    Table size can be i*points/2, i=1,2,...
;
; Macro Call - metr2a   points,data,coef,coefsize
;
;      points     number of points (2 - 2,147,483,648, power of 2)
;      data       start of data buffer
;      coef       start of 1/2 cycle sine/cosine table
;      coefsize   number of table points in sine/cosine table
;                  = i*points/2, i=1,2,...    (1 - 2,147,483,648)
;
;
; ar  ----> |   Radix 2   | ----> ar'
; ai  ----> |  Butterfly  | ----> ai'
; br  ----> |  A'=A+B*Wk  | ----> br'
; bi  ----> |  B'=A-B*Wk  | ----> bi'
;                  ^   ^
;                  |   |
;                  |   |
;               wr  wi
;
; wrk = cosine(k*pi/points) table
; wik = sine(k*pi/points)   table
;
; ar' = ar + (wr*br + wi*bi)
; ai' = ai + (wr*bi - wi*br)
; br' = ar -  wr*br - wi*bi = ar - (wr*br + wi*bi)
; bi' = ai -  wr*bi + wi*br = ai - (wr*bi - wi*br)
;

      move    #points,d1.l
      move    #@cvi(@log(points)/@log(2)+0.5),n1
      move    #data,r2
      move    #coef,m2
      move    #coefsize,d2.l

      move    #0,m6
      move    #-1,m0
      clr     d0         m0,m1
```

```
        inc     d0          m0,m4
        lsr     d2          m0,m5
        move    d2.l,n6

        do      n1,_end_pass
        move    r2,r0
        move    d0.l,n2
        lsr     d1      m2,r6
        dec     d1      d1.l,n0
        move    d1.l,n1
        move    n0,n4
        move    n0,n5
        lea     (r0)+n0,r1
        lea     (r0)-,r4
        lea     (r1)-,r5

        do      n2,_end_grp
        move                                x:(r6)+n6,d9.s y:,d8.s
        move                                x:(r1)+,d6.s   y:,d7.s
        fmpy.s  d8,d7,d3                                   y:(r5),d2.s
        fmpy.s  d9,d6,d0                                   y:(r4),d5.s
        fmpy.s  d9,d7,d1                                   y:(r1),d7.s

        do      n0,_end_bfy
        fmpy    d8,d6,d2 fadd.s    d3,d0  x:(r0),d4.s    d2.s,y:(r5)+
        fmpy    d8,d7,d3 faddsub.s d4,d0  x:(r1)+,d6.s   d5.s,y:(r4)+
        fmpy    d9,d6,d0 fsub.s    d1,d2  d0.s,x:(r4)    y:(r0)+,d5.s
        fmpy    d9,d7,d1 faddsub.s d5,d2  d4.s,x:(r5)    y:(r1),d7.s
_end_bfy
        move                              x:(r0)+n0,d7.s d2.s,y:(r5)+n5
        move                              x:(r1)+n1,d7.s d5.s,y:(r4)+n4
_end_grp
        move    n2,d0.l
        lsl     d0                        n0,d1.l
_end_pass
```

B.1.15.2 Faster Radix 2 Decimation in Time FFT

```
; Complex, Radix 2 Cooley-Tukey Decimation in Time FFT
; This program has not been exhaustively tested and may contain errors.
```

```
;
; Faster FFT using Programming Tricks found in Typical FORTRAN Libraries
;
;      First two passes combined as a four butterfly loop since
;            multiplies are trivial.
;            2.25 cycles internal (4 cycles external) per Radix 2
;            butterfly.
;      Middle passes performed with traditional, triple-nested DO loop.
;            4 cycles internal (8 cycles external) per Radix 2 butterfly
;            plus overhead.  Note that a new pipelining technique is
;            being used to minimize overhead.
;      Next to last pass performed with double butterfly loop.
;            4.5 cycles internal (8.5 cycles external) per Radix 2
;            butterfly.
;      Last pass has separate single butterfly loop.
;            5 cycles internal (9 cycles external) per Radix 2
;            butterfly.
;
;      For 1024 complex points, average Radix 2 butterfly = 3.8 cycles
;      internal and 7.35 cycles external, assuming a single external
;      data bus.
;
;      Because of separate passes, minimum of 32 points using these
;      optimizations.  Approximately 150 program words required.
;      Uses internal X and Y Data ROMs for twiddle factor coefficients
;      for any size FFT up to 1024 complex points.
;
;      Assuming internal program and internal data memory (or two
;      external data buses), 1024 point complex FFT is 1.57 msec at
;      75 nsec instruction rate.  Assuming internal program and
;      external data memory, 1024 point complex FFT is 2.94 msec
;      at 75 nsec instruction rate.
;
; First two passes
;
;      9 cycles internal, 1.77X faster than 4 cycle Radix 2 bfy
;      16 cycles external, 2.0X faster than 4 cycle Radix 2 bfy
;
;      r0 = a pointer in and out
;      r6 = a pointer in
```

```
;       r4 = b pointer in and out
;       r1 = c pointer in and out
;       r5 = d pointer in and out
;       n5 = 2
;

        move      #points,d1.l
        move      #passes,d9.l
        move      #data,d0.l
        move      #coef,m2
        move      #coefsize,d2.l

        lsr       d1         d0.l,r0
        lsr       d1         r0,r2
        add       d1,d0      d1.l,d8.l
        add       d1,d0      d0.l,r4
        add       d1,d0      d0.l,r1
        lsr       d2         d0.l,r5
        lsr       d2         r0,r6
        move      #2,n5
        move      d2.l,n6
        move      #-1,m0
        move      m0,m1
        move      m0,m4
        move      m0,m5
        move      m0,m6

        move                                x:(r0),d1.s
        move                                x:(r1),d0.s
        move                                x:(r5)-,d2.s
        move                                             y:(r5)+,d4.s
        faddsub.s d1,d0                      x:(r4),d5.s
        faddsub.s d5,d2                                   y:(r4),d7.s
;
;       Combine first two passes with trivial multiplies.
;
        do      d8.l,_twopass

        faddsub.s  d0,d2                                   y:(r5),d6.s
        faddsub.s  d7,d6                   d2.s,x:(r0)+  y:(r6)+,d3.s
```

```
        faddsub.s  d1,d7                  d0.s,x:(r4)   y:(r1)+,d2.s
        faddsub.s  d3,d2                  d1.s,x:(r5)-
        faddsub.s  d2,d6                  x:(r0)-,d1.s  d4.s,y:(r5)+n5
        faddsub.s  d3,d5                  x:(r1)-,d0.s  d2.s,y:(r4)+
        faddsub.s  d1,d0                  x:(r5),d2.s   d6.s,y:(r0)+
        ftfr.s     d5,d4                  x:(r4),d5.s   d3.s,y:(r1)
        faddsub.s  d5,d2                  d7.s,x:(r1)+  y:(r4),d7.s
_twopass
        move                                            d4.s,y:(r5)+

;
; Middle passes
;
        tfr     d9,d3      #4,d0.l
        clr     d2         d8.l,d1.l
        sub     d0,d3      d2.l,m6

        do      d3.l,_end_pass
        move    d0.l,n2
        move    r2,r0
        lsr     d1         m2,r6
        dec     d1         d1.l,n0
        dec     d1         d1.l,n1
        move    d1.l,n3
        move    n0,n4
        move    n0,n5
        lea     (r0)+n0,r1
        move    r0,r4
        move    r1,r5
        move                             x:(r6)+n6,d9.s y:,d8.s
        move                                            y:(r1),d7.s
        fmpy.s d8,d7,d3                  x:(r1)+,d6.s
        fmpy.s d9,d6,d0
        fmpy.s d9,d7,d1                                 y:(r1),d7.s
        fmpy   d8,d6,d2  fadd.s    d3,d0  x:(r0),d4.s
        fmpy   d8,d7,d3  faddsub.s d4,d0  x:(r1)+,d6.s

        do      n2,_end_grp

        do      n3,_end_bfy
```

```
      fmpy  d9,d6,d0   fsub.s    d1,d2  d0.s,x:(r4)    y:(r0)+,d5.s
      fmpy  d9,d7,d1   faddsub.s d5,d2  d4.s,x:(r5)    y:(r1),d7.s
      fmpy  d8,d6,d2   fadd      d3,d0  x:(r0),d4.s    d2.s,y:(r5)+
      fmpy  d8,d7,d3   faddsub.s d4,d0  x:(r1)+,d6.s   d5.s,y:(r4)+
_end_bfy
      move      (r1)+n1
      fmpy  d9,d6,d0   fsub.s    d1,d2  d0.s,x:(r4)    y:(r0)+,d5.s
      fmpy  d9,d7,d1   faddsub.s d5,d2  d4.s,x:(r5)    y:(r1),d7.s
      fmpy  d8,d6,d2   fadd.s    d3,d0  x:(r0),d4.s    d2.s,y:(r5)+
      move                              x:(r6)+n6,d9.s y:,d8.s
      fmpy  d8,d7,d3   faddsub.s d4,d0  x:(r1)+,d6.s   d5.s,y:(r4)+
      fmpy  d9,d6,d0   fsub.s    d1,d2  d0.s,x:(r4)    y:(r0)+n0,d5.s
      fmpy  d9,d7,d1   faddsub.s d5,d2  d4.s,x:(r5)    y:(r1),d7.s
      fmpy  d8,d6,d2   fadd.s    d3,d0  x:(r0),d4.s    d2.s,y:(r5)+n5
      fmpy  d8,d7,d3   faddsub.s d4,d0  x:(r1)+,d6.s   d5.s,y:(r4)+n4
_end_grp
      move     n2,d0.l
      lsl      d0      n0,d1.l
_end_pass

;
; next to last pass
;
      move      d0.l,n2
      move      r2,r0
      move      r0,r4
      lea      (r0)+2,r1
      move      r1,r5
      move      m2,r6
      move      #3,n0
      move      n0,n1
      move      n0,n4
      move      n0,n5
      move                              x:(r6)+n6,d9.s  y:,d8.s
      move                                              y:(r1),d7.s
      fmpy.s d8,d7,d3                   x:(r1)+,d6.s
      fmpy.s d9,d6,d0
      fmpy.s d9,d7,d1                                   y:(r1),d7.s
      fmpy   d8,d6,d2  fadd.s    d3,d0  x:(r0),d4.s
      fmpy   d8,d7,d3  faddsub.s d4,d0  x:(r1)+n1,d6.s
```

```
        do      n2,_end_next
        fmpy   d9,d6,d0  fsub.s    d1,d2  d0.s,x:(r4)     y:(r0)+,d5.s
        fmpy   d9,d7,d1  faddsub.s d5,d2  d4.s,x:(r5)     y:(r1),d7.s
        fmpy   d8,d6,d2  fadd.s    d3,d0  x:(r0),d4.s     d2.s,y:(r5)+
        move                              x:(r6)+n6,d9.s  y:,d8.s
        fmpy   d8,d7,d3  faddsub.s d4,d0  x:(r1)+,d6.s    d5.s,y:(r4)+
        fmpy   d9,d6,d0  fsub.s    d1,d2  d0.s,x:(r4)     y:(r0)+n0,d5.s
        fmpy   d9,d7,d1  faddsub.s d5,d2  d4.s,x:(r5)     y:(r1),d7.s
        fmpy   d8,d6,d2  fadd.s    d3,d0  x:(r0),d4.s     d2.s,y:(r5)+n5
        fmpy   d8,d7,d3  faddsub.s d4,d0  x:(r1)+n1,d6.s  d5.s,y:(r4)+n4
_end_next

;
; last pass
;
        move      n2,d0.l
        lsl       d0      r2,r0
        move      d0.l,n2
        move      r0,r4
        lea      (r0)+,r1
        move      r1,r5
        move      m2,r6
        move      #2,n0
        move      n0,n1
        move      n0,n4
        move      n0,n5
        move                              x:(r6)+n6,d9.s  y:,d8.s
        move                                              y:(r1),d7.s
        fmpy.s d8,d7,d3                   x:(r1)+n1,d6.s
        fmpy.s d9,d6,d0
        fmpy.s d9,d7,d1                                   y:(r1),d7.s
        fmpy   d8,d6,d2  fadd.s    d3,d0  x:(r0),d4.s
        move                              x:(r6)+n6,d9.s  y:,d8.s
        fmpy   d8,d7,d3  faddsub.s d4,d0  x:(r1)+n1,d6.s

        do      n2,_end_last
        fmpy   d9,d6,d0  fsub.s    d1,d2  d0.s,x:(r4)     y:(r0)+n0,d5.s
        fmpy   d9,d7,d1  faddsub.s d5,d2  d4.s,x:(r5)     y:(r1),d7.s
        fmpy   d8,d6,d2  fadd.s    d3,d0  x:(r0),d4.s     d2.s,y:(r5)+n5
```

```
        move                              x:(r6)+n6,d9.s  y:,d8.s
        fmpy   d8,d7,d3  faddsub.s d4,d0  x:(r1)+n1,d6.s  d5.s,y:(r4)+n4
_end_last
```

B.1.15.3 Radix 4 Decimation in Frequency FFT

```
fftr4z macro   points,data,coef,table,temp
fftr4z ident   1,1
;
; Radix 4 Decimation in Frequency In-Place FFT Routine
;
;    Complex input and output data
;       Real data in X memory
;       Imaginary data in Y memory
;    Normally ordered input data
;    Digit reversed output data
;    Coefficient lookup table
;       Full cycle sinewave in Y memory
;       Coefficient table can be generated by "sinewave" macro.
;
;
; Macro Call - mfftr4z   points,data,coef,table,temp
;
;       points     number of points (4-16384, power of 4)
;       data       starting address of data buffer
;       coef       starting address of sinewave table
;       table      size of sinewave table
;       temp       starting address of temporary storage area
;
; Cooley-Tukey Radix 4 FFT Algorithm
;
;
; ar,ai                          ar',ai'
; br,bi         Radix 4          br',bi'
; cr,ci         Butterfly        cr',ci'
; dr,di                          dr',di'
;
;
;       t1 = ar + cr
;       t2 = ar - cr
```

```
;
;       t3 = dr + br
;       t4 = dr - br
;
;       t5 = ai + ci
;       t6 = ai - ci
;
;       t7 = bi + di
;       t8 = bi - di
;
;       t9 = t2 + t8
;       t10 = t2 - t8
;

;       t11 = t6 + t4
;       t12 = t6 - t4
;
;       ar’ = t1 + t3
;       t13 = t1 - t3
;
;       ai’ = t5 + t7
;       t14 = t5 - t7
;
;       br’ = t9*wr1 + t11*wi1
;       bi’ = t11*wr1 - t9*wi1
;
;       cr’ = t13*wr2 + t14*wi2
;       ci’ = t14*wr2 - t13*wi2
;
;       dr’ = t10*wr3 + t12*wi3
;       di’ = t12*wr3 - t10*wi3
;
; Address pointers are organized as follows:
;
;       r0 = ar,ai,br,bi pointer        n0 = butterflies per group
;       r1 = wr (cos) pointer           n1 = rotation factor
;       r2 = temp storage pointer       n2 = groups per pass
;       r3 = group index counter        n3 = rotation factor
;       r4 = cr,ci,dr,di pointer        n4 = butterflies per group
;       r5 = wi (sin) pointer           n5 = rotation factor
```

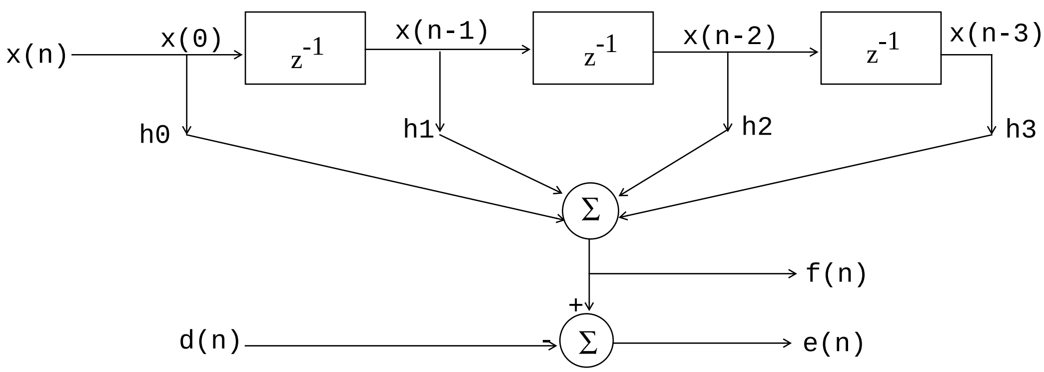
```
;       r6 = temp storage pointer       n6 = not used
;       r7 = not used                   n7 = not used
;
; Alters Data ALU Registers
;       d0      d4      d8
;       d1      d5      d9
;       d2      d6
;       d3      d7
;
; Alters Address Registers
;       r0      n0      m0
;       r1      n1      m1
;       r2      n2      m2
;       r3      n3      m3
;       r4      n4      m4
;       r5      n5      m5
;       r6              m6

;
; Alters Program Control Registers
;       pc      sr
;
; Uses 6 locations on System Stack
;
; This program has not been exhaustively tested and may contain errors.
;
;                                                            ICycles Prog
;                                                            Word    Cycle
    page                    ;
    move    #points/4,n0    ;initialize butterflies per group   2    2
    move    n0,n4           ;            "                      1    1
    move    #1,n2           ;initialize groups per pass         1    1
    move    #1,n3           ;initialize w rotation factor       1    1
    move    #-1,m0          ;initialize linear addressing       1    1
    move    m0,m1           ;            "                      1    1
    move    m0,m2           ;            "                      1    1
    move    m0,m3           ;            "                      1    1
    move    m0,m4           ;            "                      1    1
    move    m0,m5           ;            "                      1    1
    move    m0,m6           ;            "                      1    1
```

```
    move    #temp,r2          ;initialize temp storage pointers   2    2
    move    (r2)+,r6          ;            "                      1    1
    move    #0,r3             ;initialize group index counter     1    1
    move    #coef+table/4,r1  ;initialize wr (cos) pointer        2    2
    move    #coef,r5          ;initialize wi (sin) pointer        2    2
;
; Perform all FFT passes with triple nested DO loops
;
    do      #@cvi(@log(points)/@log(4)+0.5),_end_pass             2    3
    move    #data,r4                                              2    2

    do      n2,_end_grp                                           2    3
    move    r3,n5          ;update rotation factor                1    1
    move    n5,n1          ;            "                         1    1
    move    (r5)+n5        ;point at wi1                          1    1
    move    (r1)+n1        ;point at wr1                          1    1
    move    (r4)+n4                                               1    1
    move    r4,r0          ;point at B data (br,bi)               1    1
    move    (r4)+n4                                               1    1
    move    (r4)+n4        ;point at D data (dr,di)               1    1


    do      n0,_end_bfy                                           2    3
    move                            x:(r4),d0.s                   1    1
    move                            x:(r0),d7.s  y:(r4),d2.s      1    1
    faddsub.s d0,d7                              y:(r0)-n0,d5.s   1    1
    faddsub.s d5,d2                 x:(r0),d1.s                   1    1
    move                            x:(r4)-n4,d4.s                1    1
    move                            x:(r4),d4.s  y:(r0),d6.s      1    1
    faddsub.s d1,d4                 d2.s,x:(r2)+ y:(r4),d3.s      1    1
    faddsub.s d1,d5                              y:(r1)+n1,d8.s   1    1
    fmpy   d5,d8,d2  faddsub.s d6,d3             y:(r5)+n5,d9.s   1    1
    fmpy   d5,d9,d3  faddsub.s d6,d0 d1.s,x:(r2)- d3.s,y:(r6)     1    1
    faddsub.s d4,d7                 d0.s,d6.s    d6.s,y:(r2)      1    1
    fmpy.s d6,d9,d0                              y:(r5)+n5,d9.s   1    1
    fmpy   d6,d8,d0  fadd.s  d0,d2               y:(r1)+n1,d8.s   1    1
    fmpy   d4,d8,d3  fsub.s  d3,d0   x:(r2)+,d1.s y:(r6),d5.s     1    1
    faddsub.s d5,d1                 x:(r2)-,d6.s                  1    1
    fmpy.s d4,d9,d1                 d7.s,x:(r0)+n0 d1.s,y:        1    1
    fmpy.s d5,d8,d2                 d2.s,x:(r0)    d0.s,y:        1    1
```

```
    fmpy   d5,d9,d0  fsub.s  d1,d2                y:(r1)-n1,d8.s  1    1
    fmpy   d6,d8,d1  fadd.s  d0,d3                y:(r5)-n5,d9.s  1    1
    fmpy.s d6,d9,d0                  d3.s,x:(r4)  y:(r2),d4.s     1    1
    fmpy.s d4,d8,d3                               d2.s,y:(r4)+n4  1    1
    fmpy   d4,d9,d2  fsub.s  d0,d3                y:(r1)-n1,d8.s  1    1
    fadd.s    d2,d1                               y:(r5)-n5,d9.s  1    1
    move                             d1.s,x:(r4)  d3.s,y:         1    1
_end_bfy
    move    #coef,r5          ;point at wi0                       2    2
    move    #coef+table/4,r1  ;point at wr0                       2    2
    move    #0,r3             ;reset group index counter          1    1
_end_grp
    move    n0,d0.l           ;get butterflies per group          1    1
    lsr     d0.l              ;                                   1    1
    lsr     d0.l    n2,d1.l   ;divide butterflies/group by 4      1    1
    lsl     d1.l    d0.l,n0   ;multiply groups/pass by 4          1    1
    lsl     d1.l    n3,d0.l   ;get w rotation factor              1    1
    lsl     d0.l    d1.l,n2   ;multiply rotation factor by 4      1    1
    lsl     d0.l    n0,n4     ;                                   1    1
    move    d0.l,n3           ;                                   1    1
    move    n0,d1.l           ;check for 1 butterfly per group    1    1
    lsr     d1.l              ;                                   1    1
    jne     skip              ;                                   1    2
    move    #0,n3             ;reset rotation factor - last pass  1    1
skip    nop                   ;                                   1    1
_end_pass                     ;                                  ---  ---
    endm                      ;                                   78   82
```

The speed for 1024 points using a 75ns instruction cycle is 2.72ms, assuming internal program and internal data memory.

B.1.16 LMS ADAPTIVE FILTER



Notation and symbols:

x(n) - Input sample at time n.

d(n) - Desired signal at time n.

f(n) - FIR filter output at time n.

H(n) - Filter coefficient vector at time n. H={h0,h1,h2,h3}

X(n) - Filter state variable vector at time n. X={x0,x1,x2,x3}

u - Adaptation gain.

ntaps - Number of coefficient taps in the filter. For this example, ntaps=4.

Exact LMS Algorithm:

e(n)=d(n)-H(n)X(n) (FIR filter and error)

H(n+1)=H(n)+uX(n)e(n) (Coefficient update)

Delayed LMS Algorithm:

e(n)=d(n)-H(n)X(n) (FIR filter and error)

H(n+1)=H(n)+uX(n-1)e(n-1) (Coefficient update)

In the exact LMS algorithm, the output of the FIR filter is first calculated (f(n)) and then the coefficients are updated using the current error signal and coefficients. In the delayed LMS algorithm, the FIR filter and coefficient update is performed at the same time. The coefficients are updated with the error value and coefficients from the previous sample.

References:

"Adaptive Digital Filters and Signal Analysis", Maurice G. Bellanger Marcel Dekker, Inc. New York and Basel

"The DLMS Algorithm Suitable for the Pipelined Realization of Adaptive Filters", Proc. IEEE ASSP Workshop, Academia Sinica, Beijing, 1986

The sections of code shown describe how to initialize all registers, filter an input sample and do the coefficient update. Only the instructions relating to the filtering and coefficient update are shown as part of the benchmark. Instructions executed only once (for initialization) or instructions that may be user application dependent are not included in the benchmark.

Exact LMS Algorithm

```
ntaps    equ     4
u        equ     .01

         org     x:0
sbuf     ds      ntaps
```

```
        org     y:0
cbuf    ds      ntaps

        org     y:10
dsig    ds      1
xsig    ds      1

        org     p:$50
start
  move      #sbuf,r0            ;point to state buffer
  move      #cbuf,r4            ;point to coefficient buffer
  move      r4,r5               ;extra pointer
  move      #ntaps-1,m0         ;mod on pointers
  move      #ntaps-1,m4
  move      #ntaps-1,m5
  move      #-3,n0              ;final adjustment
  move      #u,d7.s             ;adaptation constant

main
  fclr    d1                              y:xsig,d4.s
  fclr    d0            d4.s,x:(r0)+    y:(r4)+,d5.s
  rep     #ntaps
  fmpy    d4,d5,d1  fadd.s d1,d0  x:(r0)+,d4.s  y:(r4)+,d5.s
  fadd.s  d1,d0         x:(r0)-,d4.s    y:(r4)-,d5.s

  move                                  y:dsig,d1.s
  fsub.s  d0,d1
  fmpy.s  d7,d1,d1      x:(r0)+,d4.s

  fmpy.s  d4,d1,d3                      y:(r4)+,d5.s
  fadd.s  d3,d5         x:(r0)+,d4.s
  do      #ntaps,cup
  fmpy.s  d4,d1,d3      d5.s,d0.s       y:(r4)+,d5.s
  fadd.s  d3,d5         x:(r0)+,d4.s    d0.s,y:(r5)+
cup
  move                  x:(r0)+n0,d4.s  y:(r4)-,d0.s
  jmp     main
  end
```

The FIR filter requires 1N/coefficient and the coefficient update requires 2N/coefficient for a total of 3N/coefficient.

On the delayed LMS algorithm, the coefficients are updated with the error from the previous iteration while the FIR filter is being computed for the current iteration. In the following implementation, two coefficients are updated with each pass of the loop.

Delayed LMS Algorithm

```
iter      equ   50              ;Number of LMS iterations
conv_fact equ   0.01            ;Convergence factor

          org   x:$0
state     ds    11              ;State of lms fir

          org   y:$0
coef      ds    10              ;LMS coefficients

e         dc    0.0             ;Signal error
xin       ds    1               ;Input to system
dsig      ds    1               ;Desired signal

          org   p:$100
lmstest
  move   #state,r0          ;Set up address generators
  move   #10,m0
  move   #xstate,r1
  move   #9,m1
  move   #coef,r4
  move   #9,m4
  move   #coef,r5
  move   #9,m5
  move   #xcoef,r6
  move   #9,m6

  move   #iter,d0.l
  do     d0.l,lms
  ; LMS algorithm setup
  move                                    y:e,d0.s
  move                          #conv_fact,d1.s
  fmpy.s d0,d1,d0                         y:xin,d6.s
  move                          d0.s,d9.s
  move                          d6.s,x:(r0)
  ; LMS algorithm loop
```

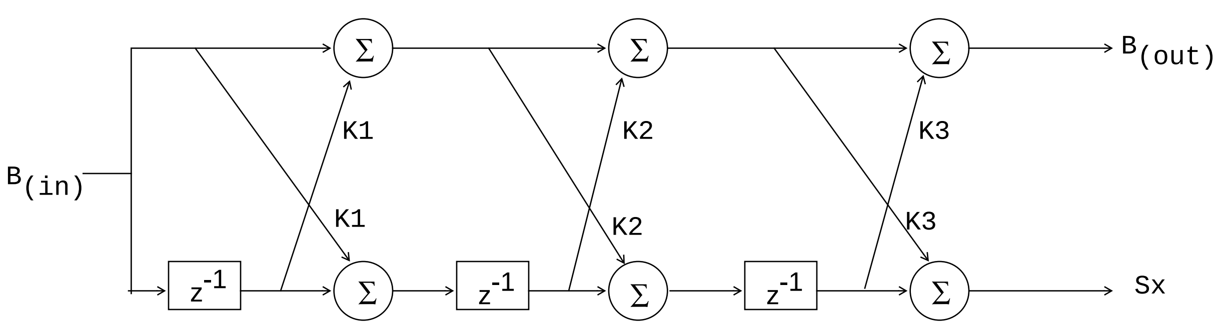
```
  move                              x:(r0)+,d6.s  y:(r4)+,d7.s
  fmpy.s d7,d6,d1                   x:(r0)+,d4.s  y:(r4)+,d5.s
  fmpy.s d9,d4,d2
  fmpy   d5,d4,d0  fadd.s d7,d2  x:(r0)+,d6.s
  do #4,_lms_loop
  fmpy   d9,d6,d3  fadd.s d0,d1                y:(r4)+,d7.s
  fmpy   d7,d6,d0  fadd.s d5,d3  x:(r0)+,d4.s  d2.s,y:(r5)+
  fmpy   d9,d4,d2  fadd.s d0,d1                y:(r4)+,d5.s
  fmpy   d5,d4,d0  fadd.s d7,d2  x:(r0)+,d6.s  d3.s,y:(r5)+
_lms_loop

  fmpy   d9,d6,d3  fadd.s d0,d1                d2.s,y:(r5)+
                   fadd.s d5,d3  (r0)-
  move                                         d3.s,y:(r5)+
  move                                         y:dsig,d2.s
  fsub.s                         d1,d2
  move                                         d2.s,y:e
lms
  nop
  nop
  end
```

The inner loop updates the coefficients and performs the FIR filtering for a speed of 2N per coefficient.

B.1.17 FIR Lattice Filter

N refers to the number of ’k’ coefficients in the lattice filter. Some filters may have other coefficients other than the ’k’ coefficients but their number may be determined from k.

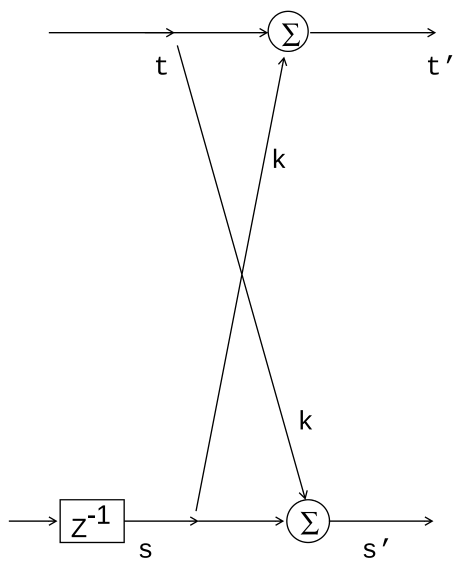
FIR LATTICE FILTER



COEFFICIENT AND STATE VARIABLE STORAGE

```
      R0                  R4
      |                   |
      v                   v
x:    S1 S2 S3 Sx     y:  k1 k2 k3

       M0=3 (mod 4)        M4=2 (mod 3)
```

SINGLE SECTION




```
equations:
t’=s*k+t, t’→t
 s’=t*k+s
```

DSP56000 IMPLEMENTATION

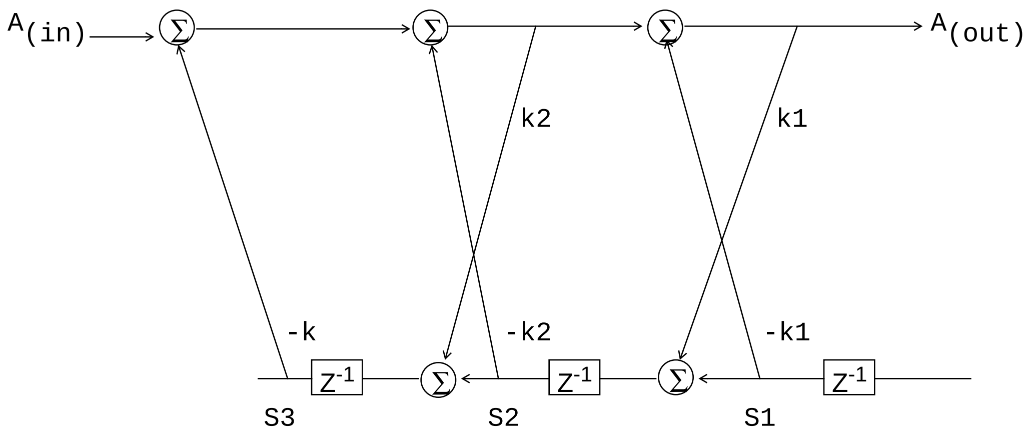
```
                                                                Program  ICycles
                                                                Words
 move      #state,r0     ;point to state variable storage
 move      #N,m0         ;N=number of k coefficients
 move      #k,r4         ;point to k coefficients
 move      #N-1,m4       ;mod for k's

 movep     y:datin,b     ;get input

 move      b,x:(r0)+  y:(r4)+,y0  ;save 1st state, get k    1       1
 do        #N,_elat               ;do each section          2       3
 move          x:(r0),a   b,y1    ;get s, copy t for mul    1       1
 macr y1,y0,a             a,y0    ;t*k+s, copy s            1       1
 macr x0,y0,b  a,x:(r0)+  y:(r4)+,y0 ;s*k+t, sv st, nxt k   1       1
_elat
 move       x:(r0)-,x0 y:(r4)-,y0 ;adj r0,r4 w/dummy loads  1       1

 movep      b,y:datout    ;output sample                  -----   -----
                                                 Totals:    7      3N+5
```

DSP96002 IMPLEMENTATION

```
                                                                Program  ICyc
                                                                Words
 move     #state,r0     ;point to state variable storage
 move     #N,m0         ;N=number of k coefficients
 move     #k,r4         ;point to k coefficients
 move     #N-1,m4       ;mod for k's

 move      y:datin,d5.s  ;get input

 move                     d5.s,x:(r0)+ y:(r4)+,d4.s ;sv s,get k  1    1
 do        #N,_elat                                 ;do filter   2    3
 fmpy d5,d4,d3            x:(r0),d0.s               ;t*k, get s  1    1
 fmpy d0,d4,d1 fadd.s d3,d0                         ;s*k,t*k+s   1    1
               fadd.s d1,d5 d0.s,x:(r0)+ y:(r4)+,d4.s ;s*k+t; s,k 1    1
_elat
 move         x:(r0)-,d0.s y:(r4)-,d7.s ;adj r0,r4 w/dummy loads  1    1

 movep        d5,y:datout            ;output sample
                                                                --- ---
                                                        Totals:  7  3N+5
```

B.1.18 All Pole IIR Lattice Filter

ALL POLE IIR LATTICE FILTER



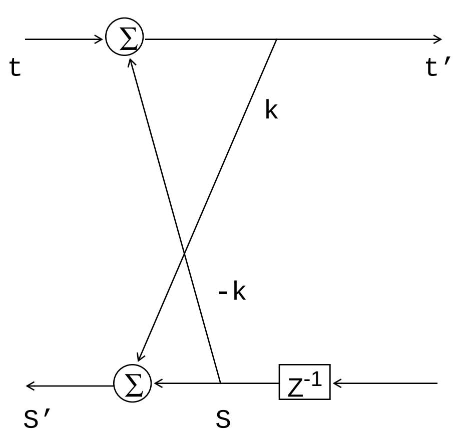
Coefficient And State Variable Storage

R0 R4

x: k1 k2 k3 y: s3 s2 s1

M0=2 (mod 3) M4=2 (mod 3)

SINGLE SECTION



EQUATIONS:

$t' = t - k * s$

$s' = s + k * t'$

$t' \rightarrow t$

DSP56000 IMPLEMENTATION

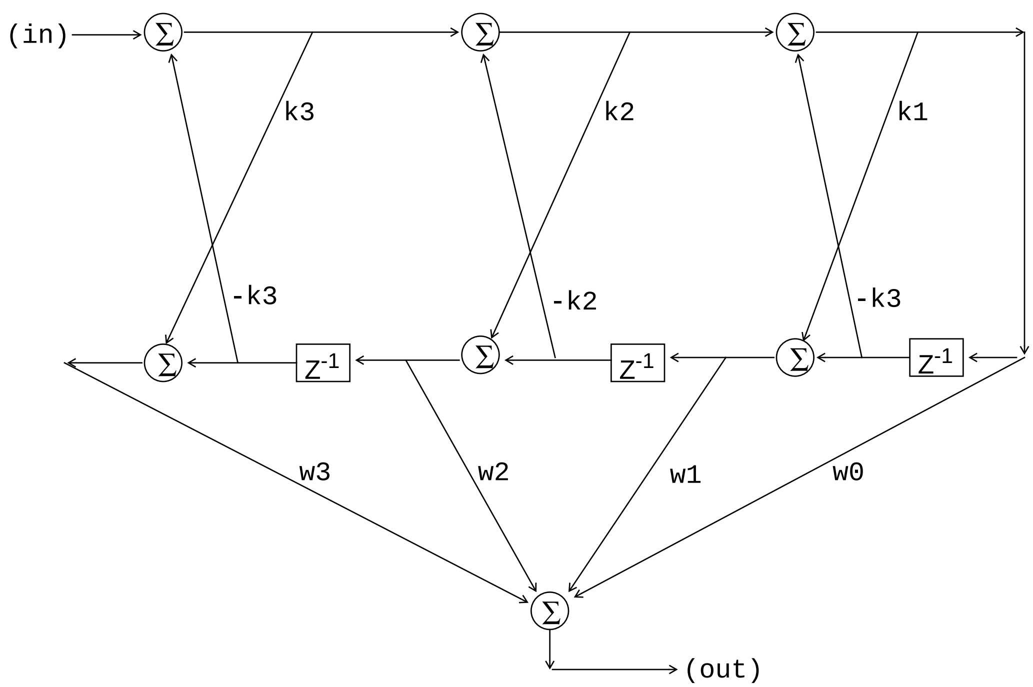
```
                                                        Program   ICycles
                                                        Words
  move      #k+N-1,r0              ;point to k
  move      #N-1,m0                ;number of k's-1
  move      #state,r4              ;point to filter states
  move      m0,m4                  ;mod for states
  movep     y:datin,a              ;get input sample
  move          x:(r0)-,x0 y:(r4)+,y0 ;first k, first s      1       1
  macr -x0,y0,a x:(r0)-,x0 y:(r4)-,y0 ;t'=t-k*s              1       1
  do    #n-1,_endlat                  ;do sections           2       3
  macr -x0,y0,a           b,y:(r4)+  ;t'-k*s, save state     1       1
  move  a,x1              y:(r4)+,b  ;copy t',get s again    1       1
  macr  x1,x0,b x:(r0)-,x0 y:(r4)-,y0 ;fnd s,get s,get k     1       1
_endlat
  move                    b,y:(r4)+  ;save second last s     1       1
  move          x:(r0)+,x0 a,y:(r4)+  ;update r0,save last s 1       1
  movep                   a,y:datout ;output sample
                                                          -----   -----
                                                             9    3N+4
```

DSP96002 IMPLEMENTATION

```
                                                        Program  ICycles
                                                        Words
 move     #k+N-1,r0               ;point to k
 move     #N-1,m0                 ;number of k's-1
 move     #state,r4               ;point to filter states
 move     m0,m4                   ;mod for states
 move     #2,n4                   ;offset for state indexing    1   1

 movep      y:datin,d1            ;get input sample

 move                     x:(r0)-,d5.s y:(r4)+,d6.s            1   1
 fmpy.s d5,d6,d3          x:(r0)-,d5.s y:(r4)-,d6.s            1   1
 fsub.s d3,d1                                                  1   1
 do   #N-1,_elat                                               2   3
 fmpy d5,d6,d0 fadd.s d0,d3                                    1   1
              fsub.s d0,d1 d6.s,d3.s    d3.s,y:(r4)+n4         1   1
 fmpy d5,d1,d0              x:(r0)-,d5.s y:(r4)-,d6.s          1   1
_elat
              fadd.s d0,d3  (r0)+                              1   1
 move                                d3.s,y:(r4)+              1   1
 move                                d1.s,y:(r4)+              1   1

 movep                               d1.s,y:datout
                                                              ---  ---
                                                     Totals:  12   3N+7
```

B.1.19 General Lattice Filter

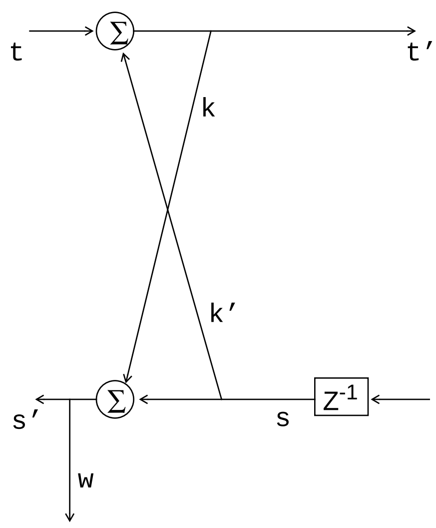
GENERAL LATTICE



COEFFICIENT AND STATE VARIABLE STORAGE

```
     r0                         r4
     |                          |
     v                          v
x: k3 k2 k1 w3 w2 w1 w0    y: s4 s3 s2 s1

   m0=6 (=2*N, mod 7)     m4=3 (=N, mod 4)
```

SINGLE SECTION



EQUATIONS:

t’=t-k*s

s’=s+k*t’

t’→t

output= sum(s’*w)

DSP56000 IMPLEMENTATION

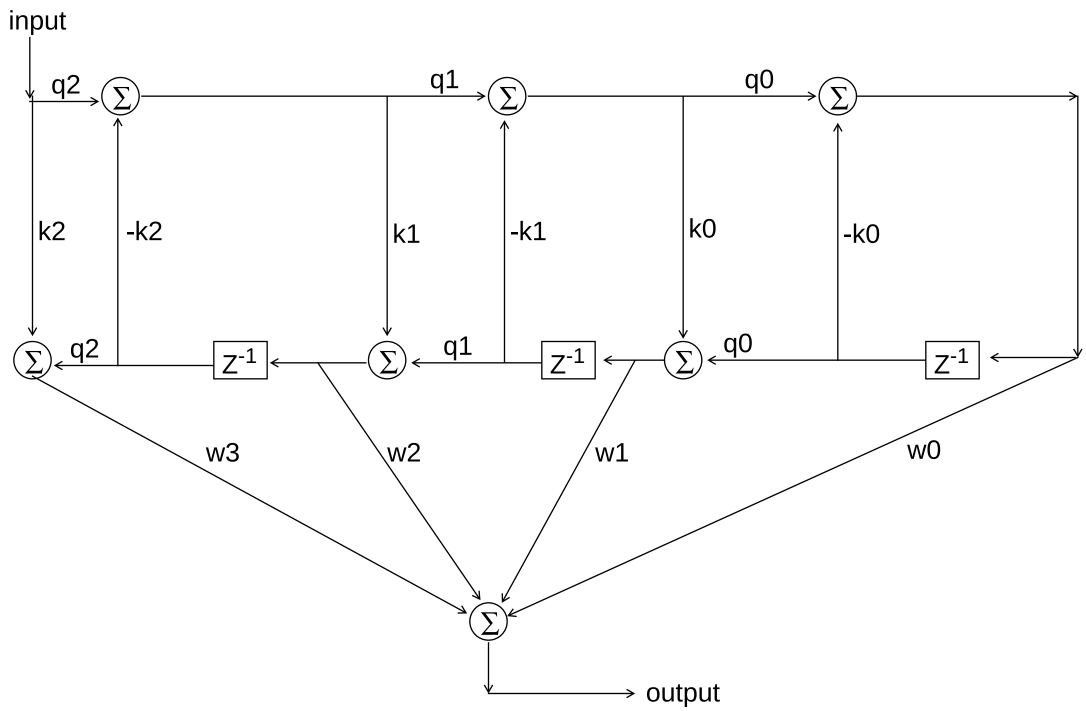
```
                                                                Program  ICycles
                                                                Words
 move   #k,r0             ;point to coefficients
 move   #2*N,m0           ;mod 2*(# of k’s)+1
 move   #state,r4         ;point to filter states
 move   #N,m4             ;mod on filter states
 movep  y:datin,a         ;get input sample
 move          x:(r0)+,x0 y:(r4)-,y0 ;get first k, first s     1    1
 do   #N,_el                         ;do filter                2    3
 macr -x0,y0,a            b,y:(r4)+  ;t-k*s, save prev s       1    1
 move          a,x1       y:(r4)+,b  ;copy t’,get s again      1    1
 macr  x1,x0,b x:(r0)+,x0 y:(r4)-,y0 ;t’*k+s,get k,get s       1    1
_el
 move                     b,y:(r4)+  ;sv scnd to 1st st        1    1
 clr   a                  a,y:(r4)+  ;save first state         1    1
 move                     y:(r4)+,y0 ;get last state           1    1
 rep   #N                                                      1    2
 mac   x0,y0,a x:(r0)+,x0 y:(r4)+,y0 ;do fir taps              1    1
 macr  x0,y0,a              (r4)+    ;finish, adj pointer      1    1
 movep a,y:datout                    ;output sample
                                                             ---   ---
                                                    Totals:  12   4N+10
```

DSP96002 IMPLEMENTATION

```
                                                                   Program   ICycles
                                                                    Words

  move    #k,r0              ;point to coefficients
  move    #2*N,m0            ;mod 2*(# of k's)+1
  move    #state,r4          ;point to filter states
  move    #N,m4              ;mod on filter states

  move p  y:datin,d1          ;get input sample

  move        #2,n4                                                  1        1
  move                         x:(r0)+,d5.s y:(r4)-,d6.s             1        1
  do   #N,_elat                                                      2        3
  fmpy d5,d6,d0  fadd.s d0,d3                                        1        1
                 fadd.s d0,d1 d6.s,d3.s    d3.s,y:(r4)+n4            1        1
  fmpy.s d5,d1,d0              x:(r0)+,d5.s y:(r4)-,d6.s             1        1
 _elat
                 fadd.s d0,d3                                        1        1
  fclr   d0                                 d3.s,y:(r4)+             1        1
  fclr   d1                                 d1.s,y:(r4)+             1        1
  move                                      y:(r4)+,d4.s             1        1
  rep   #N                                                           1        2
  fmpy d5,d4,d0  fadd.s d0,d1 x:(r0)+,d5.s y:(r4)+,d6.s              1        1
                 fadd.s d2,d3               (r4)+                    1        1

  move p  d3.s,y:datout          ;output sample
                                                                    ---      ---
                                                         Totals:    14      4N+12
```

B.1.20 Normalized Lattice Filter

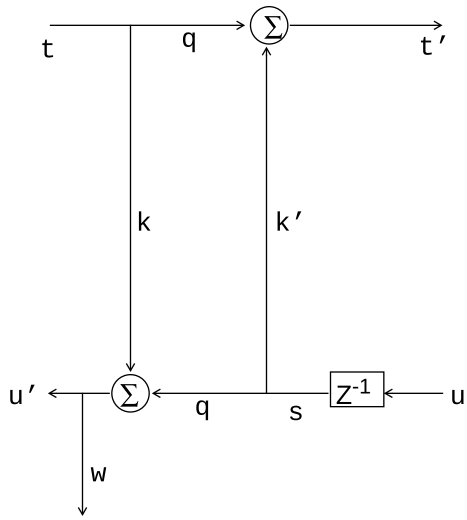
NORMALIZED LATTICE FILTER



COEFFICIENT AND STATE VARIABLE STORAGE

```
   r0                                   r4
   |                                    |
   v                                    v
X: q2 k2 q1 k1 q0 k0 w3 w2 w1 w0    Y: sx s2 s1 s0

   m0=3*N (=9, mod 10)                  m4=N  (=3, mod 4)
```

SINGLE SECTION



EQUATIONS:

t’=t*q-k*s

u’=t*k+s*q

t’→t

output=sum (w*u’)

DSP56000 IMPLEMENTATION

```
                                                                      Program  ICycles
                                                                      Words
 move       #coef,r0        ;point to coefficients
 move       #3*N,m0         ;mod on coefficients
 move       #state,r4       ;point to state variables
 move       #N,m4           ;mod on filter states

 movep      y:datin,y0      ;get input sample

 move           x:(r0)+,x1           ;get first Q in table     1      1
 do    #order,_endnlat                                         2      3
 mpy   x1,y0,a  x:(r0)+,x0 y:(r4),y1 ;q*t, get k, get s        1      1
 macr -x0,y1,a  b,y:(r4)+            ;q*t-k*s, save new s      1      1
 mpy   x0,y0,b             a,y0      ;k*t, set t'              1      1
 macr  x1,y1,b  x:(r0)+,x1           ;k*t+q*s, get next q      1      1
_endnlat
 move                      b,y:(r4)+ ;sv scnd lst st           1      1
 move                      a,y:(r4)+ ;save last state          1      1
 clr   a                   y:(r4)+,y0 ;clr acc, get fst st     1      1
 rep   #order                         ;do fir taps             1      2
 mac   x1,y0,a  x:(r0)+,x1 y:(r4)+,y0                          1      1
 macr  x1,y0,a             (r4)+      ;rnd, adj pointer        1      1

 movep a,y:datout           ;output sample
                                                               ---    ---
                                                       Totals: 13     5N+10
```

DSP96002 IMPLEMENTATION

```
                                                          Program   ICycles
                                                          Words
 move       #coef,r0        ;point to coefficients
 move       #3*N,m0         ;mod on coefficients
 move       #state,r4       ;point to state variables
 move       #N,m4           ;mod on filter states

 move p     y:datin,d5.s    ;get input sample

 move                     x:(r0)+,d6.s ;get q            1      1
 do     #N,_elat                                         2      3
;      t*q       k*w+q*s     get k    get s
 fmpy d5,d6,d2  fadd.s d1,d3 x:(r0)+,d4.s y:(r4)+,d7.s   1      1
;      k*s                           save s
 fmpy.s d4,d7,d0                       d3.s,y:(r4)+      1      1
;      t*k       w*q-k*s
 fmpy d5,d4,d1  fsub.s d0,d2                             1      1
;      q*s         t→t'   get q
 fmpy.s d6,d7,d3             d2.s,d5.s    x:(r0)+,d6.s   1      1
_elat
                fadd.s d1,d3           ;finish last t    1      1
 move                          d3.s,y:(r4)+ ;save 2nd s  1      1
 fclr   d2                     d5.s,y:(r4)+ ;save 1st s  1      1
 fclr   d3                     y:(r4)+,d7.s ;get s       1      1
 rep   #N                                                1      2
 fmpy  d6,d7,d2 fadd.s d2,d3 x:(r0)+,d6.s y:(r4)+,d7.s  ;fir   1      1
                fadd.s d2,d3              (r4)+   ;adj r4     1      1

 move p  d3.s,y:datout
                                                        ---    ---
                                              Totals:  14    5N+11
```

B.1.21 1x3 3x3 and 1x4 4x4 Matrix Multiply

1x3 3x3 Matrix Multiply

```
                                                                   Program  ICycles
                                                                   Words
move         #mat_a,r0       ;point to A matrix
move         #2,m0           ;mod 3
move         #mat_b,r4       ;point to B matrix
move         #-1,m4          ;set for linear addressing
move         #mat_c,r1       ;output C matrix

move                          x:(r0)+,d4.s y:(r4)+,d5.s ;a11,b11 1   1
fmpy.s d4,d5,d3               x:(r0)+,d4.s y:(r4)+,d5.s ;a12,b21 1   1
fmpy.s d4,d5,d0               x:(r0)+,d4.s y:(r4)+,d5.s ;a13,b31 1   1
fmpy   d4,d5,d3 fadd.s d3,d0  x:(r0)+,d4.s y:(r4)+,d5.s ;a11,b12 1   1
fmpy   d4,d5,d3 fadd.s d3,d0  x:(r0)+,d4.s y:(r4)+,d5.s ;a12,b22 1   1
fmpy.s d4,d5,d1               x:(r0)+,d4.s y:(r4)+,d5.s ;a13,b32 1   1
fmpy   d4,d5,d3 fadd.s d3,d1  x:(r0)+,d4.s y:(r4)+,d5.s ;a11,b13 1   1
fmpy   d4,d5,d3 fadd.s d3,d1  x:(r0)+,d4.s y:(r4)+,d5.s ;a12,b23 1   1
fmpy.s d4,d5,d2               x:(r0)+,d4.s y:(r4)+,d5.s ;a13,b33 1   1
fmpy   d4,d5,d3 fadd.s d3,d2               d0.s,y:(r1)+ ;save 1  1   1
                fadd.s d3,d2               d1.s,y:(r1)+ ;save 2  1   1
move                                       d2.s,y:(r1)+ ;save 3  1   1
                                                                --- ---
                                                       Totals:  12  12
```

1x4 4x4 Matrix Multiply

```
                                                                   Program  ICycles
                                                                   Words
move       #mata,r0  ;[1x4] matrix pointer, X memory
move       #matb,r4  ;[4x4] matrix pointer, Y memory
move       #matc,r1  ;output matrix, X memory

move                          x:(r0)+,d4.s y:(r4)+,d7.s ;a11,b11 1   1
fmpy.s d7,d4,d0               x:(r0)+,d3.s y:(r4)+,d7.s ;a12,b21 1   1
fmpy.s d7,d3,d1               x:(r0)+,d5.s y:(r4)+,d7.s ;a13,b31 1   1
fmpy   d7,d5,d1 fadd.s d1,d0  x:(r0)+,d6.s y:(r4)+,d7.s ;a14,b41 1   1
fmpy   d7,d6,d1 fadd.s d1,d0               y:(r4)+,d7.s ;b12     1   1
fmpy   d7,d4,d1 fadd.s d1,d0               y:(r4)+,d7.s ;b22     1   1
fmpy.s d7,d3,d2               d0.s,x:(r1)+ y:(r4)+,d7.s ;b32     1   1
fmpy   d7,d5,d2 fadd.s d2,d1               y:(r4)+,d7.s ;b42     1   1
fmpy   d7,d6,d2 fadd.s d2,d1               y:(r4)+,d7.s ;b13     1   1
```

```
fmpy   d7,d4,d0 fadd.s d2,d1              y:(r4)+,d7.s ;b23     1   1
fmpy.s d7,d3,d2              d1.s,x:(r1)+ y:(r4)+,d7.s ;b33     1   1
fmpy   d7,d5,d2 fadd.s d2,d0              y:(r4)+,d7.s ;b43     1   1
fmpy   d7,d6,d2 fadd.s d2,d0              y:(r4)+,d7.s ;b14     1   1
fmpy   d7,d4,d1 fadd.s d2,d0              y:(r4)+,d7.s ;b24     1   1
fmpy.s d7,d3,d0              d0.s,x:(r1)+ y:(r4)+,d7.s ;b34     1   1
fmpy   d7,d5,d0 fadd.s d0,d1              y:(r4)+,d7.s ;b44     1   1
fmpy   d7,d6,d0 fadd.s d0,d1                                    1   1
                fadd.s d0,d1                                    1   1
move                         d1.s,x:(r1)+                       1   1
                                                               --- ---
                                                       Totals: 19  19
```

B.1.22 NxN NxN Matrix Multiply

The matrix multiplications are for square NxN matrices. All the elements are stored in "row major" format. i.e. for the array A:

a(1,1) ... a(1,N)

a(N,1) ... a(N,N)

the elements are stored:

a(1,1), a(1,2), ..., a(1,N), a(2,1), a(2,2), ..., a(2,N), ...

The following code implements C=AB where A and B are square matrices.

DSP56000 IMPLEMENTATION

```
                                                    Program   ICycles
                                                    Words

move   #mat_a,r0                ;point to A           1          1
move   #mat_b,r4                ;point to B           1          1
move   #mat_c,r6                ;output mat C         1          1
move   #N,n0                    ;array size           1          1
move   n0,n5                                          1          1

do     #N,_rows                 ;do rows              2          3
do     #N,_cols                 ;do columns           2          3
move   r0,r1                    ;copy start of row A  1          1
move   r4,r5                    ;copy start of col B  1          1
clr    a                        ;clear sum and pipe   1          1
move             x:(r1)+,x0 y:(r5)+n5,y0              1          1
```

```
 rep    #N-1                     ;sum                   1        1
 mac    x0,y0,a   x:(r1)+,x0 y:(r5)+n5,y0               1        2
 macr   x0,y0,a   (r4)+          ;finish, next column B 1        1
 move   a,y:(r6)+                ;save output           1        1
_ecols
 move   (r0)+n0                  ;next row A            1        1
 move   #mat_b,r4                ;first element B       1        1
_erows
                                                      -----    -----
                                                        19
                                                                 |
                                ((8+(N-1))N+5)N+8 = N^3 +7*N^2 +5N+8  <-
```

At a DSP56000/1 clock speed of 20.5 MHz, a [10x10][10x10] can be computed in .1715 ms.

DSP96002 IMPLEMENTATION

	Program Words	ICycles

```
   move   #mat_a,r0                ;point to A            1        1
   move   #mat_c,r6                ;output mat C          1        1
   move   #N,n0                    ;array size            1        1
   move   n0,n5                                           1        1

   do     #N,_rows                                        2        3
   move   #mat_b,r4                ;point to B            1        1
   move   r0,r1                    ;copy start of row     1        1
   do     #N,_cols                                        2        3
   move                        r4,r5                      1        1
   fclr   d0                   (r4)+                      1        1
   fclr   d1                   x:(r1)+,d4.s y:(r5)+n5,d5.s 1       1
   rep    #N                                              1        2
   fmpy   d4,d5,d1 fadd.s d1,d0 x:(r1)+,d4.s y:(r5)+n5,d5.s 1      1
                   fadd.s d1,d0 r0,r1                     1        1
   move                                    d0.s,y:(r6)+   1        1
_cols
   move                        (r0)+n0                    1        1
_rows
                                                        -----    -----
                                       Totals:          19
                                                                   |
                                 ((N+7)N+6)N+7 = N^3 +7*N^2 +6N+7  <-
```

At a DSP96002 clock speed of 26.66 MHz, a [10x10][10x10] can be computed in .1325 ms.

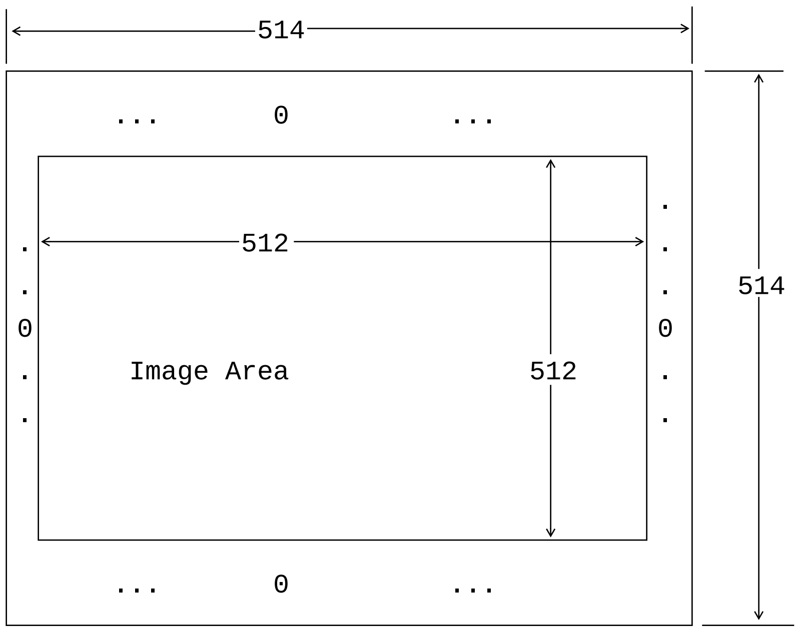
B.1.23 N Point 3x3 2-D FIR Convolution

The two dimensional FIR uses a 3x3 coefficient mask:

c(1,1) c(1,2) c(1,3)
c(2,1) c(2,2) c(2,3)
c(3,1) c(3,2) c(3,3)

Stored in Y memory in the order:
c(1,1), c(1,2), c(1,3), c(2,1), c(2,2), c(2,3), c(3,1), c(3,2), c(3,3)

The image is an array of 512x512 pixels. To provide boundary conditions for the FIR filtering, the image is surrounded by a set of zeros such that the image is actually stored as a 514x514 array. i.e.



The image (with boundary) is stored in row major storage. The first element of the array image(,) is image(1,1) followed by image(1,2). The last element of the first row is image(1,514) followed by the beginning of the next column image(2,1). These are stored sequentially in the array "im" in X memory.

Image(1,1) maps to index 0, image(1,514) maps to index 513, Image(2,1) maps to index 514 (row major storage).

Although many other implementations are possible, this is a realistic type of image environment where the actual size of the image may not be an exact power of 2. Other possibilities include storing a 512x512 image but computing only a 511x511 result, computing a 512x512 result without boundary conditions but throwing away the pixels on the border, etc.

```
r0  →image(n,m)        image(n,m+1)       image(n,m+2)
r1  →image(n+514,m)    image(n+514,m+1)   image(n+514,m+2)
r2  →image(n+2*514,m)  image(n+2*514,m+1) image(n+2*514,m+2)

r4  →FIR coefficients
r5  →output image
```

DSP56000 IMPLEMENTATION

```
                                                                 Program  ICycles
                                                                 Words
  move      #mask,r4            ;point to coefficients             1        1
  move      #8,m4               ;mod 9                             1        1
  move      #image,r0           ;top boundary                      1        1
  move      #image+514,r1       ;left of first pixel               1        1
  move      #image+2*514,r2     ;left of first pixel 2nd row       1        1

  move      #2,n1               ;adjustment for end of row         1        1
  move      n1,n2                                                  1        1

  move      #imageout,r5        ;output image                      1        1

  move      x:(r0)+,x0   y:(r4)+,y0 ;first element, c(1,1)         1        1
  do        #512,_rows                                             2        3
  do        #512,_cols                                             2        3
  mpy  x0,y0,a  x:(r0)+,x0  y:(r4)+,y0 ;c(1,2)                     1        1
  mac  x0,y0,a  x:(r0)-,x0  y:(r4)+,y0 ;c(1,3)                     1        1
  mac  x0,y0,a  x:(r1)+,x0  y:(r4)+,y0 ;c(2,1)                     1        1
  mac  x0,y0,a  x:(r1)+,x0  y:(r4)+,y0 ;c(2,2)                     1        1
  mac  x0,y0,a  x:(r1)-,x0  y:(r4)+,y0 ;c(2,3)                     1        1
  mac  x0,y0,a  x:(r2)+,x0  y:(r4)+,y0 ;c(3,1)                     1        1
  mac  x0,y0,a  x:(r2)+,x0  y:(r4)+,y0 ;c(3,2)                     1        1
  mac  x0,y0,a  x:(r2)-,x0  y:(r4)+,y0 ;c(3,3)                     1        1
  macr x0,y0,a  x:(r0)+,x0  y:(r4)+,y0 ;preload, get c(1,1)        1        1
  move                      a,y:(r5)+  ;output image sample        1        1
_rows
; adjust pointers for frame boundary
  move   x:(r0)+,x0   y:(r5)+,y1 ;adj r0,r5 w/dummy loads          1        1
  move   x:(r1)+n1,x0 y:(r5)+,y1 ;adj r1,r5 w/dummy loads          1        1
  move   (r2)+n2                 ;adj r2                           1        1
  move   x:(r0)+,x0              ;preload for next pass            1        1
_cols
                                                                  ---      ---
                                                                   28
                                    (Kernel=10N),  10N^2 +7N+12    ←
```

```
                                                          Program ICycles
                                                           Words
  move     #mask,r4          ;point to coefficients            1     1
  move     #8,m4             ;mod 9                            1     1
  move     #image,r0         ;top boundary                     1     1
  move     #image+514,r1     ;left of first pixel              1     1
  move     #image+2*514,r2   ;left of first pixel 2nd row      1     1

  move     #2,n1             ;adjustment for end of row        1     1
  move     n1,n2                                               1     1

  move     #imageout,r5      ;output image                     1     1

  move             x:(r0)+,d4.s y:(r4)+,d5.s ;preload, get c(1,1) 1  1
  fmpy.s d4,d5,d0  x:(r0)+,d4.s y:(r4)+,d6.s ;get c(1,2)       1     1
  do    #512,_rows                                             2     3
  do    #512,_cols                                             2     3
  fmpy.s d4,d6,d1              x:(r0)-,d4.s y:(r4)+,d5.s ;c(1,3)  1  1
  fmpy   d4,d5,d0 fadd.s d0,d1 x:(r1)+,d4.s y:(r4)+,d5.s ;c(2,1)  1  1
  fmpy   d4,d5,d0 fadd.s d0,d1 x:(r1)+,d4.s y:(r4)+,d5.s ;c(2,2)  1  1
  fmpy   d4,d5,d0 fadd.s d0,d1 x:(r1)-,d4.s y:(r4)+,d5.s ;c(2,3)  1  1
  fmpy   d4,d5,d0 fadd.s d0,d1 x:(r2)+,d4.s y:(r4)+,d5.s ;c(3,1)  1  1
  fmpy   d4,d5,d0 fadd.s d0,d1 x:(r2)+,d4.s y:(r4)+,d5.s ;c(3,2)  1  1
  fmpy   d4,d5,d0 fadd.s d0,d1 x:(r2)-,d4.s y:(r4)+,d5.s ;c(3,3)  1  1
  fmpy   d4,d5,d0 fadd.s d0,d1 x:(r0)+,d4.s y:(r4)+,d5.s ;c(1,1)  1  1
  fmpy   d4,d5,d0 fadd.s d0,d1 x:(r0)+,d4.s y:(r4)+,d6.s ;c(1,2)  1  1
  move                                      d1.s,y:(r5)+ ;output  1  1
_cols
  move                    x:(r0)+,d4.s y:(r5)+,d7.s ;adj r0,r5  1     1
  move                    x:(r0)+,d4.s y:(r5)+,d7.s ;load,aj r5 1     1
  fmpy.s d4,d5,d0           (r1)+n1                             1     1
  move                      (r2)+n2                             1     1
  move                    x:(r0)+,d4.s          ;load           1     1
_rows
                                                           ----------
                                          Totals:     29

                            (Kernel=10N), 10N^2 +8N+13
```

B.1.24 Table Lookup with Linear Interpolation Between Points

This performs a table lookup and linear interpolation between points in the table. It is assumed that the spacing between the known values (breakpoints) is a constant. No range checking is performed on the input number because it is assumed that previous calculations may have limiting and range checking. This can be used to approximate arbitrary functions given a set of known points (such as digital sine wave generation) or to interpolate linearly between values of a set of data such as an image.

The function to be approximated is shown below:

```
                              o           ¨ known values of function
                       o             o
  Y(i)          o


         o
      ----+------+------+------+------+
 X(i)→   1.0    6.0   11.0   16.0   21.0   ¨ indexes
          ^         / spacing between indexes is INDSPC, 5.0
                       in this example
```

FIRSTINDEX - value of the first index in the table, 1.0
in this example

Given an input value "x", the linearly interpolated value "y" from the tabulated known values is:

$$y = \frac{Y(i+1)-Y(i)}{X(i+1)-X(i)}(x-X(i)) + Y(i)$$

Program Words **ICycles**

```
;
;     Approximate d4=exp(d0) for 1.0 <= x <= 21.0
;
     page     132,60,1,1
     org     x:0
table     dc     2.7182818e+00          ;exp(1.0)
          dc     4.0342879e+02          ;exp(6.0)
          dc     5.9874141e+04          ;exp(11.0)
          dc     8.8861105e+06          ;exp(16.0)
          dc     1.3188157e+09          ;exp(21.0)

     org     p:$50
firstindex equ    1.0          ;value of first table index
```

```
indspc      equu    5.0         ;index spacing
rindspc     equ     1.0/indspc  ;reciprocal of index spacing

  move    #table,n0        ;point to start of table
  move    #firstindex,d6.s ;value of first index
  move    #rindspc,d7.s    ;reciprocal of index spacing

  fsub.s  d6,d0          ;adjust input relative to index       1     1
  fmpy.s  d7,d0,d0       ;reduce range and create index        1     1
  floor   d0,d1          ;get index                            1     1
  int     d1   d1.s,d2.s ;convert index to int,copy int part   1     1
  fsub.s  d2,d0  d1.l,r0 ;x-X(i), get ptr to table             1     1
  nop                    ;clear address ALU pipe               1     1
  move    (r0)+n0        ;point to Y(i)                        1     1
  move    x:(r0)+,d4.s   ;get Y(i)                             1     1
  move    x:(r0),d5.s    ;get Y(i+1)                           1     1
  fsub.s  d4,d5          ;Y(i+1)-Y(i)                          1     1
  fmpy.s  d0,d5,d5       ; *(x-X(i))                           1     1
  fadd.s  d5,d4          ;+Y(i)                                1     1
                                                              ---   ---
                                                     Totals:  12    12
```

B.1.25 Argument Reduction

Argument reduction (AR) is the problem of having a desired floating point number range and an argument that is outside of the range. The argument is placed inside of the desired range by adding or subtracting multiples of the desired number range. Of course, adding and subtracting multiples of a number is inherently slow and requires infinite precision. Some simple methods can be used with some assumptions on the precision of the data and relative argument sizes.

The following program performs AR when the desired range is arbitrary and the input value is arbitrary. This may be used to reduce an angle to the range of -pi to pi.

The following variables are defined:

rmin = range minimum value, -pi in this example

rmax = range maximum value, pi in this example

range = rmax-rmin, 2*pi in this example

o_range = 1.0/range

Assume the input is in d0.

```
rmin      equ  -3.14159
range     equ  2*3.14159
o_range   equ  1.0/range
```

```
                                                              Program  ICycles
                                                              Words
 move           #range,d7.s   ;load desired range
 move           #rmin,d2.s    ;load range min
 move           #o_range,d3.s ;load reciprocal of range

 fadd.s  d2,d0                ;adjust to rmin                 1     1
 fmpy.s  d0,d3,d0             ;scale the input                1     1
 floor   d0,d1                ;get integer part               1     1
 fsub.s  d1,d0                ;get fractional part            1     1
 fmpy.s  d7,d0d0              ;spread out fraction to range   1     1
 fadd.s  d2,d0                ;adjust to rmin                 1     1
                                                             ---   ---
                                                   Totals:    6     6
```

The output is in d0. Note that the constant initialization is not included in the benchmark because it does not need to be executed every time argument reduction is desired and is therefore application dependent.

If the desired range begins at zero (i.e. the desired range is zero to two pi), then the first and last fadd instructions can be deleted for a four cycle argument reduction.

This is one possible method for AR and it is efficient. This method will not work when the argument divided by the result range has no fractional part (in the current precision). This is obvious since it is the fractional part that contains the information relating to how far the scaled argument is in the reduced range. The integer part tells how many times the range has wrapped around. Typically, a good programmer will keep the argument to a few multiples of the desired range. In most practical applications, the argument may exceed the desired range by several integral values. In this case, the presented algorithms work very well. After the final reduced argument has been obtained, any increments should be made from the reduced argument to prevent eventual overflow and maintain maximum precision.

B.1.26 Non-IEEE floating-point Division

The following code segments perform the division of d0/d5. The resulting quotient is in d0. These code segments are used for a fast division without the need to conform to the error checking or error bounds of the IEEE standard.

The code uses a "convergent division" algorithm. The initial seed obtained from the FSEEDD instruction has 8 bits of accuracy. Two iterations of the convergent division algorithm provide approximately 32 bits of accuracy. For more information on the convergent division algorithm, consult "Computer Arithmetic, Principles, Architecture, and Design" by Kai Hwang, 1979, John Wiley and Sons, New York.

Non-IEEE Division Algorithm

```
                                                                Program   ICycles
                                                                 Words
fseedd d5,d4                                                       1        1
fmpy.s d5,d4,d5                       #2.0,d2.s                    2        2
fmpy   d0,d4,d0    fsub.s d5,d2       d2.s,d3.s                    1        1
fmpy.s d5,d2,d5                       d2.s,d4.s                    1        1
fmpy   d0,d4,d0    fsub.s d5,d3                                    1        1
fmpy.s d0,d3,d0                                                    1        1
                                                                  ---      ---
                                                        Totals:    7        7
```

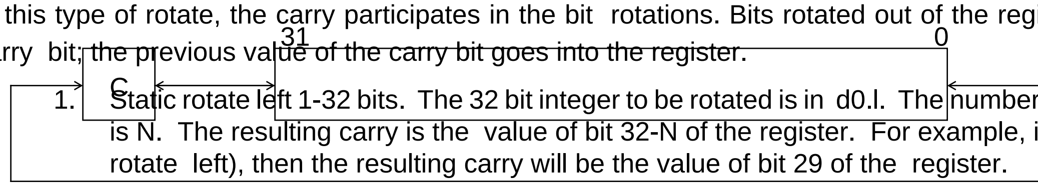
Operation table:

```
                        d0 (dividend)
                                     /
      0.0      number      infinity  /  d5 (divisor)
------------------------------------/
      NaN      NaN         NaN             0.0
      0.0     number      infinity         number
      NaN      NaN         NaN             infinity
```

B.1.27 Multibit Rotates

This describes how to perform multibit rotates using the logical barrel shifts. Both the static case (rotate by a fixed constant) and the dynamic case (rotate by a value in a register) are presented.

The following code assumes a rotating model of the form:



In this type of rotate, the carry participates in the bit rotations. Bits rotated out of the register go into the carry bit; the previous value of the carry bit goes into the register.

1. Static rotate left 1-32 bits. The 32 bit integer to be rotated is in d0.l. The number of bits to rotate is N. The resulting carry is the value of bit 32-N of the register. For example, if N=3 (three bit rotate left), then the resulting carry will be the value of bit 29 of the register.

```
                                                           Program  ICycles
                                                           Words
rol    d0         d0.l,d1.l  ;shift in carry, copy input  1    1
lsl    #N-1,d0               ;shift up, pad with zeros    1    1
lsr    #33-N,d1              ;shift down, set carry       1    1
or     d1,d0                 ;put numbers back together   1    1
```

```
                                                   --- ---
Totals:   4    4
```

2. Static rotate right 1-32 bits. The 32 bit integer to be rotated is in d0.l. The number of bits to rotate is N. The resulting carry is the value of bit N-1 of the register. For example, if N=3 (three bit rotate right), then the resulting carry will be the value of bit 2 of the register.

```
                                                             Program  ICycles
                                                             Words
      ror    d0          d0.l,d1.l  ;shift in carry, copy input 1     1
      lsr    #N-1,d0                ;shift up, pad with zeros   1     1
      lsl    #33-N,d1               ;shift down, set carry      1     1
      or     d1,d0                  ;put numbers back together  1     1
                                    ---  ---
Totals:   4     4
```

3. Dynamic rotate left 0-32 bits. The 32 bit integer to be rotated is in d0.l. The number of bits to rotate is in d2.l. In the following example, the code for checking if the shift count is zero may be eliminated if it is known that the shift count is greater than zero.

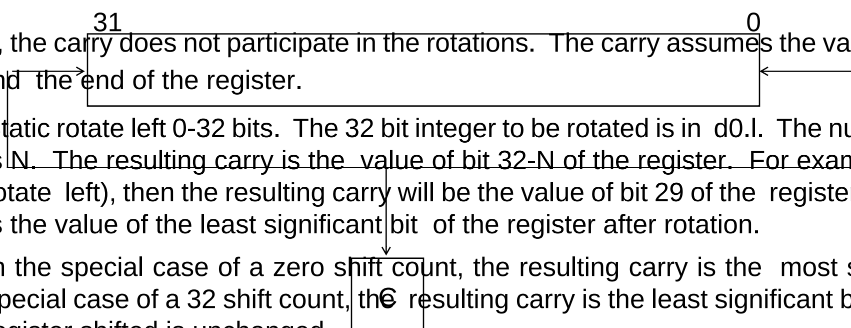
```
                                                             Program  ICycles
                                                             Words

      tst    d2                     ;see if shift count is zero  1    1
      jeq    _done                  ;yes, done                   2    2
      rol    d0          d0.l,d1.l  ;shift in carry, copy input  1    1
      dec    d2          #32,d3.l   ;dec shift count, get 32     2    2
      sub    d2,d3       d2.l,d0.h  ;get 32-shift, move count    1    1
      lsl    d0,d0       d3.l,d1.h  ;shift, move shift count     1    1
      lsr    d1,d1                  ;shift, set carry            1    1
      or     d1,d0                  ;or bits together            1    1
_done                                                           ---  ---
                                                    Totals:     10   10
```

4. Dynamic rotate right 0-32 bits. The 32 bit integer to be rotated is in d0.l. The number of bits to rotate is in d2.l. In the following example, the code for checking if the shift is zero count may be eliminated if it is known that the shift count is greater than zero.

```
                                                   Program ICycles
                                                   Words
tst    d2                  ;see if shift count is zero  1    1      jeq
_done                ;yes, done                  2    2      ror    d0
d0.l,d1.l  ;shift in carry, copy input  1    1     dec    d2        #32,d3.l
;dec shift count, get 32     2    2      sub    d2,d3      d2.l,d0.h  ;get
32-shift, move count    1    1       lsr    d0,d0      d3.l,d1.h  ;shift,
move shift count     1    1       lsl    d1,d1                  ;shift, set
carry          1   1     or    d1,d0             ;or bits together
1    1  _done                                                    ---  ---
```

```
                                                                Totals:   10   10
```

The following code assumes a rotating model of the form:



In this model, the carry does not participate in the rotations. The carry assumes the value of the bit that was rotated around the end of the register.

1. Static rotate left 0-32 bits. The 32 bit integer to be rotated is in d0.l. The number of bits to rotate is N. The resulting carry is the value of bit 32-N of the register. For example, if N=3 (three bit rotate left), then the resulting carry will be the value of bit 29 of the register. The resulting carry is the value of the least significant bit of the register after rotation.

   In the special case of a zero shift count, the resulting carry is the most significant bit. In the special case of a 32 shift count, the resulting carry is the least significant bit. In both cases, the register shifted is unchanged.

Program Words **ICycles**

```
move              d0.l,d1.l  ;copy input          1       1      lsr
#32-N,d0          ;shift first part    1     1     lsl   #N,d1
;shift other part      1       1      or     d1,d0                 ;merge
bits together   1       1
                                                    ---    ---
Totals:  4       4
```

2. Static rotate right 0-32 bits. The 32 bit integer to be rotated is in d0.l. The number of bits to rotate is N. The resulting carry is the value of bit N-1 of the register. For example, if N=3 (three bit rotate right), then the resulting carry will be the value of bit 2 of the register. The resulting carry is the value of the most significant bit of the register after rotation.

   In the special case of a zero shift count, the resulting carry is the least significant bit. In the special case of a 32 shift count, the resulting carry is the most significant bit. In both cases, the register shifted is unchanged.

Program Words **ICycles**

```
move              d0.l,d1.l  ;copy input           1       1      lsl
#32-N,d0          ;shift first part     1     1     lsr   #N,d1
```

```
;shift other part       1      1      or     d1,d0                  ;merge
bits together    1      1
                                    ---   ---
Totals:   4      4
```

3. Dynamic rotate left 0-32 bits. The 32 bit integer to be rotated is in d0.l. The number of bits to rotate is in d2.l.

   In the special case of a zero shift count, the resulting carry is the most significant bit. In the special case of a 32 shift count, the resulting carry is the least significant bit. In both cases, the register shifted is unchanged.

```
                                                                  Program    ICycles
                                                                  Words
move    #32,d1.l             ;get 32                 1      1      sub
d2,d1     d2.l,d1.h  ;32-shift, move shift   1     1      move
d1.l,d0.h  ;move other shift      1     1     lsr    d0,d0     d0.l,d1.l
;shift, copy input      1      1      lsl     d1,d1                 ;shift
other part       1      1      or      d1,d0                 ;merge bits
together    1      1
                                    ---   ---
Totals:   6      6
```

4. Dynamic rotate right 0-32 bits. The 32 bit integer to be rotated is in d0.l. The number of bits to rotate is in d2.l.

   In the special case of a zero shift count, the resulting carry is the least significant bit. In the special case of a 32 shift count, the resulting carry is the most significant bit. In both cases, the register shifted is unchanged.

```
                                                     Program   ICycles
                                                     Words
move    #32,d1.l             ;get 32               1      1      sub
d2,d1     d2.l,d1.h  ;32-shift, move shift   1     1     move
d1.l,d0.h  ;move other shift      1     1     lsl    d0,d0     d0.l,d1.l
;shift, copy input      1      1      lsr     d1,d1               ;shift
other part       1      1      or      d1,d0               ;merge bits
together    1      1
                                           ---   ---
Totals:   6      6
```

B.1.28 Bit Field Extraction/Insertion

The process of bit field extraction is performed on a 32 bit integer in the lower part of a register. A bit field of length FSIZE starting at bit position FOFF is extracted and right justified with zero or sign extension. The value of FSIZE ranges from 1-32 and the field offset ranges from 0-31. Bit field extraction and insertion operations are used in high level languages such as "structures" in C. Both the static case (extraction based on fixed constants) and the dynamic case (extraction based on the values in registers) are given. In the examples, the field to be extracted is in d0.l.

The process of bit field insertion is performed on two 32 bit integer registers. A bit field of length FSIZE from one register is shifted left by an offset FOFF and the field is then inserted into the second register. The field size FSIZE ranges from 1-32 and the field offset from the right of the register ranges from 0-31. For meaningful results, FSIZE+FOFF is less than or equal to 32. The bit field to insert is right justified in the register with zero extension to 32 bits. Both the static case (extraction based on fixed constants) and the dynamic case (extraction based on the values in registers) are given. In the examples, the field in d1.l is inserted into d0.l.

1. Static bit field extraction, zero extend.

```
                                                       Program   ICycles
                                                       Words
lsl    #32-(foff+fsize),d0   ;shift off upper bits    1       1      lsr
#32-fsize,d0          ;right justify           1       1
                                          ---    ---
Totals:  2       2
```

2. Static bit field extraction, sign extend.

```
                                                       Program   ICycles
                                                       Words
lsl    #32-(foff+fsize),d0   ;shift off upper bits    1       1
asr    #32-fsize,d0          ;right justify, sign ext 1       1
                                          ---    ---
Totals:  2       2
```

3. Dynamic bit field extraction, zero extend. Register d1.l contains FOFF, d2.l contains FSIZE.

Program Words **ICycles**

```
move            #32,d3.l    ;register size          1       1      sub
d2,d3                ;32-fsize              1       1      sub    d1,d3
d3.l,d4.h  ;32-fsize-foff, 32-fsize 1      1     move          d3.l,d0.h
;move 32-fsize-foff      1       1      lsl    d0,d0     d4.h,d0.h   ;shift
off upper bits   1      1     lsr   d0,d0            ;right justify
1       1
                                   ---   ---
```

Totals: 6 6

4. Dynamic bit field extraction, sign extend. Register d1.l contains FOFF, d2.l contains FSIZE.

Program Words **ICycles**

```
move            #32,d3.l    ;register size          1       1      sub
d2,d3                ;32-fsize              1       1      sub    d1,d3
d3.l,d4.h  ;32-fsize-foff, 32-fsize 1      1     move          d3.l,d0.h
;move 32-fsize-foff      1       1      lsl    d0,d0     d4.h,d0.h   ;shift
off upper bits   1      1     asr   d0,d0            ;right justify
1       1
                                   ---   ---
```

Totals: 6 6

5. Static bit field insertion.

Program Words **ICycles**

```
move           #-1,d2.l       ;get all ones mask      1      1     lsl
#32-fsize,d2            ;keep field fsize long   1      1      lsr     #32-
(fsize+foff),d2   ;move to insertion    1    1   andc   d2,d0
;clear field             1      1     lsl     #foff,d1              ;move
field to insert    1      1     or      d1,d0                  ;insert bit
field        1      1
                                                      ---    ---
```

Totals: 6 6

6. Dynamic bit field insertion. Register d2.l contains FOFF, d3.l contains FSIZE.

Program Words **ICycles**

```
move          #32,d4.l       ;get 32                  1     1      sub
d3,d4   #-1,d5.l       ;32-fsize, load 1's mask 2     2      sub    d2,d4
d4.l,d5.h     ;32-(fsize+foff)      1    1     lsl   d5,d5  d4.l,d5.h
;shift one's mask up     1     1      lsr    d5,d5                  ;shift
one's mask down   1     1      andc   d5,d0   d2.l,d1.h       ;invert mask
and clear   1    1     lsl    d1,d1                ;move bits to field
1     1      or     d1,d0                  ;insert bit field       1     1
                                    ---  ---
```

Totals: 9 9

7. Static bit field clear.

Program Words **ICycles**

```
move              #-1,d1.l    ;mask of all 1s        1       1      lsr
#32-fsize,d1       ;make 1s size of foff  1     1    lsl   #foff,d1
;align field          1       1      andc   d1,d0                  ;invert
mask and clear 1       1
                                   ---    ---
```

Totals: 4 4

8. Static bit field set.

Program Words **ICycles**

```
move              #-1,d1.l    ;mask of all 1s        1       1      lsr
#32-fsize,d1       ;make 1s size of foff  1     1    lsl   #foff,d1
;align field          1       1      or     d1,d0                  ;clear
field           1       1
                                   ---    ---
```

Totals: 4 4

9. Dynamic bit field clear. Register d1.l contains FOFF, d2.l contains FSIZE.

			Program Words	ICycles
move	#32,d3.l	;register size	1	1
sub	d2,d3			
	#-1,d2.l	;32-fsize, get 1s mask	2	2
move	d3.l,d3.h	;move shift count	1	1
lsr	d3,d2			
	d1.l,d1.h	;trim mask, get foff	1	1
lsl	d1,d2	;align mask	1	1
andc	d2,d0	;invert mask and clear	1	1
			---	---
Totals:			7	7

10. Dynamic bit field set. Register d1.l contains FOFF, d2.l contains FSIZE.

			Program Words	ICycles
move	#32,d3.l	;register size	1	1
sub	d2,d3			
	#-1,d2.l	;32-fsize, get 1s mask	2	2
move	d3.l,d3.h	;move shift count	1	1
lsr	d3,d2			
	d1.l,d1.h	;trim mask, get foff	1	1
lsl	d1,d2	;align mask	1	1
or	d2,d0	;clear bit field	1	1
			---	---
Totals:			7	7

B.1.29 Newton-Raphson Approximation for 1.0/SQRT(x)

The Newton-Raphson iteration can be used to approximate the function:

$$y = \frac{1.0}{\mathrm{sqrt}(x)}$$

by minimizing the function:

$$F(y) = x - \frac{1.0}{y*y}$$

Given an initial approximate value y=1/sqrt(x), the Newton-Raphson iteration for refining the estimate is:

```
y(n+1)=y(n)*(3.0-x*y*y)/2.0
```

Newton-Raphson Approximation of 1.0/SQRT(x)	Program Words	ICycles

```
seedr   d5,d4                        ;y approx 1/sqrt(x)    1        1
fmpy.s  d4,d4,d2       #.5,d7.s  ;y*y, get .5          2        2
fmpy.s  d5,d2,d2       #3.0,d3.s ;x*y*y, get 3.0       2        2
fmpy    d4,d7,d2  fsub.s d2,d3  d3.s,d6.s ;y/2, 3-x*y*y 1        1
fmpy.s  d2,d3,d4       d6.s,d3.s ;y/2*(3-x*y*y)        1        1
fmpy.s  d4,d4,d2                 ;y*y                  1        1
fmpy.s  d5,d2,d2                 ;x*y*y                1        1
fmpy    d4,d7,d2  fsub.s d2,d3  d3.s,d6.s ;y/2, 3-x*y*y 1        1
fmpy.s  d2,d3,d4                 ;y/2*(3-x*y*y)        1        1
                                                       ---      ---
                                              Totals:  11       11
```

B.1.30 Newton-Raphson Approximation for SQRT(x)

The approximation of sqrt(x) may be performed by using the Newton-Raphson iteration to first find 1.0/sqrt(x). The sqrt(x) then can be approximated by x*(1.0/sqrt(x)).

```
Newton-Raphson Approximation                                Program   ICycles
      of SQRT(x)                                            Words

seedr   d5,d4                        ;y approx 1/sqrt(x)    1        1
fmpy.s  d4,d4,d2       #.5,d7.s  ;y*y, get .5          2        2
fmpy.s  d5,d2,d2       #3.0,d3.s ;x*y*y, get 3.0       2        2
fmpy    d4,d7,d2  fsub.s d2,d3  d3.s,d6.s ;y/2, 3-x*y*y 1        1
fmpy.s  d2,d3,d4       d6.s,d3.s ;y/2*(3-x*y*y)        1        1
fmpy.s  d4,d4,d2                 ;y*y                  1        1
fmpy.s  d5,d2,d2                 ;x*y*y                1        1
fmpy    d4,d7,d2  fsub.s d2,d3  d3.s,d6.s ;y/2, 3-x*y*y 1        1
fmpy.s  d2,d3,d4                 ;y/2*(3-x*y*y)        1        1
fmpy.s  d5,d4,d4                 ;x*(1/sqrt(x))        1        1
                                                       ---      ---
                                              Totals:  12       12
```

B.1.31 Unsigned Integer Divide

The unsigned integer divide operation divides two 32 bit unsigned integers. The following code divides d0/d2 with the resulting quotient in d0 and the remainder in d1.

Unsigned 32 Bit Integer Division of d0 = d0/d2				Program Words	ICycles
eor	d1,d1		; clear d1		
do	#32,dloop		;32 quotient bits	2	3
rol	d0		;dividend bit out, q bit in	1	1
rol	d1		;put in temp	1	1
cmp	d2,d1		;check for q bit	1	1
sub	d2,d1	ifcc	;update if less	1	1
dloop					
rol	d0		;last q bit	1	1
not	d0		;complement q bits	1	1
				---	---
			Totals:	8	133

The final remainder is not produced. This program may calculate only the number of quotient bits required and has variable execution time.

Unsigned 32 Bit Integer
Division of d0 = d0/d1, d0>=d1

```
        cmp         d1,d0       d0.l,d2.m
        eor         d0,d0       iflo
        jlo         divdone                 ; divisor > dividend
        bfind       d0,d0       d3.l,d8.l
        jmi         dive2big                ;dividend has
                                            ;32 significant bits
        bfind       d1,d2       d0.h,d0.l   ;find # of quotient bits
        movei       #32,d3
        move                    d2.h,d2.l
        sub         d0,d2       d2.m,d0.l
        inc         d2          d2.l,d2.h   ;compute loop iteration count
        sub         d2,d3
        lsl         d2,d1       d3.l,d2.h   ;align divisor
        do          d2.l,divloop_fast
        cmp         d1,d0                   ;perform test subtract
        sub         d1,d0       ifhs        ;if no borrow, do subtract
        rol         d0                 ;mult remx2, save quo. bit (borrow)
divloop_fast
        not         d0          d8.l,d3.l   ;flip inverted quotient
        lsl         d2,d0                   ;clean off any remainder
        lsr         d2,d0
        jmp         divdone                 ;done
```

```
dive2big    eor         d2,d2
            do          #32,divloop_slow        ;same algorithm as 1st routine
            rol         d0
            rol         d2
            cmp         d1,d2
            sub         d1,d2       ifhs
divloop_slow rol        d0
            not         d0

divdone     end
```

The final quotient is not produced. This program may calculate only the number of quotient bits required and has variable execution time.

Unsigned 32 Bit Integer
Remainder of d0 = d0 rem d1, d0>=d1

```
            cmp         d1,d0       d0.l,d2.m
            jlo         divdone                 ;divisor > dividend
            bfind       d0,d0       #0,d2.l
            jmi         dive2big                ;dividend has
                                                ;32 significant bits
            bfind       d1,d2       d0.h,d0.l   ;find # of remainder bits
            move                    d2.h,d2.l
            sub         d0,d2       d2.m,d0.l
            inc         d2          d2.l,d2.h   ;compute loop count
            lsl         d2,d1       d2.l,d2.h   ;align divisor
            do          d2.l,remloop_fast
            cmp         d1,d0                   ;perform test subtract
            sub         d1,d0       ifhs       ;if no borrow, perform subtract
            rol         d0                      ;adjust remainder
remloop_fast lsr        d2,d0                   ;align remainder
            jmp         remdone                 ;done
dive2big    do          #32,remloop_slow        ;same algorithm as 1st routine
            rol         d0
            rol         d2
            cmp         d1,d2
            sub         d1,d2       ifhs
remloop_slow tfr        d2,d0
remdone     end
```

B.1.32 Signed Integer Divide

The signed integer divide operation divides two 32 bit signed two's complement integers. The divide operation uses a one quadrant restoring divide iteration to divide the operands. The following code divides d5/d2 with the resulting quotient in d0.

Signed 32 Bit Integer Division of d0 = d5/d2 — Program Words / ICycles

```
    eor     d2,d5    d5.l,d0.l  ;determine final sign        1        1
    abs     d2       d0.l,d3.l  ;make divisor positive       1        1
    abs     d0                  ;make dividend positive      1        1
    do      #32,dloop           ;32 quotient bits            2        3
    rol     d0                  ;dividend bit out, q bit in  1        1
    rol     d1                  ;put in temp                 1        1
    cmp     d2,d1               ;check for q bit             1        1
    sub     d2,d1     ifcc      ;update if less              1        1
dloop
    rol     d0                  ;last q bit                  1        1
    not     d0                  ;complement q bits           1        1
    tst     d5                  ;check sign of result        1        1
    neg     d0        iflt      ;negate if needed            1        1
    tst     d3
    neg     dl        iflt
                                                            ---      ---
                                                  Totals:   13       138
```

The final remainder is destroyed in the generation of the quotient. This program may calculate only the number of quotient bits required and has variable execution time.

Signed 32 Bit Integer Division of d0 = d0/d1, d0 >= d1

```
    abs         d1          d1.l,d2.l
    eor         d0,d2
    abs         d0          d2.l,d1.m
    cmp         d1,d0       d0.l,d2.m
    eor         d0,d0       iflo
    jlo         divdone
    bfind       d0,d0       d3.l,d8.l
    bfind       d1,d2       d0.h,d0.l
    movei       #32,d3
    move                    d2.h,d2.l
    sub         d0,d2       d2.m,d0.l
    inc         d2          d2.l,d2.h
    sub         d2,d3
    lsl         d2,d1       d3.l,d2.h
    do          d2.l,divloop_fast
    cmp         d1,d0
    sub         d1,d0       ifhs
    rol         d0
```

```
divloop_fast not        d0          d8.l,d3.l
            lsl         d2,d0
            lsr         d2,d0       d1.m,d2.l
            tst         d2
            neg         d0          ifmi
divdone
```

The final quotient is destroyed in the generation of the remainder. This program calculates only the number of quotient bits required and has variable execution time.

Signed 32 Bit Integer
Remainder of d0 = d0 rem d1, d0 >= d1

```
            abs         d1          d0.l,d2.l
            abs         d2          d0.l,d1.m
            cmp         d1,d2       d2.l,d2.m
            jlo         divdone
            bfind       d2,d0
            bfind       d1,d2       d0.h,d0.l
            move                    d2.h,d2.l
            sub         d0,d2       d2.m,d0.l
            inc         d2          d2.l,d2.h
            lsl         d2,d1       d2.l,d2.h
            do          d2.l,remloop_fast
            cmp         d1,d0
            sub         d1,d0       ifhs
            rol         d0
remloop_fastlsr         d2,d0       d1.m,d2.l
            tst         d2
            neg         d0          ifmi
divdone
```

B.1.33 Graphics Accept/Reject Of Polygons

In graphics applications, checks are made to determine if objects are within a viewing window. Initial checks are made to see if the object can be trivially accepted or trivially rejected. If the object can not be trivially accepted/rejected, then a clipping algorithm is used.

The following code segments perform the trivial accept/reject of a point, line or 4 point polygon within a cube.

B.1.33.1 One Point Accept/Reject

This determines if the point (x,y,z) is within a three-dimensional view cube. If the point is within the cube, the A (accept) bit of the CCR will be set. Single point accept/reject for plotting is useful for plotting of stochastic images such as fractals.

Registers:

d0 = x   d4 = limit
d1 = y   d5 = unused
d2 = z   d6 = unused
d3 = unused   d7 = unused

Memory Map:

X Memory   Y Memory
Xmin   ← r0
Xmax
Ymin
Ymax
Zmin
Zmax

Single Point Accept/Reject

```
                                                          Program   ICycles
                                                          Words
ori    #$80,ccr                       ;set accept bit        1         1
move                  y:(r0)+,d4.s    ;get window minimum    1         1
fcmp   d4,d0          y:(r0)+,d4.s    ;x-Xmin                1         1
fcmp   d0,d4          y:(r0)+,d4.s    ;Xmax-x                1         1
fcmp   d4,d1          y:(r0)+,d4,s    ;y-Ymin                1         1
fcmp   d1,d4          y:(r0)+,d4.s    ;Ymax-y                1         1
fcmp   d4,d2          y:(r0)+,d4.s    ;z-Zmin                1         1
fcmp   d2,d4                          ;Zmax-z                1         1
                                                            ---       ---
                                               Totals:       8         8
```

If the point is within the limits, then the A bit of the CCR is equal to one, otherwise, the point can be rejected.

B.1.33.2 Line Accept/Reject, floating-point Version

This determines if the line from (x0,y0,z0) to (x1,y1,z1) is within a three-dimensional view cube. If the line is within the cube, the A (accept) bit of the CCR will be set. If the line is entirely outside of the cube, then the R bit will be cleared. If the line can not be accepted or rejected, then further processing is required to clip the line where it intersects with a boundary plane.

Registers:

d0 = dimension d4 = unused
d1 = limit   d5 = unused
d2 = unused   d6 = unused
d3 = unused   d7 = unused

Memory Map:

```
            X Memory    Y Memory
(n0=3)  r0 →   x0        Xmin  ← r4
               y0        Xmax
               z0        Ymin
               x1        Ymax
               y1        Zmin
               z1        Zmax
```

```
                                                        Program  ICycles
                                                        Words
ori    #$e0,ccr         ;set accept/reject/overflow bits   1      1
move         x:(r0)+n0,d0.s y:(r4)+,d1.s ;get x0,Xmin      1      1
fcmp  d1,d0  x:(r0)-n0,d0.s              ;x0-Xmin, get x1  1      1
fcmpg d1,d0                 y:(r4)+,d1.s ;x1-Xmin, Xmax    1      1
fcmp  d0,d1  x:(r0)+,d0.s                ;Xmax-x1, get x0  1      1
fcmpg d0,d1  x:(r0)+n0,d0.s y:(r4)+,d1.s ;Xmax-x0, y0,Ymin 1      1
fcmp  d1,d0  x:(r0)-n0,d0.s              ;y0-Ymin, get y1  1      1
fcmpg d1,d0                 y:(r4)+,d1.s ;y1-Ymin, Ymax    1      1
fcmp  d0,d1  x:(r0)+,d0.s                ;Ymax-y1, get y0  1      1
fcmpg d0,d1  x:(r0)+n0,d0.s y:(r4)+,d1.s ;Ymax-y0, z0,Zmin 1      1
fcmp  d1,d0  x:(r0)-n0,d0.s              ;z0-Zmin, get z1  1      1
fcmpg d1,d0                 y:(r4)+,d1.s ;z1-Zmin, Zmax    1      1
fcmp  d0,d1  x:(r0),d0.s                 ;Zmax-z1, get z0  1      1
fcmpg d0,d1                              ;Zmax-z0          1      1
                                                          ---    ---
                                                 Totals:   14     14
```

If the A bit is set, the line can be accepted. If the R bit is cleared, the line can be rejected.

B.1.33.3 Line Accept/Reject, Fixed Point Version

```
                                                        Program  ICycles
                                                        Words
ori    #e0,ccr          ;set accept/reject/infinity bits   1      1
move         x:(r0)+n0,d0.l y:(r4)+,d1.l ;get x0,Xmin      1      1
cmp  d1,d0   x:(r0)-n0,d0.l              ;x0-Xmin, get x1  1      1
cmpg d1,d0                  y:(r4)+,d1.l ;x1-Xmin, Xmax    1      1
cmp  d0,d1   x:(r0)+,d0.l                ;Xmax-x1, get x0  1      1
cmpg d0,d1   x:(r0)+n0,d0.l y:(r4)+,d1.l ;Xmax-x0, y0,Ymin 1      1
cmp  d1,d0   x:(r0)-n0,d0.l              ;y0-Ymin, get y1  1      1
cmpg d1,d0                  y:(r4)+,d1.l ;y1-Ymin, Ymax    1      1
cmp  d0,d1   x:(r0)+,d0.l                ;Ymax-y1, get y0  1      1
cmpg d0,d1   x:(r0)+n0,d0.l y:(r4)+,d1.l ;Ymax-y0, z0,Zmin 1      1
cmp  d1,d0   x:(r0)-n0,d0.l              ;z0-Zmin, get z1  1      1
cmpg d1,d0                  y:(r4)+,d1.l ;z1-Zmin, Zmax    1      1
cmp  d0,d1   x:(r0),d0.l                 ;Zmax-z1, get z0  1      1
cmpg d0,d1                               ;Zmax-z0          1      1
                                                          ---    ---
                                                 Totals:   14     14
```

If the A bit is set, the line can be accepted. If the R bit is cleared, the line can be rejected.

B.1.33.4 Four Point Polygon Accept/Reject

This determines if the polygon consisting of the points (x0,y0,z0), (x1,y1,z1), (x2,y2,z2), (x3,y3,z3) is within a three-dimensional view cube. If the polygon is within the cube, the A (accept) bit of the CCR will be set. If the polygon is entirely outside of the cube, then the R bit will be cleared. If the polygon can not be accepted or rejected, then further processing is required to clip the polygon.

Registers:

d0 = dimension d4 = unused

d1 = limit d5 = unused

d2 = unused d6 = unused

d3 = unused d7 = unused

Memory Map:

```
                X Memory        Y Memory
(n0=3)  r0 →  x0                Xmin ← r4
               y0                Xmax
               z0                Ymin
               x1                Ymax
               y1                Zmin
               z1                Zmax
               x2
               y2
               z2
               x3
               y3
               z3
```

Polygon Accept/Reject

```
                                                                Program  ICycles
                                                                Words
ori    #$e0,ccr           ;set accept/reject/overflow bits   1      1
move         x:(r0)+n0,d0.s y:(r4)+,d1.s ;get x0,Xmin      1      1

fcmp   d1,d0  x:(r0)+n0,d0.s              ;x0-Xmin, get x1  1      1
fcmp   d1,d0  x:(r0)+n0,d0.s              ;x1-Xmin, get x2  1      1
fcmp   d1,d0  x:(r0)-n0,d0.s              ;x2-Xmin, get x3  1      1
fcmpg  d1,d0                y:(r4)+,d1.s ;x3-Xmin, Xmax    1      1
fcmp   d0,d1  x:(r0)-n0,d0.s              ;Xmax-x3, get x2  1      1
fcmp   d0,d1  x:(r0)-n0,d0.s              ;Xmax-x2, get x1  1      1
fcmp   d0,d1  x:(r0)+,d0.s                ;Xmax-x1, get x0  1      1
fcmpg  d0,d1  x:(r0)+n0,d0.s y:(r4)+,d1.s ;Xmax-x0, y0,Ymin 1      1

fcmp   d1,d0  x:(r0)+n0,d0.s              ;y0-Ymin, get y1  1      1
fcmp   d1,d0  x:(r0)+n0,d0.s              ;y1-Ymin, get y2  1      1
fcmp   d1,d0  x:(r0)-n0,d0.s              ;y2-Ymin, get y3  1      1
fcmpg  d1,d0                y:(r4)+,d1.s ;y3-Ymin, ymax    1      1
fcmp   d0,d1  x:(r0)-n0,d0.s              ;Ymax-y3, get y2  1      1
fcmp   d0,d1  x:(r0)-n0,d0.s              ;Ymax-y2, get y1  1      1
```

```
       fcmp  d0,d1  x:(r0)+,d0.s                ;Ymax-y1, get y0  1     1
       fcmpg d0,d1  x:(r0)+n0,d0.s y:(r4)+,d1.s ;Ymax-y0, z0,Zmin 1     1

       fcmp  d1,d0  x:(r0)+n0,d0.s              ;z0-Zmin, get z1  1     1
       fcmp  d1,d0  x:(r0)+n0,d0.s              ;z1-Zmin, get z2  1     1
       fcmp  d1,d0  x:(r0)-n0,d0.s              ;z2-Zmin, get z3  1     1
       fcmpg d1,d0                 y:(r4)+,d1.s ;z3-Zmin, Zmax    1     1
       fcmp  d0,d1  x:(r0)-n0,d0.s              ;Zmax-z3, get z2  1     1
       fcmp  d0,d1  x:(r0)-n0,d0.s              ;Zmax-z2, get z1  1     1
       fcmp  d0,d1  x:(r0)+,d0.s                ;Zmax-z1, get z0  1     1
       fcmpg d0,d1                              ;Zmax-z0          1     1
                                                                 ---   ---
                                                        Totals:  26    26
```

If the A bit is set, the polygon can be accepted, if the R bit is cleared, the polygon can be rejected.

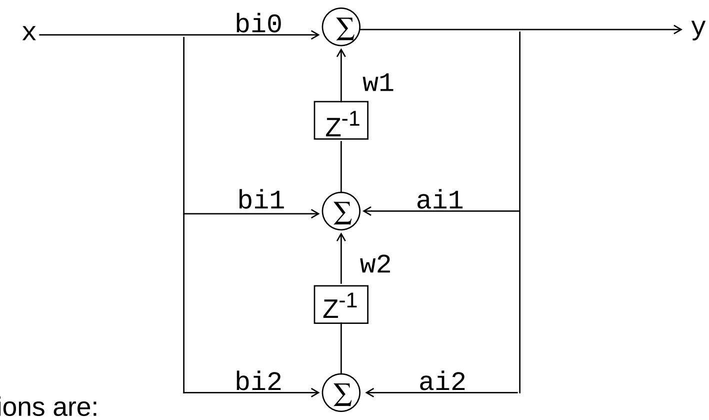
B.1.33.5 Four Point Polygon Accept/Reject (looped)

Polygon Accept/Reject

```
                                                     Program  Icycles
                                                     Words
  ori   #$e0,ccr         ;set accept/reject/overflow bits   1      1
  move         x:(r0)+n0,d0.s y:(r4)+,d1.s ;get x0,Xmin      1      1

  do    #3,clip                                             2      3
  fcmp  d1,d0  x:(r0)+n0,d0.s              ;d0-Dmin, get d1  1      1
  fcmp  d1,d0  x:(r0)+n0,d0.s              ;d1-Dmin, get d2  1      1
  fcmp  d1,d0  x:(r0)-n0,d0.s              ;d2-Dmin, get d3  1      1
  fcmpg d1,d0                 y:(r4)+,d1.s ;d3-Dmin, Dmax    1      1
  fcmp  d0,d1  x:(r0)-n0,d0.s              ;Dmax-x3, get d2  1      1
  fcmp  d0,d1  x:(r0)-n0,d0.s              ;Dmax-x2, get d1  1      1
  fcmp  d0,d1  x:(r0)+,d0.s                ;Dmax-x1, get d0  1      1
  fcmpg d0,d1  x:(r0)+n0,d0.s y:(r4)+,d1.s ;Dmax-x0, d0,Dmin 1      1
clip                                                        ---    ---
                                                   Totals:  12     26
```

B.1.34 Cascaded Five Coefficient Transpose IIR Filter

The cascaded transpose IIR filter has a filter section:



The filter equations are:

```
    y  = x*bi0 + w1
    w1 = x*bi1 + y*ai1 + w2
    w2 = x*bi2 + y*a2
```

Program Words **ICycles**

```
nsec equ   3

    org    x:0
coef
    dc    .93622314E-04     ;/* section  1 B0 */
    dc    .18724463E-03     ;/* section  1 B1 */
    dc    .19625904E+01     ;/* section  1 A1 */
    dc    .93622314E-04     ;/* section  1 B2 */
    dc    -.96296486E+00    ;/* section  1 A2 */

    dc    .94089162E-04     ;/* section  2 B0 */
    dc    .18817832E-03     ;/* section  2 B1 */
    dc    .19723768E+01     ;/* section  2 A1 */
    dc    .94089162E-04     ;/* section  2 B2 */
    dc    -.97275320E+00    ;/* section  2 A2 */

    dc    .94908880E-04     ;/* section  3 B0 */
    dc    .18981776E-03     ;/* section  3 B1 */
    dc    .19895605E+01     ;/* section  3 A1 */
    dc    .94908880E-04     ;/* section  3 B2 */
    dc    -.98994009E+00    ;/* section  3 A2 */
```

```
    org    y:0
w1    dsm    nsec
w2    dsm    nsec

    org    p:$100
    move   #coef,r0
    move   #5*nsec-1,m0
    move   #w1,r4
    move   #nsec-1,m4
    move   #w2,r5
    move   m4,m5
;
;    input in d7
;
    move   x:(r0)+,d4.s            ;get b0                   1      1
    do     #nsec,tran                                        2      3
    fmpy   d7,d4,d0  fadd.s d1,d2 x:(r0)+,d4.s  y:(r4),d5.s   1      1
    fmpy   d7,d4,d1  fadd.s d5,d0 x:(r0)+,d4.s  y:(r5),d6.s   1      1
    fmpy   d0,d4,d2  fadd.s d6,d1 x:(r0)+,d4.s  d2.s,y:(r5)+  1      1
    fmpy   d7,d4,d2  fadd.s d2,d1 x:(r0)+,d4.s  d0.s,d7.s     1      1
    fmpy.s d0,d4,d1               x:(r0)+,d4.s  d1.s,y:(r4)+  1      1
tran
                     fadd.s d1,d2                             1      1
    move   d2.s,y:(r5)+                                       1      1
    move   d0.s,y:$ffff
                                                             ---    ---
                                                  Totals:    10    5N+6
```

B.1.35 3-Dimensional Graphics Illumination

Illumination of objects in three dimensions consists of light from three sources: diffuse lighting from a point source, ambient light and specular lighting. Specular lighting is caused by an object directly reflecting the illumination source. The following variables describe the illumination process:

- L Direction vector to the point light source L={Lx,Ly,Lz}
- N Direction vector normal to the object N={Nx,Ny,Nz}
- Ip Intensity of the point source
- Kd Diffuse reflection constant 0<= Kd <= 1.0
- Ia Intensity of ambient light
- Ka Ambient reflection constant 0<= Ka <= 1.0

R Direction vector of reflection of the point source from the object R={Rx,Ry,Rz}

V Direction vector from the object to the viewpoint

Ks Specular reflection constant 0<= Ks <= 1.0

It should be noted that all vectors are normalized to unit magnitude.

The illumination can be described several ways depending on the complexity of the object and light source:

I=Ip Kd L*N Diffuse reflection

I=Ia Ka + Ip Kd L*N Ambient lighting and diffuse reflection

I=Ia Ka + Ip(Kd L*N + Ks(R*V)**n)
Ambient lighting, diffuse reflection and specular reflection (Phong model)

In the above equations, * represents a vector dot product such as L*N = LxNx+LyNy+LzNz and ** represents exponentiation.

Since the dot product of two normalized vectors is less than or equal to one, the term Ks(R*V)**n is less than one. The value of this term is found by using a 256 element lookup table with 256.0(R*V) as an index. The value of n is an arbitrary term that is fixed for the algorithm and depends on empirical conditions.

	X memory	Y memory
vec R0 →	Rx	Vx
	Ry	Vy
	Rz	Vz
	Lx	Nx
	Ly	Ny
	Lz	Nz
ktbl R4 →	256.0	
	address of spctbl	
	Kd	
	Ip	
	Ia	
	Ka	

3-D Graphics Illumination		Program Words	ICycles
move	#vec,r0	2	2
move	#ktbl,r4	2	2
move	x:(r0)+,d6.s y:,d7.s	1	1
fmpy.s d6,d7,d0	x:(r0)+,d6.s y:,d7.s	1	1
fmpy.s d6,d7,d1	x:(r0)+,d6.s y:,d7.s	1	1
fmpy d6,d7,d1 fadd.s d1,d0	x:(r0)+,d6.s y:,d7.s	1	1
fmpy d6,d7,d1 fadd.s d1,d0	x:(r4)+,d2.s	1	1
fmpy.s d2,d0,d0	x:(r4)+,n1	1	1
intrz d0	x:(r0)+,d6.s y:,d7.s	1	1
fmpy.s d6,d7,d0	d0.l,r1	1	1
move	x:(r0)+,d6.s y:,d7.s	1	1
fmpy d6,d7,d0 fadd.s d0,d1	x:(r1+n1),d2.s	1	2
fadd.s d0,d1	x:(r4)+,d0.s	1	1
fmpy.s d0,d1,d1	x:(r4)+,d0.s	1	1
fadd.s d1,d2	x:(r4)+,d1.s	1	1
fmpy.s d2,d0	x:(r4),d2.s	1	1
fmpy.s d1,d2,d1		1	1
fadd.s d1,d0		1	1
		---	---
	Totals:	20	21

The illumination value I is in d0.

Reference: "Fundamentals of Interactive Computer Graphics",
James D. Foley, Andries Van Dam
Addison-Wesley 1982

B.1.36 Pseudorandom Number Generation

This pseudorandom number generator requires a 32 bit seed and returns an unsigned 32 bit random number. There are no restrictions on the value of the seed. The equation for the seed is:

seed = (69069*seed + 1) mod 2**32

Pseudorandom Number Generation			Program Words	ICycles
move	x:seed,d0.l	;get seed	2	2
move	#69069,d1.l	;get constant	2	2
mpyu d0,d1,d0		;multiply	1	1
inc d0		; +1	1	1
move	d0.l,x:seed	;mod 2**32, new seed	2	2
			---	---
		Totals:	8	8

The resulting unsigned pseudorandom integer number is in d0.l.

Reference: VAX/VMS Run-Time Library Routines Reference Manual,

Volume 8C, p. RTL-433.

B.1.37 Bezier Cubic Polynomial Evaluation

Bezier polynomials are used to represent curves and surfaces in graphics. The Bezier form requires four points: two endpoints and two points other points. The four points define (in two dimensions) a convex polygon. The curve is bounded by the edges of the polygon.

A typical application of the Bezier cubic is generating character fonts for laser printers using the postscript notation.

Given the four sets of points, the cubic equation for the X coordinate is:

```
x(t)=(P1x)*(1-t)**3 + (P2x)*3*t*(t-1)**2 + (P3x)*3*t*t*(1-t) + (P4x)t**3

where:
    P1x = x coordinate of an endpoint
    P2x = a point used for defining the convex polygon
    P3x = a point used for defining the convex polygon
    P4x = x coordinate of an endpoint
    0.0 <= t <= 1.0
```

As t varies from zero to one, the x coordinate moves along the cubic from one endpoint to the other.

With a little inspiration, the equation can be factored as:

```
x(t)=-(t-1)**3*(P1X) + 3t(t-1)**2*(P2x) - 3t*t(1-t)*(P3x) + t**3*(P4x)
x(t)=(t-1)(-(t-1)**2*(P1x)+3t{(t-1)*(P2x)-t*(P3x)}) + t**3*(P4x)
```

Memory Map:

	X	Y
r4 →	t	1.0
		3.0
	P1x	
	P2x	
r0 →	P3x	
	P4x	

The P coefficients are accessed in the order: P3x,P2x,P1x,P4x.

```
        Bezier Cubic Evaluation                                       Program   ICycles
                                                                      Words
move   #Ptable+2,r0
move   #2,n0
move   #TK,r4

move                           x:(r0)-,d4.s                   1         1
move                           x:(r4)+,d0.s y:,d5.s           1         1
fmpy   d4,d0,d1 fsub.s d5,d0   x:(r0)-,d4.s d0.s,d5.s         1         1
fmpy.s d4,d0,d2                             y:(r4)-,d4.s      1         1
fmpy   d4,d5,d1 fsub.s d1,d2                                  1         1
fmpy.s d1,d2,d2                                               1         1
fmpy.s d0,d0,d1                x:(r0)+n0,d4.s                 1         1
fmpy.s d1,d4,d1                d5.s,d4.s                      1         1
fmpy   d4,d4,d1 fsub.s d1,d2                                  1         1
fmpy.s d0,d2,d2                                               1         1
fmpy.s d1,d4,d1                x:(r0)+n0,d5.s                 1         1
fmpy.s d1,d5,d1                                               1         1
fadd.s d1,d2                                                  1         1
                                                             ---       ---
                                                  Totals:     13        13
```

The result x(t) is in d2. The setup of the pointers is not included because this is application dependent and does not have to be performed for each evaluation of x(t). The first two moves may also be application dependent and be merged with other data ALU operations for a savings of two more cycles and program steps.

Reference: "Fundamentals of Interactive Computer Graphics",
James D. Foley Andries Van Dam
Addison-Wesley 1982

B.1.38 Byte/16 Bit Packing/Unpacking From/To 32 Bits

B.1.38.1 Pack Four Bytes Into a 32 Bit Word

The following packs four 8 bit bytes into a single 32 bit word. The bytes to be packed are right justified in four separate registers:

d0 = xxxA   d2 = xxxC
d1 = xxxB   d3 = xxxD

Four 8 Bit Packs			Program Words	ICycles
joinb	d0,d1	;d1 = xxAB	1	1
joinb	d2,d3	;d3 = xxCD	1	1
join	d1,d3	;d3 = ABCD	1	1
			---	---
		Totals:	3	3

B.1.38.2 Pack Two 16 Bit Words Into a 32 Bit Word

The following packs two 16 bit words into a single 32 bit word. The words to be packed are right justified in two separate registers:

d0 = xY d1 = xZ

Two 16 Bit Packs			Program Words	ICycles
join	d0,d1	;d1 = YZ	1	1
			---	---
		Totals:	1	1

B.1.38.3 Unpack a 32 Bit Word Into Four Sign-extended Bytes

The following unpacks a 32 bit word into four 8 bit sign-extended bytes in separate registers.

Four 8 Bit Unpacks				Program Words	ICycles
move		#data,d3.l	;get data		
split	d3,d1		;d1=ssAB, d3=ABCD	1	1
splitb	d1,d0		;d0=sssA, d1=ssAB	1	1
extb	d1		;d1=sssB	1	1
splitb	d3,d2		;d2=sssC	1	1
extb	d3		;d3=sssD	1	1
				---	---
			Totals:	5	5

B.1.38.4 Unpack a 32 Bit Word Into Two Sign-extended 16 Bit Words

The following unpacks a 32 bit word into two 16 bit sign-extended 16 bit words.

Two 16 Bit Unpacks				Program Words	ICycles
move		#data,d0.l	;get data		
split	d0,d1		;d1=sX, d0=XY	1	1
ext	d1		;d1=sY	1	1
				---	---
			Totals:	2	2

B.1.39 Nth Order Polynomial Evaluation for Two Points

```
;An Nth order polynomial c1X^N + c2X^N-1 + ...cNX + cN+1 can be factored
;and represented as ((c1X + c2)X + c3)X + ...) + cN+1. This routine
;evaluates the polynomial at both X = s and X = t.
;
;Memory Map :   X             Y
;
;       r1 ->   s             t
;               .
;               .
;       r0 ->   c1
;               c2
;               c3
;               .
;               .
;               cN+1
;  Setup N equ  order of polynomial

  move #coef,r0
  move #2_pts,r1
  move x:(r1)+,d5.s y:,d4.s                    ; s, t
  move x:(r0)+,d1.s                            ; c1
  move d1.s,d0.s
; Inner loop for evaluating 2 consecutive points

  do #N,_loop
  fmpy.x d1,d5,d1  x:(r0)+,d2.s                ; c(n)*s, c(n+1)
  fmpy   d0,d4,d0  fadd.x d2,d1                ; c(n)*t, c(n)*s+c(n+1)
  fadd.x d2,d0                                 ; c(n)*t+c(n+1) _loop
```

B.1.40 Graphics BITBLT (Bit Block Transfer)

The bit block transfer (BITBLT) is an operation that transfers a bit field from one area of memory to another. Four parameters describe the BITBLT operation:

SOURCE - The source address of the block to be transferred. Data transferred from the source starts at the lsb of the first data word.

COUNT - The number of words to transfer from the source field. This must be greater than zero.

DEST - Destination starting address.

OFFSET - The starting bit number of the destination word that the transfer is to start. The offset is in the range of 0-31.

Note that the source data starts at the lsb of the first word whereas the destination starts at an arbitrary offset from the lsb.

B.1.40.1 32 Bit Block Transfer

```
          32 Bit Block Transfer                                   Program   ICycles
             BITBLT                                               Words
       org   x:0
source ds      1      ;source address
dest   ds      1      ;destination address
offset ds      1      ;bit number start (0-31)
count  ds      1      ;number of 32 bit source words

  org  p:$50
  move          x:offset,d0.l ;get output bit position         2     2
  move          #32,d1.l      ;get 32                          1     1
  sub  d0,d1    x:source,r0   ;32-offset, point to source      2     2
  move          x:dest,r1     ;point to destination address    2     2
  move          d1.l,d1.h     ;move shift factor               1     1
  move          y:(r1),d4.l   ;get first bits of dest          1     1
  lsl  d1,d4    d0.l,d0.h     ;shift bits, move shift fact     1     1
  move          x:count,n0    ;get source word count           2     2

  do   n0,bitblt              ;do transfer                     2     3
  lsr  d1,d4    y:(r0)+,d5.l  ;shift old bits, get source bits 1     1
  lsl  d0,d5    d5.l,d3.l     ;shift new bits, save new bits   1     1
  or   d4,d5    d3.l,d4.l     ;merge bits, save new as old bit 1     1
  move          d5.l,y:(r1)+  ;save new dest field             1     1
bitblt
  lsr  d1,d4    y:(r1),d5.l   ;shift old bits, get dest bits   1     1
  lsr  d0,d5                  ;shift dest bits                 1     1
  lsl  d0,d5                  ;shift dest bits back            1     1
  or   d4,d5                  ;part of dest with source bits   1     1
  move          d5.l,y:(r1)   ;save new destination bits       1     1
                                                              ---   ---
                                                     Totals:  24   4N+20
```

Where N represents 32 bits transferred. At a 13.5 MIPS, a total of (13.5/4)*32 = 108 MBits/Second transfer rate is possible.

B.1.40.2 64 Bit Block Transfer

A more efficient implementation of BITBLT may be performed by transferring 64 bits at a time. Thus, the value of COUNT specifies the number of 64 bit transfers (two 32 bit words).

```
            64 Bit Block Transfer                                     Program   ICycles
               BITBLT                                                 Words
        org   x:0
source  ds      1      ;source address
dest    ds      1      ;destination address
offset  ds      1      ;bit number start (0-31)
count   ds      1      ;number of 64 bit source words

  org  p:$50
  move          x:offset,d0.l ;get output bit position          2     2
  move          x:offset,d0.l ;get output bit position          2     2
  move          #32,d1.l      ;get 32                           2     2
  sub   d0,d1   x:source,r0   ;32-offset, get source address    2     2
  move          x:dest,r1     ;point to destination address     2     2
  move          d1.l,d1.h     ;move shift factor                1     1
  lsl   d1,d4   d0.l,d0.h     ;shift bits, move shift factor    1     1
  move          x:count,n0    ;get source word count            2     2
  move          (r1)-         ;backup pointer                   1     1
  move          y:(r1)+,d6.l  ;init pipe                        1     1
  move          y:(r1)-,d4.l  ;get first bits of dest           1     1
  move          y:(r0)+,d5.l  ;get source bits                  1     1
  do    n0,bitblt             ;do transfer                      2     3
  lsr   d1,d4   d6.l,y:(r1)+                                    1     1
  lsl   d0,d5   d5.l,d3.l                                       1     1
  or    d4,d5   y:(r0)+,d6.l                                    1     1
  lsr   d1,d3   d5.l,y:(r1)+                                    1     1
  lsl   d0,d6   d6.l,d4.l                                       1     1
  or    d3,d6   y:(r0)+,d5.l                                    1     1
bitblt
  move          d6.l,y:(r1)+                                    1     1
  lsr   d1,d4   y:(r1),d5.l   ;shift old bits, get dest bits    1     1
  lsr   d0,d5                 ;shift dest bits                  1     1
  lsl   d0,d5                 ;shift dest bits back             1     1
  or    d4,d5                 ;part of dest with source bits    1     1
  move          d5.l,y:(r1)   ;save new destination bits        1     1
                                                               ---   ---
                                                      Totals:   32   6N+27
```

Where N represents 64 bits transferred. At a 13.5 MIPS, a total of (13.5/6)*64 = 144 MBits/Second transfer rate is possible.

B.1.41 64x64 Bit Unsigned Multiply

This performs a double precision unsigned integer multiply. The 64 bit integer is formed by the concatenation of two 32 bit registers.

Let X = A:B and Y = C:D, then X*Y can be written as:

```
          A   B
        * C   D
   ------------------
   +         B * D
   +     A * D
   +     B * C
   + A * C
   ------------------
   = W   X   Y   Z
```

64x64 Bit Unsigned Multiply
d3:d7:d6:d4 = d0:d1 * d2:d3

			Program Words	ICycles
mpyu	d0,d2,d7		1	1
mpyu	d0,d3,d5		1	1
mpyu	d1,d3,d4	d7.h,d3.l	1	1
mpyu	d1,d2,d6	d4.h,d0.l	1	1
move		d6.h,d2.l	1	1
add	d0,d5	d5.h,d1.l	1	1
addc	d1,d2		1	1
inc	d3	ifcs	1	1
add	d5,d6		1	1
addc	d2,d7		1	1
inc	d3	ifcs	1	1
			---	---
		Totals:	11	11

B.1.42 Signed Reciprocal Generation

This generates a fast approximation to 1/x.

Approximation of 1/d1 16 Bit Accuracy			Program Words	ICycles
fseedd	d1,d6		1	1
fmpy.s	d1,d6,d1	#2.0,d4.s	2	2
fsub.s	d1,d4		1	1
fmpy.s	d6,d4,d1		1	1
			---	---
		Totals:	5	5

Approximation of 1/d1 32 Bit Accuracy			Program Words	ICycles
fseedd	d1,d6		1	1
fmpy.s	d1,d6,d1	#2.0,d4.s	2	2
fsub.s	d1,d4	d4.s,d3.s	1	1
fmpy.s	d1,d4,d1		1	1
fmpy	d6,d4,d1	fsub.s d1,d3	1	1
fmpy.s	d1,d3,d1		1	1
			---	---
		Totals:	7	7

B.1.43 Line Drawing

B.1.43.1 Floating-Point Incremental Line Drawing Algorithm

This algorithm generates points along a line given the endpoints. As the coordinate along one axis is incremented in fixed point, the other coordinate is incremented in floating-point and then converted to fixed point. A full line drawing algorithm which draws lines in all directions is given below.

Registers:

d0 = temporary	d4 = temporary x1	d8 =
d1 = temporary	d5 = temporary y1	d9 = 2.0
d2 = x1 (dx)	d6 = x0 and xScreen	
d3 = y1 (dy)	d7 = y0 and yScreen	

Program Words **ICycles**

```
; Calculate dx and dy
  fsub.s d6,d2  d2.s,d4.s                             1    1
  fsub.s d7,d3  d3.s,d5.s                             1    1
; Determine whether to increment x or y
  fcmpm  d3,d2                                        1    1
  fjge   _inc_x                                       2    2
; Switch endpoints if necessary
_inc_y
  ftst   d3                    d2.s,d0.s              1    1
  ftfr.s d4,d6                 fflt                   1    1
  ftfr.s d5,d7                 fflt                   1    1
; Fix y0 and dy
  int    d7                    d3.s,d1.s              1    1
  int    d1                                           1    1
  neg    d1                    iflt                   1    1
  jeq    _draw1_y                                     2    2
; Calculate dx/dy
  fseedd d3,d4                                        1    1
  fmpy.s d3,d4,d5              d9.s,d2.s              1    1
  fmpy   d0,d4,d0 fsub.s d5,d2 d2.s,d3.s              1    1
  fmpy.s d5,d2,d5              d2.s,d4.s              1    1
  fmpy   d0,d4,d0 fsub.s d5,d3                        1    1
  fmpy.s d0,d3,d0              d6.s,d2.s              1    1
; Draw first point
  int    d6                                           1    1
  jsr    _draw_point                       application dependent
; d0 = dx/dy  d1 = dy  d6 = x0  d7 = y0
  do     d1.l,_end_y                                  2    3
  fadd.x d0,d2                                        1    1
  inc    d7                    d2.s,d6.s              1    1
  int    d6                                           1    1
  jsr    _draw_point                       application dependent
_end_y
  rts                                                 2    2

_draw1_y
  int d6                                              1    1
  jsr    _draw_point                       application dependent
  rts                                                 2    2
```

```
; Switch endpoints if necessary
_inc_x
  ftst   d2                   d3.s,d0.s                    1    1
  ftfr.s d4,d6                fflt                         1    1
  ftfr.s d5,d7                fflt                         1    1
; Fix x0 and dx
  int    d6                   d2.s,d1.s                    1    1
  int    d1                                                1    1
  neg    d1                   iflt                         1    1
  jeq                         _draw1_x                     2    2
; Calculate dy/dx
  fseedd d2,d4                                             1    1
  fmpy.s d2,d4,d5             d9.s,d2.s                    1    1
  fmpy   d0,d4,d0 fsub.s d5,d2 d2.s,d3.s                   1    1
  fmpy.s d5,d2,d5             d2.s,d4.s                    1    1
  fmpy   d0,d4,d0 fsub.s d5,d3                             1    1
  fmpy.s d0,d3,d0             d7.s,d2.s                    1    1
; Draw first point
  int    d7                                                1    1
  jsr    _draw_point                          application dependent
; d0 = dy/dx  d1 = dx  d6 = x0  d7 = y0
  do     d1.l,_end_x                                       2    3
  fadd.x d0,d2                                             1    1
  inc    d6                   d2.s,d7.s                    1    1
  int    d7                                                1    1
  jsr    _draw_point                          application dependent
_end_x
  rts                                                      2    2
_draw1_x
  int d7                                                   1    1
  jsr    _draw_point                          application dependent
  rts                                                      2    2
```

Performance:

Trivial case: (single point) 16 cycles

Other cases: 25 + 3n cycles

B.1.43.2 Integer Incremental Line Drawing Algorithm

This implementation of line drawing uses Bresenham's algorithm. This algorithm uses only integer operations to generate the points.

```
; Bresenham Line Drawing Implementation
;
; When entering subroutine, the registers must
; be set as follows:
;
;   d0 =          d4 =
;   d1 =          d5 =
;   d2 = x1       d6 = x0
;   d3 = y1       d7 = y0
;
; When entering a line drawing loop, the registers
; are set as follows:
;
;   d6 = x0
;   d7 = y0
;   d4 = dmajor
;   d5 = n0 = dminor
;   r0 = dmajor/2
;   m0 = dmajor - 1
  org    p:$50
; Calculate dx and dy
_line
  sub    d6,d2    d2.l,d4.l
  sub    d7,d3    d3.l,d5.l
; Determine whether to increment x or y
  tst    d2       d2.l,d0.l
  neg    d2       iflt
  tst    d3       d3.l,d1.l
  neg    d3       iflt
  cmp    d3,d2
  jge    _inc_x
; Increment y case
; If dy is negative, switch endpoints and sign of dx and dy
_inc_y
  tst    d1
  tfr    d4,d6    iflt
  tfr    d5,d7    iflt
```

```
  neg    d1       iflt
  neg    d0       iflt
  tst    d0
  jlt    _set_y_xn

; Increment y, dx positive case
; Set up registers
_set_y_xp
  lsr    d1       d1.l,d2.l
  dec    d2       d2.l,d4.l
  move            d1.l,r0
  move            d2.l,m0
  move            d0.l,n0
  move            d0.l,d5.l
; Draw first point
  jsr   _draw_point
; Draw additional points
  do    d4.l,_line_y_xp
  inc   d7       r0,d2.l
  add   d5,d2    (r0)+n0
  cmp   d4,d2
  inc   d6       ifge
  jsr   _draw_point
_line_y_xp
  rts
; Increment y, dx negative case
; Set up registers
_set_y_xn
  lsr    d1       d1.l,d2.l
  dec    d2       d2.l,d4.l
  neg    d0       d1.l,r0
  move            d2.l,m0
  move            d0.l,n0
  move            d0.l,d5.l
; Draw first point
  jsr   _draw_point
; Draw additional points
  do    d4.l,_line_y_xn
  inc   d7       r0,d2.l
  add   d5,d2    (r0)+n0
```

```
  cmp   d4,d2
  dec   d6       ifge
  jsr   _draw_point
_line_y_xn
  rts

; Increment x case
; If dx is negative, switch endpoints and sign of dx and dy
_inc_x
  tst    d0
  jeq    _draw1
  tfr    d4,d6    iflt
  tfr    d5,d7    iflt
  neg    d0       iflt
  neg    d1       iflt
  tst    d1
  jlt    _set_x_yn
; Increment x, dy positive case
; Set up registers
_set_x_yp
  lsr    d0       d0.l,d2.l
  dec    d2       d2.l,d4.l
  move            d0.l,r0
  move            d2.l,m0
  move            d1.l,n0
  move            d1.l,d5.l
; Draw first point
  jsr   _draw_point
; Draw additional points
  do    d4.l,_line_x_yp
  inc   d6       r0,d2.l
  add   d5,d2    (r0)+n0
  cmp   d4,d2
  inc   d7       ifge
  jsr   _draw_point
_line_x_yp
  rts
; Increment x, dy negative case
; Set up registers
_set_x_yn
```

```
    lsr    d0       d0.l,d2.l
    dec    d2       d2.l,d4.l
    neg    d1       d0.l,r0
    move            d2.l,m0
    move            d1.l,n0
    move            d1.l,d5.l

  ; Draw first point
    jsr   _draw_point
  ; Draw additional points
    do    d4.l,_line_x_yn
    inc   d6       r0,d2.l
    add   d5,d2    (r0)+n0
    cmp   d4,d2
    dec   d7       ifge
  _draw1
    jsr   _draw_point
  _line_x_yn
    rts
  ; Draw a single point
  _draw_point
    move           d6.l,x:(r1)+   d7.l,y:
    rts
```

B.1.44 Wire-Frame Graphics Rendering

WIRE-FRAME RENDITION OF A THREE DIMENSIONAL POLYLINE
ON THE MOTOROLA DSP96002
Version 1.00

OVERVIEW

This program displays a three dimensional polyline in two dimensions. The points of the polyline, as defined in the input list, are projected into two dimensions using the perspective transformation. The projected points are output to a display list that can be drawn by a graphics engine or a fast drawing program.

In order to maximize speed, two loops perform the graphics transformations: the trivial accept loop and the trivial reject loop.

The trivial accept loop assumes that the last displayed point was inside the viewing pyramid and thus not clipped. It pulls a new point from the input list, converts it to clipping space and checks if it is inside the viewing pyramid. If so, the routine performs the perspective transformation, scales and translates the point so it lies within the viewing window, and finally adds it to the display list.

If the point is found to lie outside the viewing pyramid, an algorithm to clip a single point is performed and the program enters the trivial reject loop.

The trivial reject loop assumes that the last displayed point was outside the viewing pyramid. It pulls a new point from the input list, converts it to clipping space and checks if the line joining the new point and the last point can be trivially rejected. Trivial rejection occurs when both points of a line lie outside of a clipping plane. When this occurs, the current point is saved and the trivial reject loop repeats.

Should the line not be trivially rejected but the current point is accepted, an algorithm to clip a single point is performed. If the current point is not accepted, two-point clipping is performed.

PERFORMANCE

All times are given in instruction cycles.

```
Accept loop
  First point                        38
  Each additional point              39

Accept single point clip
  Minimum (single plane)             68
  Maximum (three planes)             94

Reject loop
  Each point                         37

Reject single point clip
  Minimum (single plane)             89
  Maximum (three planes)            115

Reject double clip line drawn
  Minimum (two single planes)       145
  Maximum (six planes)              206

Reject double clip line rejected
  Minimum (two single planes)       112
  Maximum (six planes)              173
```

The DSP96002 has an instruction cycle time of 74ns and will transform 347K points/sec in the accept loop. In the reject loop, 365K points can be rejected each second.

INPUT

Before calling the polyline generator, address register r1 should point to the area in X memory which contains the X, Y and Z coordinates of the input points. Data register d7.l should contain the number of points in the polyline in the form of a 32-bit integer.

OUTPUT

Address register r5 should point to a display list data area when the polyline generator is called. Afterwards, the display list will be in the following format:

```
Polygon1:  X1,Y1
           X2,Y2
           X3,Y3
                                Xn,Yn

Delimiter  -1.0

Polygon2:  X1,Y1
           X2,Y2
                      Delimiter  -1.0
          PolygonM:  X1,Y1

           Xn,Yn
           -2.0
```

All coordinates are in IEEE single-precision floating-point format to speed up the DSP96002 floating-point incremental line drawing algorithm.

ADDRESS REGISTER USAGE

Four address registers are used:

- r0 input list
- r1 temporary coordinates
- r4 transformation matrix, scale and offset for 2D transformation
- r5 output list
- r6 miscellaneous scratchpad memory

The following memory map results:

X Memory   Y Memory

```
  r0  →  Xobj0
  n0=0.0 Yobj0
         Zobj0
         Xobj1

  r1 →   Xnew         Znew
  n1=2   Ynew         Wnew
  m1=3   Xold         Zold
         Yold         Wold

 r4 →                Matrix1,1
  n4=2   Matrix4,1    Matrix2,1
  m4=13               Matrix3,1
                      Matrix1,2
         Matrix4,2    Matrix2,2
                      Matrix3,2
                      Matrix1,3
         Matrix4,3    Matrix2,3
                      Matrix3,3
                      Matrix1,4
         Matrix4,4    Matrix2,4
                      Matrix3,4
         Xscale       Xoffset
         Yscale       Yoffset

                      Xout0  ←  r5
                      Yout0     n5=-1.0
                      Xout1
                      Yout1

         TempCount    TOld,Xtemp ¨ r6  (temporaries)
                      Ytemp
                      Wtemp
```

Several registers hold constants that speed up calculations. These are:

d8 = 1.0 for double point clipping

d9 = 2.0 for division

n0 = 0.0 for z limit test and double point clipping

n5 = -1.0 for end of polyline marker

TRIVIAL ACCEPT LOOP

The transformation from object space to screen space is performed in lines 19-33. This is a {1x4}{4x4} matrix multiplication but because the W coordinate of the {1x4} input vector {X Y Z W} is always equal to one, four multiplications can be eliminated.

Lines 39-47 determine if the point is within the viewing pyramid. The FCMP s,d instruction is designed to clear the sticky accept (A) bit (bit 7 in the CCR) whenever s > d. By switching the order of the operands, the FCMP instruction can be used to test both the maximum and minimum boundaries of a window. To test acceptance, the A bit is set in line 40 and the X and Y coordinates are compared to the boundaries -W and W. The Z coordinate is compared to the boundaries 0 and W. If the A bit remains set, the point is within the viewing pyramid and is transformed to screen coordinates.

If the A bit is clear, the reject loop is entered. Note that the A bit is only affected by the CMP, CMPG, FCMP and FCMPG instructions.

The reciprocal 1/W is calculated in lines 53-58. The result is accurate to approximately 32 bits. It is multiplied by the X coordinate and then by the X scale to scale the data to the output screen. The coordinate is then translated to screen space. The procedure is repeated for the Y coordinate and the coordinates are added to the display list.

For additional points the accept loop code is almost identical to the first point code except that if the new point is not within the viewing pyramid, a jump to a single point clipping routine is performed.

ACCEPT LOOP SINGLE POINT CLIPPING CODE

The method used for clipping a line when one point is inside the viewing pyramid and one point is outside is a special case of a general clipping algorithm presented in [1] and is used in the double point clipping code.

Suppose that the line between points P1 and P2 was rejected because the x coordinate of P2, x2, was larger than w2. Then,

y2 = y1 + t (y2 - y1)

where

$$t = \frac{w1 - x1}{(w1 - x1) - (w2 - x2)}$$

Substituting the value of t results in the determinant

```
y2 =   | y2  w2-y2 |
       | y1  w1-y1 |
    -------------------
     (w1-x1) - (w2-x2)
```

The equations for z2 and w2 are analogous. Since w2 has the same denominator as x2, y2 and z2, and these will be divided by w2 in the perspective transformation, the division shown above does not need to be performed.

Lines 151-162 determine which planes that the point is outside and call the appropriate clipping routines. These routines (lines 520-617) calculate the determinants and return with the resulting coordinates in the data registers.

The resulting point is transformed using the perspective transformation, scaled and translated in lines 168-186. A code (-1.0) is stored in the display list to indicate that the next line to be drawn is not joined with the current one. Control is then transferred to the trivial reject loop.

TRIVIAL REJECT LOOP

The trivial reject loop starts with the {1x4}{4x4} matrix multiplication to transform the input point to clipping space. Next, the line joining the current point and the previously rejected point is tested for trivial rejection. As mentioned earlier, trivial rejection occurs whenever both of the endpoints lie outside of one clipping plane.

A sticky bit called Local Reject (LR) is defined as bit 5 of the CCR. It is cleared by the FCMP s,d instruction whenever s <= d. In other words, the LR bit is cleared whenever the FCMP instruction finds the coordinate inside of the boundary.

An additional instruction, FCMPG, is needed because trivial rejection occurs when both points are outside of any boundary plane. Thus, an additional sticky bit called Reject (R) (bit 6 of the CCR) is used to "remember" that a trivial reject has occurred after comparisons against one boundary plane. The FCMPG instruction affects R and is performed as the last comparison to a boundary plane. When FCMPG s,d is executed, the R flag is cleared if the previous point was outside of the boundary (LR is set) and the current point is outside of the boundary (s > d). The FCMPG instruction also resets the LR bit to 1 for comparison to the next boundary plane.

To perform the trivial reject test, the LR and R bits are set to 1. The two points are tested against the X = -W boundary plane and then tested against the X = W plane etc. The first point is tested using FCMP and the second point is tested using FCMPG to clear the R bit if both comparisons were outside of the boundary. At the end of these comparisons, if the R bit is 0, the line was trivially rejected. With this definition, the trivial rejection test can be generalized to a polygon with any number of points. The execution time is of order 6N cycles where N is the number of points.

The lines 225-236 perform the trivial reject test. Should the line be trivially rejected, the new coordinates are stored for the next comparison and the reject loop repeats.

If the line is not trivially rejected, a check is made to determine if the current point is accepted. If so, control is transferred to the reject loop single point clip routine. Otherwise the double point routine is entered.

REJECT LOOP SINGLE POINT CLIPPING CODE

The reject loop single point clipping code is very similar to the analogous code in the accept loop. It calls the same clipping subroutines in lines 520-617. Then the point that was just calculated is transformed, scaled and translated and stored in the output list (lines 305-321). Finally, the new point (which was accepted) is transformed, scaled and translated (lines 327-345). Control is transferred to the accept loop.

REJECT LOOP DOUBLE POINT CLIPPING CODE

Lines 359-492 are a direct implementation of a clipping algorithm using endpoint coordinates given in {1}. The clipping method using determinants is not powerful enough to handle the cases where the line is rejected but not trivially rejected. Thus, the line parameters t1 and t2 are calculated explicitly. The t1 parameter is calculated based on the coordinates of the old point and the t2 parameter is calculated based on the current point.

These parameters are calculated by a set of double point clipping subroutines in lines 631-853. These subroutines are called based on the coordinates in lines 359-395.

The line is checked for rejection which occurs when t1 > t2. If the line is not rejected, the plane intersections are interpolated based on t1 and t2 (lines 409-431). Then the two new points are transformed, scaled and translated in lines 437-478. Control is then transferred to the reject loop.

If the line is rejected, control is transferred to the reject loop after some housekeeping is performed.

TERMINATION CODE

Lines 499-509 swallow the line delimiter code (-1.0) if it is the last coordinate in the display list. Then it adds the end of display list code (-2.0) to the display list and exits.

REFERENCE

{1} William M. Newman and Robert F. Sproull, Principles of Interactive Computer Graphics, (New York: McGraw-Hill, 1979).

```
;
; WIRE-FRAME RENDITION OF A THREE DIMENSIONAL POLYLINE
;              ON THE MOTOROLA DSP96002
;
;          Version 1.00  18-Nov-88
;
;
;
;---------------------------------------------------------
;
;                    First point
;
;---------------------------------------------------------


; Transform to clip space
```

```
;                                                          Words   ICycles
wf3d
  move                         x:(r0)+,d0.s               ;X        1  1
  move                         x:(r0)+,d5.s   y:(r4)+,d4.s ;Y   M11 1  1
  fmpy.s d4,d0,d2              x:(r4)+,d3.s   y:,d4.s      ;M41 M21 1  1
  fmpy   d4,d5,d3 fadd.s d3,d2 x:(r0)+,d6.s   y:(r4)+,d4.s ;Z   M31 1  1
  fmpy   d4,d6,d3 fadd.s d3,d2 x:(r1)+n1,d1.s y:(r4)+,d4.s ;r1+ M12 1  1
  fmpy   d4,d0,d1 fadd.s d3,d2 x:(r4)+,d3.s   y:,d4.s      ;M42,M22 1  1
  fmpy   d4,d5,d3 fadd.s d3,d1                y:(r4)+,d4.s ;    M32 1  1
  fmpy   d4,d6,d3 fadd.s d3,d1 d2.s,x:(r1)+   y:(r4)+,d4.s ;Xo  M13 1  1
  fmpy   d4,d0,d2 fadd.s d3,d1 x:(r4)+,d3.s   y:,d4.s      ;M43 M23 1  1
  fmpy   d4,d5,d3 fadd.s d3,d2                y:(r4)+,d4.s ;    M33 1  1
  fmpy   d4,d6,d3 fadd.s d3,d2 d1.s,x:(r1)-   y:(r4)+,d4.s ;Yo  M14 1  1
  fmpy   d4,d0,d1 fadd.s d3,d2 x:(r4)+,d3.s   y:,d4.s      ;M44 M24 1  1
  fmpy   d4,d5,d3 fadd.s d3,d1                y:(r4)+,d4.s ;    M34 1  1
  fmpy   d4,d6,d3 fadd.s d3,d1                d2.s,y:(r1)  ;    Zo  1  1
                  fadd.s d3,d1 x:(r1)+,d0.s               ;Xo       1  1


; Test if point is within viewing pyramid

  fneg.s d1                    d1.s,d2.s                  ;         1  1
  ori    #$80,ccr                                         ;         1  1
  fcmp   d1,d0                                            ;         1  1
  fcmp   d0,d2                 x:(r1)-,d5.s               ;Yo       1  1
  fcmp   d1,d5                 n0,d4.s                    ;         1  1
  fcmp   d5,d2                                y:(r1)+,d6.s ;    Zo  1  1
  fcmp   d4,d6                                            ;         1  1
  fcmp   d6,d2                                            ;         1  1
  jclr   #7,sr,_reject_entry                              ;         2  3


; Calculate reciprocal 1/W


  fseedd d2,d6                                            ;         1  1
  fmpy.s d2,d6,d1              d9.s,d4.s                  ;         1  1
                  fsub.s d1,d4 d4.s,d3.s      d2.s,y:(r1)+ ;    Wo  1  1
  fmpy.s d1,d4,d1                                         ;         1  1
  fmpy   d6,d4,d1 fsub.s d1,d3                            ;         1  1
  fmpy.s d1,d3,d1              x:(r4)+,d4.s   y:,d3.s      ;Xs  Xf  1  1
```

```
; Multiply coordinates by 1/W, scale and add offset

  fmpy.s d0,d4,d2                                          ;        1  1
  fmpy.s d2,d1,d2              x:(r4)+,d4.s   y:,d6.s      ;Ys  Yf  1  1
  fmpy   d5,d4,d3 fadd.s d3,d2                             ;        1  1
  fmpy.s d3,d1,d3                             d2.s,y:(r5)+ ;        1  1
                  fadd.s d6,d3 x:(r0)+,d0.s                ;        1  1
  dec    d7                                   d3.s,y:(r5)+ ;    Y1  1  1



;---------------------------------------------------------
;
;                    Accept loop
;
;---------------------------------------------------------


; Transform point to clip space


_accept_loop
  move                         x:(r0)+,d5.s   y:(r4)+,d4.s ;Y   M11 1  1
  fmpy.s d4,d0,d2              x:(r4)+,d3.s   y:,d4.s      ;M41 M21 1  1
  fmpy   d4,d5,d3 fadd.s d3,d2 x:(r0)+,d6.s   y:(r4)+,d4.s ;Z   M31 1  1
  fmpy   d4,d6,d3 fadd.s d3,d2                y:(r4)+,d4.s ;    M12 1  1
  fmpy   d4,d0,d1 fadd.s d3,d2 x:(r4)+,d3.s   y:,d4.s      ;M42,M22 1  1
  fmpy   d4,d5,d3 fadd.s d3,d1                y:(r4)+,d4.s ;    M32 1  1
  fmpy   d4,d6,d3 fadd.s d3,d1 d2.s,x:(r1)+   y:(r4)+,d4.s ;Xn  M13 1  1
  fmpy   d4,d0,d2 fadd.s d3,d1 x:(r4)+,d3.s   y:,d4.s      ;M43 M23 1  1
  fmpy   d4,d5,d3 fadd.s d3,d2                y:(r4)+,d4.s ;    M33 1  1
  fmpy   d4,d6,d3 fadd.s d3,d2 d1.s,x:(r1)-   y:(r4)+,d4.s ;Yn  M14 1  1
  fmpy   d4,d0,d1 fadd.s d3,d2 x:(r4)+,d3.s   y:,d4.s      ;M44 M24 1  1
  fmpy   d4,d5,d3 fadd.s d3,d1                y:(r4)+,d4.s ;    M34 1  1
  fmpy   d4,d6,d3 fadd.s d3,d1                d2.s,y:(r1)  ;    Zn  1  1
                  fadd.s d3,d1 x:(r1)+,d0.s                ;Xn      1  1
```

```
; Determine if point is within view volume

  fneg.s d1                   d1.s,d2.s                          ;        1  1
  ori    #$80,ccr                                                ;        1  1
  fcmp   d1,d0                                 d2.s,y:(r1)  ;    Wn  1  1
  fcmp   d0,d2                x:(r1)-,d5.s                       ;Yn      1  1
  fcmp   d1,d5                n0,d4.s                            ;        1  1
  fcmp   d5,d2                                 y:(r1)-,d6.s ;    Zn  1  1
  fcmp   d4,d6                d7.l,x:(r6)                        ;        1  1
  fcmp   d6,d2                d6.s,d7.s                          ;        1  1
  jclr   #7,sr,_accept_clip                                      ;        2  3

; Calculate reciprocal 1/W

  fseedd d2,d6                                                   ;        1  1
  fmpy.s d2,d6,d1             d9.s,d4.s                          ;        1  1
                fsub.s d1,d4 d4.s,d3.s      d2.s,y:(r1)- ;    Wo  1  1
  fmpy.s d1,d4,d1             d0.s,x:(r1)+   d7.s,y:        ;Xo  Zo  1  1
  fmpy   d6,d4,d1 fsub.s d1,d3 d5.s,x:(r1)+                  ;Yo      1  1
  fmpy.s d1,d3,d1             x:(r4)+,d4.s   y:,d3.s        ;Xs  Xf  1  1

; Multiply coordinates by 1/W, scale and add offset

  fmpy.s d0,d4,d2                                                ;        1  1
  fmpy.s d2,d1,d2             x:(r4)+,d4.s   y:,d6.s        ;Ys  Yf  1  1
  fmpy   d5,d4,d3 fadd.s d3,d2 x:(r6),d7.l                       ;        1  1
  fmpy.s d3,d1,d3                              d2.s,y:(r5)+ ;        1  1
                fadd.s d6,d3 x:(r0)+,d0.s                       ;        1  1
  dec    d7                                    d3.s,y:(r5)+ ;    Y1  1  1
  jne    _accept_loop                                            ;        2  2
  jmp    _end                                                    ;        2  2
```

```
;----------------------------------------------------------
;
;            Accept loop single-clip routine
;
;----------------------------------------------------------


; Dispatch to single-plane clipping routines


_accept_clip
  fsub.s d0,d2                d2.s,d1.s                  ;        1  1
  fjslt  _clip1_xp                                       ;        2  2
  fadd.s d0,d1                d1.s,d2.s                  ;        1  1
  fjslt  _clip1_xn                                       ;        2  2
  fsub.s d5,d2                d2.s,d1.s                  ;        1  1
  fjslt  _clip1_yp                                       ;        2  2
  fadd.s d5,d1                d1.s,d2.s                  ;        1  1
  fjslt  _clip1_yn                                       ;        2  2
  fsub.s d6,d2                d2.s,d1.s                  ;        1  1
  fjslt  _clip1_zp                                       ;        2  2
  ftst   d6                                              ;        1  1
  fjslt  _clip1_zn                                       ;        2  2


; Calculate reciprocal 1/W


  fseedd d1,d6                                           ;        1  1
  fmpy.s d1,d6,d1             d9.s,d4.s                  ;        1  1
                fsub.s d1,d4 d4.s,d3.s      y:(r1)+n1,d2.s ; r1+2 1  1
  fmpy.s d1,d4,d1             x:(r1)+n1,d2.s y:,d7.s     ;Yn  Wn  1  1
  fmpy   d6,d4,d1 fsub.s d1,d3 d2.s,x:(r1)+   d7.s,y:    ;Yo  Wo  1  1
  fmpy.s d1,d3,d1             x:(r4)+,d4.s   y:,d3.s     ;Xs  Xf  1  1


; Multiply coordinates by 1/W, scale and add offset

  fmpy.s d0,d4,d2             x:(r1)+n1,d0.s y:,d7.s     ;Xn  Zn  1  1
```

```
  fmpy.s d2,d1,d2              x:(r4)+,d4.s   y:,d6.s      ;Ys  Yf  1  1
  fmpy   d5,d4,d3 fadd.s d3,d2 d0.s,x:(r1)+n1 d7.s,y:      ;Xo  Zo  1  1
  fmpy.s d3,d1,d3              x:(r6),d7.l                 ;Cnt     1  1
                  fadd.s d6,d3 x:(r0)+,d0.s   d2.s,y:(r5)+ ;X       1  1
  move                                        d3.s,y:(r5)+ ;    Y1  1  1
  dec    d7                                   n5,y:(r5)+   ;   -1.0 1  1
  jne    _reject_loop                                      ;        2  2
  jmp    _end                                              ;        2  2

;---------------------------------------------------------
;
;                    Reject loop
;
;---------------------------------------------------------



; Transform point to clip space

_reject_entry
  dec    d7                                   d2.s,y:(r1)+ ;    Wo  1  1
  move                         x:(r0)+,d0.s  y:(r4)+n4,d4.s ;X r4+2 1  1

_reject_loop
  move                         x:(r0)+,d5.s  y:(r4)+,d4.s  ;Y   M11 1  1
  fmpy.s d4,d0,d2              x:(r4)+,d3.s  y:,d4.s       ;M41 M21 1  1
  fmpy   d4,d5,d3 fadd.s d3,d2 x:(r0)+,d6.s  y:(r4)+,d4.s  ;    M31 1  1
  fmpy   d4,d6,d3 fadd.s d3,d2               y:(r4)+,d4.s  ;    M12 1  1
  fmpy   d4,d0,d1 fadd.s d3,d2 x:(r4)+,d3.s  y:,d4.s       ;M42 M22 1  1
  fmpy   d4,d5,d3 fadd.s d3,d1               y:(r4)+,d4.s  ;    M32 1  1
  fmpy   d4,d6,d3 fadd.s d3,d1 d2.s,x:(r1)+  y:(r4)+,d4.s  ;Xn  M13 1  1
  fmpy   d4,d0,d2 fadd.s d3,d1 x:(r4)+,d3.s  y:,d4.s       ;M43 M23 1  1
  fmpy   d4,d5,d3 fadd.s d3,d2               y:(r4)+,d4.s  ;    M33 1  1
  fmpy   d4,d6,d3 fadd.s d3,d2 d1.s,x:(r1)-  y:(r4)+,d4.s  ;Yn  M14 1  1
  fmpy   d4,d0,d1 fadd.s d3,d2 x:(r4)+,d3.s  y:,d4.s       ;M44 M24 1  1
  fmpy   d4,d5,d3 fadd.s d3,d1               y:(r4)+,d4.s  ;    M34 1  1
  fmpy   d4,d6,d3 fadd.s d3,d1               d2.s,y:(r1)-  ;    Zn  1  1
                  fadd.s d3,d1                             ;        1  1


; Determine trivial rejection
```

```
  ori    #$e0,ccr                                       ;        1  1
  fneg.s d1                   d1.s,d5.s      y:(r1)-,d2.s ;    Wo  1  1
  fneg.s d2                   x:(r1)+n1,d6.s d2.s,d4.s    ;Xo      1  1
  fcmp   d2,d6                x:(r1)-,d0.s                ;Xn      1  1
  fcmpg  d1,d0                (r4)+n4                     ;r4+2    1  1
  fcmp   d6,d4                                            ;        1  1
  fcmpg  d0,d5                x:(r1)+n1,d6.s              ;Yo      1  1
  fcmp   d2,d6                x:(r1)+,d3.s                ;Yn      1  1
  fcmpg  d1,d3                                            ;        1  1
  fcmp   d6,d4                                            ;        1  1
  fcmpg  d3,d5                              y:(r1)+n1,d6.s ;Zo      1  1
  fcmp   d6,d4                              y:(r1)+n1,d2.s ;Zn      1  1
  fcmpg  d2,d5                n0,d4.s                     ;        1  1
  fcmp   d4,d6                                            ;        1  1
  fcmpg  d4,d2                                            ;        1  1
  jset   #6,sr,_reject_clip                               ;        2  3

; Save new point

  move                        d0.s,x:(r1)+   d2.s,y:      ;Xo Zo   1  1
  move                        d3.s,x:(r1)+   d5.s,y:      ;Yo Wo   1  1
  dec    d7                   x:(r0)+,d0.s                ;X       1  1
  jne    _reject_loop                                     ;        2  2
  jmp    _end                                             ;        2  2


;---------------------------------------------------------
;
;              Reject loop clipping routine
;
;---------------------------------------------------------

; Determine if new point is within view volume

_reject_clip
  ori     #$80,ccr                                        ;        1  1
  fcmp    d1,d0               (r1)-                       ;r1-     1  1
  fcmp    d1,d3                              d5.s,y:(r1)+ ;    Wn  1  1
  fcmp    d4,d2                                           ;        1  1
```

```
  fcmp    d0,d5                                          ;        1  1
  fcmp    d3,d5                                          ;        1  1
  fcmp    d2,d5                                          ;        1  1
  jclr    #7,sr,_r_clip2                                 ;        2  3


;---------------------------------------------------------
;
;            Reject loop single-clip routine
;
;---------------------------------------------------------

; Dispatch to clipping routines

  move                      x:(r1)+,d0.s   y:,d6.s      ;Xo  Zo  1  1
  move                      x:(r1)+n1,d5.s y:,d2.s      ;Yo  Wo  1  1
  move                      d7.l,x:(r6)                 ;Cnt     1  1
  fsub.s d0,d2              d2.s,d1.s                   ;        1  1
  fjslt  _clip1_xp                                      ;        2  2
  fadd.s d0,d1              d1.s,d2.s                   ;        1  1
  fjslt  _clip1_xn                                      ;        2  2
  fsub.s d5,d2              d2.s,d1.s                   ;        1  1
  fjslt  _clip1_yp                                      ;        2  2
  fadd.s d5,d1              d1.s,d2.s                   ;        1  1
  fjslt  _clip1_yn                                      ;        2  2
  fsub.s d6,d2              d2.s,d1.s                   ;        1  1
  fjslt  _clip1_zp                                      ;        2  2
  ftst   d6                                             ;        1  1
  fjslt  _clip1_zn                                      ;        2  2

; Calculate reciprocal 1/W (old point)


  fseedd d1,d6                                          ;        1  1
  fmpy.s d1,d6,d1           d9.s,d4.s                   ;        1  1
                fsub.s d1,d4               d4.s,d3.s    ;        1  1
  fmpy.s d1,d4,d1                          (r4)-n4      ;   r4-2 1  1
  fmpy   d6,d4,d1 fsub.s d1,d3                          ;        1  1
  fmpy.s d1,d3,d1           x:(r4)+,d4.s   y:,d3.s      ;Xs  Xf  1  1
```

```
; Multiply coordinates by 1/W, scale and add offset (old point)

  fmpy.s d0,d4,d2                                          ;        1  1
  fmpy.s d2,d1,d2              x:(r4)-,d4.s y:,d6.s        ;Ys   Yf  1  1
  fmpy   d5,d4,d3 fadd.s d3,d2                             ;        1  1
  fmpy.s d3,d1,d3                           d2.s,y:(r5)+   ;     X1  1  1
                  fadd.s d6,d3              y:(r1)+n1,d2.s ;     Wn  1  1
  move                                      d3.s,y:(r5)+   ;     Y1  1  1

; Calculate reciprocal 1/W (new point)

  fseedd d2,d6                                             ;        1  1
  fmpy.s d2,d6,d1              d9.s,d4.s                   ;        1  1
                  fsub.s d1,d4 d4.s,d3.s      d2.s,y:(r1)+ ;     Wo  1  1
  fmpy.s d1,d4,d1              x:(r1)+n1,d0.s y:,d2.s      ;Xn   Zn  1  1
  fmpy   d6,d4,d1 fsub.s d1,d3 d0.s,x:(r1)-   d2.s,y:      ;Xo   Zo  1  1
  fmpy.s d1,d3,d1              x:(r4)+,d4.s   y:,d3.s      ;Xs   Xf  1  1

; Multiply coordinates by 1/W, scale and add offset (new point)

  fmpy.s d0,d4,d2              x:(r1)+n1,d5.s              ;Yn      1  1
  fmpy.s d2,d1,d2              x:(r4)+,d4.s   y:,d6.s      ;Ys   Yf  1  1
  fmpy   d5,d4,d0 fadd.s d3,d2 d5.s,x:(r1)+                ;Yo      1  1
  fmpy.s d0,d1,d5              x:(r0)+,d0.s   d2.s,y:(r5)+ ;X    X1  1  1
                  fadd.s d5,d3 x:(r6),d7.l                 ;Cnt     1  1
  dec    d7                                   d3.s,y:(r5)+ ;     Y1  1  1
  jne    _accept_loop                                      ;        2  2
  jmp    _end                                              ;        2  2
```

```
;-----------------------------------------------------------
;
;             Double point clipping routine
;
;-----------------------------------------------------------


; Dispatch to old point clipping routines


_r_clip2
  move                         d7.l,x:(r6)    y:(r1)+,d1.l ;Cnt r1+ 1  1
  move                                        y:(r1)-,d1.s ;    Wo  1  1
  move                         x:(r1)+,d5.s                ;Xo      1  1
  move                         n0,d7.s                     ;        1  1
                  fsub.s d1,d5 d5.s,d6.s                   ;        1  1
  fjsgt  _clip2_xop                                        ;        2  2
                  fadd.s d1,d6 x:(r1)-,d5.s                ;Yo      1  1
  fjslt  _clip2_xon                                        ;        2  2
                  fsub.s d1,d5 d5.s,d6.s                   ;        1  1
  fjsgt  _clip2_yop                                        ;        2  2
                  fadd.s d1,d6              y:(r1)+n1,d5.s ;Zo      1  1
  fjslt  _clip2_yon                                        ;        2  2
                  fsub.s d1,d5 d5.s,d6.s                   ;        1  1
  fjsgt  _clip2_zop                                        ;        2  2
  ftst   d6                    x:(r1)+,d5.s                ;Xn      1  1
  fjslt  _clip2_zon                                        ;        2  2
  move                                      d7.s,y:(r6)    ;    to  1  1


; Dispatch to new point clipping routines


  move                                      y:(r1),d1.s    ;    Wn  1  1
  move                         d8.s,d7.s                   ;    tn  1  1
                  fsub.s d1,d5 d5.s,d6.s                   ;        1  1
  fjsgt  _clip2_xnp                                        ;        2  2
                  fadd.s d1,d6 x:(r1)-,d5.s                ;Yn      1  1
  fjslt  _clip2_xnn                                        ;        2  2
                  fsub.s d1,d5 d5.s,d6.s                   ;        1  1
```

```
  fjsgt  _clip2_ynp                                      ;        2  2
                  fadd.s d1,d6             y:(r1)+n1,d5.s ;Zn      1  1
  fjslt  _clip2_ynn                                      ;        2  2
                  fsub.s d1,d5 d5.s,d6.s                 ;        1  1
  fjsgt  _clip2_znp                                      ;        2  2
  ftst   d6                                              ;        1  1
  fjslt  _clip2_znn                                      ;        2  2

; Check for rejection

  move                        x:(r1)+n1,d3.s y:(r6),d5.s  ;Xo  to  1  1
  fcmp   d5,d7                d7.s,d4.s                   ;        1  1
  fjlt   _clip2_reject                                   ;        2  2

; Calculate end point coordinates: X

  move                        x:(r1)+n1,d6.s              ;Xn      1  1
                  fsub.s d3,d6 d6.s,x:(r1)-               ;Xo      1  1
  fmpy.s d4,d6,d1                                         ;        1  1
  fmpy   d5,d6,d2 fadd.s d3,d1 x:(r1)+n1,d6.s              ;Yn      1  1
                  fadd.s d3,d2 x:(r1),d3.s                ;Yo      1  1

; Calculate end point coordinates: Y

                  fsub.s d3,d6 d6.s,x:(r1)+n1 d1.s,y:(r6)+ ;Yo  Xnd 1  1
  fmpy.s d4,d6,d1             d2.s,d0.s                   ;        1  1
  fmpy   d5,d6,d2 fadd.s d3,d1             y:(r1)+n1,d6.s ;    Wn  1  1
                  fadd.s d3,d2             y:(r1)+n1,d3.s ;    Wo  1  1

; Calculate end point coordinates: W

                  fsub.s d3,d6             d1.s,y:(r6)+   ;Ynd     1  1
  fmpy.s d4,d6,d1             d2.s,d7.s    y:(r1)+n1,d4.s ;    Wn  1  1
```

```
  fmpy   d5,d6,d2 fadd.s d3,d1              d4.s,y:(r1)+   ;    Wo  1  1
                  fadd.s d3,d2              d1.s,y:(r6)    ;Wnd     1  1

; Calculate reciprocal 1/W (old point)

  fseedd d2,d6                                             ;        1  1
  fmpy.s d2,d6,d1              d9.s,d4.s                   ;        1  1
                  fsub.s d1,d4 d4.s,d3.s                   ;        1  1
  fmpy.s d1,d4,d1                             (r4)-n4      ;   r4-2 1  1
  fmpy   d6,d4,d1 fsub.s d1,d3                             ;        1  1
  fmpy.s d1,d3,d1              x:(r4)+,d4.s   y:,d3.s      ;Xs  Xf  1  1

; Multiply coordinates by 1/W, scale and add offset (old point)

  fmpy.s d0,d4,d2                                          ;        1  1
  fmpy.s d2,d1,d2              x:(r4)-,d4.s y:,d6.s        ;Ys  Yf  1  1
  fmpy   d7,d4,d3 fadd.s d3,d2              y:(r1)+n1,d4.s ;    Zn  1  1
  fmpy.s d3,d1,d3                           d4.s,y:(r1)+n1 ;    Zo  1  1
                  fadd.s d6,d3              d2.s,y:(r5)+   ;    X1  1  1
  move                                      y:(r6)-,d1.s   ;    Wnd 1  1
  move                                      d3.s,y:(r5)+   ;    Y1  1  1

; Calculate reciprocal 1/W (new point)

  fseedd d1,d6                                             ;        1  1
  fmpy.s d1,d6,d1              d9.s,d4.s                   ;        1  1
                  fsub.s d1,d4 d4.s,d3.s     y:(r6)-,d5.s  ;    Ynd 1  1
  fmpy.s d1,d4,d1                            y:(r6),d0.s   ;    Xnd 1  1
  fmpy   d6,d4,d1 fsub.s d1,d3                             ;        1  1
  fmpy.s d1,d3,d1              x:(r4)+,d4.s  y:,d3.s       ;Xs  Xf  1  1

; Multiply coordinates by 1/W, scale and add offset (old point)
```

```
  fmpy.s d0,d4,d2                                       ;        1  1
  fmpy.s d2,d1,d2              x:(r4)+,d4.s   y:,d6.s      ;Ys   Yf  1  1
  fmpy   d5,d4,d3 fadd.s d3,d2                          ;        1  1
  fmpy.s d3,d1,d3              x:(r6),d7.l              ;        1  1
                  fadd.s d6,d3 x:(r0)+,d0.s   d2.s,y:(r5)+ ;X    X1  1  1
  move                                        d3.s,y:(r5)+ ;    Y1  1  1
  dec    d7                                   n5,y:(r5)+   ;   -1.0 1  1
  jne    _reject_loop                                   ;        2  2
  jmp    _end                                           ;        2  2



; Reject double-clipped line

_clip2_reject
  move                         x:(r6),d7.l              ;        1  1
  move                         x:(r1)+n1,d0.s y:,d1.s      ;Xn   Zn  1  1
  move                         d0.s,x:(r1)-   d1.s,y:      ;Xo   Zo  1  1
  move                         x:(r1)+n1,d0.s y:,d1.s      ;Yn   Wn  1  1
  move                         d0.s,x:(r1)+   d1.s,y:      ;Yo   Wo  1  1
  dec    d7                    x:(r0)+,d0.s             ;        1  1
  jne    _reject_loop                                   ;        2  2


; Terminate endpoint list and exit

_end
  move                         n5,d0.s                  ;-1.0    1  1
  move                         (r5)-                    ;        1  1
  move                                        y:(r5),d1.s  ;        1  1
  fcmp   d0,d1                                          ;        1  1
  fjeq  _end1                                           ;        2  2
  move                         (r5)+                    ;        1  1

_end1
  move                         #-2.0,d0.s               ;        2  2
  move                                        d0.s,y:(r5)+ ;        1  1
  rts                                                   ;        2  2
```

```
;------------------------------------------------------------
;
;             Single point clipping routines
;
;------------------------------------------------------------

; x = w boundary

_clip1_xp
  move                                           y:(r1)-,d4.s ;W1      1  1
  fmpy.s d2,d4,d3              x:(r1)+,d0.s   d2.s,d7.s    ;X1      1  1
                fsub.s d0,d4 x:(r1)-,d0.s                  ;Y1      1  1
  fmpy.s d1,d4,d1                                          ;        1  1
  fmpy   d4,d5,d2 fsub.s d3,d1 d0.s,d5.s                   ;        1  1
  fmpy.s d5,d7,d3                                          ;        1  1
  fmpy   d4,d6,d3 fsub.s d3,d2                y:(r1)+,d4.s ;Z1      1  1
  fmpy.s d4,d7,d2              d2.s,d5.s                   ;        1  1
                fsub.s d2,d3 d1.s,d0.s                     ;        1  1
  move                         d3.s,d6.s                   ;        1  1
  rts                                                      ;        2  2


; x = -w boundary


_clip1_xn
  move                                           y:(r1)-,d4.s ;W1      1  1
  fmpy.s d1,d4,d3              x:(r1)+,d0.s   d1.s,d7.s    ;X1      1  1
                fadd.s d0,d4 x:(r1)-,d0.s                  ;Y1      1  1
  fmpy.s d2,d4,d2                                          ;        1  1
  fmpy   d4,d5,d1 fsub.s d3,d2 d0.s,d5.s                   ;        1  1
  fmpy.s d5,d7,d3                                          ;        1  1
  fmpy   d4,d6,d3 fsub.s d3,d1                y:(r1)+,d4.s ;Z1      1  1
  fmpy.s d4,d7,d1              d1.s,d5.s                   ;        1  1
                fsub.s d1,d3 d2.s,d0.s                     ;        1  1
  fneg.s d0                    d3.s,d6.s                   ;        1  1
  rts                                                      ;        2  2
```

```
; y = w boundary

_clip1_yp
  move                                        y:(r1),d4.s  ;W1      1  1
  fmpy.s d2,d4,d3              x:(r1)-,d5.s   d2.s,d7.s    ;Y1      1  1
                fsub.s d5,d4 x:(r1),d5.s                   ;X1      1  1
  fmpy.s d1,d4,d1                                          ;        1  1
  fmpy   d0,d4,d2 fsub.s d3,d1                             ;        1  1
  fmpy.s d5,d7,d3                                          ;        1  1
  fmpy   d4,d6,d3 fsub.s d3,d2                y:(r1)+,d4.s ;Z1      1  1
  fmpy.s d4,d7,d2              d2.s,d0.s                   ;        1  1
                fsub.s d2,d3 d1.s,d5.s                     ;        1  1
  move                         d3.s,d6.s                   ;        1  1
  rts                                                      ;        2  2

; y = -w boundary

_clip1_yn
  move                                        y:(r1),d4.s  ;W1      1  1
  fmpy.s d1,d4,d3              x:(r1)-,d5.s   d1.s,d7.s    ;Y1      1  1
                fadd.s d5,d4 x:(r1),d5.s                   ;X1      1  1
  fmpy.s d2,d4,d2                                          ;        1  1
  fmpy   d0,d4,d1 fsub.s d3,d2                             ;        1  1
  fmpy.s d5,d7,d3                                          ;        1  1
  fmpy   d4,d6,d3 fsub.s d3,d1                y:(r1)+,d4.s ;Z1      1  1
  fmpy.s d4,d7,d1              d1.s,d0.s                   ;        1  1
                fsub.s d1,d3 d2.s,d5.s                     ;        1  1
  fneg.s d5                    d3.s,d6.s                   ;        1  1
  rts                                                      ;        2  2

; Clip at z = w boundary

_clip1_zp
  move                                        y:(r1)-,d4.s ;W1      1  1
```

```
  fmpy.s d2,d4,d3               d2.s,d7.s      y:(r1),d6.s  ;Z1      1  1
                fsub.s d6,d4 x:(r1)+,d6.s                   ;X1      1  1
  fmpy.s d1,d4,d1                                           ;        1  1
  fmpy   d0,d4,d2 fsub.s d3,d1                              ;        1  1
  fmpy.s d6,d7,d3                                           ;        1  1
  fmpy   d4,d5,d3 fsub.s d3,d2 x:(r1),d4.s                  ;Y1      1  1
  fmpy.s d4,d7,d2               d2.s,d0.s                   ;        1  1
                fsub.s d2,d3 d1.s,d6.s                      ;        1  1
  move                          d3.s,d5.s                   ;        1  1
  rts                                                       ;        2  2


; Clip at z = 0 boundary


_clip1_zn
  move                                         y:(r1)-,d2.s ;W1      1  1
  fmpy.s d2,d6,d2                              y:(r1),d4.s  ;Z1      1  1
  fmpy.s d1,d4,d1               x:(r1)+,d7.s                ;X1      1  1
  fmpy   d0,d4,d2 fsub.s d2,d1                              ;        1  1
  fmpy.s d6,d7,d0               x:(r1),d7.s                 ;Y1      1  1
  fmpy   d6,d7,d3 fsub.s d0,d2                              ;        1  1
  fmpy.s d4,d5,d5               d2.s,d0.s                   ;        1  1
                fsub.s d3,d5 n0,d6.s                        ;        1  1
  rts                                                       ;        2  2



;---------------------------------------------------------
;
;            Double point clipping routines
;
;---------------------------------------------------------


; XOld = WOld boundary


_clip2_xop
  move                          (r1)+n1                     ;        1  1
```

```
  move                                    y:(r1)-,d3.s ;Wn      1  1
                fadd.s d3,d5 x:(r1)-,d3.s   d5.s,d0.s    ;Xn      1  1
                fsub.s d3,d5                             ;        1  1
  fseedd d5,d4                                           ;        1  1
  fmpy.s d5,d4,d5              d9.s,d2.s                 ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d2 d2.s,d3.s                 ;        1  1
  fmpy.s d5,d2,d5              d2.s,d4.s                 ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d3                           ;        1  1
  fmpy.s d0,d3,d0                                        ;        1  1
  fcmp   d7,d0                                           ;        1  1
  ftfr.s d0,d7    ffgt                                   ;        1  1
  rts                                                    ;        2  2

; XOld = -WOld boundary

_clip2_xon
  move                         (r1)-                     ;        1  1
  move                                    y:(r1)-,d3.s ;Wn      1  1
                fsub.s d3,d6 x:(r1)+n1,d3.s d6.s,d0.s    ;Xn      1  1
                fsub.s d3,d6                             ;        1  1
  fseedd d6,d4                                           ;        1  1
  fmpy.s d6,d4,d6              d9.s,d2.s                 ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d2 d2.s,d3.s                 ;        1  1
  fmpy.s d6,d2,d6              d2.s,d4.s                 ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d3                           ;        1  1
  fmpy.s d0,d3,d0                                        ;        1  1
  fcmp   d7,d0                                           ;        1  1
  ftfr.s d0,d7    ffgt                                   ;        1  1
  rts                                                    ;        2  2

; YOld = WOld boundary

_clip2_yop
  move                         (r1)-                     ;        1  1
  move                                    y:(r1),d3.s  ;Wn      1  1
                fadd.s d3,d5 x:(r1)+,d3.s   d5.s,d0.s    ;Yn      1  1
```

```
                fsub.s d3,d5                           ;        1  1
  fseedd d5,d4                                         ;        1  1
  fmpy.s d5,d4,d5              d9.s,d2.s               ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d2 d2.s,d3.s               ;        1  1
  fmpy.s d5,d2,d5              d2.s,d4.s               ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d3                         ;        1  1
  fmpy.s d0,d3,d0                                      ;        1  1
  fcmp   d7,d0                                         ;        1  1
  ftfr.s d0,d7    ffgt                                 ;        1  1
  rts                                                  ;        2  2


; YOld = -WOld boundary


_clip2_yon
  move                         (r1)+                   ;        1  1
  move                                    y:(r1),d3.s  ;Wn      1  1
                fsub.s d3,d6 x:(r1)-,d3.s   d6.s,d0.s  ;Yn      1  1
                fsub.s d3,d6                           ;        1  1
  fseedd d6,d4                                         ;        1  1
  fmpy.s d6,d4,d6              d9.s,d2.s               ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d2 d2.s,d3.s               ;        1  1
  fmpy.s d6,d2,d6              d2.s,d4.s               ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d3                         ;        1  1
  fmpy.s d0,d3,d0                                      ;        1  1
  fcmp   d7,d0                                         ;        1  1
  ftfr.s d0,d7    ffgt                                 ;        1  1
  rts                                                  ;        2  2



; ZOld = WOld boundary

_clip2_zop
  move                         (r1)+                   ;        1  1
  move                                    y:(r1)-,d3.s ;Wn      1  1
                fadd.s d3,d5 d5.s,d0.s    y:(r1),d3.s  ;Zn      1  1
                fsub.s d3,d5                           ;        1  1
  fseedd d5,d4                                         ;        1  1
```

```
  fmpy.s d5,d4,d5              d9.s,d2.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d2 d2.s,d3.s                     ;        1  1
  fmpy.s d5,d2,d5              d2.s,d4.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d3                               ;        1  1
  fmpy.s d0,d3,d0                                            ;        1  1
  fcmp   d7,d0                                               ;        1  1
  ftfr.s d0,d7    ffgt                                       ;        1  1
  rts                                                        ;        2  2


; ZOld = 0 boundary

_clip2_zon
  move                         (r1)-                         ;        1  1
  move                                        y:(r1)+,d3.s   ;Zn      1  1
                  fsub.s d3,d6 d6.s,d0.s                     ;        1  1
  fseedd d6,d4                                               ;        1  1
  fmpy.s d6,d4,d6              d9.s,d2.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d2 d2.s,d3.s                     ;        1  1
  fmpy.s d6,d2,d6              d2.s,d4.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d3                               ;        1  1
  fmpy.s d0,d3,d0                                            ;        1  1
  fcmp   d7,d0                                               ;        1  1
  ftfr.s d0,d7    ffgt                                       ;        1  1
  rts                                                        ;        2  2


; XNew = WNew boundary

_clip2_xnp
  move                         (r1)+n1                       ;        1  1
  move                                        y:(r1)-,d0.s   ;Wo      1  1
  move                         x:(r1)-,d2.s                  ;Xo      1  1
                  fsub.s d2,d0                               ;        1  1
                  fadd.s d0,d5                               ;        1  1
  fseedd d5,d4                                               ;        1  1
  fmpy.s d5,d4,d5              d9.s,d2.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d2 d2.s,d3.s                     ;        1  1
  fmpy.s d5,d2,d5              d2.s,d4.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d3                               ;        1  1
```

```
  fmpy.s d0,d3,d0                                            ;        1  1
  fcmp   d7,d0                                               ;        1  1
  ftfr.s d0,d7    fflt                                       ;        1  1
  rts                                                        ;        2  2

; XNew = -WNew boundary

_clip2_xnn
  move                         (r1)-                         ;        1  1
  move                                        y:(r1)-,d3.s ;Wo      1  1
  move                         x:(r1)+n1,d2.s                ;Xo      1  1
                  fadd.s d3,d2                               ;        1  1
                  fsub.s d6,d2 d2.s,d0.s                     ;        1  1
  fseedd d2,d4                                               ;        1  1
  fmpy.s d2,d4,d6              d9.s,d2.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d2 d2.s,d3.s                     ;        1  1
  fmpy.s d6,d2,d6              d2.s,d4.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d3                               ;        1  1
  fmpy.s d0,d3,d0                                            ;        1  1
  fcmp   d7,d0                                               ;        1  1
  ftfr.s d0,d7    fflt                                       ;        1  1
  rts                                                        ;        2  2


; YNew = WNew boundary

_clip2_ynp
  move                         (r1)-                         ;        1  1
  move                         x:(r1)+,d2.s   y:,d0.s        ;Yo  Wo  1  1
                  fsub.s d2,d0                               ;        1  1
                  fadd.s d0,d5                               ;        1  1
  fseedd d5,d4                                               ;        1  1
  fmpy.s d5,d4,d5              d9.s,d2.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d2 d2.s,d3.s                     ;        1  1
  fmpy.s d5,d2,d5              d2.s,d4.s                     ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d3                               ;        1  1
  fmpy.s d0,d3,d0                                            ;        1  1
  fcmp   d7,d0                                               ;        1  1
  ftfr.s d0,d7    fflt                                       ;        1  1
  rts                                                        ;        2  2
```

```
; YNew = -WNew boundary

_clip2_ynn
  move                         (r1)+                          ;        1  1
  move                         x:(r1)-,d2.s   y:,d3.s         ;Yo  Wo  1  1
                fadd.s d3,d2                                  ;        1  1
                fsub.s d6,d2 d2.s,d0.s                        ;        1  1
  fseedd d2,d4                                                ;        1  1
  fmpy.s d2,d4,d6              d9.s,d2.s                      ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d2 d2.s,d3.s                      ;        1  1
  fmpy.s d6,d2,d6              d2.s,d4.s                      ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d3                                ;        1  1
  fmpy.s d0,d3,d0                                             ;        1  1
  fcmp   d7,d0                                                ;        1  1
  ftfr.s d0,d7    fflt                                        ;        1  1
  rts                                                         ;        2  2

; ZNew = WNew boundary


_clip2_znp
  move                         (r1)+                          ;        1  1
  move                                        y:(r1)-,d0.s    ;Wo      1  1
  move                                        y:(r1),d2.s     ;Zo      1  1
                fsub.s d2,d0                                  ;        1  1
                fadd.s d0,d5                                  ;        1  1
  fseedd d5,d4                                                ;        1  1
  fmpy.s d5,d4,d5              d9.s,d2.s                      ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d2 d2.s,d3.s                      ;        1  1
  fmpy.s d5,d2,d5              d2.s,d4.s                      ;        1  1
  fmpy   d0,d4,d0 fsub.s d5,d3                                ;        1  1
  fmpy.s d0,d3,d0                                             ;        1  1
  fcmp   d7,d0                                                ;        1  1
  ftfr.s d0,d7    fflt                                        ;        1  1
  rts                                                         ;        2  2


; ZNew = 0 boundary
```

```
_clip2_znn
  move                         d6.s,d0.s      y:(r1),d6.s  ;Zo      1  1
                fsub.s d0,d6 d6.s,d0.s                     ;        1  1
  fseedd d6,d4                                             ;        1  1
  fmpy.s d6,d4,d6              d9.s,d2.s                   ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d2 d2.s,d3.s                   ;        1  1
  fmpy.s d6,d2,d6              d2.s,d4.s                   ;        1  1
  fmpy   d0,d4,d0 fsub.s d6,d3                             ;        1  1
  fmpy.s d0,d3,d0                                          ;        1  1
  fcmp   d7,d0                                             ;        1  1
  ftfr.s d0,d7    fflt                                     ;        1  1
  rts                                                      ;        2  2
```

B.1.45 Walsh-Hadamard Transforms

The Walsh-Hadamard transform (WHT) is an orthogonal transform requiring only additions and subtractions. The transform can be decomposed similar to the fast fourier transform (FFT) to yield a fast implementation of the WHT.

B.1.45.1 In-place WHT

Since the WHT requires 2 loads and 2 stores per butterfly, the maximum throughput for a WHT butterfly is 4 cycles. This implementation executes 2 butterflies in 8 cycles on the inner loop for a 4N per butterfly execution speed. The last stage is split out and also executes 2 butterflies in 8 cycles for each pass of the loop.

In this example, a 16 point transform is performed. The input data are in X:0-f and the output is in x:0-f in bit reversed order.

Execution speed for a 1024 point WHT is 1.68 milliseconds at 13.5 MIPS.

```
    page    132,60,1,1
;
;    Implements the Walsh-Hadamard Transform
;

iord   equ    4         ;order of transform=log2(npoints)
n      equ    1<<iord    ;length of transform

       org    x:0
data
```

```
        dc     0.0000000E+00
        dc     2.000000
        dc     3.000000
        dc     8.000000
        dc     9.000000
        dc     12.00000
        dc     15.00000
        dc     19.00000
        dc     20.00000
        dc     22.00000
        dc     23.00000
        dc     24.00000
        dc     25.00000
        dc     26.00000
        dc     27.00000
        dc     28.00000


        org     p:$100
start
        move    #1,d7.l         ;number of groups
        move    #n/4,d6.l       ;number of butterflies/group

        move    #data,r0        ;upper leg pointer
        move    #n/2,n0         ;offset between groups
        move    #n-1,m0         ;mod N

        move    #data+n/2,r4    ;lower leg pointer
        move    #n/2,n4         ;offset between groups

        do     #iord-1,_stage   ;do stages
        do     d7.l,_grp        ;do groups
        do     d6.l,_bfly       ;do butterflies
        move             x:(r0)+,d0.s    ;upper leg 1
        move             x:(r4)+,d1.s    ;lower leg 1
        faddsub.s d0,d1  x:(r0)-,d2.s    ;upper leg 2, point back to 1
        move             x:(r4)-,d3.s    ;lower leg 2, point back to 1
        faddsub.s d2,d3  d1.s,x:(r0)+    ;save upper 1, point to 2
        move             d0.s,x:(r4)+    ;save lower 1, point to 2
        move             d3.s,x:(r0)+    ;save upper 2, point to next
```

```
    move              d2.s,x:(r4)+    ;save lower 2, point to next
_bfly
    move    x:(r0)+n0,d0.s  y:(r4)+n4,d1.s    ;adjust r0,r4
_grp
    lsr    d6    d6.l,n0     ;bflys/2, make old value new offset
    lsl    d7    n0,n4       ;ngroups*2, move new offset
    lea    (r0)+n0,r4        ;new lower leg pointer
_stage
    move   #3,n0             ;offset between 2 butterflies-1
    move   n0,n4             ;same
    move   (r4)+             ;point r4 to second bfly
    do     #n/4,_laststage   ;do last stage, 2 bflys at a time
    move             x:(r0)+,d0.s   ;get upper of bfly 1
    move             x:(r0)-,d1.s   ;get lower of bfly 1, point to upper
    faddsub.s d0,d1  x:(r4)+,d2.s   ;get upper of bfly 2
    move             x:(r4)-,d3.s   ;get lower of bfly 1, point to upper
    faddsub.s d2,d3  d1.s,x:(r0)+   ;save upper 1
    move             d0.s,x:(r0)+n0 ;save lower 1, point to next group
    move             d3.s,x:(r4)+   ;save upper 2
    move             d2.s,x:(r4)+n4 ;save lower 2, point to next group
_laststage
    end
```

B.1.45.2 Out-of-place WHT

Since the WHT requires 2 loads and 2 stores per butterfly, the maximum throughput for a WHT butterfly is 4 cycles. However, if the data is split between two memories, then the 2 loads and 2 stores can be performed in 2 cycles. Thus, it is possible to execute each butterfly in 2 cycles. This implementation takes the input data in a single memory space and on the first stage of the transform, splits the data into X and Y memory. The middle stages then perform 4 WHT butterflies in 8 cycles. The last stage is split out and also performs 4 WHT butterflies in 8 cycles. Thus, except for the first stage, all WHT butterflies are performed in 2 cycles.

In this example, a 16 point transform is performed. The input data are in X:0-f and the output is split between X and Y memory. The first 8 output values are at x:0-7 and the next 8 output values are at y:0-7 in bit reversed order starting at x:0. To increase execution speed, an extra block of memory is used at y:0-7. Thus, with this algorithm, an extra block of memory is required in Y memory equal to one-half of the transform data size in X memory.

If both X and Y memory are on the same port (A or B), then all X and Y memory references are performed on the same port. Thus, the WHT butterfly executes in 4 cycles. This gives an execution speed of 1.64 milliseconds at 13.5 MIPS. However, if X memory is on port A and Y memory is on port B, then the memory bandwidth is doubled and an X memory access and Y memory access can occur in a single cycle. This gives an execution speed of 0.939 milliseconds at 13.5 MIPS.

```
      page      132,60,1,1
;
;      Implements the Walsh-Hadamard Transform
;

iord  equ      4           ;order of transform=log2(npoints)
n     equ      1<<iord      ;length of transform

      org      x:$1000
data
      dc      0.0000000E+00
      dc      2.000000
      dc      3.000000
      dc      8.000000
      dc      9.000000
      dc      12.00000
      dc      15.00000
      dc      19.00000
      dc      20.00000
      dc      22.00000
      dc      23.00000
      dc      24.00000
      dc      25.00000
      dc      26.00000
      dc      27.00000
      dc      28.00000


      org      p:$100
start
      move   #data,r0          ;point to upper leg
      move   #data+n/2,r4      ;point to lower leg
      do     #n/4,_firststage  ;do first stage. split into X and Y
      move             x:(r0)+,d0.s   ;get upper leg of bfly 1
      move             x:(r4)+,d1.s   ;get lower leg of bfly 1
      faddsub.s d0,d1  x:(r0)-,d2.s   ;get upper leg of bfly 2
      move             x:(r4)+,d3.s   ;get lower leg of bfly 2
      faddsub.s d2,d3  d1.s,x:(r0)    ;save sum 1
      move             d0.s,y:(r0)+   ;save dif 1
      move             d3.s,x:(r0)    ;save sum 2
```

```
        move             d2.s,y:(r0)+   ;save dif 2

_firststage
        nop
        nop
        move   #data,r0          ;point to data
        move   #n/2-1,m0         ;mod n/2
        move   #n/4,n0           ;offset to next group

        move   #data+n/4,r4      ;point to lower leg of half
        move   #n/4,n4           ;offset to next group

        move   #1,d8.l           ;number of groups/stage
        move   #n/8,d9.l         ;number of bflys/group

        do     #iord-2,_mid      ;do middle part of transform
        move   d8.l,d7.l         ;get group count
        do     d7.l,_grps        ;do groups
        move   d9.l,d7.l         ;get bfly count
        do     d7.l,_bfly        ;do bflys
        move             x:(r0)+,d0.s  y:,d4.s    ;upper x,y #1
        move             x:(r4)+,d1.s  y:,d5.s    ;lower x,y #1
        faddsub.s d0,d1  x:(r0)-,d2.s  y:,d6.s    ;upper x,y #2
        faddsub.s d4,d5  x:(r4)-,d3.s  y:,d7.s    ;lower x,y #2
        faddsub.s d2,d3  d1.s,x:(r0)+  d5.s,y:    ;save sum x,y #1
        faddsub.s d6,d7  d0.s,x:(r4)+  d4.s,y:    ;save dif x,y #1
        move             d3.s,x:(r0)+  d7.s,y:    ;save sum x,y #2
        move             d2.s,x:(r4)+  d6.s,y:    ;save dif x,y #2
_bfly
        move   x:(r0)+n0,d0.s  y:(r4)+n4,d1.s      ;adj r0,r4
_grps
        move   d9.l,d7.l         ;get # bflys/stage
        lsr    d7        d7.l,n0 ;divide # bflys by 2, divide offset by 2
        move   d7.l,d9.l         ;save # bflys/stage

        move   d8.l,d6.l         ;get # of groups/stage
        lsl    d6        n0,n4   ;multiply # groups by 2,copy offset
        move   d6.l,d8.l         ;save new # groups/stage

        lea   (r0)+n0,r4         ;update other pointer
```

```
_mid
      move   #3,n0              ;new offset
      move   n0,n4              ;copy
     move   (r4)+              ;point to second butterfly
      do     #n/8,_laststage   ;do last stage, 4 bflys at a time
      move             x:(r0)+,d0.s   y:,d4.s      ;upper x,y #1
      move             x:(r0)-,d1.s   y:,d5.s      ;lower x,1 #1
      faddsub.s d0,d1 x:(r4)+,d2.s   y:,d6.s      ;upper x,y #2
      faddsub.s d4,d5 x:(r4)-,d3.s   y:,d7.s      ;lower x,y #2
      faddsub.s d2,d3 d1.s,x:(r0)+   d5.s,y:      ;save upper x,y #1
      faddsub.s d6,d7 d0.s,x:(r0)+n0 d4.s,y:      ;save lower x,y #1
      move             d3.s,x:(r4)+   d7.s,y:      ;save upper x,y #2
      move             d2.s,x:(r4)+n4 d6.s,y:      ;save lower x,y #2
_laststage
      end
```

If it is desired to have the results in a single memory, then the last pass of the above algorithm can be modified to merge the data from X memory and Y memory back into X memory as the butterflies are performed. Each butterfly is read from a separate memory space but the outputs are written to a single memory space. This executes in 3 cycles per butterfly on the final stage. Note that the last stage performs 4 butterflies per loop and the loop takes 12 cycles for an average of 3 cycles per butterfly on the final stage.

```
    move    #data+n/2,r5     ;pointer to move back to X
      move    #3,n0            ;new offset
      move    n0,n4            ;copy
      move    (r4)+            ;point to second butterfly
      do      #n/8,_laststage  ;do last stage, 4 bflys at a time
      move             x:(r0)+,d0.s  y:,d4.s    ;upper x,y #1
      move             x:(r0)-,d1.s  y:,d5.s    ;lower x,1 #1
      faddsub.s d0,d1 x:(r4)+,d2.s  y:,d6.s    ;upper x,y #2
      faddsub.s d4,d5 x:(r4)-,d3.s  y:,d7.s    ;lower x,y #2
      faddsub.s d2,d3 d1.s,x:(r0)+              ;save upper x #1
      move             d5.s,x:(r5)+              ;move upper #1 back to X
      faddsub.s d6,d7 d0.s,x:(r0)+n0            ;save lower x #1
      move             d4.s,x:(r5)+              ;move lower #1 back
      move             d3.s,x:(r4)+              ;save upper x,y #2
      move             d7.s,x:(r5)+              ;move upper #2 back
      move             d2.s,x:(r4)+n4            ;save lower x,y #2
      move             d6.s,x:(r5)+              ;move lower #2 back
_laststage
```

B.1.46 Evaluation of LOG(x)

Floating-point evaluation of $\log_2(x)$ can be performed by representing x as s*(2**e) where s is the significand and e is the unbiased exponent. Then, $\log_2(s*(2**e)) = \log_2(s) + e$. After extracting the significand s, $\log_2(s)$ can be evaluated with a polynomial. By adding the unbiased exponent, $\log_2(x)$ results. Various execution speeds and accuracies may be determined by using different order polynomials.

```
          page     132,60,1,1
           org      x:0
     polyc
           dc      0.6681523e-02       ;**8
           dc      -0.6736254e-01      ;**7
           dc      0.2584541e+00       ;**6
           dc      -0.3676691e+00      ;**5
           dc      -0.4461204e+00      ;**4
           dc      0.2740512e+01       ;**3
           dc      -0.5236615e+01      ;**2
           dc      0.6184454e+01       ;**1
           dc      -0.3072334e+01      ;**0

           org      p:$100
     ;
     ;      calculate d2=log2(d0)
     ;
                                                     Program   ICycles
                                                     Words
         getexp   d0,d7      #polyc,r0                  2         2
         fgetman  d0,d0                                 1         1
         fclr     d2         x:(r0)+,d1.s               1         1
         do       #9,_log2sig                           2         3
         fmpy.x   d2,d0,d2                              1         1
         fadd.x   d1,d2      x:(r0)+,d1.s               1         1
     _log2sig
         float.x  d7                                    1         1
         fadd.s   d7,d2                                 1         1
                                                       ---       ---
                                           Totals:      10        27
```

B.1.47 Evaluation of EXP2(x)

Floating-point evaluation of exp2(x) can be performed by representing x as i+f where f is the fractional part and i is the greatest integer in x that does not exceed x. Then, exp2(i+f) = exp2(f)*(2**i). After extracting the fractional part f, exp2(f) can be evaluated with a polynomial. By scaling by the integer part, exp2(x) results. Various execution speeds and accuracies may be determined by using different order polynomials.

```
        page    132,60,1,1
        org     x:0
polyc
        dc      -0.5770606e-03      ;**8
        dc       0.2093549e-02      ;**7
        dc      -.02777411e-02      ;**6
        dc       0.3357901e-02      ;**5
        dc       0.8940958e-02      ;**4
        dc       0.5558203e-01      ;**3
        dc       0.2402348e+00      ;**2
        dc       0.6931450e+00      ;**1
        dc       0.1000000e+01      ;**0

        org     p:$100
;
;       calculate d2=exp2(d0)
;
                                                       Program   ICycles
                                                       Words
    floor    d0,d7       #polyc,r0                        2         2
    fsub.x   d7,d0                                        1         1
    fclr     d2          x:(r0)+,d1.s                     1         1
    int      d7                                           1         1
    do       #9,_log2sig                                  2         3
    fmpy.x   d2,d0,d2    d7.l,d7.h                        1         1
    fadd.x   d1,d2       x:(r0)+,d1.s                     1         1
_log2sig
    fscale.s d7,d2                                        1         1
                                                         ---       ---
                                              Totals:    10        27
```

B.1.48 Vector Cross Product

The cross product of two vectors is always perpendicular to both of the vectors making this vector useful for 3D graphics, shading, and illumination. The three dimensional cross product a X b where a and b are {1 x 3} vectors can be written as the determinant:

```
 i  j  k
ax ay az
bx by bz
```

where i, j and k are the unit vectors in the x, y and z directions respectively. Expanding this determinant yields:

```
cx = ay bz - az by
cy = az bx - ax bz
cz = ax by - ay bx
```

where vector c is the cross product of a and b.

Memory Map: X Y

```
     r0 → ax       .
    m0=2  ay       .
  (mod 3) az       .
           .      bx ← r4
           .      by    m0=2
           .      bz    (mod 3)
     r1→ cx        .
          cy       .
          cz       .
```

```
move   #aaddr,r0               ; set up pointers
move   #2,m0
move   #baddr,r4
move   #2,m4
move   #caddr,r1
                                                          Program  ICycles
                                                          Words
move                         x:(r0)+,d6.s y:(r4)-,d7.s ;ax bx   1   1
move                         x:(r0)+,d6.s y:(r4)-,d7.s ;ay bz   1   1
fmpy.s d6,d7,d3              x:(r0)+,d6.s y:(r4)-,d7.s ;az by   1   1
fmpy.s d6,d7,d2                           y:(r4)-,d7.s ;   bx   1   1
fmpy   d6,d7,d1 fsub.s d2,d3 x:(r0)+,d6.s y:(r4)-,d7.s ;ax bz   1   1
fmpy.s d6,d7,d0              d3.s,x:(r1)+ y:(r4)-,d7.s ;cx by   1   1
fmpy   d6,d7,d3 fsub.s d0,d1 x:(r0)+,d6.s y:(r4)-,d7.s ;ay bx   1   1
fmpy.s d6,d7,d2              d1.s,x:(r1)+              ;cy      1   1
```

```
                fsub.s d2,d3                          ;        1   1
        move                   d3.s,x:(r1)+           ;cz      1   1
                                                              --- ---
                                                    Totals:   10  10
```

B.1.49 Power Function X**Y

Power Function XY**

X = Single Precision Float, Y = 5 Bit Integer

```
                                                          Program   ICycles
                                                          Words
;
;      d1.s = d4.s**d0.l
;
        andi    #0,ccr                ;clear ccr bits          1     1
        move    sr,d3.l               ;get sr                  1     1
        or      d0,d3      #1.0,d1.s  ;set ccr bits            2     2
        move    d3.l,sr               ;move power to CCR bits  1     1
        fmpy.x  d1,d4,d1  ifcs   ;bit 0, carry                 1     1
        fmpy.x  d4,d4,d4  ifal   ;do multiply w/o ccr update   1     1
        fmpy.x  d1,d4,d1  ifvs   ;bit 1, overflow              1     1
        fmpy.x  d4,d4,d4  ifal   ;do multiply w/o ccr update   1     1
        fmpy.x  d1,d4,d1  ifeq   ;bit 2, zero                  1     1
        fmpy.x  d4,d4,d4  ifal   ;do multiply w/o ccr update   1     1
        fmpy.x  d1,d4,d1  ifmi   ;bit 3, negative              1     1
        fmpy.x  d4,d4,d4  ifal   ;do multiply w/o ccr update   1     1
        fmpy.s  d1,d4,d1  ffinf  ;bit 4, infinity              1     1
                                                              ---   ---
                                                    Totals:   14    14
```

Power Function XY**

X = Single Precision Float, Y = 32 Bit Unsigned Integer

```
                                                          Program   ICycles
                                                          Words
;
;      d1.s = d4.s**d0.l
;
        move    #1.0,d1.s           ;initialize power          2     2
        do      #32,pwr                                        2     3
        lsr     d0                  ;get lsb                   1     1
        fmpy.x  d1,d4,d1  ifcs      ;multiply if bit set       1     1
```

```
        fmpy.x  d4,d4,d4            ;scale power              1     1
pwr                                                          ---   ---
                                                 Totals:     7    100
```

Power Function XY**

X = Single Precision Float, Y = 32 Bit Unsigned Integer

```
                                                        Program   ICycles
                                                        Words
;
;       d1.s = d4.s**d0.l
;
        bfind   d0,d0       #32,d2.l    ;how many bits          2     2
        move    d0.h,d3.l                                       1     1
        sub     d3,d2       #1.0,d1.s   ;initialize power       2     2
        do      d2.l,pwr                                        2     3
        lsr     d0                      ;get lsb                1     1
        fmpy.x  d1,d4,d1    ifcs        ;multiply if bit set    1     1
        fmpy.x  d4,d4,d4                ;scale power            1     1
pwr                                                            ---   ---
                                                 Totals:        10   3N+8
```

where N is the bit position of the most significant "one" bit in Y plus 1.

Power Function XY**

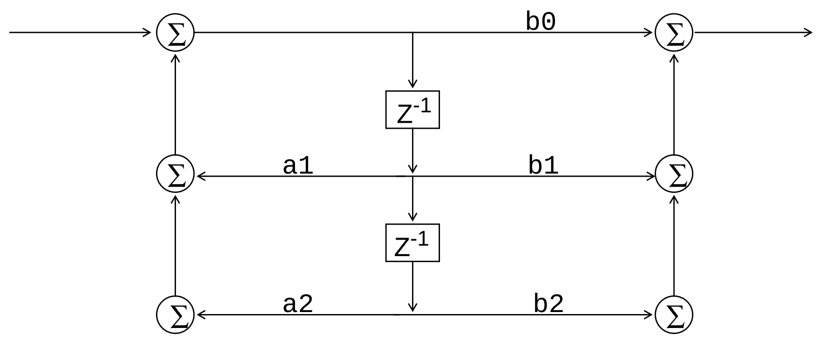
X = Single Precision Float, Y = Single Precision Float

```
                                                        Program   ICycles
                                                        Words
logc
        dc      0.6681523e-02       ;**8
        dc      -0.6736254e-01      ;**7
        dc      0.2584541e+00       ;**6
        dc      -0.3676691e+00      ;**5
        dc      -0.4461204e+00      ;**4
        dc      0.2740512e+01       ;**3
        dc      -0.5236615e+01      ;**2
        dc      0.6184454e+01       ;**1
        dc      -0.3072334e+01      ;**0
expc
        dc      -0.5770606e-03      ;**8
```

```
        dc       0.2093549e-02     ;**7
        dc      -.02777411e-02     ;**6
        dc       0.3357901e-02     ;**5
        dc       0.8940958e-02     ;**4
        dc       0.5558203e-01     ;**3
        dc       0.2402348e+00     ;**2
        dc       0.6931450e+00     ;**1
        dc       0.1000000e+01     ;**0

;
;        d2.s = d4.s**d0.s = exp2(d0 * log2(d4))
;
;        calculate d2=log2(d4)
;
        getexp   d4,d7      #logc,r0     ;get exponent         2     2
        fgetman  d4,d4                   ;get mantissa         1     1
        fclr     d2         x:(r0)+,d1.s ;clr sum, get coef    1     1
        do       #9,_log                 ;do log2(man)         2     3
        fmpy.x   d2,d4,d2                ;sum*x                1     1
        fadd.x   d1,d2      x:(r0)+,d1.s ;sum*x+coef, coef     1     1
_log
        float.x  d7                      ;float exponent       1     1
        fadd.s   d7,d2                   ;add log2(man)        1     1
;
        fmpy.x   d2,d0,d0                ;y*log2(x)            1     1
;
;        calculate d2=exp2(d0)
;
        floor    d0,d7     #expc,r0      ;get lowest int       2     2
        fsub.x   d7,d0                   ;get fraction part    1     1
        fclr     d2        x:(r0)+,d1.s                        1     1
        int      d7                      ;get lowest int       1     1
        do       #9,_exp                                       2     3
        fmpy.x   d2,d0,d2  d7.l,d7.h     ;sum*x, move scale    1     1
        fadd.x   d1,d2     x:(r0)+,d1.s  ;+coef, get next coef 1     1
_exp
        fscale.s d7,d2                   ;exp2(y*log2(x))      1     1
                                                              ---   ---
                                                    Totals:    21    55
```

B.1.50 Cascaded Five Coefficient Biquad Filter

Filter Section:



Program Words **ICycles**

```
nsec   equu   3
       org     x:0
states ds      2*nsec
       org     y:0
coef
    dc    -.68461698E+00    ;/* section  1 A2 */
    dc    .16526726E+01     ;/* section  1 A1 */
    dc    .83384343E-02     ;/* section  1 B2 */
    dc    .16676869E-01     ;/* section  1 B1 */
    dc    .83384343E-02     ;/* section  1 B0 */

    dc    -.75893794E+00    ;/* section  2 A2 */
    dc    .17255842E+01     ;/* section  2 A1 */
    dc    .90060414E-02     ;/* section  2 B2 */
    dc    .18012083E-01     ;/* section  2 B1 */
    dc    .90060414E-02     ;/* section  2 B0 */

    dc    -.90446499E+00    ;/* section  3 A2 */
    dc    .18683517E+01     ;/* section  3 A1 */
    dc    .25061846E+00     ;/* section  3 B2 */
    dc    .50123692E+00     ;/* section  3 B1 */
    dc    .25061846E+00     ;/* section  3 B0 */
;
;    input in d2
;
    move    #states,r0
```

```
        move    #coef,r4
        nop
        fclr    d1                      x:(r0)+,d4.s  y:(r4)+,d6.s 1    1
        do      #nsec,loop                                         2    3
        fmpy    d4,d6,d0  fadd.s d1,d2  x:(r0)-,d5.s  y:(r4)+,d6.s 1    1
        fmpy    d5,d6,d1  fadd.s d2,d0  d5.s,x:(r0)+  y:(r4)+,d6.s 1    1
        fmpy    d6,d4,d1  fadd.s d1,d0                y:(r4)+,d6.s 1    1
        fmpy.s  d6,d5,d2                d0.s,x:(r0)+  y:(r4)+,d4.s 1    1
        fmpy    d4,d0,d1  fadd.s d1,d2  x:(r0)+,d4.s  y:(r4)+,d6.s 1    1
    loop
        move    d2.s,y:output
                                                                  ---  ---
                                                       Totals:   8   5N+4
```

B.1.51 Four Quadrant Trigonometric SINE (CORDIC Algorithm)

```
    page    132,60,1,1
    opt    mex,cex
tabsize    equ    16

    org    x:0
scale    set    1.0
tantab
tanarg   set    45.0*3.14159/180.0
         dup    tabsize
scale    set    scale*@cos(tanarg)
         dc     @tan(tanarg)
tanarg   set    tanarg/2.0
         endm

    org    p:$100
;
;    Do argument reduction, input in d6 in degrees
;
  move              #-180.0,d7.s     ;get range min
  fadd.x  d7,d6     #1.0/360.0,d5.s  ;adjust to min, get range
  fmpy.x  d5,d6,d6                   ;reduce range
  floor   d6,d5                      ;get int part
  fsub.x  d5,d6     #360.0,d5.s      ;get frac part, spread
  fmpy.x  d5,d6,d6                   ;spread fraction part to range
  fadd.x  d7,d6                      ;adjust to min
;
;    Input angle in d6 in degrees, -180 < d6 < 180
;
  fabs.x  d6        d6.s,d3.s        ;make positive, save sign
```

```
  move              #90.0,d7.s       ;get pi/2
  fcmp    d7,d6    #180.0,d7.s      ;see if greater than 90
  fsub.x  d6,d7    ffge             ;reduce to less than 90
  ftfr.x  d6,d7    fflt             ;copy if no change
;
;    First quadrant CORDIC trig computation
;    Input angle in d7 in degrees
;    Output d1=sine, d0=cosine
;
  move              #tantab,r0       ;point to tangent table
  fclr    d1       #scale,d0.s      ;y=0, x=scale
  fclr    d5       #45.0,d6.s       ;z=0,alp=45
  do      #tabsize,_cordic
  fcmp    d5,d7    x:(r0)+,d4.s     ;angle < z? get tangent
  fneg.x  d4       fflt             ;yes, rotate cw
  fsub.x  d6,d5    fflt             ;yes, subtract angle
  fadd.x  d6,d5    ffge             ;no, add angle for ccw
  fmpy.x  d1,d4,d2                  ;y*tan
  fmpy    d0,d4,d2 fsub.x d2,d0     ;x*tan, x’=x-y*tan
  fadd.x  d2,d1                     ;y’=y+x*tan
  fscale.x #-1,d6                   ;alp=alp/2
_cordic
  fcopys.s d3,d1                    ;fix sign of sine
  end
```

	Program Words	ICycles
Argument Reduction	10	10
Quadrantizing	7	7
CORDIC Algorithm	16	8N+9
	----	-------
Totals:	33	8N+26

B.1.52 Four Quadrant Trigonometric COSINE (CORDIC Algorithm)

```
    page    132,60,1,1
    opt    mex,cex
tabsize    equ    16
```

```
    org    x:0
scale    set    1.0
tantab
tanarg   set    45.0*3.14159/180.0
         dup    tabsize
scale    set    scale*@cos(tanarg)
         dc     @tan(tanarg)
tanarg   set    tanarg/2.0
         endm

    org    p:$100

;
;    Do argument reduction, input in d6 in degrees
;
  move              #-180.0,d7.s      ;get range min
  fadd.x  d7,d6     #1.0/360.0,d5.s   ;adjust to min, get range
  fmpy.x  d5,d6,d6                    ;reduce range
  floor   d6,d5                       ;get int part
  fsub.x  d5,d6     #360.0,d5.s       ;get frac part, spread
  fmpy.x  d5,d6,d6                    ;spread fraction part to range
  fadd.x  d7,d6                       ;adjust to min
;
;    Input angle in d6 in degrees, -180 < d6 < 180
;
  fabs.x  d6        #90.0,d7.s        ;make positive, get pi/2
  move              d6.s,d3.s         ;save new sign
  fcmp    d7,d6     #180.0,d7.s       ;see if greater than 90
  fsub.x  d6,d7     ffge              ;reduce to less than 90
  ftfr.x  d6,d7     fflt              ;transfer if no change
  fneg.x  d3        ffge              ;flip if other quadrant
;
;    First quadrant CORDIC trig computation
;    Input angle in d7 in degrees
;    Output d1=sine, d0=cosine
;
  move              #tantab,r0        ;point to tangent table
  fclr    d1        #scale,d0.s       ;y=0, x=scale
  fclr    d5        #45.0,d6.s        ;z=0,alp=45
  do      #tabsize,_cordic
  fcmp    d5,d7     x:(r0)+,d4.s      ;angle < z? get tangent
  fneg.x  d4        fflt              ;yes, rotate cw
  fsub.x  d6,d5     fflt              ;yes, subtract angle
  fadd.x  d6,d5     ffge              ;no, add angle for ccw
  fmpy.x  d1,d4,d2                    ;y*tan
  fmpy    d0,d4,d2  fsub.x d2,d0      ;x*tan, x’=x-y*tan
```

```
  fadd.x  d2,d1                     ;y'=y+x*tan
  fscale.x  #-1,d6                  ;alp=alp/2
_cordic
  fcopys.s  d3,d0                   ;fix sign of cosine
  end
```

	Program Words	ICycles
Argument Reduction	10	10
Quadrantizing	8	8
CORDIC Algorithm	16	8N+9
	----	-------
Totals:	34	8N+27

B.1.53 Four Quadrant Trigonometric TANGENT (CORDIC Algorithm)

```
    page    132,60,1,1
    opt    mex,cex
tabsize    equ    16

    org    x:0
scale    set    1.0
tantab
tanarg   set    45.0*3.14159/180.0
         dup    tabsize
scale    set    scale*@cos(tanarg)
         dc     @tan(tanarg)
tanarg   set    tanarg/2.0
         endm

    org    p:$100

;
;    Do argument reduction, input in d6 in degrees
;
  move             #-180.0,d7.s     ;get range min
  fadd.x  d7,d6    #1.0/360.0,d5.s  ;adjust to min, get range
  fmpy.x  d5,d6,d6                  ;reduce range
  floor   d6,d5                     ;get int part
  fsub.x  d5,d6    #360.0,d5.s      ;get frac part, spread
  fmpy.x  d5,d6,d6                  ;spread fraction part to range
  fadd.x  d7,d6                     ;adjust to min
;
```

```
;    Input angle in d6 in degrees, -180 < d6 < 180
;
  fabs.x  d6       d6.s,d3.s        ;make positive, save sign
  move             #90.0,d7.s       ;get pi/2
  fcmp    d7,d6    #180.0,d7.s      ;see if greater than 90
  fsub.x  d6,d7    ffge             ;reduce to less than 90
  ftfr.x  d6,d7    fflt             ;transfer if no change
  fneg.x  d3       ffge             ;flip if other quadrant
;
;    First quadrant CORDIC trig computation
;    Input angle in d7 in degrees
;    Output d1=sine, d0=cosine
;
  move             #tantab,r0       ;point to tangent table
  fclr    d1       #scale,d0.s      ;y=0, x=scale
  fclr    d5       #45.0,d6.s       ;z=0,alp=45
  do      #tabsize,_cordic
  fcmp    d5,d7    x:(r0)+,d4.s     ;angle < z? get tangent
  fneg.x  d4       fflt             ;yes, rotate cw
  fsub.x  d6,d5    fflt             ;yes, subtract angle
  fadd.x  d6,d5    ffge             ;no, add angle for ccw
  fmpy.x  d1,d4,d2                  ;y*tan
  fmpy    d0,d4,d2 fsub.x d2,d0     ;x*tan, x'=x-y*tan
  fadd.x  d2,d1                     ;y'=y+x*tan
  fscale.x #-1,d6                   ;alp=alp/2
_cordic
  fcopys.s  d3,d0                   ;fix sign of tangent
  ftfr.s  d1,d0    d0.s,d1.s        ;exchange d0<-->d1
  fseedd  d1,d4                     ;d0/d1
  ftfr.s  d4,d1       ffinf
  fneg.s  d1          ffinf
  fmpy.s  d1,d4,d1                    #2.0,d2.s
  fmpy    d0,d4,d0    fsub.s d1,d2    d2.s,d3.s
  fmpy.s  d1,d2,d1                    d2.s,d4.s
  fmpy    d0,d4,d0    fsub.s d1,d3
  fmpy.s  d0,d3,d0                   ;tangent
  end
```

	Program Words	**ICycles**
Argument Reduction	10	10
Quadrantizing	8	8
CORDIC Algorithm	16	8N+9
Division/Error Check	10	10
	----	-------

Totals: 44 8N+37

B.1.54 [NxN] by [NxN] Matrix Multiplication (Modulo-Aligned)

```
;This routine performs an [NxN] by [NxN] matrix multiplication
;for the 96000 floating-point DSP chip.  Sample data is given
;for N=4.  The data for all matrices is stored in row major
;format.  For example, take the matrix A:
;
;                  A(1,1) ... A(1,N)
;                  .       .     .
;                  .       .     .
;                  A(N,1) ... A(N,N)
;
;Matrix A's elements are stored as such:
;amatrix  dc A(1,1),A(1,2),...,A(1,N),A(2,1),A(2,2),...,A(2,N), ...
;
;Matrices A and C are in X memory, while matrix B is in Y memory.
;Since modulo N**2 addressing is used for all matrices, the first
;k least significant bits of the address of the beginning of any
;matrix storage area must be equal to zero, where 2**k >= N**2.
;
;This routine takes
;
;      16 + n(3 + n(2 +n(1) + 2) + 2)
;
;      = n**3 + 4n**2 + 5n + 16 instruction cycles to complete.
;
;
;
                                                    Program ICycles
                                                    Words
    page 132,60,1,1
N    equ 4
N_sqr   equ N*N
        org x:$0
amatrix    dc .1,.2,.3,.4
    dc .5,.6,.7,.8
    dc .9,.1,.2,.3
    dc .4,.5,.6,.7
    org x:$20
cmatrix    ds N_sqr
    org y:$0
bmatrix    dc .5,.5,.5,.5
```

```
    dc .5,.5,.5,.5
    dc .5,.5,.5,.5
    dc .5,.5,.5,.5
    org p:$100
    move #amatrix,r0                                                  1     1
    move #N,n0                                                        1     1
    move #N_sqr-1,m0    ; modulo N-squared addressing                 1     1
    move #bmatrix,r4                                                  1     1
    move #cmatrix,r1                                                  1     1
    move n0,n4                                                        1     1
    move m0,m4                                                        1     1
    move n0,n1                                                        1     1
    move m0,m1                                                        1     1
    fclr d1    x:(r0)+,d0.s y:(r4)+n4,d4.s                            1     1
    fclr d3    d1.s,d7.s                                              1     1
    do   #N,endall                                                    2     3
    do   #N,endcol                                                    2     3
    rep  #N                                                           1     2
    fmpy d0,d4,d3 fadd.s d3,d1 x:(r0)+,d0.s y:(r4)+n4,d4.s            1     1
                  fadd.s d3,d1 d7.s,d3.s                              1     1
    fclr d1 d1.s,x:(r1)+n1                                            1     1
endcol
    move (r4)+              ; increment r4                            1     1
    move (r1)+              ; increment r1                            1     1
endall                                                               ---   ---
                                                        Totals:  21  n**3
                                                                   +4n**2
                                                                   +5n
                                                                   +16
```

B.1.55 [4x4] by [4x4] Matrix Multiplication (Modulo-Aligned)

```
;This routine performs a [4x4] by [4x4] matrix multiplication
;for the 96000 floating-point DSP chip.  Sample data is given.
;The data for all matrices is stored in row major
;format.  For example, take the matrix A:
;
;                  A(1,1) ... A(1,N)
;                  .       .     .
;                  .       .     .
;                  A(N,1) ... A(N,N)
;
;Matrix A’s elements are stored as such:
```

```
;amatrix  dc A(1,1),A(1,2),...,A(1,N),A(2,1),A(2,2),...,A(2,N), ...
;
;Matrix A is in X memory, while matrices B and C are in Y memory.
;Since modulo N**2 addressing is used for all matrices, the first
;k least significant bits of the address of the beginning of any
;matrix storage area must be equal to zero, where 2**k >= N**2.
;
;This routine takes
;                  15 + 4*18 = 87 instruction cycles to complete.
;
;
;
                                                       Program ICycles
                                                       Words
    page 132,60,1,1
N    equ 4
N_sqr    equ N*N
            org x:$0
amatrix        dc .1,.2,.3,.4
        dc .5,.6,.7,.8
        dc .9,.1,.2,.3
        dc .4,.5,.6,.7
        org y:$0
bmatrix        dc .5,1.0,.5,.5
        dc .5,1.0,.5,.5
        dc .5,1.0,.5,.5
        dc .5,1.0,.5,.5
        org y:$20
cmatrix     ds N_sqr
    org p:$100
   move #amatrix,r0                                             1     1
    move #N,n4                                                  1     1
    move #N_sqr-1,m0    ; modulo-N addressing                   1     1
    move #bmatrix,r4                                            1     1
    move #cmatrix+N_sqr-1,r5                                    1     1
    move m0,m4                                                  1     1
    move n4,n5                                                  1     1
    move m0,m5                                                  1     1
    fclr d1 x:(r0)+,d4.s                                        1     1
    fclr d5 y:(r4)+n4,d8.s                                      1     1
    do   #4,endall                                              2     3
    fmpy.s d4,d8,d3            x:(r0)+,d4.s y:(r4)+n4,d0.s 1     1
    fmpy d4,d0,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d6.s 1     1
    fmpy d4,d6,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r5)+,d2.s
                                          ;junk into d2.s   1     1
```

```
    fmpy d4,d8,d3 fadd.s d3,d1 x:(r0)+,d4.s d5.s,d2.s      1     1
    fmpy d4,d0,d3 fadd.s d3,d2 x:(r0)+,d4.s d1.s,y:(r5)+n5 1     1
    fmpy d4,d6,d3 fadd.s d3,d2 x:(r0)+,d4.s                1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s                1     1
    fmpy d4,d8,d3 fadd.s d3,d2 x:(r0)+,d4.s d5.s,d1.s      1     1
    fmpy d4,d0,d3 fadd.s d3,d1 x:(r0)+,d4.s d2.s,y:(r5)+n5 1     1
    fmpy d4,d6,d3 fadd.s d3,d1 x:(r0)+,d4.s                1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s                1     1
    fmpy d4,d8,d3 fadd.s d3,d1 x:(r0)+,d4.s d5.s,d2.s      1     1
    fmpy d4,d0,d3 fadd.s d3,d2 x:(r0)+,d4.s d1.s,y:(r5)+n5 1     1
    fmpy d4,d6,d3 fadd.s d3,d2 x:(r0)+,d4.s d5.s,d1.s      1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+,d0.s
                                         ;junk into d0.s   1     1
                  fadd.s d3,d2              y:(r4)+n4,d8.s 1     1
    move d2.s,y:(r5)+n5                                    1     1
endall                                                   ---   ---
                                                 Totals: 30    87
```

B.1.56 [8x8] by [8x8] Matrix Multiplication (Modulo-Aligned)

```
;This routine performs an [8x8] by [8x8] matrix multiplication
;for the 96000 floating-point DSP chip.  Sample data is given
;for N=8.  The data for all matrices is stored in row major
;format.  For example, take the matrix A:
;
;                  A(1,1) ... A(1,N)
;                  .       .     .
;                  .       .     .
;                  A(N,1) ... A(N,N)
;
;Matrix A's elements are stored as such:
;amatrix  dc A(1,1),A(1,2),...,A(1,N),A(2,1),A(2,2),...,A(2,N), ...
;
;Matrix A is in X memory, while matrices B and C are in Y memory.
;Since modulo N**2 addressing is used for all matrices, the first
;k least significant bits of the address of the beginning of any
;matrix storage area must be equal to zero, where 2**k >= N**2.
;
;This routine takes 15 + 8*74 = 607 instruction cycles.
;
;
;
                                                  Program ICycles
```

```
                                                          Words
    page 132,60,1,1
N    equ 8
N_sqr   equ N*N
        org x:$0
amatrix    dc .1,.2,.3,.4,.1,.2,.3,.4
    dc .5,.6,.7,.8,.5,.6,.7,.8
    dc .9,.1,.2,.3,.9,.1,.2,.3
    dc .4,.5,.6,.7,.4,.5,.6,.7
    dc .1,.2,.3,.4,.1,.2,.3,.4
    dc .5,.6,.7,.8,.5,.6,.7,.8
    dc .9,.1,.2,.3,.9,.1,.2,.3
    dc .4,.5,.6,.7,.4,.5,.6,.7
    org y:$0
bmatrix    dc .5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5
    org y:$40
cmatrix    ds N_sqr
    org p:$100
    move #amatrix,r0                                      1     1
    move #N,n4                                            1     1
    move #N_sqr-1,m0    ; modulo-N addressing             1     1
    move #bmatrix,r4                                      1     1
    move #cmatrix,r5                                      1     1
    move m0,m4                                            1     1
    move n4,n5                                            1     1
    move m0,m5                                            1     1
    fclr d1    x:(r0)+,d4.s                               1     1
    fclr d5    y:(r4)+n4,d7.s                             1     1
;
    do   #8,endall                                        2     3
    fmpy.s d4,d7,d3            x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
```

```
move d1.s,y:(r5)+n5 d5.s,d2.s                            1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s1                           1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                            1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                            1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                            1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                            1     1
```

```
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+,d1.s
                                       ;junk to d1.s     1     1
                  fadd.s d3,d2            y:(r4)+n4,d7.s 1     1
    move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
    move (r5)+                                             1     1
endall                                                    ---   ---
                                               Totals: 86   607
```

B.1.57 [16x16] by [16x16] Matrix Multiplication (Modulo Aligned)

```
;This routine performs a [16x16] by [16x16] matrix multiplication
;for the 96000 floating-point DSP chip.  Sample data is given
;for N=16.  The data for all matrices is stored in row major
;format.  For example, take the matrix A:
;
;                  A(1,1) ... A(1,N)
;                  .      .     .
;                  .      .     .
;                  A(N,1) ... A(N,N)
;
;Matrix A’s elements are stored as such:
;amatrix  dc A(1,1),A(1,2),...,A(1,N),A(2,1),A(2,2),...,A(2,N), ...
;
;Matrix A is in X memory, while matrices B and C are in Y memory.
;Since modulo N**2 addressing is used for all matrices, the first
;k least significant bits of the address of the beginning of any
;matrix storage area must be equal to zero, where 2**k >= N**2.
;
```

```
;This routine takes 15 + 16(18 + 14*17 + 18) = 4399 instruction
cycles.
;
;
;
                                                    Program ICycles
                                                    Words
    page 132,60,1,1
N    equ 16
N_sqr   equ N*N
        org x:$0
amatrix    dc .1,.2,.3,.4,.1,.2,.3,.4,1,1,1,1,1,1,1,1
    dc .5,.6,.7,.8,.5,.6,.7,.8,1,1,1,1,1,1,1,1
    dc .9,.1,.2,.3,.9,.1,.2,.3,1,1,1,1,1,1,1,1
    dc .4,.5,.6,.7,.4,.5,.6,.7,1,1,1,1,1,1,1,1
    dc .1,.2,.3,.4,.1,.2,.3,.4,1,1,1,1,1,1,1,1
    dc .5,.6,.7,.8,.5,.6,.7,.8,1,1,1,1,1,1,1,1
    dc .9,.1,.2,.3,.9,.1,.2,.3,1,1,1,1,1,1,1,1
    dc .4,.5,.6,.7,.4,.5,.6,.7,1,1,1,1,1,1,1,1
    dc .1,.2,.3,.4,.1,.2,.3,.4,1,1,1,1,1,1,1,1
    dc .5,.6,.7,.8,.5,.6,.7,.8,1,1,1,1,1,1,1,1
    dc .9,.1,.2,.3,.9,.1,.2,.3,1,1,1,1,1,1,1,1
    dc .4,.5,.6,.7,.4,.5,.6,.7,1,1,1,1,1,1,1,1
    dc .1,.2,.3,.4,.1,.2,.3,.4,1,1,1,1,1,1,1,1
    dc .5,.6,.7,.8,.5,.6,.7,.8,1,1,1,1,1,1,1,1
    dc .9,.1,.2,.3,.9,.1,.2,.3,1,1,1,1,1,1,1,1
    dc .4,.5,.6,.7,.4,.5,.6,.7,1,1,1,1,1,1,1,1
    org y:$0
bmatrix    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    dc .5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5,.5
    org y:$100
cmatrix    ds N_sqr
```

```
     org p:$100
     move #amatrix,r0                                       1     1
     move #N,n4                                             1     1
     move #N_sqr-1,m0    ; modulo-N addressing              1     1
     move #bmatrix,r4                                       1     1
     move #cmatrix,r5                                       1     1
     move m0,m4                                             1     1
     move n4,n5                                             1     1
     move m0,m5                                             1     1
     fclr d1    x:(r0)+,d4.s                                1     1
     fclr d5    y:(r4)+n4,d7.s                              1     1
;
     do   #16,endall                                        2     3
     fmpy.s d4,d7,d3            x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
     fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
```

```
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
```

```
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
```

```
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
```

```
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
```

```
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d1 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
move d1.s,y:(r5)+n5 d5.s,d2.s                          1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
```

```
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+n4,d7.s 1     1
    fmpy d4,d7,d3 fadd.s d3,d2 x:(r0)+,d4.s y:(r4)+,d1.s
                                         ;junk to d1.s     1     1
                  fadd.s d3,d2              y:(r4)+n4,d7.s 1     1
    move d2.s,y:(r5)+n5 d5.s,d1.s                          1     1
    move (r5)+                                             1     1
endall                                                    ---   ---
                                               Totals: 286  4399
```

B.1.58 Sine Wave Oscillators

```
                                                  Program  Icycles
                Double Integrator Oscillator      Words

        page   132,60,1,1
fs      equ    8192.0          ;sampling frequency
f0      equ    256.0           ;center frequency
mag     equ    1.0             ;magnitude
scale   equ    (2.0*@sin(3.14159*f0/fs))*(2.0*@sin(3.14159*f0/fs))
output  equ    $ffff           ;output file

    org     p:$100
    move    #scale,d7.s         ;init scale factor
    fclr    d6    #mag,d5.s     ;init magnitudes
    do      #100,_gen           ;do 100 points
    fmpy.x  d7,d6,d0                                    1      1
    fadd.s  d0,d5                                       1      1
    fsub.x  d5,d6                                       1      1

    move    d5.s,y:output       ;output signal
_gen                                                   ---    ---
                                              Totals:  3      3
```

```
                                                   Program  Icycles
                Second Order Oscillator               Words
```

```
        page   132,60,1,1
fs      equ    8000.0          ;sampling frequency
f0      equ    320.0           ;center frequency
scale   equ    2.0*@cos(2.0*3.14159*f0/fs)
mag     equ    1.0*@sin(2.0*3.14159*f0/fs)
output  equ    $ffff

    org    p:$100
    move    #scale,d7.s        ;init scale factor
    fclr    d6    #mag,d5.s    ;init magnitudes
    do      #200,_gen          ;generate 200 points
    fmpy.s  d6,d7,d6       d6.s,d4.s                    1      1
    fsub.s  d5,d6          d4.s,d5.s                    1      1

    move    d5.s,y:output
_gen                                                   ---    ---
                                              Totals:  2      2
```

B.1.59 DTMF Generation

```
                                                      Program  Icycles
                DTMF Generation                       Words
        page    132,60,1,1
fs      equ    8000.0     ;sampling frequency
f0      equ    697.0      ;frequency 0
scale0  equ    2.0*@cos(2.0*3.14159*f0/fs)
mag0    equ    1.0*@sin(2.0*3.14159*f0/fs)
f1      equ    1209.0     ;frequency 1
scale1  equ    2.0*@cos(2.0*3.14159*f1/fs)
mag1    equ    1.0*@sin(2.0*3.14159*f1/fs)
output  equ    $ffff

    org    p:$100
    move    #scale0,d7.s        ;init scale0 factor
    fclr    d6    #mag0,d5.s    ;init magnitude0
    move    #scale1,d3.s        ;init scale1
    fclr    d2    #mag1,d1.s    ;init magnitude1
    do    #4096,_gen
    fmpy.s    d6,d7,d6    d6.s,d4.s                     1      1
    fsub.s    d5,d6       d4.s,d5.s                     1      1
    fmpy.s    d2,d3,d2    d2.s,d0.s                     1      1
    fsub.s    d1,d2       d0.s,d1.s                     1      1
```

```
    fadd.s    d4,d0                                          1      1
    move    d0.s,y:output
_gen                                                        ---    ---
                                                   Totals:  5      5
```

B.2 IEEE STANDARD CONFORMANCE FUNCTIONS

B.2.1 IEEE Remainder

B.2.2 IEEE floating-point Round to Integer

The IEEE standard section 5.5 specifies that it shall be possible to round a floating-point number to an integral valued floating point number in the same format. If the rounding mode is round to nearest, the rounded result is even if the difference between the rounded result and the unrounded operand is exactly one half.

```
                                                       Program    ICycles
                                                       Words
        fint    d0      ;round to nearest integer        1          1
                                                        ---        ---
                                                         1          1
```

The FINT instruction rounds the number in d0 according to the current rounding mode.

B.2.3 IEEE floating-point to Decimal String

B.2.4 IEEE Decimal String to floating-point

B.2.5 Format Conversions

The IEEE standard states that it shall be possible to convert between all supported floating-point formats and all supported integer formats. Conversions between floating-point integers and integer formats shall be exact unless an exception arises. If the floating point number is infinity, a NaN or overflows the integer data type, then the invalid operation is signaled.

Some conversions may require range checking to signal an error if the source would produce an invalid result for the destination data type. In some programming languages, the programmer is responsible for the correct value of the source and the conversion of an out of range source produces an erroneous result. In the conversion descriptions, conversions that require range checking will perform the actual range

checking on the source and either jump to an error handling procedure or return a valid result. The programs provided may vary depending on the application.

The following data types and abbreviations will be used:

- I - Signed 32 bit integer
- U - Unsigned 32 bit integer
- SP - Single precision floating-point

All conversion examples assume that the value to be converted is in d0 if floating-point or in d0.l if fixed point.

I → U

```
                                                  Program   ICycles
                                                  Words
tst    d0                 ;check for in range        1         1
jmi    _negerr            ;if negative, error        1         2
                                                    ---       ---
                                                     2         3
```

I → SP

```
                                                  Program   ICycles
                                                  Words
float.s  d0               ;convert to SP float       1         1
                                                    ---       ---
                                                     1         1
```

U → I

```
                                                  Program   ICycles
                                                  Words
tst     d0                ;see if msb is set         1         1
jmi     _toobig           ;if set, too big           1         2
                                                    ---       ---
                                                     2         3
```

U → SP

```
                                                  Program   ICycles
                                                  Words
floatu.s   d0.1           ;convert                   1         1
                                                    ---       ---
                                                     1         1
```

SP → I

```
                                                   Program  ICycles
                                                   Words
int     d0              ;convert to integer          1       1
jset    #20,sr,_error   ;jump if invalid op set      2       3
                                                    ---     --
-                                                    3       4
```

SP → U

```
                                                   Program  ICycles
                                                   Words
intu    d0              ;convert to integer          1       1
jset    #20,sr,_error   ;jump if invalid op set      2       3
                                                    ---     ---
                                                     3       4
```

B.3 IEEE RECOMMENDED FUNCTIONS AND PREDICATES

The following functions are recommended by the IEEE-754 standard but are not required.

Functions that require explicit knowledge of the variable precision and may lead to families of functions on high level languages are 4, 5, and 10. Functions 1 and 2 have an arithmetic form (signals IOP if the source is a NaN) and a non-arithmetic form.

B.3.1 Copysign(x,y)

Copysign(x,y) returns y with the sign of x.

Arithmetic Implementation Of Copysign(d1,d0)

```
                                                   Program  ICycles
                                                   Words
fcopys.s  d1,d0    ;copy sign of d1 to d0            1       1
                                                    ---     ---
                                          Totals:    1       1
```

Non-Arithmetic Implementation Of Copysign(d1,d0)

```
                                                   Program  ICycles
                                                   Words
bclr      #31,d0.h          ;clear sign bit          2       2
jclr      #31,d1.h,_bitclr  ;sign bit clear          2       3
bset      #31,d0.h          ;set sign bit            2       2
_bitclr                                             ---     ---
                                          Totals:    6       7
                                                    43
```

B.3.2 -x

The arithmetic form signals IOP if x is a signalling NaN. The non-arithmetic form copies x with its sign complemented.

Arithmetic Implementation Of -d0

```
                                        Program  ICycles
                                        Words
fneg.s   d0      ;change sign bit         1        1
                                         ---      ---
                              Totals:     1        1
```

Non-Arithmetic Implementation Of -d0

```
                                        Program  ICycles
                                        Words
bchg     #31,d0.h   ;change sign bit      1        2
                                         ---      ---
                              Totals:     1        2
```

B.3.3 Scalb(y,N)

Scalb(y,N) returns y*(2**N) for integral values of N without computing 2**N. This is an arithmetic function.

Arithmetic Implementation Of d0*(2d1.h)**

```
                                        Program  ICycles
                                        Words
fscale.s d1.h,d0    ;scale d0             1        1
                                         ---      ---
                              Totals:     1        1
```

B.3.4 Logb(x)

Logb(x) returns the unbiased exponent of x, a signed integer in the format of x, except that logb(NaN) is a NaN, logb(infinite) is +infinity, and logb(0) is -infinity and signals the division by zero exception. When x is positive and finite, the expression scalb(x,-logb(x)) lies strictly between 0 and 2; it is less than 1 only when x is denormalized. This is an arithmetic function.

Arithmetic Implementation Of d0=logb(d1)

```
                                                        Program  ICycles
                                                        Words
ninf     equ     $ff800000     ;negative infinity
     ftst   d1      d1.s,d0.s  ;check input, copy         1        1
     fjun   _done              ;done if nan               2        3
     fjinf  _done              ;done if infinity          2        3
     fjne   _notzero           ;jump if non-zero          2        3
```

```
        move   #ninf,d0.s          ;set -infinity result      2      2
        ori    #2,er               ;set DZ in ER              1      1
        ori    #2,ier              ;set DZ in IER             1      1
        jmp    _done               ;done                      2      2
_notzero
        getexp d1,d0   #-126,d3.l ;get exponent              2      2
        cmp    d3,d0               ;cmp to SP exp min         1      1
        tfr    d3,d0   iflt        ;limit if denorm           1      1
        float.s   d0               ;convert to SP FP          1      1
_done                                                        ---    ---
                                                   Totals:   18      *
```

Execution Time:

```
Nan         4
Infinity    7
Zero       16
In-range   15
```

B.3.5 Nextafter(x,y)

Nextafter(x,y) returns the next representable neighbor of x in the direction toward y. The following special cases arise: if x=y, then the result is x without any exception being signaled; otherwise, if either x or y is a quiet NaN, then the result is one or the other input NaNs. Overflow is signaled when x is finite but nextafter(x,y) is infinite; underflow is signaled when nextafter(x,y) lies strictly between +/-2**(Emin); in both cases, inexact is signaled.

The x argument of the nextafter(x,y) must be a single precision number and not a single-extended number. This is an arithmetic function.

Implementation of nextafter(d0,d4)
d0 for single precision numbers:

```
                                                          Program  ICycles
                                                          Words
    ftst    d4                                              1        1
    ftfr.s  d4,d0      ffun                                 1        1
    ftst    d0         d0.s,d1.l                            1        1
    fjor    _not_nan                                        2        2
    move               #$7fffffff,d0.s                      2        2
    jmp    _ok                                          2     2  _not_nan
    fjinf   _ok                                             2        2
    bclr    #31,d1.l                                        2        2
    neg     d1         ifcs                                 1        1
    fcmp    d0,d4      #$00800000,d3.s                      2        2
    inc     d1         ffgt                                 1        1
    dec     d1         fflt                                 1        1
    tst     d1         #$80000000,d2.l                      2        2
    neg     d1         ifmi                                 1        1
    or      d2,d1      ifmi                                 1        1
    move               d1.l,d0.s                            1        1
    fcmpm   d3,d0                                           1        1
    fjge    _not_denorm                                     2        2
    ori     #$5,er                                          1        1
    ori     #$5,ier                                         1        1
_not_denorm
    fjninf  _ok                                             2        2
    ori     #$9,er                                          1        1
    ori     #$9,ier                                         1        1
_ok                                                        ---      ---
                                                Totals:    32       *
```

Execution Timing in ICycles

```
    Either operand a NaN:      9
    X is + or - infinity:      7
    Result is normalized:      26
    Result denormalized:       24
    Result overflowed:         26
```

B.3.6 Finite(x)

Finite(x) returns the value TRUE if -inf<x<+inf, and returns FALSE otherwise. This is an arithmetic function.

d1=Finite(d0)

				Program Words	ICycles
ftst	d0	#0,d1.l	;set ccr bits	2	2
inc	d1	ffinf	;set true if infinite	1	1
				---	---
			Totals:	3	3

B.3.7 Isnan(x)

Isnan(x) returns the value TRUE if x is a NaN, and returns FALSE otherwise. This is an arithmetic function.

d1=Isnan(d0)

				Program Words	ICycles
ftst	d0	#0,d1.l	;set ccr bits	2	2
inc	d1	ffun	;set true if NaN	1	1
				---	---
			Totals:	3	3

B.3.8 x<>y

x<>y is TRUE only when x<y or x>y, and is distinct from x=/=y which means NOT(x=y). This is an arithmetic function.

d2=d0<>d1

				Program Words	ICycles
fcmp	d0,d1	#0,d2.l	;set ccr bits	2	2
inc	d2	ffgl	;set true if GL	1	1
				---	---
			Totals:	3	3

When comparing two values, GL is true if the values are not equal and both values being compared are valid floating-point numbers. The GL condition is false if either number is a NaN even though the values are not equal.

B.3.9 Unordered(x,y) or x?y

Unordered(x,y), or x?y, returns the value TRUE if x is unordered with y, and returns FALSE otherwise. This is an arithmetic function.

d2=d0?d1				Program Words	ICycles
fcmp	d0,d1	#0,d2.l	;set ccr bits	2	2
inc	d2	ffun	;set true if unordered	1	1
				---	---
			Totals:	3	3

B.3.10Class(x)

Class(x) tells which of the following ten classes x falls into:

1. signaling NaN
2. quiet NaN
3. -infinity
4. negative normalized nonzero
5. negative denormalized
6. -0
7. +0
8. positive denormalized
9. positive normalized nonzero
10. +infinity

Class(x) function applies to single precision floating-point numbers.

d1=class(d0)

```
                                                        Program  ICycles
                                                        Words
    ftst    d0             ;test d0                       1        1
    fjor    _notnan        ;jump if ordered               2        3
    jset    #5,er,_tsnan   ;check signaling nan bit       2        3
    jmp     _tqnan         ;quiet nan                     2        2
_notnan
    fjne    _notz          ;jump if not zero              2        3
    fjmi    _tmzer         ;type is minus zero            2        3
    jmp     _tpzer         ;type is plus zero             2        3
_notz
    fjninf  _finite        ;jump if finite                2        3
    fjmi    _tminf         ;minus infinity                2        3
    jmp     _tpinf         ;plus infinity                 2        2
_finite
    fjge    _pos           ;see if positive               2        3
    jset    #30,d0.h,_tmdnrm ;denormalized                2        3
    jmp     _tmnorm        ;normalized                    2        2
_pos
    jset    #30,d0.h,_tpdnrm ;denormalized                2        3
    jmp     _tpnorm        ;normalized                    2        2
_tpinf
    inc     d1                                            1        1
_tpnorm
    inc     d1                                            1        1
_tpdnrm
    inc     d1                                            1        1
_tpzer
    inc     d1                                            1        1
_tmzer
    inc     d1                                            1        1
_tmdnrm
    inc     d1                                            1        1
_tmnorm
    inc     d1                                            1        1
_tminf
    inc     d1                                            1        1
_tqnan
    inc     d1                                            1        1
_tsnan                                                   ---      ---
                                              Totals:    38        *
```

Execution Times:

```
Signaling not a number      -  7
Quiet not a number          - 10
Negative infinity           - 15
Negative normalized nonzero - 21
Negative denormalized       - 20
Negative zero               - 15
Positive zero               - 19
Positive denormalized       - 23
Positive normalized nonzero - 26
Positive infinity           - 25
```

Note the following code assignments:

```
Signaling not a number      - 0
Quiet not a number          - 1
Negative infinity           - 2
Negative normalized nonzero - 3
Negative denormalized       - 4
Negative zero               - 5
Positive zero               - 6
Positive denormalized       - 7
Positive normalized nonzero - 8
Positive infinity           - 9
```

B.4 IEEE DOUBLE PRECISION USING SOFTWARE EMULATION

Note: The following programs have not been exhaustively tested and may contain errors.

B.4.1 IEEE Double Precision Addition

```
;
; Double Precision IEEE floating-point Addition For The DSP96002
;
;    D0 + D1 → D1
;
;
;
; Alters Data ALU Registers
;       d0.h    d0.m    d0.l
;       d1.h    d1.m    d1.l
;               d2.m    d2.l
;                       d3.l
```

```
;       d4.h            d4.l
;       d5.h            d5.l
;                       d6.l
;                       d7.l
;
; Alters Program Control Registers
;       pc      sr
;
;
; Version 1.0
; Latest Revision - 01-Aug-88
;
            section     ieeeadd
emsk    equ     $7ff            ; exponent mask
eden    equ     $1              ; denorm exponent
grsmsk  equ     $700            ; GRS (guard-round-sticky) bits mask
grmsk   equ     $fffffe00       ; GR (guard-round) bits mask
smsk   equ    $ffffff00      ; mask to clear bits to right of the sticky
bit
onemsk  equ    $1ff           ; mask to set bits to right of the round bit
inum    equ     $100            ; increment number
imsk    equ     $7fffffff       ; infinity mask
qnane   equ     $7ff            ; quiet NaN exponent
qnanmh  equ     $7fffffff       ; quiet NaN mantissa high
qnanml  equ     $ffffffff       ; quiet NaN mantissa low
maxnum  equ     $fffff800       ; low part of maximum number
;
sdptest                         ; double precision add subroutine
;
; Clear ER portion of status register
;
        andi    #0,er
;
; Check for Maximum and Minimum Exponents
;
        move    #0,d6.l                 ; addend 0 flag
        move    d0.h,d4.l               ; get exp0
        move    #emsk,d7.l              ; get exponent mask
        and     d7,d4      d0.m,d2.l  ; delete tags, get m0.h
        cmp     d7,d4      d1.m,d3.l  ; check max exp, get m1.h
        jeq     _mant1                  ; jump if exp0 = max exp
        move    d1.h,d5.l               ; get exp1
        and     d7,d5      #1,d6.l    ; delete tags, a1 flag
        cmp     d7,d5      #0,d2.m    ; check max exp, sticky=0
        jeq     _mant2                  ; jump if exp1 = max exp
        tst     d4    #0,d6.l         ; check min exp, a0 flag
        jeq     _mant3                  ; jump if exp0 = min exp
        tst     d5    #1,d6.l         ; check min exp, a1 flag
        jeq     _mant4                  ; jump if exp1 = min exp
        jmp     _nadd                   ; jump to normalized add
        ;
        ; Check if Addend 0 is Infinity
        ;
_mant1  move    #imsk,d7.l       ; get infinity mask
```

```
        and     d7,d2             ; remove implied one bit
        tst     d2                ; check m0.high = zero
        jne     _nan0             ; jump if nan
        tst     d0                ; check m0.low = zero
        jne     _nan0             ; jump if nan
        move    #emsk,d7.l        ; get exponent mask
        move    d1.h,d5.l         ; get exp1
        and     d7,d5             ; delete tags
        cmp     d7,d5             ; check for max exp
        jne     _inf1             ; jump if a1 is not inf or NaN
        ;
        ; Check if Addend 1 is Infinity
        ;
_mant2  move    #imsk,d7.l        ; get infinity mask
        and     d7,d3             ; remove implied one bit
        tst     d3                ; check m1.high = zero
        jne     _nan1             ; jump if nan
        tst     d1                ; check m1.low = zero
        jne     _nan1             ; jump if nan
        jclr    #0,d6.l,_binf     ; jump if a0 and a1 are inf
        ori     #$10,ccr          ; set infinity bit
        jmp     _done             ; a1 is infinity
        ;
        ; Check for Case: (+Inf) + (-Inf) = QNaN
        ;
_inf1   ftfr.x  d0,d1             ; move result to d1
            ori     #$10,ccr          ; set infinity bit
        jmp     _done             ; a0 is infinity
_binf   ftst    d0                ; check sign of a0
        jmi     _minf             ; jump if a0 is -inf
        ftst    d1                ; check sign of a1
        jmi     _inan             ; (+inf) + (-inf) = QNaN
        ori     #$10,ccr          ; set infinity bit
        jmp     _done             ; a0 and a1 are +inf
_minf   ftst    d1                ; check sign of a1
        jpl     _inan             ; (-inf) + (+inf) = QNaN
        ori     #$10,ccr          ; set infinity bit
        jmp     _done             ; a0 and a1 are -inf
        ;
        ; Check for NaNs
        ;
_nan0   jclr    #30,d0.m,_inan    ; jump if a0 is a SNaN
        move    #emsk,d7.l        ; get exponent mask
        move    d1.h,d5.l         ; get exp1
        and     d7,d5             ; delete tags
        cmp     d7,d5             ; check for max exp
        jne     _qnan             ; jump if a1 is not a NaN
        move    #imsk,d7.l        ; get infinity mask
        and     d7,d3             ; remove implied one bit
        tst     d3                ; check mant1.high = zero
        jne     _nan1             ; jump if a1 is a NaN
        tst     d1                ; check mant1.low = zero
        jeq     _qnan             ; jump if a1 is infinity
_nan1   jset    #30,d1.m,_qnan    ; jump if a1 is a QNaN
```

```
_inan   ori     #$10,ier          ; set invalid operation bit
_qnan   move    #qnane,d1.h       ; get QNaN exponent
        move    #qnanmh,d1.m      ; get QNaN mantissa high
        move    #qnanml,d1.l      ; get QNaN mantissa low
        ori     #$20,ccr          ; set Not-a-Number bit
        jmp     _done             ; result is a NaN
        ;
        ; Check if Addend 0 is a Denormalized Number
        ;
_mant3  tst     d2                ; check mant0.high = zero
        jne     _den0             ; jump if a0 is a denorm
        tst     d0                ; check mant0.low = zero
        jne     _den0             ; jump if a0 is a denorm
        cmp     d7,d5             ; check min exp for a1
        jgt     _done             ; a1 is the answer
        ;
        ; Check if Addend 1 is a Denormalized Number
        ;
_mant4  tst     d3                ; check mant1.high = zero
        jne     _den1             ; jump if a1 is a denorm
        tst     d1                ; check mant1.low = zero
        jne     _den1             ; jump if a1 is a denorm
        jclr    #0,d6.l,_bzero    ; jump if both are zero
        jmp     _tfr              ; move result to d1
        ;
        ; Addend 0 is a Denormalized Number
        ;
_den0   bset    #0,d0.h           ; get denorm exponent
        inc     d4                ;       "
        tst     d5                ; check if a1 is a denorm
        jgt     _ftz              ; jump if a1 is a normal number
        tst     d3                ; check mant1.high = zero
        jne     _bden             ; jump if a1 is a denorm
        tst     d1                ; check mant1.low = zero
        jne     _bden             ; jump if a1 is a denorm
        jmp     _tfr              ; move result to d1
_bden   bset    #0,d1.h           ; get denorm exponent
        inc     d5                ;       "
_ftz    jclr    #27,sr,_nadd      ; jump to add for ieee mode
        jmp     _done             ; a0 is flushed-to-zero
        ;
        ; Addend 1 is a Denormalized Number
        ;
_den1   jclr    #0,d6.l,_done     ; a1 is the answer
        bset    #0,d1.h           ; get denorm exponent
        inc     d5                ;       "
        jclr    #27,sr,_nadd      ; jump to add for ieee mode
_tfr    move    #eden,d7.l        ; get denorm exponent
        move    d0.h,d5.l         ; get expr
        move    #emsk,d4.l        ; get exponent mask
        and     d4,d5             ; delete tags and sign
        cmp     d7,d5      d0.h,d5.l ; compare exps, get expr
        jne     _tmov             ; jump if not a denorm
        move    d0.m,d3.l         ; get mantr.high
```

```
        tst     d3                ; test mantr.high, get expr
        dec     d5    ifpl.u      ; decrement expr if no int bit
_tmov   move    d5.l,d1.h         ; move result to d1
        move    d0.m,d1.m         ;       "
        move    d0.l,d1.l         ;       "
        jmp     _done             ; a0 is the answer
        ;
        ; Both Addends are Zero
        ;
_bzero  move    d0.h,d4.l         ; get exp0
        move    d1.h,d5.l         ; get exp1
        eor     d4,d5             ; check for opposite signs
        jclr    #31,d5.l,_done    ; jump if same signs
        bclr    #31,d1.h          ; set result as positive
        jclr    #22,sr,_done      ; jump if round bit r1 = zero
        jset    #21,sr,_done      ; jump if round bit r0 = one
        bset    #31,d1.h          ; set result as negative
        jmp     _done             ; result is negative zero
;
;
;
; ******************************************
; *** DP Addition for Normalized Numbers ***
; ******************************************
;
;
; Compare Exponents
;
_nadd   cmp     d4,d5      d1.h,d6.l  ; compare exps, get expr
        jgt     _pos                  ; jump if exp1 > exp0
        jeq     _add                  ; jump if exp0 = exp1
;
; *** Case: Exp0 > Exp1 ***
;
        ;
        ; Align Mantissas
        ;
        sub     d5,d4      d0.h,d6.l  ; get shift, get expr
        move    #55,d7.l              ; get number of bits
        cmp     d7,d4            ; check for shift > 55
        jgt     _setst0          ; jump if shift > 55
        do      d4.l,_end1       ; align mantissas
        lsr     d3               ; shift right m1.h
        ror     d1               ; shift right m1.l and GRS1
        jclr    #8,d1.l,_cclr1   ; jump if sticky bit clear
        move    #1,d2.m          ; set sticky bit
_cclr1  nop                      ;
        ;
        ; Calculate Sticky Bit
        ;
_end1   move    #grmsk,d7.l      ; get GR mask
            and     d7,d1            ; remove bits right of round bit
        jclr    #0,d2.m,_add     ; jump if sticky = 0
        bset    #8,d1.l          ; put in sticky bit
```

```
        jmp     _add            ;
        ;
        ; Set Sticky Bit for Shift > 55 Bits
        ;
_setst0 move    #0,d3.l         ; get number for addition
        move    #inum,d1.l      ;       "
        jmp     _add            ;
;
; *** Case: Exp1 > Exp0 ***
;
        ;
        ; Align Mantissas
        ;
_pos    sub     d4,d5      d1.h,d6.l  ; get shift, get expr
        move    #55,d7.l              ; get number of bits
        cmp     d7,d5           ; check for shift > 55
        jgt     _setst1         ; jump if shift > 55
        do      d5.l,_end2      ; align mantissas
        lsr     d2              ; shift right m0.h
        ror     d0              ; shift right m0.l and GRS0
        jclr    #8,d0.l,_cclr2  ; jump if sticky bit clear
        move    #1,d2.m         ; set sticky bit
_cclr2  nop                     ;
        ;
        ; Calculate Sticky Bit
        ;
_end2   move    #grmsk,d7.l     ; get GR mask
        and     d7,d0           ; remove bits right of round bit
        jclr    #0,d2.m,_add    ; jump if sticky = 0
        bset    #8,d1.l         ; put in sticky bit
        jmp     _add            ;
        ;
        ; Set Sticky Bit for Shift > 55 Bits
        ;
_setst1 move    #0,d2.l         ; get number for addition
        move    #inum,d0.l      ;       "
;
; Check the Signs of the Addends
;
_add    jset    #31,d0.h,_neg1  ; jump if a0 negative
        jset    #31,d1.h,_neg2  ; jump if a1 negative
        jmp     _fadd           ; jump to addition for a0+,a1+
;
; *** Case: Addend 0 is Negative,
;           Addend 1 is Positive ***
;
_neg1   jset    #31,d1.h,_nset  ; jump if a1 negative
        sub     d0,d1           ; subtract for case: a0-,a1+
        subc    d2,d3           ;         "
        jcc     _zchk           ; jump if result is positive
        bset    #31,d6.l        ; set result as negative
        move    #inum,d7.l      ; get increment number
        not     d1              ; get 2's comp of result
        not     d3              ;         "
```

```
        add     d7,d1           ;         "
        jcc     _zchk           ;         "
        inc     d3              ;         "
        jmp     _zchk           ;
;
; *** Case: Addend 0 is Positive,
;           Addend 1 is Negative ***
;
_neg2   bclr    #31,d6.l        ; set result as positive
        sub     d1,d0           ; subtract for case: a0+,a1-
        subc    d3,d2           ;         "
        jcc     _cclr3          ; jump if result is positive
        bset    #31,d6.l        ; set result as negative
        move    #inum,d7.l      ; get increment number
        not     d0              ; get 2's comp of result
        not     d2              ;         "
        add     d7,d0           ;         "
        jcc     _cclr3          ;         "
        inc     d2              ;         "
_cclr3  move    d0.l,d1.l       ; get mantr.low and GRS bits
        move    d2.l,d3.l       ; get mantr.high
;
; Check result equal zero (do not want to normalize)
;
_zchk   move    #smsk,d7.l      ; get sticky mask
        and     d7,d1           ; remove bits right of sticky
        tst     d3              ; check mantr.high = zero
        jne     _snrm           ; normalize result
        tst     d1              ; check mantr.low = zero
        jne     _snrm           ; normalize result
        move    #0,d6.l         ; set expr = zero
        ;
        ; Check for Special Case (Round toward -infinity)
        ;
        jclr    #22,sr,_rnd     ; jump if round bit r1 = zero
        jset    #21,sr,_rnd     ; jump if round bit r0 = one
        bset    #31,d6.l        ; set result to negative zero
        jmp     _rnd            ; check rounding mode
;
; Normalize for Opposite Sign Cases
;
_snrm   jset    #31,d3.l,_rnd   ; jump if result normalized
        move    #emsk,d7.l      ; get exp mask
        move    d6.l,d5.l       ; get expr
        and     d7,d5           ; delete tags
        move    #eden,d4.l      ; get denorm exponent
        jclr    #8,d1.l,_st0    ; jump if sticky bit = 0
        move    #onemsk,d7.l    ; get one mask
_st1    or      d7,d1           ; set bits right of round bit
_st0    cmp     d4,d5           ; test expr = zero
            jle     _rnd            ; jump if denormalized number
        dec     d6              ; decrement expr
        dec     d5              ; decrement expr copy
        lsl     d1              ; shift mantr.l left
```

```
        rol     d3              ; shift mantr.h left
        jset    #31,d3.l,_rnd   ; jump if result normalized
        jclr    #8,d1.l,_st0    ; jump if sticky bit = 0
        jmp     _st1            ; jump if sticky bit = 1
;
; *** Cases: 1) Addend 0 is Negative,
;               Addend 1 is Negative
;            2) Addend 0 is Positive,
;               Addend 1 is Positive ***
;
_nset   bset    #31,d6.l        ; set result as negative
_fadd   add     d0,d1           ; add for case: a0-,a1-
        addc    d2,d3           ;     and case: a0+,a1+
        jcc     _rnd            ; jump if number normalized
        lsr     d3              ; shift right mantr.h
        ror     d1              ; shift right mantr.low
        jclr    #8,d1.l,_cclr4  ; jump if sticky bit = 0
        bset    #8,d1.l         ; set sticky bit
_cclr4  bset    #31,d3.l        ; set bit 31 of mantr.high
        inc     d6              ; increment expr
;
; Check if Result is Infinity
;
        move    #emsk,d7.l      ; get exp mask
        move    d6.l,d5.l       ; get expr
        and     d7,d5           ; delete tags
        cmp     d7,d5           ; check max exp
        jne     _rnd            ; jump if no overflow
        jset    #31,d6.l,_ninf  ; jump if result is -infinity
        ;
        ; Positive Infinity
        ;
        jset    #22,sr,_rmchk   ; jump if rounding bit r1 = 1
        jclr    #21,sr,_setinf  ; jump if rounding bit r0 = 0
        jmp     _setbig         ; round toward zero case
        ;
        ; Negative Infinity
        ;
_ninf   jclr    #21,sr,_setinf  ; jump if rounding bit r0 = 0
        ;
        ; Result is Largest Number Less Than Infinity
        ;
_setbig dec     d6              ; get big exponent
        move    #qnanml,d1.m    ; get mantr.high
        move    #maxnum,d1.l    ; get mantr.low
        ori     #$09,ier        ; set OVF and INX bits in IER
        ori     #$09,er         ; set OVF and INX bits in ER
        jmp     _emove          ; get expr
_rmchk  jclr    #21,sr,_setbig  ; round toward -inf case
        ;
        ; Result is Infinity
        ;
_setinf move    #0,d1.l         ; set result to infinity
        move    #0,d1.m         ;         "
```

```
        move    d6.l,d1.h       ;         "
        ori     #$10,ccr        ; set infinity bit
        ori     #$09,ier        ; set OVF and INX bits in IER
        ori     #$09,er         ; set OVF and INX bits in ER
        jmp     _done           ; result is infinity
;
; Begin Rounding the Result
;
        ;
        ; Check for Denormalized Numbers
        ;
_rnd    move    #eden,d7.l      ; get denorm exponent
        move    d6.l,d5.l       ; get expr
        move    #emsk,d4.l      ; get exponent mask
        and     d4,d5           ; delete tags and sign
        cmp     d7,d5           ; compare exponents
        jne     _remst          ; jump if not a denorm
        tst     d3              ; test mantr.high
        dec     d6    ifpl.u    ; decrement expr if no int bit
        ;
        ; Remove Bits to Right of the Sticky Bit
        ;
_remst  move    #smsk,d7.l      ; get sticky mask
        and     d7,d1           ; remove bits right of sticky
        ;
        ; Check GRS Bits Equal Zero
        ;
        move    d1.l,d5.l       ; get register with GRS bits
        move    #grsmsk,d7.l    ; get GRS mask
        and     d7,d5           ; get GRS bits
        tst     d5              ; check GRS bits = zero
        jeq     _lmove          ; jump if no rounding required
        ori     #$1,ier         ; set inexact result bit
        ori     #$1,er          ; set inexact result bit
        ;
        ; Check Rounding Mode
        ;
        jset    #21,sr,_r1chk   ; jump if rounding bit r0 = 1
        jset    #22,sr,_rminf   ; jump if round toward -infinity
;
; Round to nearest even
;
        jclr    #10,d5.l,_lmove ; check guard bit
        bclr    #10,d5.l        ; delete G bit
        tst     d5              ; check sticky and round bits
        jne     _addone         ; jump if S or R bits = 1
        jset    #11,d1.l,_addone ; add one if LSB of result = 1
        jmp     _lmove          ; no rounding required
_r1chk  jclr    #22,sr,_lmove   ; jump if round toward zero
;
; Round toward +infinity
;
        jclr    #31,d6.l,_addone ; add one if positive
        jmp     _lmove           ; get result in d1
```

```
;
; Round toward -infinity
;
_rminf  jclr    #31,d6.l,_lmove ; no rounding if positive
_addone move    #$800,d7.l      ; get increment number
        add     d7,d1           ; add one to lsb
        jcc     _acar           ; jump if no carry
        inc     d3              ; increment mantr.high
_acar   jcc     _lmove          ; jump if result normalized
        lsr     d3              ; shift right mantr.high
        ror     d1              ; shift right mantr.low
        inc     d6              ; increment expr
        ;
        ; Check if Result is Infinity
        ;
        move    #emsk,d7.l      ; get exp mask
        move    d6.l,d5.l       ; get expr
        and     d7,d5           ; delete tags
        cmp     d7,d5           ; check for max exp
        jne     _lmove          ; jump if no overflow
        move    #0,d1.l         ; set result to infinity
        move    #0,d1.m         ;        "
        ori     #$10,ccr        ; set infinity bit
        ori     #$09,ier        ; set OVF and INX bits in IER
        ori     #$09,er         ; set OVF and INX bits in ER
        jmp     _emove          ; get infinity exponent
;
; Get Result in D1
;
_lmove  move    d3.l,d1.m       ; move mantr.high to d1
_emove  move    d6.l,d1.h       ; move expr to d1
_done   nop                     ;
        nop                     ;
        nop                     ;
        rts                     ; end of subroutine
            endsec
```

B.4.2 IEEE Double Precision Subtraction

```
;
; Double Precision IEEE floating-point Subtraction
;
;    D0 - D1 → D1
;
;
;
; Alters Data ALU Registers
;       d0.h    d0.m    d0.l
;       d1.h    d1.m    d1.l
;               d2.m    d2.l
;                       d3.l
```

```
;       d4.h            d4.l
;       d5.h            d5.l
;                       d6.l
;                       d7.l
;
; Alters Program Control Registers
;       pc      sr
;
;
; Version 1.0
; Latest Revision - 01-Aug-88
;
           section     ieeesub
emsk    equ     $7ff            ; exponent mask
eden    equ     $1              ; denorm exponent
grsmsk  equ     $700            ; GRS (guard-round-sticky) bits mask
grmsk   equ     $fffffe00       ; GR (guard-round) bits mask
smsk    equ     $ffffff00       ; mask to clear bits to right of the sticky bit
onemsk  equ     $1ff            ; mask to set bits to right of the round bit
inum    equ     $100            ; increment number
imsk    equ     $7fffffff       ; infinity mask
qnane   equ     $7ff            ; quiet NaN exponent
qnanmh  equ     $7fffffff       ; quiet NaN mantissa high
qnanml  equ     $ffffffff       ; quiet NaN mantissa low
maxnum  equ     $fffff800       ; low part of maximum number
;
sdptest                         ; double precision subtraction subroutine
;
; Clear ER portion of status register
;
        andi    #0,er
;
; Check for Maximum and Minimum Exponents
;
        bchg    #31,d1.h              ; change sign of addend 1
        move    #0,d6.l               ; addend 0 flag
        move    d0.h,d4.l             ; get exp0
        move    #emsk,d7.l            ; get exponent mask
        and     d7,d4      d0.m,d2.l  ; delete tags, get m0.h
        cmp     d7,d4      d1.m,d3.l  ; check max exp, get m1.h
        jeq     _mant1                ; jump if exp0 = max exp
        move    d1.h,d5.l             ; get exp1
        and     d7,d5      #1,d6.l    ; delete tags, a1 flag
        cmp     d7,d5      #0,d2.m    ; check max exp, sticky=0
        jeq     _mant2                ; jump if exp1 = max exp
        tst     d4    #0,d6.l         ; check min exp, a0 flag
        jeq     _mant3                ; jump if exp0 = min exp
        tst     d5    #1,d6.l         ; check min exp, a1 flag
        jeq     _mant4                ; jump if exp1 = min exp
        jmp     _nadd                 ; jump to normalized add
        ;
        ; Check if Addend 0 is Infinity
        ;
_mant1  move    #imsk,d7.l       ; get infinity mask
```

```
        and     d7,d2             ; remove implied one bit
        tst     d2                ; check m0.high = zero
        jne     _nan0             ; jump if nan
        tst     d0                ; check m0.low = zero
        jne     _nan0             ; jump if nan
        move    #emsk,d7.l        ; get exponent mask
        move    d1.h,d5.l         ; get exp1
        and     d7,d5             ; delete tags
        cmp     d7,d5             ; check for max exp
        jne     _inf1             ; jump if a1 is not inf or NaN
        ;
        ; Check if Addend 1 is Infinity
        ;
_mant2  move    #imsk,d7.l        ; get infinity mask
        and     d7,d3             ; remove implied one bit
        tst     d3                ; check m1.high = zero
        jne     _nan1             ; jump if nan
        tst     d1                ; check m1.low = zero
        jne     _nan1             ; jump if nan
        jclr    #0,d6.l,_binf     ; jump if a0 and a1 are inf
        ori     #$10,ccr          ; set infinity bit
        jmp     _done             ; a1 is infinity
        ;
        ; Check for Case: (+Inf) + (-Inf) = QNaN
        ;
_inf1   ftfr.x  d0,d1             ; move result to d1
            ori     #$10,ccr          ; set infinity bit
        jmp     _done             ; a0 is infinity
_binf   ftst    d0                ; check sign of a0
        jmi     _minf             ; jump if a0 is -inf
        ftst    d1                ; check sign of a1
        jmi     _inan             ; (+inf) + (-inf) = QNaN
        ori     #$10,ccr          ; set infinity bit
        jmp     _done             ; a0 and a1 are +inf
_minf   ftst    d1                ; check sign of a1
        jpl     _inan             ; (-inf) + (+inf) = QNaN
        ori     #$10,ccr          ; set infinity bit
        jmp     _done             ; a0 and a1 are -inf
        ;
        ; Check for NaN
        ;
_nan0   jclr    #30,d0.m,_inan    ; jump if a0 is a SNaN
        move    #emsk,d7.l        ; get exponent mask
        move    d1.h,d5.l         ; get exp1
        and     d7,d5             ; delete tags
        cmp     d7,d5             ; check for max exp
        jne     _qnan             ; jump if a1 is not a NaN
        move    #imsk,d7.l        ; get infinity mask
        and     d7,d3             ; remove implied one bit
        tst     d3                ; check mant1.high = zero
        jne     _nan1             ; jump if a1 is a NaN
        tst     d1                ; check mant1.low = zero
        jeq     _qnan             ; jump if a1 is infinity
_nan1   jset    #30,d1.m,_qnan    ; jump if a1 is a QNaN
```

```
_inan   ori     #$10,ier          ; set invalid operation bit
_qnan   move    #qnane,d1.h       ; get QNaN exponent
        move    #qnanmh,d1.m      ; get QNaN mantissa high
        move    #qnanml,d1.l      ; get QNaN mantissa low
        ori     #$20,ccr          ; set Not-a-Number bit
        jmp     _done             ; result is a NaN
        ;
        ; Check if Addend 0 is a Denormalized Number
        ;
_mant3  tst     d2                ; check mant0.high = zero
        jne     _den0             ; jump if a0 is a denorm
        tst     d0                ; check mant0.low = zero
        jne     _den0             ; jump if a0 is a denorm
        cmp     d7,d5             ; check min exp for a1
        jgt     _done             ; a1 is the answer
        ;
        ; Check if Addend 1 is a Denormalized Number
        ;
_mant4  tst     d3                ; check mant1.high = zero
        jne     _den1             ; jump if a1 is a denorm
        tst     d1                ; check mant1.low = zero
        jne     _den1             ; jump if a1 is a denorm
        jclr    #0,d6.l,_bzero    ; jump if both are zero
        jmp     _tfr              ; move result to d1
        ;
        ; Addend 0 is a Denormalized Number
        ;
_den0   bset    #0,d0.h           ; get denorm exponent
        inc     d4                ;       "
        tst     d5                ; check if a1 is a denorm
        jgt     _ftz              ; jump if a1 is a normal number
        tst     d3                ; check mant1.high = zero
        jne     _bden             ; jump if a1 is a denorm
        tst     d1                ; check mant1.low = zero
        jne     _bden             ; jump if a1 is a denorm
        jmp     _tfr              ; move result to d1
_bden   bset    #0,d1.h           ; get denorm exponent
        inc     d5                ;       "
_ftz    jclr    #27,sr,_nadd      ; jump to add for ieee mode
        jmp     _done             ; a0 is flushed-to-zero
        ;
        ; Addend 1 is a Denormalized Number
        ;
_den1   jclr    #0,d6.l,_done     ; a1 is the answer
        bset    #0,d1.h           ; get denorm exponent
        inc     d5                ;       "
        jclr    #27,sr,_nadd      ; jump to add for ieee mode
_tfr    move    #eden,d7.l        ; get denorm exponent
        move    d0.h,d5.l         ; get expr
        move    #emsk,d4.l        ; get exponent mask
        and     d4,d5             ; delete tags and sign
        cmp     d7,d5      d0.h,d5.l ; compare exps, get expr
        jne     _tmov             ; jump if not a denorm
        move    d0.m,d3.l         ; get mantr.high
```

```
        tst     d3                 ; test mantr.high, get expr
        dec     d5    ifpl.u       ; decrement expr if no int bit
_tmov   move    d5.l,d1.h          ; move result to d1
        move    d0.m,d1.m          ;       "
        move    d0.l,d1.l          ;       "
        jmp     _done              ; a0 is the answer
        ;
        ; Both Addends are Zero
        ;
_bzero  move    d0.h,d4.l          ; get exp0
        move    d1.h,d5.l          ; get exp1
        eor     d4,d5              ; check for opposite signs
        jclr    #31,d5.l,_done     ; jump if same signs
        bclr    #31,d1.h           ; set result as positive
        jclr    #22,sr,_done       ; jump if round bit r1 = zero
        jset    #21,sr,_done       ; jump if round bit r0 = one
        bset    #31,d1.h           ; set result as negative
        jmp     _done              ; result is negative zero
;
;
;
; ******************************************
; *** DP Addition for Normalized Numbers ***
; ******************************************
;
;
; Compare Exponents
;
_nadd   cmp     d4,d5      d1.h,d6.l  ; compare exps, get expr
        jgt     _pos                  ; jump if exp1 > exp0
        jeq     _add                  ; jump if exp0 = exp1
;
; *** Case: Exp0 > Exp1 ***
;
        ;
        ; Align Mantissas
        ;
        sub     d5,d4      d0.h,d6.l  ; get shift, get expr
        move    #55,d7.l              ; get number of bits
        cmp     d7,d4             ; check for shift > 55
        jgt     _setst0           ; jump if shift > 55
        do      d4.l,_end1        ; align mantissas
        lsr     d3                ; shift right m1.h
        ror     d1                ; shift right m1.l and GRS1
        jclr    #8,d1.l,_cclr1    ; jump if sticky bit clear
        move    #1,d2.m           ; set sticky bit
_cclr1  nop                       ;
        ;
        ; Calculate Sticky Bit
        ;
_end1   move    #grmsk,d7.l       ; get GR mask
            and     d7,d1             ; remove bits right of round bit
        jclr    #0,d2.m,_add      ; jump if sticky = 0
        bset    #8,d1.l           ; put in sticky bit
```

```
        jmp     _add            ;
        ;
        ; Set Sticky Bit for Shift > 55 Bits
        ;
_setst0 move    #0,d3.l         ; get number for addition
        move    #inum,d1.l      ;       "
        jmp     _add            ;
;
; *** Case: Exp1 > Exp0 ***
;
        ;
        ; Align Mantissas
        ;
_pos    sub     d4,d5      d1.h,d6.l  ; get shift, get expr
        move    #55,d7.l              ; get number of bits
        cmp     d7,d5           ; check for shift > 55
        jgt     _setst1         ; jump if shift > 55
        do      d5.l,_end2      ; align mantissas
        lsr     d2              ; shift right m0.h
        ror     d0              ; shift right m0.l and GRS0
        jclr    #8,d0.l,_cclr2  ; jump if sticky bit clear
        move    #1,d2.m         ; set sticky bit
_cclr2  nop                     ;
        ;
        ; Calculate Sticky Bit
        ;
_end2   move    #grmsk,d7.l     ; get GR mask
        and     d7,d0           ; remove bits right of round bit
        jclr    #0,d2.m,_add    ; jump if sticky = 0
        bset    #8,d1.l         ; put in sticky bit
        jmp     _add            ;
        ;
        ; Set Sticky Bit for Shift > 55 Bits
        ;
_setst1 move    #0,d2.l         ; get number for addition
        move    #inum,d0.l      ;       "
;
; Check the Signs of the Addends
;
_add    jset    #31,d0.h,_neg1  ; jump if a0 negative
        jset    #31,d1.h,_neg2  ; jump if a1 negative
        jmp     _fadd           ; jump to addition for a0+,a1+
;
; *** Case: Addend 0 is Negative,
;           Addend 1 is Positive ***
;
_neg1   jset    #31,d1.h,_nset  ; jump if a1 negative
        sub     d0,d1           ; subtract for case: a0-,a1+
        subc    d2,d3           ;         "
        jcc     _zchk           ; jump if result is positive
        bset    #31,d6.l        ; set result as negative
        move    #inum,d7.l      ; get increment number
        not     d1              ; get 2's comp of result
        not     d3              ;         "
```

```
        add     d7,d1           ;         "
        jcc     _zchk           ;         "
        inc     d3              ;         "
        jmp     _zchk           ;
;
; *** Case: Addend 0 is Positive,
;           Addend 1 is Negative ***
;
_neg2   bclr    #31,d6.l        ; set result as positive
        sub     d1,d0           ; subtract for case: a0+,a1-
        subc    d3,d2           ;         "
        jcc     _cclr3          ; jump if result is positive
        bset    #31,d6.l        ; set result as negative
        move    #inum,d7.l      ; get increment number
        not     d0              ; get 2's comp of result
        not     d2              ;         "
        add     d7,d0           ;         "
        jcc     _cclr3          ;         "
        inc     d2              ;         "
_cclr3  move    d0.l,d1.l       ; get mantr.low and GRS bits
        move    d2.l,d3.l       ; get mantr.high
;
; Check result equal zero (do not want to normalize)
;
_zchk   move    #smsk,d7.l      ; get sticky mask
        and     d7,d1           ; remove bits right of sticky
        tst     d3              ; check mantr.high = zero
        jne     _snrm           ; normalize result
        tst     d1              ; check mantr.low = zero
        jne     _snrm           ; normalize result
        move    #0,d6.l         ; set expr = zero
        ;
        ; Check for Special Case (Round toward -infinity)
        ;
        jclr    #22,sr,_rnd     ; jump if round bit r1 = zero
        jset    #21,sr,_rnd     ; jump if round bit r0 = one
        bset    #31,d6.l        ; set result to negative zero
        jmp     _rnd            ; check rounding mode
;
; Normalize for Opposite Sign Cases
;
_snrm   jset    #31,d3.l,_rnd   ; jump if result normalized
        move    #emsk,d7.l      ; get exp mask
        move    d6.l,d5.l       ; get expr
        and     d7,d5           ; delete tags
        move    #eden,d4.l      ; get denorm exponent
        jclr    #8,d1.l,_st0    ; jump if sticky bit = 0
        move    #onemsk,d7.l    ; get one mask
_st1    or      d7,d1           ; set bits right of round bit
_st0    cmp     d4,d5           ; test expr = zero
            jle     _rnd            ; jump if denormalized number
        dec     d6              ; decrement expr
        dec     d5              ; decrement expr copy
        lsl     d1              ; shift mantr.l left
```

```
        rol     d3               ; shift mantr.h left
        jset    #31,d3.l,_rnd    ; jump if result normalized
        jclr    #8,d1.l,_st0     ; jump if sticky bit = 0
        jmp     _st1             ; jump if sticky bit = 1
;
; *** Cases: 1) Addend 0 is Negative,
;               Addend 1 is Negative
;            2) Addend 0 is Positive,
;               Addend 1 is Positive ***
;
_nset   bset    #31,d6.l         ; set result as negative
_fadd   add     d0,d1            ; add for case: a0-,a1-
        addc    d2,d3            ;     and case: a0+,a1+
        jcc     _rnd             ; jump if number normalized
        lsr     d3               ; shift right mantr.h
        ror     d1               ; shift right mantr.low
        jclr    #8,d1.l,_cclr4   ; jump if sticky bit = 0
        bset    #8,d1.l          ; set sticky bit
_cclr4  bset    #31,d3.l         ; set bit 31 of mantr.high
        inc     d6               ; increment expr
;
; Check if Result is Infinity
;
        move    #emsk,d7.l       ; get exp mask
        move    d6.l,d5.l        ; get expr
        and     d7,d5            ; delete tags
        cmp     d7,d5            ; check max exp
        jne     _rnd             ; jump if no overflow
        jset    #31,d6.l,_ninf   ; jump if result is -infinity
        ;
        ; Positive Infinity
        ;
        jset    #22,sr,_rmchk    ; jump if rounding bit r1 = 1
        jclr    #21,sr,_setinf   ; jump if rounding bit r0 = 0
        jmp     _setbig          ; round toward zero case
        ;
        ; Negative Infinity
        ;
_ninf   jclr    #21,sr,_setinf   ; jump if rounding bit r0 = 0
        ;
        ; Result is Largest Number Less Than Infinity
        ;
_setbig dec     d6               ; get big exponent
        move    #qnanml,d1.m     ; get mantr.high
        move    #maxnum,d1.l     ; get mantr.low
        ori     #$09,ier         ; set OVF and INX bits in IER
        ori     #$09,er          ; set OVF and INX bits in ER
        jmp     _emove           ; get expr
_rmchk  jclr    #21,sr,_setbig   ; round toward -inf case
        ;
        ; Result is Infinity
        ;
_setinf move    #0,d1.l          ; set result to infinity
        move    #0,d1.m          ;         "
```

```
        move    d6.l,d1.h       ;         "
        ori     #$10,ccr        ; set infinity bit
        ori     #$09,ier        ; set OVF and INX bits in IER
        ori     #$09,er         ; set OVF and INX bits in ER
        jmp     _done           ; result is infinity
;
; Begin Rounding the Result
;
        ;
        ; Check for Denormalized Numbers
        ;
_rnd    move    #eden,d7.l      ; get denorm exponent
        move    d6.l,d5.l       ; get expr
        move    #emsk,d4.l      ; get exponent mask
        and     d4,d5           ; delete tags and sign
        cmp     d7,d5           ; compare exponents
        jne     _remst          ; jump if not a denorm
        tst     d3              ; test mantr.high
        dec     d6    ifpl.u    ; decrement expr if no int bit
        ;
        ; Remove Bits to Right of the Sticky Bit
        ;
_remst  move    #smsk,d7.l      ; get sticky mask
        and     d7,d1           ; remove bits right of sticky
        ;
        ; Check GRS Bits Equal Zero
        ;
        move    d1.l,d5.l       ; get register with GRS bits
        move    #grsmsk,d7.l    ; get GRS mask
        and     d7,d5           ; get GRS bits
        tst     d5              ; check GRS bits = zero
        jeq     _lmove          ; jump if no rounding required
        ori     #$1,ier         ; set inexact result bit
        ori     #$1,er          ; set inexact result bit
        ;
        ; Check Rounding Mode
        ;
        jset    #21,sr,_r1chk   ; jump if rounding bit r0 = 1
        jset    #22,sr,_rminf   ; jump if round toward -infinity
;
; Round to nearest even
;
        jclr    #10,d5.l,_lmove ; check guard bit
        bclr    #10,d5.l        ; delete G bit
        tst     d5              ; check sticky and round bits
        jne     _addone         ; jump if S or R bits = 1
        jset    #11,d1.l,_addone ; add one if LSB of result = 1
        jmp     _lmove          ; no rounding required
_r1chk  jclr    #22,sr,_lmove   ; jump if round toward zero
;
; Round toward +infinity
;
        jclr    #31,d6.l,_addone ; add one if positive
        jmp     _lmove           ; get result in d1
```

```
;
; Round toward -infinity
;
_rminf  jclr    #31,d6.l,_lmove ; no rounding if positive
_addone move    #$800,d7.l      ; get increment number
        add     d7,d1           ; add one to lsb
        jcc     _acar           ; jump if no carry
        inc     d3              ; increment mantr.high
_acar   jcc     _lmove          ; jump if result normalized
        lsr     d3              ; shift right mantr.high
        ror     d1              ; shift right mantr.low
        inc     d6              ; increment expr
        move    #emsk,d7.l      ; get exp mask
        move    d6.l,d5.l       ; get expr
        and     d7,d5           ; delete tags
        cmp     d7,d5           ; check for max exp
        jne     _lmove          ; jump if no overflow
        move    #0,d1.l         ; set result to infinity
        move    #0,d1.m         ;        "
        ori     #$10,ccr        ; set infinity bit
        ori     #$09,ier        ; set OVF and INX bits in IER
        ori     #$09,er         ; set OVF and INX bits in ER
        jmp     _emove          ; get infinity exponent
;
; Get Result in D1
;
_lmove  move    d3.l,d1.m       ; move mantr.high to d1
_emove  move    d6.l,d1.h       ; move expr to d1
_done   nop                     ;
        nop                     ;
        nop                     ;
        rts                     ; end of subroutine
            endsec
```

B.4.3 IEEE Double Precision Multiplication

```
;
;*******************************************************************
;*******************************************************************
; ***   IEEE Double Extended Precision Multiply Operation
; ***
; ***   The routine was implemented as a unsigned multiply routine.
; ***
; ***      64-bit input operand format (immediately before multiply):
; ***               i.fff...fl
; ***
; ***      67-bit intermediate result format (immediately after post norm):
; ***               i.fff...flgrs
; ***         where
; ***            i = integer bit
```

```
; ***            f = fraction bits, initially bits in mantissas
; ***            l = least significant fraction bit, initially in mantissas
; ***            g = guard bit
; ***            r = round bit
; ***            s - sticky bit
; ***
; ***
; ***   Routine Inputs:
; ***      d6 - IEEE double extended precision operand 1   (destroyed)
; ***      d7 - IEEE double extended precision operand 2   (destroyed)
; ***
; ***   Routine Outputs:
; ***      d5 - IEEE double extended precision result
; ***
; ***   Registers Used:
; ***      d0.l - general purpose usage
; ***      d0.m - unbiased operand 2 exponent
; ***           - unbiased result exponent
; ***      d0.h - MSB contains the XOR of the sign bits
; ***      d1.l - general purpose usage
; ***      d1.m - unbiased operand 1 exponent
; ***           - loop index for denormalizing upon underflow.
; ***      d1.h - LSB contains the sticky bit
; ***      d2.m,l - partial product and intermediate calculations
; ***      d3.m,l - partial product and intermediate calculations
; ***      d4.m,l - partial product and intermediate calculations
; ***      d5.m,l - partial product and intermediate calculations
; ***
; ***   NOTES: Currently ignores the FR, P, RP bits.
; ***          Assumes that operands are NOT UNnormalized numbers
; ***          Code size greatly decreased if "depftst" macro
; ***             becomes a routine
; ***
           section ieeemult
SR_MASK equ     $ffff80c0       ; Status Register Mask, resets cond. codes
EXP_MSK equ     $7ff            ; Mask for exponent field, 16-bits
EBIAS   equ     $3ff            ; Exponent bias for IEEE double precision
EMAX    equ     $3ff            ; Max exp for normalized double precision val
EMIN    equ     $fffffc02       ; Min exp for normalized double precision val
EDEN    equ     $fffffc01       ; Exp for denormalized double precision val
MAX     equ     $7ff            ; Max exp (biased), indicating infs & NaNs
SMSK    equ     $1ff            ; Mask for sticky bit calculation
INUM    equ     $800            ; Increment for LSB
sdptest                         ; double precision multiplication subroutine

; ****** Define Program Macros ******

depftst macro   op,tmp1,tmp2

; This macro performs the "ftst" function on DEP floating pt vars.
; It sets the NAN,I,N,Z bits in the CCR register appropriately.
;
;    op = register name of the form "Dn" containing the floating pt var
;         in all 96 bits of the register (not destroyed)
```

```
;    tmp1 = register name of the form "Dn", and is a temporary var which
;           uses the lowest 32 bits of the register (Dn.L is destroyed)
;    tmp2 = register name of the form "Dn", and is a temporary var which
;           uses the lowest 32 bits of the register (Dn.L is destroyed)
;
; Note that op, tmp1, and tmp2 must all be different registers.

        andi    #$c3,ccr                  ;

        jclr    #31,op.h,_chkrst          ;
        ori     #$8,ccr                   ;

_chkrst move    op.h,tmp1.l               ;
        move    #EXP_MSK,tmp2.l           ;
        and     tmp2,tmp1                 ;

        tst     op                        ;
        jneq    _chknan                   ;
        tst     tmp1    op.m,tmp2.l       ;
        jneq    _maxexp                   ;
        tst     tmp2                      ;
        jneq    _chknan                   ;
        ori     #$4,ccr                   ;
        jmp     _done                     ;
_maxexp move    #MAX,tmp2.l               ;
        cmp     tmp1,tmp2     op.m,tmp1.l ;
        jne     _done                     ;

        bclr    #31,tmp1.l                ;
        tst     tmp1                      ;
        jneq    _nan                      ;
        andi    #$b,ccr                   ;
        ori     #$10,ccr                  ;
        jmp     _done                     ;
_chknan move    #MAX,tmp2.l               ;
        cmp     tmp1,tmp2                 ;
        jne     _done                     ;
_nan    andi    #$b,ccr                   ;
        ori     #$20,ccr                  ;
        jset    #30,op.m,_done            ;
        ori     #$20,er                   ;
_done
        endm



; ****** Reset Processor Flags ******

        movec   sr,d0.l                 ;
        move    #SR_MASK,d1.l           ;
        and     d1,d0                   ;
        movec   d0.l,sr                 ;
```

```
; ****** Flush DeNorms to 0 if Fast Mode ******

        jclr    #27,sr,_chksgn          ;
        move    #$80000000,d1.l         ;
        jset    #31,d6.m,_chkop2        ;
        fclr    d6      d6.h,d0.l       ;
        and     d1,d0                   ;
        move    d0.l,d6.h               ;
_chkop2 jset    #31,d7.m,_chksgn        ;
        fclr    d7      d7.h,d0.l       ;
        and     d1,d0                   ;
        move    d0.l,d7.h               ;



; ****** Sign Bit Calculation ******

_chksgn move    d6.h,d0.l                 ;
        move    d7.h,d1.l                 ;
        eor     d1,d0                     ;
        move    d0.l,d0.h                 ;



; ****** Check Input Operands ******

_chkops
        depftst d6,d0,d1                  ;
        jeq     _op1_0                    ;
        jset    #4,sr,_op1inf             ;
        jset    #5,sr,_op1nan             ;

        depftst d7,d0,d1                  ;
        jeq     _op2_0                    ;
        jset    #4,sr,_op2inf             ;
        jset    #5,sr,_op2nan             ;



; ****** Extract Exponents ******
;       ------ Should be able to use FGETEXP here on double-extended ------

        move    d7.h,d0.l                 ;
        move    #EXP_MSK,d1.l             ;
        and     d1,d0                     ;
        tst     d0                        ;
        jne     _ebias1                   ;
        inc     d0                        ;
_ebias1 move    #EBIAS,d1.l               ;
        sub     d1,d0                     ;
        move    d0.l,d0.m                 ;

        move    d6.h,d0.l                 ;
```

```
        move    #EXP_MSK,d1.l             ;
        and     d1,d0                     ;
        tst     d0                        ;
        jne     _ebias2                   ;
        inc     d0                        ;
_ebias2 move    #EBIAS,d1.l               ;
        sub     d1,d0                     ;
        move    d0.l,d1.m                 ;



; ****** Extract Mantissas ******

        move    #0,d6.h                 ;
        move    #0,d7.h                 ;



; ****** Normalize any Denorms ******

        jset    #31,d6.m,_nrmop2        ;
        move    d6.m,d0.l               ;
        tst     d0                      ;
        jneq    _op1nrm                 ;
        move    d1.m,d0.l               ;
        move    #32,d1.l                ;
        sub     d1,d0                   ;
        move    d0.l,d1.m               ;
        move    d6.l,d6.m               ;
        move    #0,d6.l                 ;
        jset    #31,d6.m,_nrmop2        ;
_op1nrm                                 ;    normalize
        asl     d6      d6.m,d0.l       ;
        rol     d0                      ;
        move    d0.l,d6.m               ;
        move    d1.m,d0.l               ;
        dec     d0                      ;
        move    d0.l,d1.m               ;
        jclr    #31,d6.m,_op1nrm        ;

_nrmop2 jset    #31,d7.m,_domul         ;
        move    d7.m,d0.l               ;
        tst     d0                      ;
        jneq    _op2nrm                 ;
        move    d0.m,d0.l               ;
        move    #32,d1.l                ;
        sub     d1,d0                   ;
        move    d0.l,d0.m               ;
        move    d7.l,d7.m               ;
        move    #0,d7.l                 ;
        jset    #31,d7.m,_domul         ;
_op2nrm                                 ;    normalize operand 2
        asl     d7      d7.m,d0.l       ;
        rol     d0                      ;
```

```
        move    d0.l,d7.m               ;
        move    d0.m,d0.l               ;
        dec     d0                      ;
        move    d0.l,d0.m               ;
        jclr    #31,d7.m,_op2nrm        ;
_domul



; ****** Initial Exponent Processing ******

        move    d0.m,d0.l               ;
        move    d1.m,d1.l               ;
        add     d0,d1                   ;
        inc     d1                      ;
        move    d1.l,d0.m               ;



; ****** Calculate Partial Products (A:B * C:D) ******

        mpyu    d6,d7,d2                ;
        move    d6.m,d0.l               ;
        mpyu    d0,d7,d3                ;
        move    d7.m,d1.l               ;
        mpyu    d1,d6,d4                ;
        mpyu    d0,d1,d5                ;



; ****** Sum Partial Products ******

        move    #0,d1.h                 ;
        tst     d2      d2.m,d0.l       ;
        jeq     _addpps                 ;
        move    #1,d1.h                 ;

_addpps
        add     d0,d3                   ;
        rol     d1          d3.m,d2.l   ;
        add     d4,d3       d4.m,d0.l   ;

        addc    d2,d0                   ;
        rol     d2                      ;
        ror     d1          d5.m,d4.l   ;
        addc    d0,d5                   ;

        move    #0,d0.l                 ;
        addc    d0,d4                   ;
        ror     d2                      ;
        addc    d0,d4                   ;

                                        ; At this point,
                                        ;    d4.l = most significant 32 bits,
```

```
                                        ;    d5.l = next most significant word,
                                        ;    d3.l = next most significant word,
                                        ;    and least significant word info
                                        ;    is in the sticky bit.
                                        ;
                                        ; Upper 96 bits = d4.l:d5.l:d3.l, and
                                        ; the lowest 32 bits have been ORed
                                        ; into the sticky bit.


; ****** Continue Calculating Sticky Bit ******

        move    #SMSK,d0.l              ;
        move    d5.l,d2.l               ;
        and     d0,d2                   ;
        tst     d2                      ;
        jeq     _stlow                  ;
        move    #1,d1.h                 ;

_stlow  tst     d3                      ;
        jeq     _post                   ;
        move    #1,d1.h                 ;


; ****** Post Normalization ******

_post   jset    #31,d4.l,_undr          ;
_ptop   asl     d3                      ;
        rol     d5                      ;
        rol     d4      d0.m,d0.l       ;
        move    d0.m,d0.l               ;
        dec     d0                      ;
        move    d0.l,d0.m               ;
        jclr    #31,d4.l,_ptop          ;


; ****** Underflow Check ******
_undr
        move    d0.m,d0.l               ;
        move    #EMIN,d1.l              ;
        sub     d1,d0                   ;
        jpl     _rnd                    ;

        jset    #27,sr,_ret0            ;
                                        ;
        move    #-52,d1.l               ;
        cmp     d1,d0                   ;
        jmi     _rnd0                   ;
```

```
        move    #EDEN,d0.m              ;
        abs     d0                      ;
        move    d0.l,d1.m               ;
        do      d1.m,_dnrmq             ;
        lsr     d4                      ;
        ror     d5                      ;
        ror     d3                      ;
_dnrmq
        jset    #0,d1.h,_sundr          ;
        jset    #9,d5.l,_sundr          ;
        jset    #10,d5.l,_sundr         ;
        jmp     _asml                   ;


; ****** Round ******

_rnd    jset    #10,d5.l,_inex          ;
        jset    #9,d5.l,_inex           ;
        jset    #0,d1.h,_inex           ;
        jmp     _endrnd                 ;
_sundr
        ori     #$04,er                 ;
        ori     #$04,ier                ;
_inex                                   ;
        ori     #$01,er                 ;
        ori     #$01,ier                ;
        jclr    #22,sr,_nxt             ;
        jset    #21,sr,_pinf            ;
        jclr    #31,d0.h,_endrnd        ;
        jmp     _add1                   ;
_nxt                                    ;
        jclr    #21,sr,_rn              ;
        jmp     _endrnd                 ;
_pinf   jset    #31,d0.h,_endrnd        ;
        jmp     _add1                   ;

_rn     jclr    #10,d5.l,_endrnd        ;
        jset    #9,d5.l,_add1           ;
        jset    #0,d1.h,_add1           ;
        jset    #11,d5.l,_add1          ;
        jmp     _endrnd                 ;
                                        ;

_add1   move    #INUM,d0.l              ;
        add     d0,d5                   ;
        move    #0,d0.l                 ;
        addc    d0,d4                   ;

        jcc     _den                    ;
        move    d0.m,d0.l               ;
        inc     d0                      ;
        move    d0.l,d0.m               ;
        lsr     d4                      ;
        ror     d5                      ;
```

```
        jmp     _endrnd                 ;

_den    move    #EDEN,d1.l              ;
        move    d0.m,d0.l               ;
        cmp     d1,d0                   ;
        jne     _endrnd                 ;
        jclr    #31,d4.l,_asml          ;
        inc     d0                      ;
        move    d0.l,d0.m               ;
        jmp     _asml                   ;


_rnd0                                   ; Reaches here if value is too small
                                        ;    to denormalize.
        ori     #$05,er                 ;
        ori     #$05,ier                ;
        jset    #22,sr,_nxt1            ;
        jclr    #21,sr,_rn1             ;
        jmp     _ret0                   ;

_pinf1  jset    #31,d0.h,_ret0          ;
        jmp     _retsml                 ;

_rn1    move    #-56,d1.l               ;
        cmp     d1,d0                   ;
        jle     _grs0                   ;
        move    #-53,d1.l               ;
_grsl   cmp     d1,d0                   ;
        jeq     _rnrnd                  ;
        lsr     d4                      ;
        dec     d0                      ;
        jmp     _grsl                   ;
_grs0   move    #0,d4.l                 ;
_rnrnd  jclr    #31,d4.l,_ret0          ;
        jset    #30,d4.l,_retsml        ;
        jset    #0,d1.h,_retsml         ;
        jmp     _ret0                   ;

_nxt1                                   ;
        jset    #21,sr,_pinf1           ;
        jclr    #31,d0.h,_ret0          ;

_retsml jset    #27,sr,_ret0            ;
        move    #0,d5.h                 ;
        move    d5.h,d5.m               ;
        move    #$800,d5.l              ;
        jmp     _putsgn                 ;

_endrnd



; ****** Overflow Check ******
```

```
        move    #EMAX,d1.l              ;
        move    d0.m,d0.l               ;
        cmp     d1,d0                   ;
        jle     _asml                   ;
        ori     #$09,er                 ;
        ori     #$09,ier                ;

        jclr    #22,sr,_next            ;
        jset    #21,sr,_posinf          ;
        jset    #31,d0.h,_retinf        ;
        jmp     _retlrg                 ;
_posinf
        jclr    #31,d0.h,_retinf        ;
        jmp     _retlrg                 ;
_next                                   ;
        jclr    #21,sr,_retinf          ;
                                        ;
_retlrg move    #$ffffffff,d5.m         ;
        move    #$ffffffff,d5.l         ;
        move    #MAX,d0.l               ;
        dec     d0                      ;
        move    d0.l,d5.h               ;
        jmp     _putsgn                 ;


; ****** Assemble Result into IEEE Format ******

_asml   move    d4.l,d5.m               ;

        move    d0.m,d0.l               ;
        move    #EBIAS,d1.l             ;
        add     d1,d0                   ;
        move    d0.l,d5.h               ;

        jclr    #31,d0.h,_done          ;
        bset    #31,d5.h                ;



; ****** Exit Routine ******

_putsgn jclr    #31,d0.h,_done          ;
        bset    #31,d5.h                ;
        jmp     _done                   ;




; ****** Zero Operand Detected (or denorm in FAST mode) ******

_op2_0  depftst d6,d0,d1                ;
```

```
        jset    #5,sr,_op1nan           ;
        jset    #4,sr,_operr            ;
        jmp     _ret0                   ;

_retinf move    #0,d5.l                 ;
        move    d5.l,d5.m               ;
        move    #MAX,d5.h               ;
        ori     #$10,ccr                ;
        jmp     _putsgn                 ;

_op1_0  depftst d7,d0,d1                ;
        jset    #5,sr,_op2nan           ;
        jset    #4,sr,_operr            ;

_ret0   move    #0,d5.h                 ;
        move    d5.h,d5.m               ;
        move    d5.m,d5.l               ;
        bset    #2,sr                   ;
        jmp     _putsgn                 ;

_operr  bset    #12,sr                  ;
        bset    #20,sr                  ;
        bset    #4,sr                   ;
        move    #$ffffffff,d5.l         ;
        move    #$ffffffff,d5.m         ;
        move    #$7ff,d5.h              ;
        jclr    #31,d0.h,_done          ;
        bset    #31,d5.h                ;
        jmp     _done                   ;



; ****** Infinity Operand Detected ******

_op2inf depftst d6,d0,d1                ;
        jset    #5,sr,_op1nan           ;
        jeq     _operr                  ;
        jmp     _retinf                 ;

_op1inf depftst d7,d0,d1                ;
        jset    #5,sr,_op2nan           ;
        jeq     _operr                  ;
        jmp     _retinf                 ;



; ****** NaN Operand Detected ******

_op2nan jset    #13,sr,_op2sn           ;
        ftfr.x  d7,d5                   ;
        jmp     _done                   ;
_op1nan jset    #13,sr,_op1sn           ;
        bset    #4,sr                   ;
        ftfr.x  d6,d5                   ;
```

```
        depftst d7,d0,d1                ;
        jset    #5,sr,_snan2            ;
        jmp     _done                   ;
_snan2  jset    #13,sr,_op2sn           ;
        jmp     _done                   ;

_op2sn  ftfr.x  d7,d6                   ;
_op1sn  bset    #30,d6.m                ;
        ftfr.x  d6,d5                   ;
        bset    #4,sr                   ;
        bset    #13,sr                  ;
        bset    #20,sr                  ;
_done   nop                             ;
        nop
        nop
        rts                             ; end of subroutine
            endsec
```

B.5 NON-IEEE DOUBLE PRECISION USING SOFTWARE EMULATION

```
dplib    ident    1,0
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; EXTENDED DOUBLE PRECISION floating-point SUBROUTINE LIBRARY
;
         page     132,60,1,1
;
; equates
;
exp      equ      0                ;offset to exponent
sign     equ      1                ;offset to sign
ms       equ      2                ;offset to most significant word
ls       equ      3                ;offset to least significand word
bias     equ      $1fffffff        ;exponent bias
dptemp   equ      $1fc             ;temporary storage in top 4 internal
                                   ;x memory locations
         page
         org      x:dptemp         ;double precision register
         ds       1                ;exponent
         ds       1                ;sign: 0=+, 1=-
         ds       2                ;64 bit significand
;
         page
         org      p:
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; IEEE2DPLIB - Convert floating-point number in d0 to an internal
```

```
;       extended precision number.
;
; Entry point:  ieee2dplib:  c(r0) ← convert(d0)
;
; Input:  r0 contains the lowest address of the 4-word internal
;             extended precision number
;         d0 contains the DSP96002 floating-point number.
;             The DSP96002 has the following floating-point formats:
;                SP normalized (24 bit mantissa)
;                SP denormalized
;                SEP normalized (32 bit mantissa)
;                SEP denormalized (encoded as DP normalized)
;                DP normalized
; The SP denormalized is encoded using the U tag.  All other encodings
; appear the same with varying amount of significand bits.
;
; Output:  r0 points to the lowest address of a double precision
;             number in non-IEEE double precision format.
;
; Error checking:
;          NaNs      - Not converted, internal A register not affected
;          +/- inf   - Limited to maximum internal format value
;
; Alters:  D0.L,D1.L,D2.L,D0.H,D1.H
;
ieee2dplib
         ftst     d0                     ;check input
         fjor     _notnan                ;ok if not nan
         rts                             ;no conversion
_notnan  fjeq     uflow                  ;if zero, set zero
         clr      d1                     ;get zero for sign
         bclr     #31,d0.h               ;get sign and clear sign bit
         inc      d1       ifcs          ;if sign bit is set, inc
         ftst     d0       d1.l,x:(r0+sign)    ;reset flags, save sign
         fjinf    oflow                  ;limit if infinity
         jset     #30,d0.h,_dodenorm     ;do denorm if U tag is set
         move     d0.m,x:(r0+ms)         ;save ms of significand
         move     d0.l,x:(r0+ls)         ;save ls of significand
         move     d0.h,d0.l              ;get dp exponent
         move     #$1ffffc00,d1.l        ;get bias adjustment
         add      d1,d0                  ;new bias
         move     d0.l,x:(r0)            ;set exponent
         rts
_dodenorm
         move     d0.m,d0.l              ;get denormed sp significand
         bfind    d0,d1                  ;find first 1
         clr      d2       d1.h,d1.l     ;get a 0, move shift
         lsl      d1.h,d0  d2.l,x:(r0+ls)  ;norm ms, set 0 ls
         move     d0.l,x:(r0+ms)         ;set ms
         move     #$1fffff81,d0.l        ;get exponent
         sub      d1,d0                  ;sub denorm shift to get new bias
         move     d0.l,x:(r0)            ;set exponent
         rts
```

```
        page
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DPLIB2IEEE - Convert internal double precision format to a double
;               precision format in d0.
;
; Entry point:  dplib2ieee:  d0 ← convert(c(r0))
;
; Input:  r0 contains the lowest address of the 4-word internal
;             extended precision number
;
; Output:  The returned format is DSP96002 extended precision
;          floating-point format.  Typical calling sequences:
;
;         jsr     dplib2ieee       ;convert to register format
;         move    d0.d,L:0         ;save as dp format
;
;         jsr     dplib2ieee       ;convert to register format
;         ftfr.s  d0,d0            ;round to sp
;         move    d0.s,x:0         ;save as sp format
;
;         jsr     dplib2ieee       ;convert to register format
;         move    d0.d,d0.ml       ;convert to IEEE dp format
;         move    d0.ml,l:0        ;save IEEE dp format
;
; Alters:  D0.L,D1.L,D0.M,D0.H
;
dplib2ieee
        move     x:(r0),d0.l           ;get internal exponent
        move     #$200003fe,d1.l       ;max limit for register
        cmp      d1,d0     #$1ffffc01,d1.l   ;compare to max, get min
        jhi      _setinf               ;too big for register, set inf
        cmp      d1,d0     #$1ffffc00,d1.l   ;compare to min, get adjust
        jlo      _setzero              ;return zero
        sub      d1,d0     x:(r0+ms),d0.m    ;adjust exponent, get ms
        move     d0.l,d0.h             ;move exponent
        move     x:(r0+ls),d0.l        ;get ls part
_fixsign move    x:(r0+sign),d1.l      ;get sign
        jclr     #0,d1.l,_ok           ;jump if bit clear
        bset     #31,d0.h              ;set negative
_ok     rts
_setinf  move    #$7f800000,d0.s       ;get infinity
        jmp      _fixsign
_setzero fclr    d0                    ;get 0
        jmp      _fixsign
        page
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_ABS _ Absolute value of a double precision number
;
; Entry point:  dp_abs:  c(r0+sign)← 0  (make the number positive);
```

```
;
; Input:  r0 contains the lowest address of the 4-word internal
;              extended precision number
;
; Output:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.l
;
dp_abs   clr      d0.l
         move     d0.l,x:(r0+sign)     ;clear the sign word
         rts
         page
;
```

; MOTOROLA DSP96002 DPLIB - VERSION 1.0

;

; DP_ADD - Add two double precision numbers.

```
;
; Entry point:  dp_add:  c(r0) ← c(r0) + c(r1)
;
; Input:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Output:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.L,D1.L,D2.L,D3.L,D4.L,D5.L,D6.L,D7.L,D0.H,D1.H
;
dp_add   move     x:(r0+ms),d0.l       ;get c(r0)_ms
         move     x:(r0+ls),d1.l       ;get c(r0)_ls
         move     x:(r1+ms),d2.l       ;get c(r1)_ms
         move     x:(r1+ls),d3.l       ;get c(r1)_ls
         move     x:(r0),d4.l          ;get c(r0) exponent
         move     x:(r1),d5.l          ;get c(r1) exponent
         move     d4.l,d6.l            ;copy of c(r0) exponent
         cmp      d5,d4      #63,d7.l  ;compare exponents
         jeq      addmant              ;exponents are equal
         jpl      abig                 ;c(r0) exponent is greater
;
; X has a larger exponent than c(r0)
;
xbig     sub      d4,d5      d5.l,d4.l ;c(r1) exponent is greater
         cmp      d5,d7      #31,d7.l  ;is |r0(exp)-r1(exp)| > 63?
         jmi      aequalx              ;yes, then c(r0) + c(r1) = c(r1)
         cmp      d5,d7      #32,d7.l  ;is |r0(exp)-r1(exp)| > 31?
         jge      dshifta              ;no, shift both c(r0) words
         sub      d7,d5      d0.l,d1.l ;yes, shift ms to ls
         clr      d0.l       d5.l,d0.h ;# of shifts to be performed
         lsr      d0.h,d1              ;align the c(r0) mantissa
         jmp      addmant              ;add the mantissas
;
; A has a larger exponent than X
;
```

```
abig     sub      d5,d6                 ;c(r0) exponent is greater
         cmp      d6,d7      #31,d7.l  ;is |r0(exp)-r1(exp)| > 63?
         jmi      aequala               ;yes, then c(r0) + c(r1) = c(r0)
         cmp      d6,d7      #32,d7.l  ;is |r0(exp)-r1(exp)| > 31?
         jge      dshiftx               ;no, shift both c(r1) words
         sub      d7,d6      d2.l,d3.l ;yes, shift ms to ls
         clr      d2.l       d6.l,d0.h ;# of shifts to be performed
         lsr      d0.h,d3               ;align the mantissas
;
; Add the two mantissas together
;
addmant  move     x:(r0+sign),d6.l      ;get c(r0) sign
         move     x:(r1+sign),d7.l      ;get c(r1) sign
         cmp      d7,d6                 ;are the signs the same?
         jmi      apos                  ;c(r0) > 0 and c(r1) < 0
         jeq      signseq               ;c(r0) and X have the same sign
;
; Calculate the result assuming that c(r1) > 0 and c(r0) < 0
;
aneg     cmp      d2,d0                 ;compare mantissas
         jne      decid                 ;if ms's are equal, test ls's
         cmp      d3,d1      #0,d7.l   ;compare ls of mantissas
         jeq      dp_clr                ;clear reg_a if same magnitude
decid    jcc      r1fromr0              ;if c(r0) > c(r1), c(r0) - c(r1)
r0fromr1 move     d7.l,x:(r0+sign)      ;make sign positive
         sub      d1,d3                 ;subtract c(r0) from c(r1)
         subc     d0,d2      d3.l,d1.l ;calculate c(r0)_ms
         move     d2.l,d0.l             ;put result in c(r0) register
;
; Normalize the result
;
subnorm  jeq      msis0                 ;test ms word
         bfind    d0,d0                 ;find out how many zeros in ms
         lsl      d0.h,d0    d1.l,d2.l ;shift c(r0)_ms
         lsl      d0.h,d1    #32,d7.l  ;shift c(r0)_ls
         move     d0.h,d3.l             ;copy # of shifts
         sub      d3,d7                 ;# of opposite dir. shifts
         move     d7.l,d0.h             ;move # of shifts to .h reg.
         lsr      d0.h,d2               ;get bits to go from ls to ms
         or       d2,d0                 ;shift in bits from ls to ms
         sub      d3,d4                 ;decrement the exponent
         jmp      leave                 ;make sure the exp is valid
msis0    tst      d1.l       #32,d3.l  ;test if ls is zero
         jeq      dp_clr                ;zero reg_a if yes
         bfind    d1,d0      d1.l,d0.l ;find out how many zeros in ls
         lsl      d0.h,d0    d0.h,d2.l ;get bits to go from ls to ms
         add      d2,d3                 ;include shifts from ms
         sub      d3,d4      #0,d1.l   ;decrement the exponent
         jmp      leave                 ;make sure the exp is valid
signseq  add      d3,d1                 ;add lower words
         addc     d2,d0                 ;add upper words
addnorm  jcc      leave                 ;test for carry
         ror      d0.l                  ;normalize the sum
         ror      d1.l                  ;shift ms and ls
```

```
         inc      d4.l                 ;increment the exponent
         jmp      leave                ;check for overflow
;
; Calculate the result assuming that c(r0) > 0 and X < 0
;
apos     cmp      d2,d0                ;compare mantissas
         jne      decide               ;if ms's are equal, test ls's
         cmp      d3,d1      #1,d7.l   ;compare ls of mantissas
         jeq      dp_clr               ;clear reg_a if same magnitude
decide   jcc      r1fromr0             ;if c(r0) > c(r1), c(r0) - c(r1)
         jmp      r0fromr1             ;subtract c(r0) from c(r1)
r1fromr0 sub      d3,d1                ;subtract c(r1) from c(r0)
         subc      d2,d0               ;calculate c(r0)_ms
         jmp      subnorm              ;normalize result
;
; Shift the c(r0) 64 bit significand
;
dshifta  move     d5.l,d0.h            ;# of shifts in .h register
         lsr      d0.h,d1    d0.l,d6.l ;shift ls, copy ms
         lsr      d0.h,d0    #32,d7.l  ;shift ms
         sub      d5,d7                ;calc. # opposite dir. shifts
         move     d7.l,d0.h            ;# of shifts in .h register
         lsl      d0.h,d6              ;get bits to be shifted to ls
         or       d6,d1                ;shift in bits from ms to ls
         jmp      addmant
;
; Shift the c(r1) 64 bit significand
;
dshiftx  move     d6.l,d0.h            ;# of shifts in .h register
         lsr      d0.h,d3    d2.l,d1.h ;shift ls, copy ms
         lsr      d0.h,d2    #32,d7.l  ;shift ms
         sub      d6,d7      d1.h,d6.l ;calc. # opposite dir. shifts
         move     d7.l,d0.h            ;# of opposite dir. shifts
         lsl      d0.h,d6              ;get bits to be shifted to ls
         or       d6,d3                ;shift in bits from ms to ls
         jmp      addmant
;
; Replace c(r0) with c(r1)  (c(r0) is insignificant compared to c(r1))
;
aequalx  move     x:(r1+sign),d0.l
         move     d0.l,x:(r0+sign)     ;c(r0)_sign ← c(r1)_sign
         move     d3.l,x:(r0+ls)       ;c(r0)_ls ← c(r1)_ls
         move     d2.l,x:(r0+ms)       ;c(r0)_ms ← c(r1)_ms
         move     d4.l,x:(r0)          ;c(r0)_exp ← c(r1)_exp
;
; Leave c(r0) unchanged  (c(r1) is insignificant compared to c(r0))
;
aequala  rts
;
; Place the result of the operation in c(r0)
;
leave    move     d0.l,x:(r0+ms)       ;store c(r0)_ms
         move     d1.l,x:(r0+ls)       ;store c(r0)_ls
         move     d4.l,x:(r0)          ;store c(r0) exponent
```

```
        jmp      echeck
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_CLR - Set the double precision number to zero.
;
; Entry point:  dp_clr:  c(r0) = 0
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.L
;
dp_clr   clr      d0.l                 ;get a 0
         move     d0.l,x:(r0)
         move     d0.l,x:(r0+sign)
         move     d0.l,x:(r0+ms)
         move     d0.l,x:(r0+ls)
         rts
         page
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_CMP - Compare the two double precision numbers.
;
; Entry point:  dp_cmp:  c(r0)-c(r1)  (set condition codes)
;
; Inputs:  r0 contains the lowest address of the 4-word internal
;             extended precision number
;
; Outputs:  none
;
; CCR CONDITION CODES:
;
;      C - NOT AFFECTED.
;      V - ALWAYS CLEARED
;      Z - SET IF RESULT IS ZERO, CLEARED OTHERWISE.
;      N - SET IF RESULT IS NEGATIVE, CLEARED OTHERWISE.
;      I - NOT AFFECTED.
;     LR - NOT AFFECTED.
;      R - NOT AFFECTED.
;      A - NOT AFFECTED.
;
;      The following Jcc branch conditions can be used after
;      calling dp_cmp.  The other branch conditions should not
;      be used.
;
;      "cc" Mnemonic                  Condition
;      EQ - equal                     Z = 1
```

```
;      GE - greater than or equal     N eor V = 0
;      GT - greater than              Z + (N eor V) = 0
;      LE - less than or equal        Z + (N eor V) = 1
;      LT - less than                 N eor V = 1
;      NE - not equal                 Z = 0
;
; Alters:  D0.L,D1.L,D2.L
;
dp_cmp   move     x:(r0+sign),d0.l     ;get sign
         tst      d0       x:(r0),d1.l ;get exponent
         jeq      _pos1                ;positive
         bset     #31,d1.l             ;set sign bit
_pos1    move     x:(r1+sign),d0.l     ;get sign
         tst      d0       x:(r1),d2.l ;get exponent
         jeq      _pos2                ;positive
         bset     #31,d2.l             ;set sign bit
_pos2    cmp      d2,d1                ;compare signs and exponents
         jeq      _same1               ;more if same
         rts                           ;conditions are set
_same1   move     x:(r0+ms),d1.l       ;get ms parts
         move     x:(r1+ms),d2.l
         cmp      d2,d1                ;compare
         jeq      _same2               ;more if same
         rts                           ;conditions are set
_same2   move     x:(r0+ls),d1.l       ;get ls parts
         move     x:(r1+ls),d2.l
         cmp      d2,d1                ;do final compare
         rts
;
```

```
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
;DP_COPYS-Copy sign from one double precision number to another.
;
; Entry point:  dp_copys:  c(r0+sign) ← c(r1+sign)
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;          r1 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.L
;
dp_copys move     x:(r0+ms),d0.l       ;get ms
         tst      d0                   ;test for zero
         jne      notzero              ;if not zero, copy the sign
         move     x:(r0+ls),d0.l       ;get ls
         tst      d0                   ;test for zero
         jne      notzero              ;if not zero, copy the sign
         rts
notzero  move     x:(r1+sign),d0.l     ;get sources sign
```

```
        move    d0.l,x:(r0+sign)     ;apply to destination
        rts
        page
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_DIV - Divide two double precision numbers.
;
; Entry point:  dp_div:  c(r0) ← c(r0)/c(r1)
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;          r1 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; NOTE:  Error checking:
;            0/0 returns 0
;            finite/0 returns saturated value
;
; Alters:  D0.L,D1.L,D4.L,D5.L,D6.L,D7.L
;
dp_div  move    x:(r1),d0.l          ;get divisor exponent
        tst     d0       x:(r1),d1.l ;test, get ms
        jne     _notdiv0             ;non-zero
        tst     d1                   ;test
        jne     _notdiv0
        move    x:(r0+sign),d0.l     ;get sign
        move    x:(r1+sign),d1.l     ;get sign
        eor     d0,d1    x:(r0),d0.l ;new sign, get dividend exp
        move    d1.l,x:(r0+sign)     ;save new sign
        tst     d0       x:(r0+ms),d1.l    ;test, get ms
        jne     oflow                ;finite/0 => overflow
        tst     d1                   ;test
        jne     oflow                ;finite/0 => overflow
        jmp     uflow                ;0/0 => zero
_notdiv0 move   x:(r0),d1.l          ;get exponent a
        move    x:(r1),d0.l          ;get exponent x
        sub     d0,d1    #bias,d0.l  ;new exponent, get bias
        add     d0,d1                ;Add bias back
        move    x:(r0+ms),d7.l       ;a ms
        move    x:(r0+ls),d6.l       ;a ls
        move    x:(r1+ms),d5.l       ;x ms
        move    x:(r1+ls),d4.l       ;x ls
        cmp     d5,d7                ;compare magnitudes
        jhi     _startdiv            ;dividend>divisor
        jlo     _adj                 ;adjust if <
        cmp     d4,d6                ;extend comparison
        jhs     _startdiv            ;dividend >= divisor
_adj    lsr     d5                   ;scale down divisor
        ror     d4
```

```
         dec      d1                     ;and adjust exponent
_startdiv
         move     d1.l,x:(r0)            ;save new exponent
;
;      unsigned fractional divide: d7:d6 / d5:d4 = d3:d2
;
         do       #64,_divloop
         cmp      d5,d7                   ;compare ms word
         jhi      _big                    ;dividend > divisor
         jlo      _small                  ;dividend < divisor
         cmp      d4,d6                   ;compare ls word
         jhs      _big                    ;dividend >= divisor
_small   andi     #$fe,ccr                ;set 0 q bit
         jmp      _q
_big     sub      d4,d6                   ;adjust remainder
         subc     d5,d7
         ori      #$01,ccr                ;set 1 q bit
_q       rol      d2                      ;move in q bit
         rol      d3
         lsl      d6                      ;adjust remainder
         rol      d7
_divloop move     d3.l,x:(r0+ms)          ;save ms
         move     d2.l,x:(r0+ls)          ;save ls
         move     x:(r0+sign),d0.l        ;get sign
         move     x:(r1+sign),d1.l        ;get sign
         eor      d1,d0                   ;new sign
         move     d0.l,x:(r0+sign)        ;save sign
         jmp      echeck                  ;check for errors
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_INT - Truncate a double precision number to an integer.
;
; Entry point:  dp_int:  c(r0) ← truncate to integer ← c(r0)
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D2.L,D3.L,D4.L,D7.L,D0.H
;
dp_int   move     x:(r0),d4.l            ;get exponent
         move     #bias,d7.l             ;get bias
         sub      d7,d4    #64,d2.l      ;calculate how far to shift
         jmi      dp_clr                 ;if a fraction, zero the number
         cmp      d2,d4    #32,d2.l      ;is A > 2**63
         jpl      aequala                ;yes, out of range
         cmp      d2,d4    #$80000000,d3.l    ;is A > 2**31
         jpl      rndls                  ;yes, last valid digit is in ls
         clr      d1.l     d4.l,d0.h     ;no, put # shifts in .h register
         asr      d0,d3    x:(r0+ms),d0.l  ;zero ls, create trunc. mask
```

```
         and      d3,d0    d1.l,x:(r0+ls)  ;truncate to an integer
         move     d0.l,x:(r0+ms)       ;store the result
         rts
rndls    sub      d2,d4    x:(r0+ls),d1.l  ;calculate # shifts, get ls
         move     d4.l,d0.h            ;put # shifts in .h register
         asr      d0,d3                ;create the truncation mask
         and      d3,d1                ;truncate to an integer
         move     d1.l,x:(r0+ls)       ;store the result
         rts
         page
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_MAC - Multiply two double precision numbers and
;          accumulate the sum.
;
; Entry point:  dp_mac: c(r0) ← c(r0) + c(r1) * c(r2)
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;          r1 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.L,D1.L,D2.L,D3.L,D4.L,D5.L,D6.L,D7.L,D8.L,D9.L
;
dp_mac   move     r0,d8.l              ;store the r0 pointer
         move     #dptemp,r0           ;get temporary pointer
         jsr      dp_mpy               ;multiply (r1)*(r2)
         move     r1,d9.l              ;store the r1 pointer
         move     #dptemp,r1           ;point to result
         move     d8.l,r0              ;restore the r0 pointer
         jsr      dp_add               ;accumulate the result
         move     d9.l,r1              ;restore the r1 pointer
         rts
         page
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_MOVE - Copy floating-point value from one address to another
;
; Entry point:  dp_move:  c(r0) ← c(r1)
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;          r1 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
```

```
dp_move  move     x:(r1),d0.l       ;move exponent
         move     d0.l,x:(r0)
         move     x:(r1+sign),d0.l ;move sign
         move     d0.l,x:(r0+sign)
         move     x:(r1+ms),d0.l   ;move ms
         move     d0.l,x:(r0+ms)
         move     x:(r1+ls),d0.l   ;move ls
         move     d0.l,x:(r0+ls)
         rts
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_MPY - Multiply two double precision numbers.
;
; Entry point:  dp_mpy:  c(r0) <- c(r1) * c(r2)
;
;                c d
;             *  a b
;                d b
;              c b
;              d a
;            c a
;            x y
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;          r1 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.L,D1.L,D2.L,D3.L,D4.L,D5.L,D6.L,D7.L,D0.M
;
dp_mpy   clr      d4         x:(r1+ms),d2.l      ;get c
         move                x:(r2+ms),d3.l      ;get a
         mpyu     d2,d3,d0   x:(r2+ls),d5.l      ;c*a, get b
         mpyu     d5,d2,d2   d0.m,d1.l           ;c*b, move high
         move     d2.m,d2.l
         add      d2,d0      x:(r1+ls),d2.l      ;add to low, get d
         addc     d4,d1      x:(r2),d5.l         ;get exponent
         mpyu     d2,d3,d2   x:(r2+sign),d7.l    ;d*a
         move     d2.m,d2.l
         add      d2,d0      x:(r1),d6.l         ;add to low
         addc     d4,d1      #bias,d4.l          ;get bias
         jeq      uflow                          ;if *0, set 0 result
         inc      d5         ifmi                ;if normed
         jmi      _nonorm
         lsl      d0                             ;if not normed
         rol      d1                             ;if not normed
_nonorm  add      d5,d6      d1.l,x:(r0+ms)      ;do exponents
         sub      d4,d6      d0.l,x:(r0+ls)      ;1 bias
         move     x:(r1+sign),d1.l
```

```
         eor      d7,d1      d6.l,x:(r0)         ;new sign, save exp
         move                d1.l,x:(r0+sign)    ;save sign
         jmp      echeck                         ;go check for errors
         page
;
; Check for overflow and underflow and saturate or flush to zero
;
echeck   move     x:(r0),d0.l      ;get exponent
         jset     #31,d0.l,uflow   ;bit 31 indicates underflow
         jset     #30,d0.l,oflow   ;bit 30 indicates overflow
         rts                       ;no errors
oflow    move     #$3fffffff,d0.l  ;max exponent
         move     d0.l,x:(r0)
         move     #$ffffffff,d0.l  ;max significand
         move     d0.l,x:(r0+ms)
         move     d0.l,x:(r0+ls)
         rts
uflow    clr      d0               ;min exponent and significand
         move     d0.l,x:(r0)
         move     d0.l,x:(r0+sign)
         move     d0.l,x:(r0+ms)
         move     d0.l,x:(r0+ls)
         rts
         page
;
```

; MOTOROLA DSP96002 DPLIB - VERSION 1.0

;

; DP_NEG - Negate the double precision number pointed to by r0.

```
;
; Entry point:  dp_neg:  c(r0) = -c(r0)
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.L,D1.L
;
dp_neg   move     x:(r0+ms),d0.l   ;get mantissa ms
         move     x:(r0+ls),d1.l   ;get mantissa ls
         or       d1,d0            ;check to see if zero
         jeq      negzero          ;can't have negative zero
         move     x:(r0+sign),d0.l ;get sign
         neg      d0               ;negate
         inc      d0               ;correct
         move     d0.l,x:(r0+sign) ;save sign
negzero  rts                       ;and return
;
```

; MOTOROLA DSP96002 DPLIB - VERSION 1.0

;

; DP_SCALE: scale the double precision number

```
;
```

```
; Entry point:  dp_scale:  c(r0) ← c(r0) * 2**r1
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;          r1 contains an integer number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.L,D1.L
;
; NOTE:  r1 contains an integer. (It does NOT point to an address.)
;
dp_scale move     r1,d0.l          ;put scale factor in data register
         move     x:(r0),d1.l      ;get exponent
         add      d0,d1      #$3fffffff,d0.l     ;scale the number
         jvc      scle             ;scale if no overflow
         move     d0.l,x:(r0)      ;  if overflow,
         move     #$ffffffff,d0.l  ;  set the result
         move     d0.l,x:(r0+ms)   ;  to the maximum
         move     d0.l,x:(r0+ls)   ;  number achievable
         rts
scle     move     d1.l,x:(r0)      ;save scaled exponent
         rts
         page
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_SQRT - Find the square root of a double precision number.
;
; Entry point:  dp_sqrt:  c(r0) ← sqrt(c(r0))
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Outputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;
; Alters:  D0.L,D1.L,D2.L,D3.L,D4.L,D5.L,D6.L,D7.L
;
dp_sqrt  move     x:(r0+ms),d0.l   ;get most significant
         tst      d0         x:(r0),d1.l         ;check, get exponent
         jne      _ok              ;not zero
         tst      d1               ;check ls
         jne      _ok              ;not zero
         rts                       ;if already 0, return
_ok
         move      x:(r0+sign),d0.l              ;get sign
         tst      d0               ;check for negative
         jne      uflow            ;return 0
         move     #bias,d2.l       ;get bias
         sub      d2,d1      #1,d0.l            ;unbias exponent
         lsr      d1               ;square root of exponent
```

```
        inc      d0         ifcs  ;if odd exponent, use 2 bits
        add      d2,d1            ;restore exponent bias
        move     d1.l,x:(r0)      ;store it
        move     x:(r0+ms),d7.l   ;get ms
        move     x:(r0+ls),d6.l   ;get ls
        clr      d4         #0,d5.l            ;clear RR
        clr      d3         #0,d2.l            ;clear DR
        do       d0.l,_initshift  ;initial shift
        lsl      d6               ;shift 2 bits from d7:d6 (SQR)
        rol      d7
        rol      d4               ;to d5:d4 (RR)
        rol      d5
_initshift
        do       #62,_sqrt        ;take root of SQR into DR
        lsl      d2         d4.l,d0.l          ;(dr<<2)|1, copy rr
        rol      d3         d5.l,d1.l
        lsl      d2
        rol      d3
        inc      d2               ;set lsb
        sub      d2,d0            ;temp=rr-(dr<<2)|1
        subc     d3,d1
        jcs      _ofl             ;overflow
        lsr      d3         d0.l,d4.l     ;shift dr back only 1 bit
        ror      d2         d1.l,d5.l     ;rr=temp
        inc      d2               ;root bit=1
        jmp      _next
_ofl    lsr      d3               ;shift dr back only 1 bit
        ror      d2               ;root bit=0
_next   lsl      d6               ;shift 2 bits from d7:d6 (SQR)
        rol      d7
        rol      d4               ;to d5:d4 (RR)
        rol      d5
        lsl      d6               ;shift 2 bits from d7:d6 (SQR)
        rol      d7
        rol      d4               ;to d5:d4 (RR)
        rol      d5
_sqrt   lsl      d2               ;adjust to msb
        rol      d3
        lsl      d2               ;adjust to msb
        rol      d3
        move     d3.l,x:(r0+ms)   ;save ms part
        move     d2.l,x:(r0+ls)   ;save ls part
        rts
;
```

; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_SUB - Double precision subtraction.
;

```
; Entry point:  dp_sub:  c(r0) ← c(r0) - c(r1)
;
; Inputs:  r0 contains the lowest address of a 4-word internal
;             extended precision number
;          r1 contains the lowest address of a 4-word internal
```

```
;              extended precision number
;
; Outputs:  r0 contains the lowest address of the 4-word internal
;              extended precision number with the result
;
; Alters:  D0.L,D1.L,D2.L,D3.L,D4.L,D5.L,D6.L,D7.L,D0.H,D1.H
;
dp_sub   jsr      dp_neg           ;negate the operand
         jsr      dp_add           ;add the numbers
         jmp      dp_neg           ;negate the operand
;
; MOTOROLA DSP96002 DPLIB - VERSION 1.0
;
; DP_TST - Test a double precision operand.  (The same as "TST.")
;
; Entry point:  dp_tst:  c(r0) - 0  (Set the flags)
;
; Inputs:  r0 contains the lowest address of the 4-word internal
;              extended precision number
;
; Outputs:  none
;
; CCR CONDITION CODES:
;
;       C - NOT AFFECTED.
;       V - ALWAYS CLEARED
;       Z - SET IF RESULT IS ZERO, CLEARED OTHERWISE.
;       N - SET IF RESULT IS NEGATIVE, CLEARED OTHERWISE.
;       I - NOT AFFECTED.
;      LR - NOT AFFECTED.
;       R - NOT AFFECTED.
;       A - NOT AFFECTED.
;
;       The following Jcc branch conditions can be used after
;       calling dp_tst.  The other branch conditions should not
;       be used.
;
;       "cc" Mnemonic                  Condition
;       EQ - equal                     Z = 1
;       GE - greater than or equal     N eor V = 0
;       GT - greater than              Z + (N eor V) = 0
;       LE - less than or equal        Z + (N eor V) = 1
;       LT - less than                 N eor V = 1
;       NE - not equal                 Z = 0
;
; Alters:  D0.L,D1.L,D2.L
;
dp_tst   move     x:(r0+ms),d0.l   ;get ms
         tst      d0         x:(r0+sign),d1.l    ;test ms = 0, get sign
         jeq      mszero           ;if zero, check if ls = 0
sgntst   tst      d1         #-1,d0.l             ;test the sign
         move     #2,d1.l          ;get offset for negative sign
         add      d1,d0      ifeq  ;make d0 same sign as (r0)
```

```
         tst      d0              ;set the correct flags
         rts
mszero   move     x:(r0+ls),d0.l  ;get ls
         tst      d0         #0,d2.l             ;check if ls = 0
         jne      sgntst          ;if not, check sign
         tst      d2              ;set the correct flags
         rts
;
; END OF DPLIB
;
         end
```

; **Double precision FIR example**

```
;
;  "data" and "coef" are assumed to be in DPLIB format.
;  Other variables are assumed to be in IEEE DP format.
;
         org     x:0
ntaps    equ     8
data     ds      4*ntaps
coef     ds      4*ntaps
p        ds      1
a        ds      1
ieee_in  ds      1
ieee_out ds      1
;
         org     p:$100
start
         move    #data,r2        ;point to data
         move    #coef,r3        ;point to coefficients
         move    #p,r4           ;temp product
         move    #a,r5           ;product accumulator
         move    #4,n2           ;dp size
         move    n2,n3
         move    #4*ntaps-1,m2   ;mod buffer size
         move    m2,m3
_loop
         move    l:ieee_in,d0.d  ;get ieee number
         move    r2,r0           ;point r0 to data buffer
         jsr     ieee2dplib      ;convert register to dp and save
         do      #ntaps,_dpfir
         move    r4,r0           ;point to product variable
         move    r3,r1           ;point to coefficients
         jsr     dp_mpy          ;multiply, result in p
         move    r5,r0           ;point to accumulations
         move    r4,r1           ;point to product variable
         jsr     dp_add          ;add them together
         move    (r2)+n2         ;shift to next dp data value
         move    (r3)+n3         ;move to next dp coefficient
_dpfir
         move    r5,r0           ;point to result
         jsr     dplib2ieee      ;convert to a value in d0
```

```
         move    d0.d,l:ieee_out ;output as dp ieee number
         move    (r2)-n2         ;delete last sample
         jmp     _loop
```

; **NxN by NxN Matrix Multiplication Example**

```
;
;                        Multiply Two Matrices:  AB = C
;
; ***NOTE:  All numbers are assumed to be in DPLIB format.
;
         org     x:0
order    equ     3              ;Nth order system
elements equ     order*order
a        ds      4*elements     ;matrix A stored starting at x:$00
         org     x:64
b        ds      4*elements     ;matrix B stored starting at x:$40
         org     x:128
c        ds      4*elements     ;matrix C stored starting at x:$80
;
         org     p:$100
start    move    #a,r2          ;r2 points to matrix A elements
         move    #b,r1          ;r1 points to matrix B elements
         move    #c,r0          ;r0 points to matrix C (the result)
         move    #4,n0          ;offset for 4 word numbers
         move    #order*4,n1    ;offset for one row
         move    #4,n2          ;offset for 4 word numbers
         move    #(elements-order+1)*4,n3
         move    #(order+1)*4,n4
         move    n1,n5
         move    #(elements*4)-1,m0
         move    #(elements*4)-1,m1
         move    #(elements*4)-1,m2
;
3x3mult  do      #order,rows    ;calculate each row of the result
         do      #order,columns ;calculate each column of the result
         jsr     dp_mpy         ;multiply the first row-column elements
         move    (r1)+n1        ;update B offset for next column element
         move    (r2)+n2        ;update A offset for next row element
         jsr     dp_mac         ;accumulate the inner products
         move    (r1)+n1        ;update B offset for next column element
         move    (r2)+n2        ;update A offset for next row element
         jsr     dp_mac         ;accumulate the inner products
         move    n3,n2          ;update A offset to return to column 1
         move    n4,n1          ;update B offset for next column
         move    (r0)+n0        ;update result matrix pointer
         move    (r1)+n1        ;point to a row 1 element
         move    (r2)+n2        ;point to a column 1 element
         move    n0,n2          ;restore A offset
         move    n5,n1          ;restore B offset
columns  move    n1,n2          ;update A offset for row shift
         move    #b,r1          ;point to B column 1
         move    (r2)+n2        ;point to the next A row
         move    n0,n2          ;restore A offset
rows     stop                   ;resultant matrix finished
         include "dplib"
         end
```

B.6 STANDARD BENCHMARK SUMMARY

Benchmark	56000/1 Word	56000/1 Icyc	DSP96000 Word	DSP96000 Icyc
B.1.1 Real Multiply	3	3	3	3
B.1.2 N Real Multiplies	8	2N+7	8	2N+7
B.1.3 Real Update	4	4	4	4
B.1.4 N Real Updates	10	2N+9	10	2N+9
B.1.5 N Term Real Convolution (FIR)	10	1N+12	10	1N+12
B.1.6 N Term Real*Complex Convolution	10	2N+9	9	2N+8
B.1.7 Complex Multiply	6	6	7	7
B.1.8 N Complex Multiplies	12	4N+9	12	4N+9
B.1.9 Complex Update	7	7	8	8
B.1.10 N Complex Updates	13	4N+10	15	4N+12
	13	5N+9	17	4N+14
B.1.11 N Term Complex Convolution (FIR)	11	4N+8	11	4N+8
B.1.12 Nth Order Power Series	9	2N+8	9	2N+8
B.1.13 2nd Order Real Biquad Filter	7	7	7	7
B.1.14 N Cascaded 2nd Order Biquads	17	4N+16	19	4N+18
B.1.15 Radix 2 FFT Butterfly	6	6N	4	4N
B.1.16 Adaptive True LMS Filter				3N
Adaptive Delayed LMS Filter				2N
B.1.17 FIR Lattice Filter	7	3N+5	7	3N+5
B.1.18 All Pole IIR Lattice Filter	9	3N+4	12	3N+7
B.1.19 General Lattice Filter	12	4N+10	14	4N+12
B.1.20 Normalized Lattice Filter	13	5N+10	14	5N+11
B.1.21 [1x3 [3x3 Matrix Multiply			12	12
[1x4 [4x4 Matrix Multiply			19	19
B.1.22 [NxN [NxN Matrix Multiply	19	N^3+7N^2+5N+8	19	N^3+7N^2+6N+7
B.1.23 N Point 3x3 2-Dimensional FIR	28	$10N^2+7N+12$	29	$10N^2+8N+13$
B.1.24 Table Lookup with Interpolation			12	12
B.1.25 Argument Reduction			6	6
B.1.26 Non-IEEE Floating-Point Division				
No Error Checking			7	7
With Divide By Zero Checking			9	9
With Divide By Infinity Checking			8	8
With Divide By Zero And Infinity Checking			10	10
B.1.27 Multibit Rotates				
With Carry, Static			4	4
With Carry, Dynamic			9	9
Without Carry, Static			4	4
Without Carry, Dynamic			6	6

Figure B-1. Standard Benchmark Summary

Benchmark	DSP96000 Word	DSP96000 Icyc
B.1.28 Bit Field Extraction/Insertion		
Static Field Extraction, Zero Extend	2	2
Static Field Extraction, Sign Extend	2	2
Dynamic Field Extraction, Zero Extend	6	6
Dynamic Field Extraction, Sign Extend	6	6
Static Field Insertion	6	6
Dynamic Field Insertion	9	9
Static Field Clear	4	4
Static Field Set	4	4
Dynamic Field Clear	7	7
Dynamic Field Set	7	7
B.1.29 Newton-Raphson Approximation of 1.0/SQRT(x)	11	11
B.1.30 Newton-Raphson Approximation of SQRT(x)	12	12
B.1.31 Unsigned 32 Bit Integer Division/Remainder	8	133
Unsigned 32 Bit Integer Division	16	3N+14
Unsigned 32 Bit Integer Remainder	14	3N+12
B.1.32 Signed 32 Bit Integer Division/Remainder	13	138
Signed 32 Bit Integer Division	21	3N+19
Signed 32 Bit Integer Remainder	17	3N+15
B.1.33 Trivial Accept/Reject In Three Dimensions		
Single Point	8	8
Polyline (Fixed Point)	14	14
Polyline (floating-point)	14	14
Four Point Polygon (in-line)	26	26
Four Point Polygon (looped)	12	29
B.1.34 Cascaded Five Coefficient Transpose IIR Filter	10	5N+6
B.1.35 3-D Graphics Illumination	20	21
B.1.36 Pseudorandom Number Generation	8	8
B.1.37 Bezier Cubic Polynomial Evaluation	13	13
B.1.39 Nth Order Polynomial Evaluation for Two Points	12	
B.1.38 Byte/16 Bit Packing/Unpacking From/To 32 Bits		
Four 8 Bit Byte Packs Into 32 Bits	3	3
Two 16 Bit Word Packs Into 32 Bits	1	1
Four 8 Bit Byte Unpacks From 32 Bits	5	5
Two 16 Bit Word Unpacks From 32 Bits	2	2
B.1.40 Graphics BITBLT (Bit Block Transfer)		
32 Bits/Iteration	24	4N+20
64 Bits/Iteration	32	6N+27
B.1.41 64x64 Bit Unsigned Integer Multiply	11	11
B.1.42 Signed Reciprocal Approximation		
16 Bit	5	5
32 Bit	7	7
B.1.43 Incremental Line Drawing		
floating-point		3N/pt
Fixed Point (Bresenham’s algorithm)		4N/pt
B.1.44 Three Dimensional Wire-Frame Rendition		
B.1.45 Walsh-Hadamard Transforms		
B.1.46 Evaluation of LOG2(x)	10	27

Figure B-1. Standard Benchmark Summary (continued)

Benchmark	DSP96000 Word	Icyc
B.1.47 Evaluation of EXP2(x)	10	27
B.1.487 Vector Cross Product	10	10
B.1.49 Power Function X**Y, X = Single Precision FP		
Y = 5 Bit Integer, Straight	14	14
Y = 32 Bit Unsigned Integer, Looped	7	100
Y = 32 Bit Unsigned Integer, Variable Loop	10	3N+8
Y = Single Precision FP	21	55
B.1.50 Cascaded Five Coefficient Biquad Filter	8	5N+4
B.1.51 CORDIC Sine (4 Quadrant/Argument Reduction)	33	8N+26
B.1.52 CORDIC Cosine (4 Quadrant/Argument Reduction)	34	8N+27
B.1.53 CORDIC Tangent (4 Quadrant/Argument Reduction)	44	8N+37
B.1.54 [NxN by [NxN Matrix Multiplication		
(Modulo-Aligned)	21	n^3+4n^2 +5n+16
B.1.55 [4x4 by [4x4 Matrix Multiplication		
(Modulo-Aligned)	30	87
B.1.56 [8x8 by [8x8 Matrix Multiplication		
(Modulo-Aligned)	86	607
B.1.57 [16x16 by [16x16 Matrix Multiplication		
(Modulo-Aligned)	286	4399
B.1.58 Double Integrator Oscillator	3	3
Second Order Oscillator	2	2
B.1.59DTMF Signal Generator	5	5
IEEE Standard Conformance Benchmarks		
B.2.1 IEEE Floating-point Remainder		
B.2.2 IEEE Floating-point Round to Integer	1	1
B.2.3 IEEE Floating-point to Decimal String		
B.2.4 IEEE Decimal String to F.P.		
B.2.5 Format Conversions		
Signed 32 Bit Integer to:		
Unsigned 32 Bit Integer	2	3
Single Precision floating-point	1	1
Unsigned 32 Bit Integer to:		
Signed 32 Bit Integer	2	3
Single Precision floating-point	1	1
Single Precision to:		
Signed 32 Bit Integer	3	4
Unsigned 32 Bit Integer	3	4

Figure B-1. Standard Benchmark Summary (continued)

IEEE Recommended Functions and Predicates Benchmark		DSP96000 Word	Icyc
B.3.1	Copysign(x,y)		
	Arithmetic	1	1
	Non-arithmetic	6	7
B.3.2	-x		
	Arithmetic	1	1
	Non-arithmetic	1	2
B.3.3	Scalb(y,N)	1	1
B.3.4	Logb(x)	18	
	x = NaN		4
	x = Infinity		7
	x = Zero		16
	x = In-range		15
B.3.5	Nextafter(x,y)	32	
	Either operand a NaN		9
	X is signed infinity		7
	Result is normalized		26
	Result is denormalized		24
	Result overflowed		26
B.3.6	Finite(x)	3	3
B.3.7	Isnan(x)	3	3
B.3.8	x<>y	3	3
B.3.9	Unordered(x,y)	3	3
B.3.10	Class(x)	38	
	Signaling not a number		7
	Quiet not a number		10
	Negative infinity		15
	Negative normalized nonzero		21
	Negative denormalized		20
	Negative zero		15
	Positive zero		19
	Positive denormalized		23
	Positive normalized nonzero		26
	Positive infinity		25

Figure B-1. Standard Benchmark Summary (continued)

IEEE Double Precision Using Software Emulation	TYPICAL	WORST CASE	FULLY TESTED
B.4.1 ADDITION	6.86 us	29.1 us	YES
B.4.2 SUBTRACTION	7.01 us	29.2 us	YES
B.4.3 MULTIPLICATION	13.58 us	39.5 us	YES

Non-IEEE Double Precision Using Software Emulation	Instruction Cycles		
	TYPICAL	WORST CASE	BEST CASE
B.5.1 CONVERT NUMBERS:			
-TO DPLIB FORMAT (IEEE2DPLIB)	23	33	6
-TO IEEE FORMAT (DPLIB2IEEE)	20	23	16
B.5.2 ABSOLUTE VALUE (DP_ABS)	5	5	5
B.5.3 ADDITION (DP_ADD)	69	86	22
B.5.4 CLEAR (DP_CLR)	9	9	9
B.5.5 COMPARE (DP_CMP)	22	30	14
B.5.6 COPY SIGN (DP_COPYS)	13	16	11
B.5.7 DIVISION (DP_DIV)	852	1020	32
B.5.8 ROUND TO AN INTEGER (DP_INT)	16	20	10
B.5.9 MAC (DP_MAC)	109	149	61
B.5.10 COPY A NUMBER (DP_MOVE)	15	15	15
B.5.11 MULTIPLICATION (DP_MPY)	109	51	27
B.5.12 NEGATE (DP_NEG)	14	14	14
B.5.13 SCALE (DP_SCALE)	11	13	8
B.5.14 SQUARE ROOT (DP_SQRT)	1317	1413	9
B.5.15 SUBTRACTION (DP_SUB)	103	120	56
B.5.16 TEST A NUMBER (DP_TST)	13	14	12

Note: typical execution times are the average of all possible paths through the algorithm.

Figure B-1. Standard Benchmark Summary (continued)

APPENDIX C
IEEE ARITHMETIC

C.1 FLOATING-POINT NUMBER STORAGE AND ARITHMETIC

C.1.1 General

The IEEE standard for binary floating point arithmetic provides for the compatibility of floating-point numbers across all implementations which use the standard by defining bit-level encoding of floating-point numbers. Maximum mathematical accuracy, with respect to roundoff errors, is achieved by optimally scaling floating-point numbers by using a normalized exponential notation. Error bounds are guaranteed by the standard for the basic mathematical operations (add, subtract, multiply, divide, square root, round to nearest integer, conversion to and from integers and conversion to and from decimal strings). The standard also defines error handling for five floating point exceptions: invalid operation, divide by zero, overflow, underflow and inexact result.

The standard defines two data storage formats which are identical across implementations (basic formats): Single Precision (SP) and Double Precision (DP). It also specifies the use of two implementation-dependent encodings (extended formats): Single Extended Precision (SEP) and Double Extended Precision (DEP), on which it only places some general constraints, and for which bit-level encodings are not defined. The extended formats are consequently implementation-dependent and should never be used for representation of numbers which are to be shared across different processors (i. e., stored).

Each format provides representation of the following elements:

1. **Floating-point numbers** of the form:

$$X = (-1)^{S} 2^{E} (b_0 \bullet b_1 b_2 \cdots_{p-1})$$

where:

$s = 0$ or 1

E = an integer between E_{min} and E_{max} , inclusive.

$b_i = 0$ or 1

2. **Infinities:** $+\infty$ and $-\infty$
3. **" Not-a-Numbers (NaNs) "**. NaNs are special symbolic elements, encoded in the floating point format. They can appear as operands and/or as results of arithmetic operations. The standard provides two types of NaNs:
4. **Quiet NaNs (QNaNs):** are encodings of information regarding meaningless or invalid results.

Examples of QNaNs are results of operations such as 0/0, ∞–∞, ∞/∞, etc. Encodings of QNaNs are intended to provide some kind of retrospective diagnostic information concerning the origin of the NaN. Since this information needs to remain available even after a large number of arithmetic operations, QNaNs "propagate" unchanged through arithmetic operations and format conversions. QNaNs can thus occur as operands of an arithmetic operation. If one or more QNaN occur as operands, the result is a quiet NaN, and no floating point exception is signaled. Hence the name "quiet" NaN. The standard specifies that at least one QNaN must be supported.

5. **Signaling NaNs (SNaNs):** Signaling NaNs are used only in systems with arithmetic-like enhancements that are not defined by the standard. As opposed to QNaNs, they are never generated by the DSP96002 arithmetic. They can, however, appear as operands in arithmetic operations (as generated by other processors, for instance). In this case, they always signal the "Invalid Operation" floating point exception. The returned result is a QNaN.

Floating point operands in the DSP96002 are either 32 bits long (Single Real), 64 bits long (Double Real) or 96 bits long (Register operand). The operand size is either explicitly encoded in the instruction or implicitly defined by the instruction operation. The following sections describe the details of each operand type.

C.1.2 DSP96002 Floating Point Storage Format in Memory

DP and SP are the only floating point formats for which the IEEE standard provides bit-level definitions. Since the DSP96002 is designed for multiprocessing applications, where data in memory can be shared among different processors, SP and DP are the only formats supported for memory storage of floating point numbers.

SP numbers are represented by 32 bits in memory, and can be located in either X: or Y: data spaces. DP numbers take up 64 bits in memory, and can thus only be stored in long (L:) memory space.

The basic formats (SP and DP) contain three fields in their binary representation, as shown in Figure C-1. These fields are described as:

1. Sign Bit (s): The sign bit denotes the sign of the number, in a signed magnitude notation. When s=0, the number is positive. When s=1, the number is negative. Note that floating-point numbers do not use a two’s complement notation.

2. Exponent Field (e): The exponent of SP and DP numbers is stored as a positive (biased) integer:

   $$e = E + \text{bias}$$

   where E is the actual exponent of the floating point number as explained later in this section. e is also used in conjunction with the fractional field f to encode non-numerical values (infinities and NaNs).

   For SP, the exponent consists of 8 bits (bits 23 through 30) , and the bias equals 127. The biased exponent e can thus take on integer values between 0 (denoted by e_{min}-1) and 255 (denoted by e_{max}+1) inclusive.

   For DP, the exponent consists of 11 bits (bits 52 through 62) , and the bias equals 1023. Values for the biased exponent e thus fall between 0 (e_{min}-1) and 2047 (e_{max}+1), inclusive. Table C-1 summarizes these values for SP and DP.

3. Fraction (f): The fractional field consists of bits b_i:

31	30 23	22 0
S	8-bit biased exponent	23-bit fraction

Single Precision (SP)

63	62 52	51 0
S	11-bit biased exponent	52-bit fraction

Double Precision (DP)

Figure C-1. SP and DP IEEE Formats

	p-1	bias	e_{min}	e_{max}	E_{min}	E_{max}
SP	23	127	+1	+ 254	- 126	+ 127
DP	52	1023	+1	+2046	-1022	+1023

Table C-1. Parameters for Numerical Formats

$$f = \bullet b_1 b_2 \bullet\bullet\bullet b_{p-1}$$

There are 23 fractional bits (p=24) (bits 0 through 22) in the SP format, and 52 fractional bits (p=53) (bits 0 through 51) in the DP format. Note that bit b_0 is not explicitly represented.

The sign bit, exponent, and fraction fields encode the numerical values of floating-point numbers, as well as $\pm$ 0, $\pm\infty$, and NaNs as follows:

1. Normalized Numerical Values ($E_{min} \le E \le E_{max}$): For numerical values, the biased exponent e lies between e_{min} and e_{max}, inclusive. Equivalently, the exponent E takes on values between E_{min} and E_{max} inclusive. Table C-1 summarizes these values for SP and DP. If the biased exponent e is equal to or greater than e_{min} (E is greater than E_{min}), the number in question is called normalized (i.e. the implicit integer value b0 is equal to one). Note that this integer value, b_0, is not stored in memory. Normalized numbers x are equal in value to:

$$x = (-1)^s \bullet 2^{e - bias}\ 1.f$$

where 1.f is a binary, fixed point number, i.e.:

$$1.f = 1+(o.5) \bullet b_1 + (0.25) \bullet b_2 + ... + (-\tfrac{1}{2})^{p-1} \bullet b_{p-1}$$

Therefore, the smallest magnitude of any normalized number, $X_{min,\,n}$, is equal to ($e=e_{min}$, f=0):

$$x_{min,n} = 1 \bullet 2^{e_{min} - bias} = 1 \bullet 2^{E_{min}}$$

Using the value from Table C-1, this equals approximately $1.18 \bullet 10^{-38}$ for SP numbers.

The largest normalized numerical value that can be represented equals (all b_i=1, $e=e_{max}$):

$$x_{max,n} = (2 - 0.5^{p-1})\ 2^{e_{max} - bias} = (2 - 0.25^{p-1})\ 2^{E_{max}}$$

For SP this equals approximately (using the values in Table C-1) $3.4 \bullet 10^{38}$.

2. Denormalized Numerical Values ($e = e_{min}-1, f \neq 0$): When the exponent e equals the value e_{min}-1 and the fraction field is non-zero the floating point number is called denormalized, and the implicit integer bit b0 is equal to zero. The numerical value of a denormalized number y is given by:

$$y = (-1)^s \bullet 0.f \bullet 2^{e_{min}-bias} = (-1)^s \bullet 0.f \bullet 2^{E_{min}}$$

The denormalization of the fractional part allows the representation of very small numbers near the underflow threshold. The smallest possible magnitude of any denormalized number ($f=f_{min}$) which can be represented equals:

$$y_{min} = (0.5)^{p-1} \bullet 2^{e_{min} - bias}$$

For SP denormalized numbers, this results in a smallest magnitude of approximately $1.4 \bullet 10^{-45}$.

3. Zeros ($e = e_{min}-1, f=0$): Floating point value(s) of zero are encoded by a biased exponent e equal to e_{min}-1, and a fractional field f of all zeros. Note that this encoding retains a significant sign bit: plus and minus zero are two separate entities. Figure C-2 shows the encoding of plus and minus zero in floating point format.
4. Infinities ($e = e_{max} + 1, f = 0$) Infinities are encoded in the floating point format by a biased exponent equal to e_{max}+1, and a fractional field f consisting of all zeros. The sign bit distinguishes between + and -∞. Figure C-3 shows the encodings for + and -∞ in SP and DP.
5. NaNs ($e = e_{max}+1, f \neq 0$): NaNs are encoded in the floating point format by a biased exponent equal to e_{max}+1, and a nonzero fractional field. The value of the sign bit is irrelevant in this encoding.

QNaNs (b_1=1) Quiet NaNs are represented by a fraction with MSB = 1 (and $e=e_{max}+1$). The DSP96002 only fully supports one QNaN, the "legal" QNaN as required by the standard. This QNaN is encoded by a fractional field of all ones (all b_i = 1 in f). Other types of QNaNs (DSP96002 "illegal" NaNs) may occur in multiprocessing situations (as generated by other processors) however, and do deliver well-defined results in the DSP96002. When QNaNs other than the "legal" QNaN occur as operand(s) to floating point arithmetic, the delivered result is always a "legal" QNaN. Figure C-4 shows the encoding for QNaNs.

SNaNs (b_1=0) Signaling NaNs are never generated by the DSP96002 as arithmetic results, but may appear in the DSP96002 memory as passed along by other processors. SNaNs are characterized by a MSB of the fractional field equal to 0 (and $e = e_{max}+1$). When a SNaN appears as an operand of an arithmetic instruction, the invalid operation exception is signaled, and the result is returned as a "legal" QNaN.

The two basic formats, discussed in the previous paragraphs, are the only formats which are used for representation of floating point values in the DSP96002 memory (internal and/or external). The SEP format,

31	30 … 23	22 … 0
S	0	0

Single Precision

63	62 … 52	51 … 0
S	0	0

Double Precision

Figure C-2. Encodings for + and - Zero

31	30 … 23	22 … 0
S	11..................1	0

Single Precision

63	62 … 52	51 … 0
S	11..............................1	0

Double Precision

Figure C-3. Encodings for + and - Infinity

31	30 … 23	22 … 0
X	11..................1	1111....................................1

Single Precision

63	62 … 52	51 … 0
X	11..............................1	11111111..1

Double Precision

Figure C-4. Encodings for QNaNs

generated exclusively by the DSP96002 data ALU as a result of floating point arithmetic operations, is embedded in the DP format, and is thus stored implicitly as a DP number with zeros in the lower 21 bits of the fraction.

C.1.3 DSP96002 Floating Point Storage Format in the Data ALU

The data ALU is designed to accommodate mixed-precision operands in a common format. To this end, a common DP storage format is used internal to the data ALU. SP and DP numbers from memory are automatically converted to the internal format by means of a format conversion unit, the operation of which is transparent to the user.

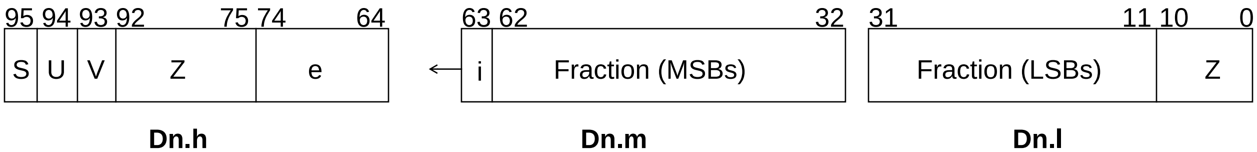
The bit-level DP representation internal to the ALU is illustrated in Figure C-5. The internal floating point format is 96 bits wide and consists of the following fields:

1. Sign of the mantissa (S) bit 95.
2. SP Unnormalized tag (U) bit 94. The U-TAG is set when writing a floating-point register with a denormalized SP number. Cleared otherwise.
3. DP Unnormalized tag (V) bit 93. The V-TAG is set when writing a floating-point register with a denormalized DP number (denormalized SEP in the DSP96002). Cleared otherwise.
4. Unused bits (Z) bits 75 through 92 and bits 0 through 10. These bits read as zeros, and should be written with zeros for future compatibility. They are cleared by floating-point moves and operations.
5. Biased Exponent (e) bits 64 through 74. Since the internal ALU format is DP, there are 11 exponent bits, with an integer bias of 1023 ($3FF). The encodings of the exponent are identical to the ones explained in the section on memory storage formats (Appendix D.1.2) .
6. Integer bit (i or b_0) bit 63. The integer bit is explicitly presented in the internal representation as bit 63 and is the integer part of the mantissa.
7. Fraction – bits 11 through 62. This is a 52-bit field representing the fractional part of the mantissa (only 31 are used by the DSP96002 floating-point ALU). The remaining bits are set to zero by floating-point ALU operations or single-precision floating-point moves. Since the internal format is DP, the fraction consists of 52 bits. The data ALU arithmetic, however, only provides results in either SP or SEP. The SEP format is the same as the DP format, except for the size of the fraction. The SEP fraction consists of only 31 bits. Consequently, the lower 21 or 29 bits of the fraction will consist of zeros when representing SEP or SP arithmetic results, respectively. When DP values are moved from memory to the data ALU, the fraction contains all 52 significant bits. However, when using these DP values as operands in a floating-point arithmetic operation, only 31 bits of the 52-bit fraction are used; the remaining bits are simply truncated. The SEP format is shown in Figure C-7.

C.1.4 IEEE Floating Point Exceptions

The IEEE standard defines five types of exceptions which must be signaled when detected. The DSP96002 implements the default "trap disabled" way of signaling exceptions: when an exception occurs, a flag is set and program execution continues. The flag remains set until cleared by the user. The different exceptions are:

1. Invalid operation: The invalid operation exception is signaled when an operand is invalid for the



S : sign
U : single precision unnormalized tag
V : double precision unnormalized
i : explicit integer

Figure C-5. DP Format in the Data ALU

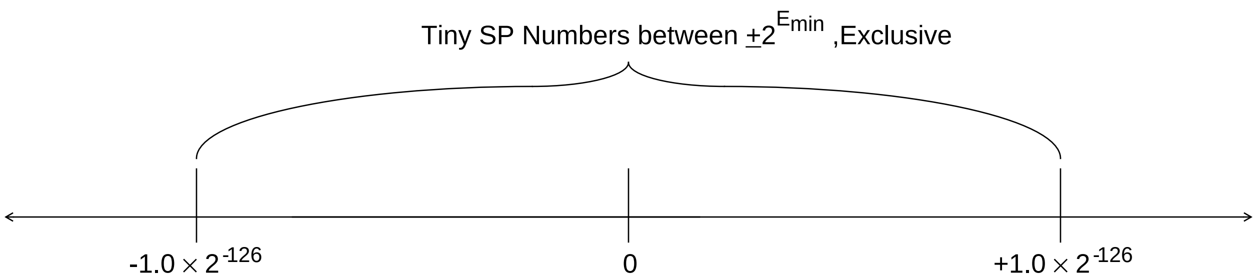


Figure C-6. Tiny Numbers on the Real Number Line

specific operation to occur. The result of an invalid operation is a QNaN, as described above. Examples of invalid operations are 0/0, ∞/∞, ∞–∞, 0×∞, etc.

2. Division by zero: The result of a division by zero is an infinity (with the correct sign), and the operation is signaled as an exception.
3. Overflow: The overflow exception is signaled when the result of an operation exceeds the largest magnitude that the result precision can accommodate. The **result** generated by the hardware **is dependent upon the rounding mode**. For **round to nearest**, an infinity with correct sign is generated. **Round to zero** results in the largest possible numerical value the result precision can accommodate, with correct sign (i. e., the result saturates). **Round to -∞** results in the largest possible numerical value the result precision can accommodate (i. e., the result saturates) when the overflow is positive. It results in -∞ when the overflow is negative. **Round to +∞** results in +∞ when the overflow is positive, and in the largest negative numerical value the result precision can accommodate (i. e., the result saturates) when the overflow is negative.
4. Underflow: Underflow is signaled when both (1) a very small (tiny) number is detected as the

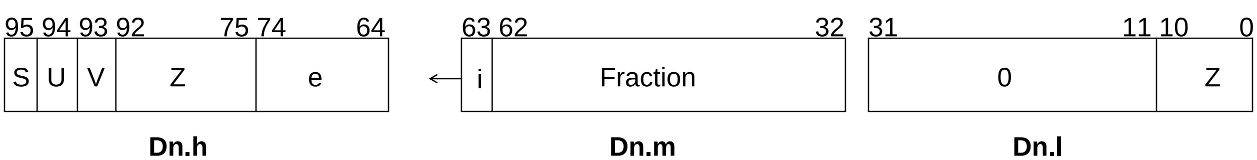


Figure C-7. SEP Format in the Data ALU

result of a floating point operation (nonzero result with true exponent smaller than the minimum exponent, see Figure C-6) and (2) loss of accuracy is detected (delivered result differs from what would have been computed if the exponent range was unbounded – i. e., cannot be accurately represented as a denormalized number due to an insufficient number of bits or round-off errors). Consider the case of floating point multiplication as an example. Let the first SP source operand have a mantissa of 1.01, with biased exponent e_{min}=1 (unbiased exponent of -126) and the second SP source operand have a mantissa of 1.0 with a biased exponent of 60 (unbiased exponent of -67). The result of a multiplication with infinite precision arithmetic would be a mantissa of 1.01 with actual (unbiased) exponent of -193 (=-126-67). Since this exponent is smaller than the smallest exponent possible in SP, the number is tiny, and since the number is so tiny that it cannot be accurately represented as a denormalized number (a mantissa having 68 bits would be required), loss of accuracy also occurs, therefore an underflow will be signaled. The delivered SP result would be a SP zero, and the underflow flag would be set. Note that the SEP format can accommodate this exponent, and thus the result of an SEP operation would not signal the underflow exception. In that case, the correct result is delivered. If the first operand of the SP multiplication has the same value as before, but the second operand has a biased exponent of 104 (actual exponent of -23), the result of an infinite-precision multiplication has a mantissa of 1.01 and an actual exponent of -149. The SP result consists of a denormalized number (i.e., tiny) with a mantissa of 0.00000000000000000000001 and biased exponent of 0. Note that the denormalization process results in loss of accuracy, and therefore the the underflow flag will be set. Finally, if the second source operand has a biased exponent of 120 (actual exponent of -7), then the resulting mantissa with infinite precision would be 1.01 as before, with an actual exponent of -133. The SP result is again denormalized (tiny) with a mantissa of 0.000000101 and a biased exponent equal to 0. Note that there is no loss in accuracy due to the normalization (no lost significant bits), and thus the underflow flag will not be set. The delivered result is the correct SP denormalized number.

5. Inexact: The inexact exception is signaled if the delivered result differs from what would have been obtained with infinite-precision arithmetic. For instance, the examples of underflow shown above deliver numerically inexact results, and thus set the inexact flag. Another example is the case where floating point numbers are rounded up or down.

C.1.5 Data ALU Block Diagram

The block diagram of the data ALU is shown in Figure C-8. The data ALU consists of four main parts:

1. Register file and automatic conversion unit: All operations in the data ALU are register-based – operands as well as results of data ALU operations are read from and written to registers. A register file consisting of ten 96-bit registers are available for storage of floating-point numbers. An automatic conversion unit converts the floating point storage format in memory to the internal DP format when moving operands and/or results from/to memory.
2. Multiply unit: A full IEEE floating-point multiply unit, delivering either a SP or SEP result in one instruction cycle.
3. Adder/Subtracter unit: A full IEEE floating-point adder/subtracter unit, which can deliver the sum as well as the difference of two operands in the same instruction cycle, to either SP or SEP.
4. Special function unit: A special function unit provides various logic functions, as well as support for divide and square root in terms of an initial seed for a fast convergent divide and square root

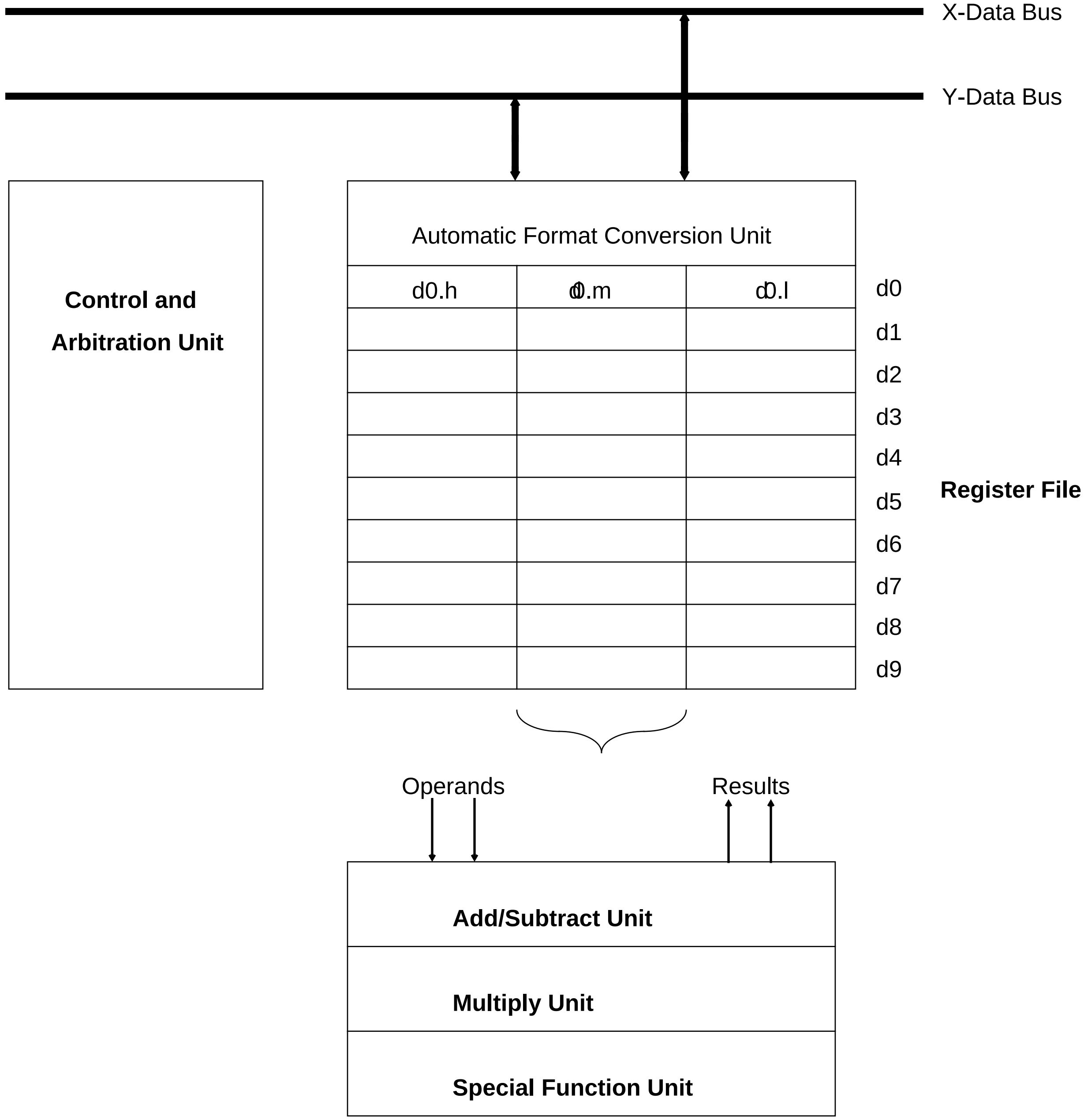


Figure C-8. The Data ALU Block Diagram

Infinite-precision result	Rounded result (to p=4 bits for example)
1.000 11100000....	1.001 (round up)
1.000 01100000....	1.000 (round down)
1.000 10000000....(absolute tie)	1.000 (round down)
1.001 10000000....(absolute tie)	1.010 (round up)

Table C-2. Example of the Round to Nearest (Even) Mode.

algorithm.

5. Controller and arbitrator: A controller/arbitrator supplies all of the control signals necessary for the operation of the data ALU.

The data ALU uses the SEP format for all of its operations: the results are automatically rounded to either SP or SEP. All of the rounding modes specified by the IEEE standard are supported. These rounding modes are:

1. Round to nearest (even): a convergent rounding mode, designed to deliver results without a rounding bias. In this case the infinite-precision result is rounded to the finite-precision result which is closest. In the case of an absolute tie, the infinite-precision result is rounded to the "nearest even" finite precision result, as is illustrated in Table C-2.
2. Round to zero: in this case, the infinite-precision result is rounded to the nearest finite-precision result which is closest to zero. Clearly, results are rounded up in this mode when negative, and down when positive.
3. Round to plus infinity: results are always rounded in the direction of plus infinity, i.e. "up".
4. Round to minus infinity: results are always rounded in the direction of minus infinity, or "down".

C.1.5.1 Register file and automatic format conversion unit

The general-purpose register file consists of ten 96-bit registers named d0..d9, as shown in Figure C-9. Each 96-bit register accommodates the DP internal floating point storage format. Each 96-bit register is obtained by the concatenation of three 32-bit registers dn.h:dn.m:dn.l. The registers dn.h, dn.m, and dn.l can be accessed as individual registers by MOVE operations and integer and logic instructions, as is further described in Appendix C.2 and in Appendix A.

The registers d0..d7 are general-purpose registers in the sense that MOVE instructions and data ALU operations do not differentiate between them. They are used for data ALU source and destination operands for most of the data ALU instructions. They can be used as operands for MOVE operations as well as for data ALU operations in the same instruction cycle: dual source operands are allowed. They can not be used as dual destinations in the same instruction cycle.

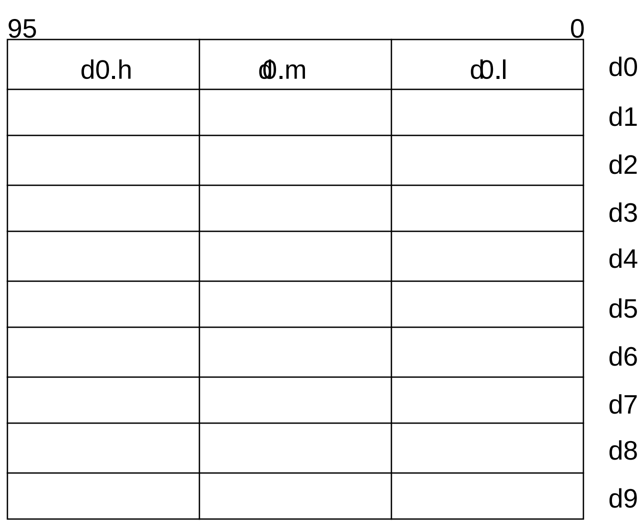
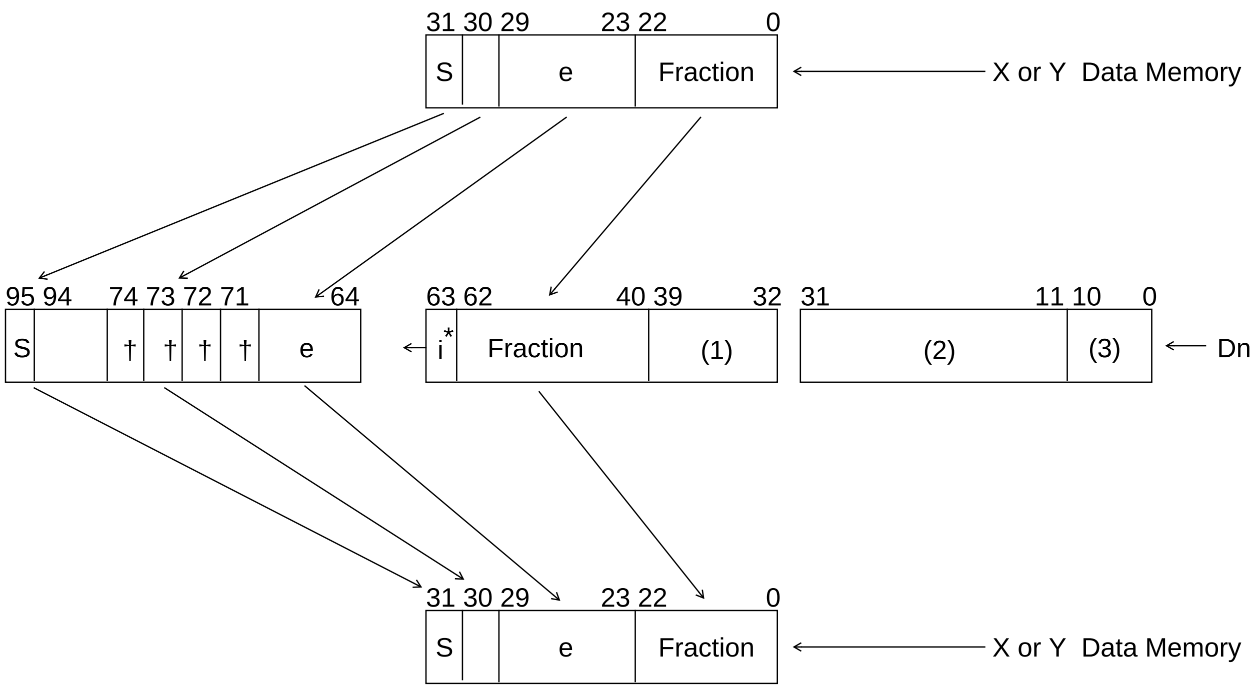


Figure C-9. The Data ALU’s Register File



Notes: * – i = 1 when normalized
i = 0 when unnormalized

† – When NaN, bits 71, 72, 73 = 1
When not NaN Bit 74 ↔ Bit 30
Bits 73, 72, 71 are complement of Bit 74.

(1) – Bits 32-39 are nonzero when the register contains a SEP floating point result or a DP floating point number.
Bits 32-39 are zero when the register contains a SP floating point number.

(2) – Bits 11-31 are only nonzero when the register contains a DP floating point number.

(3) – Bits 0-10 are always zero when representing a floating point number.

Figure C-10a. Automatic Format Conversion – Single Precision

The registers d8 and d9 are auxiliary registers which can be used for temporary data storage. Their main purpose is to allow a fast, four-cycle radix-2, decimation in time FFT butterfly kernel, though their use is certainly not limited to this application. d8 and d9 can be used as source operands in multiply operations and MOVE instructions, but can only be written as destinations of MOVE instructions.

The format conversion unit provides automatic format conversion from/to the SP and DP memory storage formats to/from the DP storage format in the data ALUs register file. The conversion is depicted in Figure C-10 and is done in a transparent fashion.

When moving **SP** numbers **into** the data ALU (see Figure C-10a), the 52-bit fraction of the DP internal format is written with the 23-bit fraction of the source in its most significant bits, and the implicit integer bit is made explicit. The remaining bits of the fraction are set equal to zero. If the number in question is denormalized (exponent = e_{min} and the first bit of the mantissa = 0), the U tag is set. In the non-IEEE "flush to zero" mode (indicated by the FZ bit in the Status Register), the number is considered zero when used as an operand for floating-point operations, although the contents of the register are not changed. In the IEEE

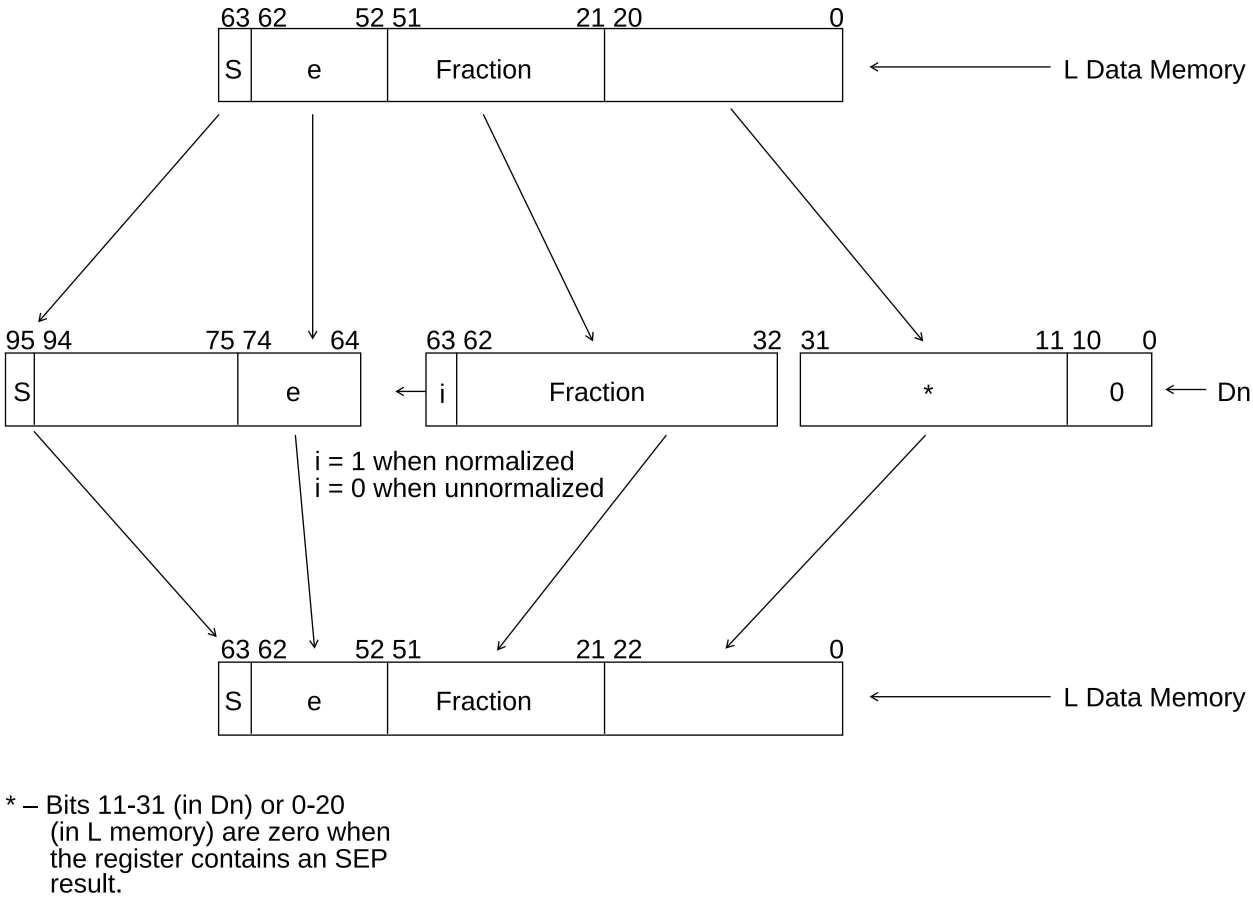


Figure C-10b. Automatic Format Conversion – Double Precision

mode, the number is "corrected" when used as an operand for floating point calculations, at the expense of extra cycles introduced for normalization.

The 8-bit exponent of the SP source is translated into an 11-bit exponent by copying the 7 least significant bits of the source exponent into the seven least significant bits of the destination. The most significant bit of the 8-bit exponent of the source is copied to the most significant bit of the exponent of the destination. The remaining 3 bits of the destination's exponent are set if the number is a NaN or infinity, otherwise they are the inverted MSB of the source's exponent. Inverting the MSB effectively changes the bias from 127 to 1023.

When moving **single precision** numbers **from** the data ALU to memory (see Figure C-10a), the above process is reversed. The 23 most significant bits of the fraction are moved to the 23 fraction bits of the destination. Note that the contents of the data ALU register may have more than 23 fractional bits if it was the result of a previous DP move or SEP arithmetic operation; in this case, the fraction is simply truncated.

The MSB of the 11-bit exponent of the source in the data ALU is moved to the MSB of the exponent of the destination. The 7 LSBs of the exponent of the source are copied to the seven LSBs of the exponent of the source. Note that if the source was not a SP number (result of a DP move or a SEP arithmetic operation), an incorrect exponent may be moved. **Therefore, care must be taken to always round results to SP before moving them to memory as single precision numbers.**

When moving **DP** numbers **into** the Data ALU from memory (see Figure C-10b), the 52 bit fraction of the

source is moved to the 52 bit fraction of the destination, and the implicit integer bit is made explicit. If the number is denormalized, the V tag is set. Again, extra cycles may be required when a denormalized number is used as an operand, depending on the FZ bit in the SR. The 11-bit exponent of the source is copied to the 11-bit exponent of the destination.

When moving **DP** numbers **from** the data ALU to memory, the above process is reversed, as shown in Figure C-10b. Note that the 52-bit fraction may actually consist of 21 zeros if the number in question was the result of a SEP arithmetic or 29 zeros in the case of a SP move. SEP arithmetic result precision can only be retained in memory by using DP moves.

C.1.5.1.1 FLOATING-POINT MOVES TO/FROM DATA ALU REGISTERS

The following sections deal with the case where a write (move in) is followed by a read (move out) without any floating-point operation being actually performed on the Data ALU register (save-restore procedure). The only way to provide correct results for save-restore procedures is to perform the same type of moves when writing and then reading the register (SP write followed by SP read or DP write followed by DP read).

C.1.5.1.1.1 Single Precision (SP) Move Of A SP Normalized Number

This section illustrates what happens when a 32-bit source (normalized single precision) is written by a single precision floating-point move and the data is stored in a Data ALU floating-point register D0-D9. Following the above operation, the Data ALU register will be read first by a single precision and then by a double precision floating-point move.

```
        - 32-bit data from source is 3F800000 (= +1.0)
             - exp      = 7F (8 bit bias)
             - mantissa = 000000 (the hidden bit is one)

        - data stored in the register
             - e     = 3FF   (correct representation with 11-bit bias)
             - I     = 1     (the number is normalized so hidden bit is 1)
             - U-TAG = 0     (cleared; the number can be used in computations
                              without adding extra cycles for normalization,
                              since it is a normalized number)
   - fraction =   00...00         - mantissa = 1.00...00
```

One should notice that both single and double precision floating-point moves out of the register will produce correct results in this case as shown in Figure C-11.

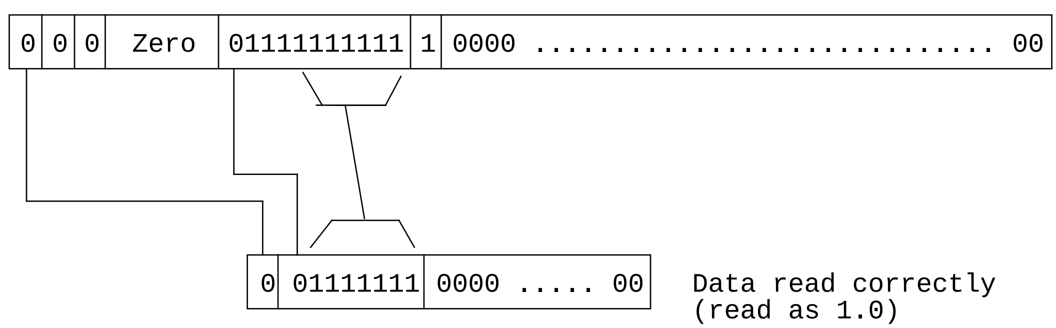
C.1.5.1.1.2 SP Move Of A SP Denormalized Number

This section describes what happens when a 32-bit denormalized, single precision number is written by a single precision floating-point move, into a Data ALU floating-point register D0-D9. Following the above operation, the Data ALU register will be read first by a single precision and then by a double precision floating-point move.

```
- 32-bit data from source is $00200000 (= +2**(-128))
     - exp      = $00     (8 bit bias)
     - mantissa = $200000 (the hidden bit is zero)
SP move into the register
     - e     = 380   (incorrect representation with 11-bit bias; the
                      correct representation would be 37F)
     - I     = 0     (the number is unnormalized)
     - U-TAG = 1     (set; the number cannot be used in computations
                      without adding extra cycles for normalization,
                      since it is unnormalized)
     - fraction = 40000000
     - mantissa = 0.010...00
```

In this last case, the U-TAG tells us that an operation using this operand will first add extra cycles to normalize it. However, an SP move will render the correct result since the format conversion presented in **Section 5.5** chooses the correct bits. One should notice that a double precision floating-point move that reads the register will yield the wrong data in this case.

SP read of the register



DP read of the register

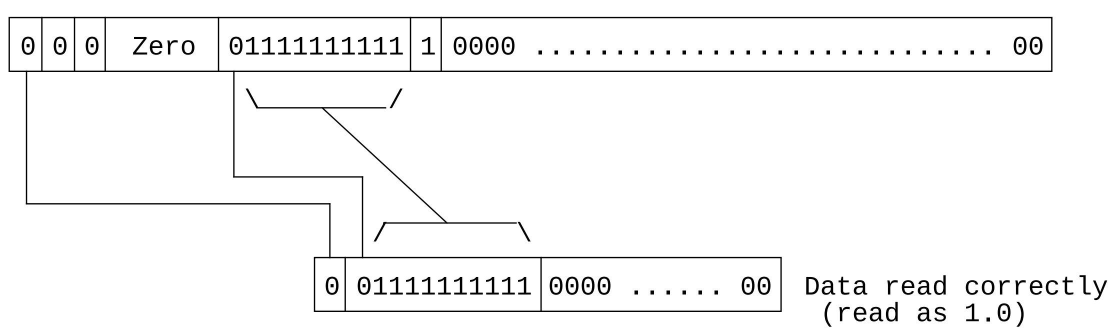


Figure C-11. Single Precision (SP) Move Of A SP Normalized Number

C.1.5.1.1.3 Denormalized Numbers In Double Precision (DP)

This section describes what happens when a 64-bit denormalized double precision number is written by a double precision floating-point move, into a Data ALU floating-point register D0-D9. Following the above operation, the Data ALU register will be read first by a single precision and then by a double precision floating-point move.

The denormalized double precision data is stored in the Data ALU register with the V tag set and the exponent set to $000 (always). The V-TAG set indicates that floating-point multiply operations will require extra cycles to wrap it ("normalize") before using it as an operand. Double precision moves will yield correct results when reading the denormalized DP from the register to memory (the V-TAG will also be set when a single extended denormalized result is obtained from a Data ALU operation).

Here is an example of a double precision denormalized number:

```
- 64 bit data from source is 0004000000000000 (= 2**(-1024))
     - exp      = $000 (11-bit bias)
     - mantissa = $4000000000000 (the hidden bit is zero)

- data stored in the register
     - e     = 000   (correct representation with 11-bit bias)
     - I     = 0     (the number is not normalized)
     - U-TAG = 0     (cleared; the number can be used in computations
                      as it is by the adder)

     - V-TAG = 1     (set; it indicates a denormalized number in DP,
                      requiring extra cycles for denormalization in
                      multiply operations)
     - fraction = 40000000
     - mantissa = 0.010...00
```

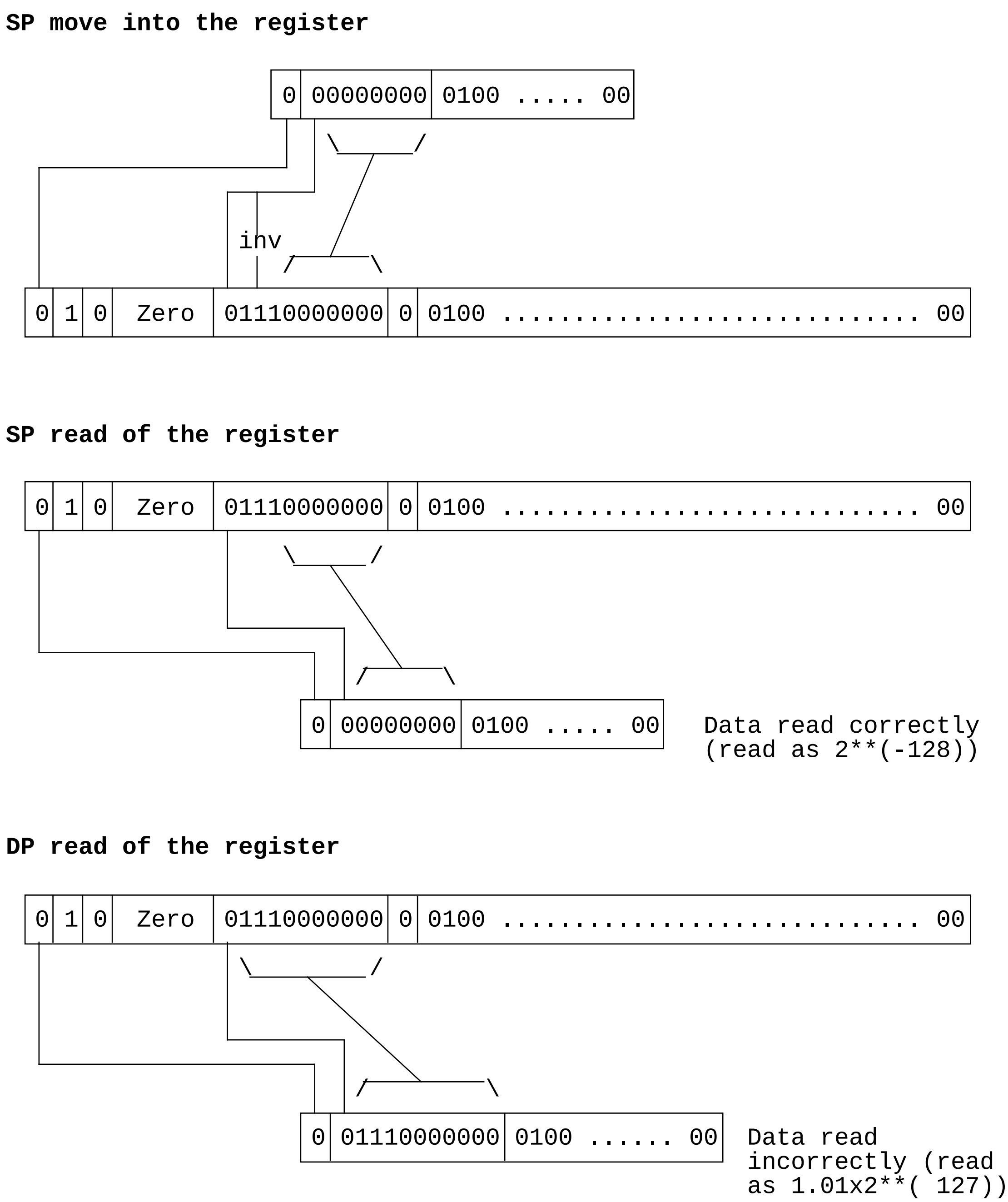
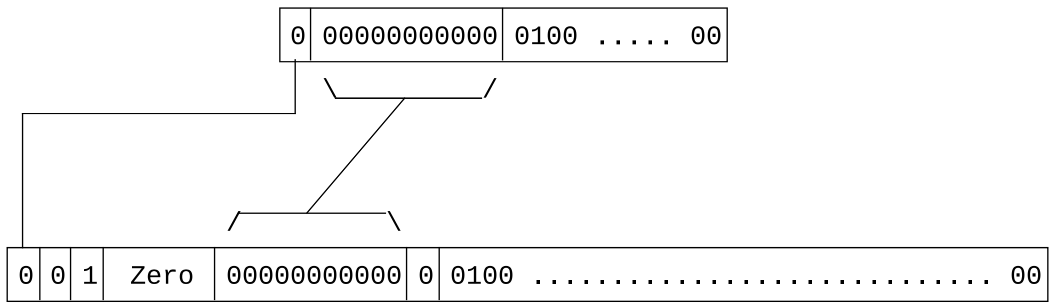


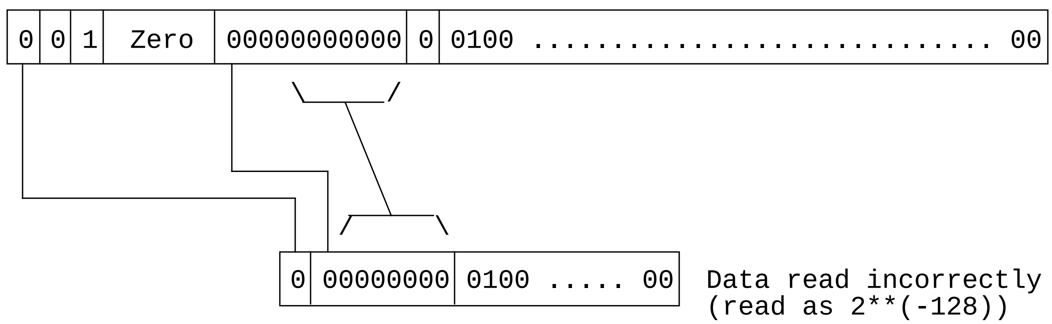
Figure C-12. SP Move Of A SP Denormalized Number

DP move into the register



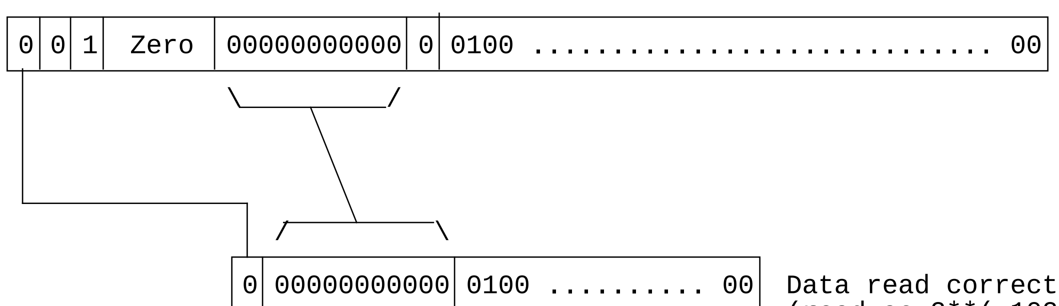
NOTE THAT THE V TAG IS SET IN THIS CASE

SP read of the register



Data read incorrectly
(read as 2**(-128))

DP read of the register



Data read correctly
(read as 2**(-1024))

C.1.5.1.1.4 Floating-Point Moves Summary

Figure C-14 summarizes what will be the result of a data move into a Data ALU register followed by a read of the same register, depending on the data range and the type of moves.

Figure C-13. Denormalized Numbers In Double Precision (DP)

MOVE IN TYPE	EXPONENT RANGE (UNBIASED)	INPUT DATA	U	V	MOVE OUT TYPE	MOVE OUT RESULT
SP	E= 128 Fraction= .0xx...xx	signaling NaN (SNAN) written as DP SNAN read as SNAN (see Note 1)	0	0	SP	CORRECT
					DP	CORRECT
SP	E= 128 Fraction= .1xx...xx	non signaling NaN (QNAN) written as DP QNAN read as QNAN (see Note 2)	0	0	SP	CORRECT
					DP	CORRECT
SP	E= 128 Fraction= .000...00	infinity in SP written as DP infinity read as infinity (all formats)	0	0	SP	CORRECT
					DP	CORRECT
SP	-127<E< 128	normalized (all formats)	0	0	SP	CORRECT
					DP	CORRECT
SP	-150<E<-126	denormalized in SP	1	0	SP	CORRECT
					DP	WRONG
DP	E= 1024 Fraction= .0xx...xx	signaling NaN (SNAN) written as DP SNAN read as SNAN (see Notes 1,3)	0	0	SP	CORRECT
					DP	CORRECT
DP	E= 1024 Fraction= .1xx...xx	non signaling NaN (QNAN) written as DP QNAN read as QNAN (see Note 2)	0	0	SP	CORRECT
					DP	CORRECT
DP	E= 1024 Fraction= .000...00	infinity in SP written as DP infinity read as infinity (all formats)	0	0	SP	CORRECT
					DP	CORRECT
DP	+127<E< 1024	no SP representation normalized in DP/SEP	0	0	SP	WRONG
					DP	CORRECT
DP	-127<E< 128	normalized (all formats)	0	0	SP	TRUNC
					DP	CORRECT
DP	-150<E<-126	denormalized in SP normalized in DP/SEP	0	0	SP	WRONG
					DP	CORRECT
DP	-1023<E<-149	no SP representation normalized in DP/SEP	0	0	SP	WRONG
					DP	CORRECT
DP	-1054<E<-1022	denormalized (in DP/SEP)	0	1	SP	WRONG
					DP	CORRECT

Figure C-14. Floating-Point Moves Summary

Note 1 The xx...xx pattern for the signaling NaNs indicates any NON-ZERO bit pattern.

Note 2 The xx...xx pattern for the non-signaling NaNs indicates any bit pattern. The DSP96002 generates all ones for QNaNs.

Tags
Note 4

Note 3 If a register is written with a SNAN using a double precision floating-point move and then the same register is read using single precision floating-point move the result will be a single precision SNAN (if the first 23 bits of the fraction are a non-zero pattern) or single precision infinity (if the first 23 bits of the fraction are a zero pattern).

Note 4 The case when both U-TAG = 1 and V-TAG = 1 is reserved for future use.

C.1.5.1.2 RESULTS OF DATA ALU FLOATING-POINT OPERATIONS

This section describes how the Data ALU floating-point operation results are stored in the Data ALU registers.

All DSP96002 Data ALU floating-point operations are executed in single extended precision, using single extended precision input operands, and return single extended or single precision results in double precision format. The results are formatted in double precision before being stored in the Data ALU registers. When performing a DP move into a register and then using that register in a DSP96002 SEP floating point operation, the mantissa of the operand will be first truncated to a SEP value, as the hardware is unable to operate on more than 32 mantissa bits. Figure C-13 explains how a DP register is used as operand for a SEP operating unit (adder/multiplier).

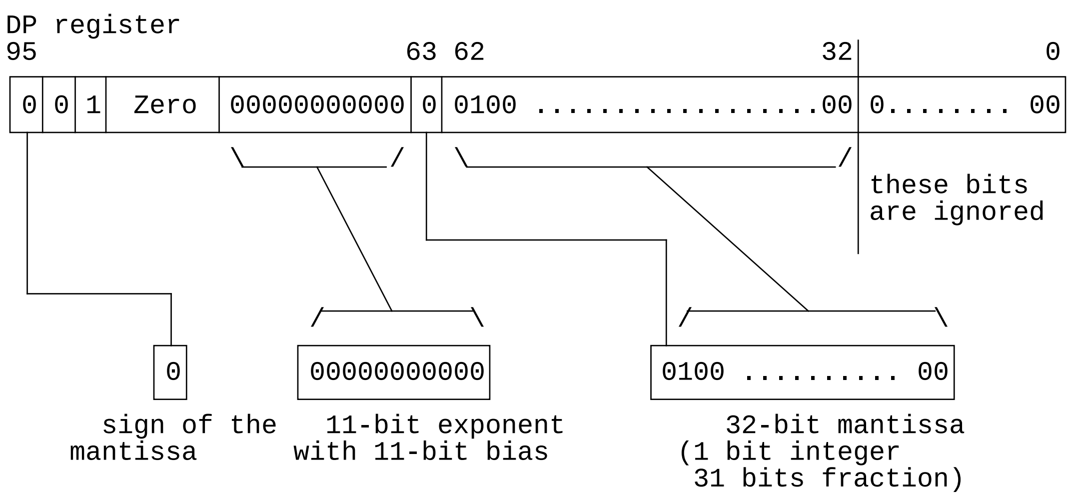


Figure C-15. DP operand in a SEP operation

The 11-bit exponent used by the SEP operating units is identical with the exponent of the original DP number loaded into the register (both have the same bias, namely $3FF). This means that the number can be used in computations directly, assuming that the least significant 21 mantissa bits are zero, otherwise a **round towards zero** occurs because the mantissa is truncated to 32 bits (21 bits of the 52-bit DP mantissa are ignored).

C.1.5.1.2.1 Results Rounded To SP

Data ALU results are rounded to SP when the instruction is specified with the .S suffix (FMPY.S, FADD.S, etc.). The rounding mode is programmed using the rounding mode bits in the status register.

C.1.5.1.2.1.1 Results Rounded To SP That Are Normalized

If the Data ALU operation result was rounded to SP and the rounded result may be represented as a normalized single precision floating-point number, the result will be stored in normalized DP format that may be read out by single and double precision moves without errors or truncation.

C.1.5.1.2.1.2 Results Rounded To SP That Are Denormalized

If the Data ALU operation result was rounded to SP and the rounded result must be represented as a denormalized single precision floating-point number, the result will be stored in unnormalized DP format with the U tag set and the I bit cleared, and it may be read out by single precision moves without errors or truncation. If the register is read by a double precision move, completely incorrect data will be obtained; see the discussion in Section C.1.5.1.1.2.

In this case, before the result is delivered, an additional Data ALU execution cycle is required in which the SEP mantissa is shifted right the required number of places for correct rounding to SP.

The presence of unnormalized numbers in DP format will add one dummy cycle followed by an additional cycle for each unnormalized DP operand to any Data ALU operation that uses them as input. During the additional cycle the unnormalized operand (U-TAG=1) is normalized, however the register itself will not be modified.

C.1.5.1.2.2 Results Rounded To SEP

Data ALU results are rounded to SEP when the instruction is specified with the .X suffix (FMPY.X, FADD.X, etc.). The rounding mode is programmed using the rounding mode bits in the status register.

C.1.5.1.2.2.1 Results Rounded To SEP That Are Normalized

If the Data ALU operation result was rounded to SEP and the rounded result may be represented as a normalized single extended precision floating-point number, the result will be stored in normalized DP format that may be read out by double precision moves without errors or truncation.

If the result stored in the register is read with a single precision move, two situations may occur:

1. The SEP exponent is in the range of the normalized SP exponent: the data read will be rounded to SP by truncating the SEP mantissa; this is equivalent to IEEE round towards zero.
2. The SEP exponent is not in the range of the normalized SP exponent: the data read will not have the right exponent. The correct value should have been infinity, zero or a denormalized SP, but the move instruction does not provide it.

C.1.5.1.2.2.2 Results Rounded To SEP That Are Denormalized

If the Data ALU operation result was rounded to SEP and the rounded result must be represented as a denormalized single extended precision floating-point number, the result will be stored in normalized DP format with the V tag set and I bit cleared, and it may be read out by double precision moves without errors

or truncation. If the register is read by a single precision move, completely incorrect data will be obtained; see the discussion in Section C.1.5.1.1.3.

C.1.5.1.2.3 Data ALU Results/Move Compatibility Summary

Figure C-16 summarizes what happens when Data ALU operation results of a certain range are stored in the destination register, and the register is read by a certain kind of move.

All cases where "move out type"=SP and "move out result"=WRONG can be corrected by rounding in the instruction (using the .S option). The case where "move out type"=SP and "move out result"=TRUNC can also be corrected by using the .S option.

C.1.5.2 Multiply unit

The multiply unit consists of a hardware multiplier, an exponent adder, and a control unit, as shown in Figure C-17. The multiply unit accepts two 44 bit input operands for floating point multiplications, each consisting of a sign bit, eleven exponent bits, the explicit integer bit, and 31 fractional bits. Note that for full double precision operands, as obtained by double precision MOVEs, the least significant 8 bits of the fraction are simply truncated. Multiply operations occur in parallel with and independent of data moves over the X and Y data buses.

The hardware multiplier accepts the two 32-bit mantissas (integer bit + 31 bit fraction), and delivers a 64 bit result, as shown in Figure C-18. This result is automatically rounded to a 32-bit mantissa for SEP arithmetic or a 24 bit mantissa for SP arithmetic, as specified by the instruction opcode. The result is stored into the mantissa portion of the destination register.

The exponent adder takes the two unsigned (i. e., biased) operand exponents, adds them together, and subtracts one bias, resulting in an 11-bit biased exponent which is stored in the exponent part of the floating point format in the destination register, as depicted in Figure C-19.

C.1.5.3 Adder/Subtracter Unit

The adder/subtracter is depicted in Figure C-20, and consists of a barrel shifter and normalization unit, an add unit, a subtract unit, an exponent comparator and update unit and a special function unit. The adder/subtracter unit accepts 44-bit floating point operands, and delivers 44-bit results. The adder/subtracter operations deliver the sum and the difference of the same two floating point operands in a single instruction cycle. In addition, the barrel shifter used for mantissa alignment in floating point additions and subtractions is used for executing multibit shifts for fixed point operation. The adder/subtracter operates in parallel with and independent of data moves over the X and Y data buses.

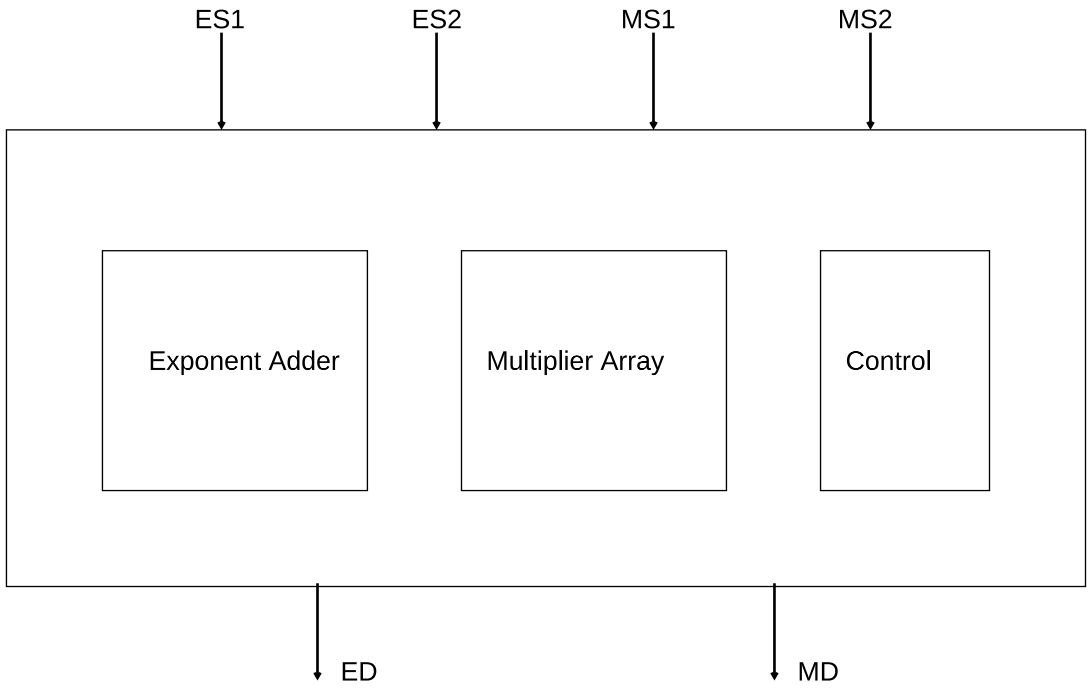
The add unit is a high speed 32-bit adder, used in all floating-point non-multiply operations. For floating point operations, 32-bit mantissas (1 integer bit and 31 fractional bits) are first "aligned" for floating point addition in the barrel shifter and normalization unit, after which they are added in the add unit. The result is then rounded to 32 bits for SEP results, and to 24 bits for SP results, as indicated by the instruction opcode. The type of rounding implemented depends on the rounding mode bits in the MR register. The rounded result is stored in the middle portion (mantissa) of the destination register. This is illustrated in Figure C-21

The subtract unit is a high speed 32-bit adder/subtracter, used in all floating-point non-multiply operations and in all fixed point operations delivering a 32-bit result. For floating point operations, 32-bit mantissas (1 integer bit and 31 fractional bits) are first "aligned" for floating point subtraction in the barrel shifter and normalization unit, after which they are subtracted in the subtract unit. The result is then rounded to 32 bits for SEP results, and to 24 bits for SP results, as indicated by the instruction opcode. The type of rounding implemented depends on the rounding mode bits in the MR register. The rounded result is stored in the middle portion (mantissa) of the destination register for floating point operations, and in the low portion for fixed-point operations. This is illustrated in Figure C-21.

The barrel shifter/normalization unit is used for the alignment of the two operand mantissas, needed for addition/subtraction of two floating point numbers. The barrel shifter is a 32-bit left-right multibit shifter, which is also used in fixed point arithmetic and logic shifting operations with a 32-bit result. For the addition of two floating point operands, the barrel shifter receives the exponent difference of the two operand exponents from the exponent comparator and update unit, and uses this difference to align the mantissas for addition. For example, if the biased exponent of the first floating point operand equals 10 and the biased exponent of the second floating point operand equals 13, the mantissa of the first operand will be right shifted by three

ROUND TO	EXPONENT RANGE BEFORE ROUND (UNBIASED)	DATA ALU OPERATION RESULT	TAGS U	TAGS V	MOVE OUT TYPE	MOVE OUT RESULT
SP	NaN operand or invalid op	non signaling NaN (QNAN) written as DP QNAN e=7FF mantissa=1.11...11	0	0	SP	CORRECT
					DP	CORRECT
SP	127<E	infinity (overflow) written as DP infinity e=7FF mantissa=1.00...00	0	0	SP	CORRECT
					DP	CORRECT
SP	127<E< 128	normalized (all formats)	0	0	SP	CORRECT
					DP	CORRECT
SP	-150 < E< –126	denormalized (in SP)	1	0	SP	CORRECT
					DP	WRONG
SP	E < –149	zero (underflow)	0	0	SP	CORRECT
					DP	CORRECT
SEP	NaN operand or invalid op	non signaling NaN (QNAN) written as DP QNAN e=7FF mantissa=1.11...11	0	0	SP	CORRECT
					DP	CORRECT
SEP	1023<E	infinity in SP and SEP written as DP infinity e=7FF mantissa=1.00...00	0	0	SP	CORRECT
					DP	CORRECT
SEP	127<E< 1024	infinity in SP normalized in SEP	0	0	SP	WRONG
					DP	CORRECT
SEP	-127<E< 128	normalized (all formats)	0	0	SP	TRUNC
					DP	CORRECT
SEP	-150< e < -126	denormalized in SP normalized in SEP	0	0	SP	WRONG
					DP	CORRECT
SEP	-1023< e < -149	zero in SP normalized in SEP	0	0	SP	WRONG
					DP	CORRECT
SEP	-1054< e< -1022	zero in SP denormalized in SEP	0	1	SP	WRONG
					DP	CORRECT
SEP	e< -1053	zero in SP zero in SEP (underflow)	0	0	SP	CORRECT
					DP	CORRECT

Figure C-16. Data ALU Results/Move Compatibility Summary



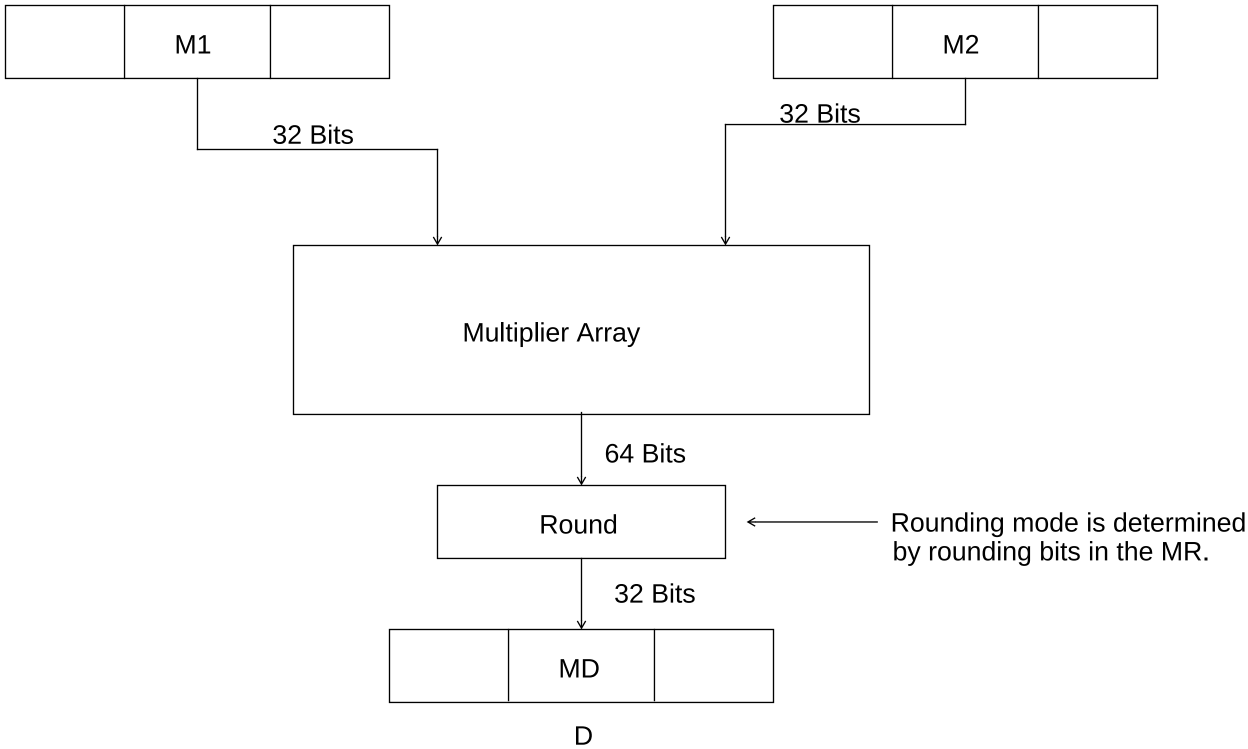
positions (3 bit shift).

The exponent comparator and update unit consists of an 11 bit subtracter, which compares the two exponents of the floating point operands, and delivers the difference to the barrel shifter for mantissa alignment. The largest of the two exponents is delivered to the exponent update unit. The exponent update unit may update this exponent for normalization of the result, after which the exponent (biased) is stored in the high portion of the destination register. This is depicted in Figure C-22.

For example, if the mantissa of the first operand in a floating point addition is 1.010...0, with biased exponent of 10, and the mantissa of the second operand is 1.000...0000, with biased exponent of 13, the exponent comparator simply delivers the difference (=3) to the barrel shifter, the first operand’s mantissa is aligned to 0.001010...0, the two mantissas are added to deliver 1.001010...0, and the result (biased) exponent equals 13. The postnormalization unit does not need to postnormalize the result in this case.

If the first operand’s mantissa is 1.010...0 with biased exponent of 13 and the second operand’s mantissa is 1.000...0 with biased exponent of 13, the exponent difference is zero and the barrel shifter does not need to realign the mantissas. The result after addition is now equal to 10.010...0, which needs to be postnormalized by adding one to the result exponent. The exponent update unit sets the result exponent (biased) equal to 14 and the result mantissa is 1.0010...0.

Finally, if the first operand’s mantissa in a floating point subtraction is 1.010...0 with biased exponent of 10, and the second operand’s mantissa is 1.00...0 with a biased exponent of 10, the result mantissa after subtraction is -0.010...0. This is not normalized, and the postnormalization unit subtracts two from the exponent. The result mantissa is -1.000...0 with a biased exponent equal to 8.



C.1.5.4 Special Function Unit

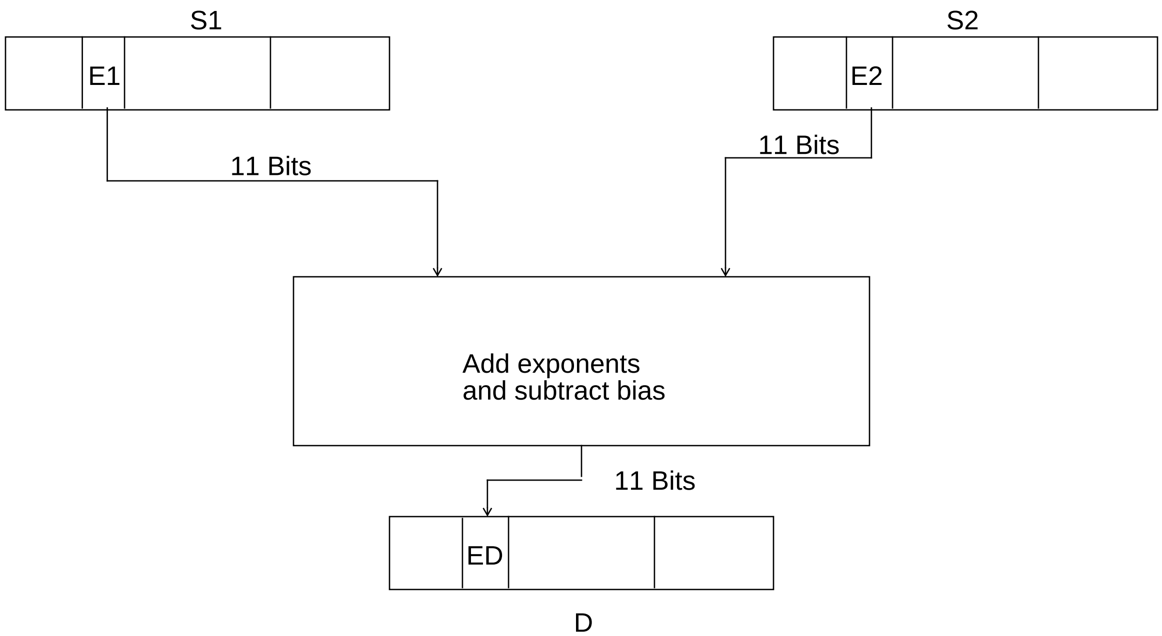
The special function unit (SFU) consists of a logic unit and a divide and square root unit. The logic unit is further described under the fixed point (integer) operations.

The divide and square root unit supports execution of the divide and square root algorithms. These algorithms are iterative, and require an initial approximation or "seed". These seeds are generated in the SFU. The FSEEDD and FSEEDR instructions provide an initial approximation to 1/x and sqrt(1/x), as is described in Appendix A.

C.1.5.5 Controller and Arbitrator Unit

The controller and arbitrator (CA) unit supplies control signals to the processing units of the data ALU and register file, and is responsible for the full implementation of the IEEE standard. Its operation is determined by the flush-to-zero (FZ) bit in the status register (SR), which determines whether or not denormalized numbers are treated as defined by the standard. In the flush-to-zero mode, all denormalized input operands are treated as zeros (although their original contents are preserved), and denormalized results are set equal to zero ("flushed-to-zero"). In the flush-to-zero mode, no additional cycles are required for the normalization of denormalized numbers as they are treated as zeros. In the IEEE mode, the standard for treatment of denormalized numbers is correctly and fully implemented. However, operations on denormalized numbers can not be performed in a single instruction cycle, except for operations done in the floating point adder when the operand is a denormalized number in SEP. The controller and arbitrator is responsible for providing the correct sequence that deals with such situations.

When denormalized numbers are detected as input operands in IEEE mode, the CA unit adds one extra cycle for entering the IEEE mode procedure. Next, one additional cycle is added for each denormalized in-



put operand. These cycles are used to normalize the input operand. The original value of the operand in the source register is not affected. **During the IEEE mode procedure all activity of the chip is suspended until the input operands have been normalized**. When denormalized output results are detected, each denormalized output result is normalized (one additional instruction cycle). There is no extra cycle penalty for entering the IEEE mode procedure when normalizing output results.

C.2 FIXED-POINT NUMBER STORAGE AND ARITHMETIC

C.2.1 General

Integer operand sizes are defined as follows:

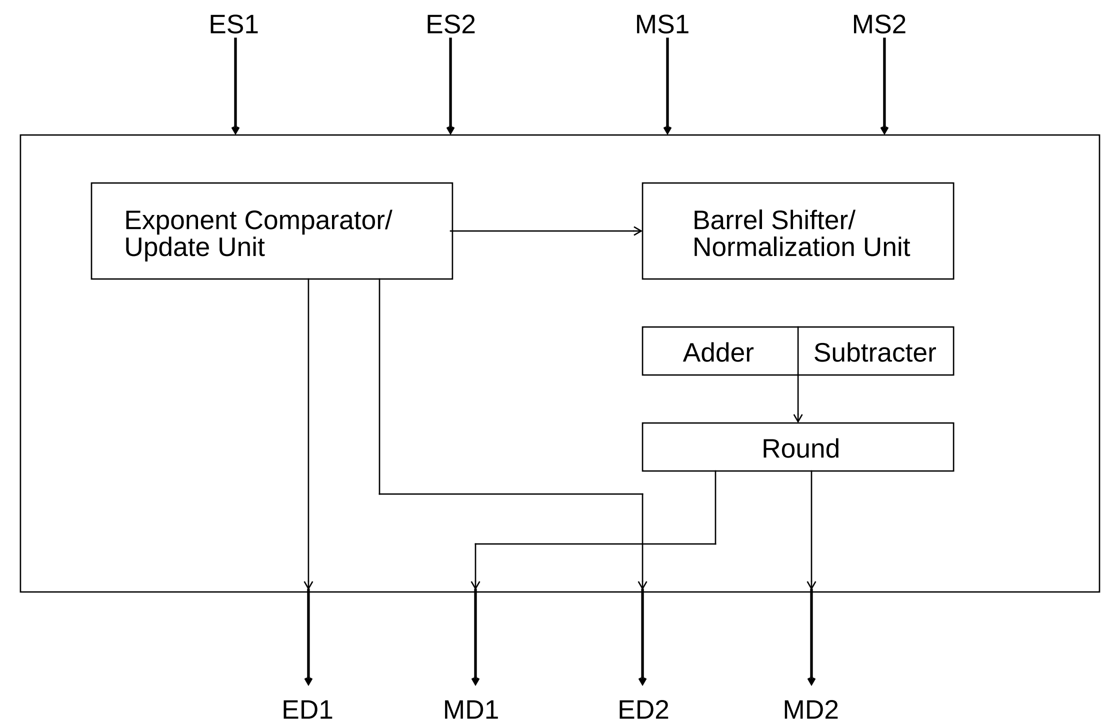
1. **Byte:** 8 bits long
2. **Short word:** 16 bits long
3. **Word:** 32 bits long
4. **Long word:** 64 bits long

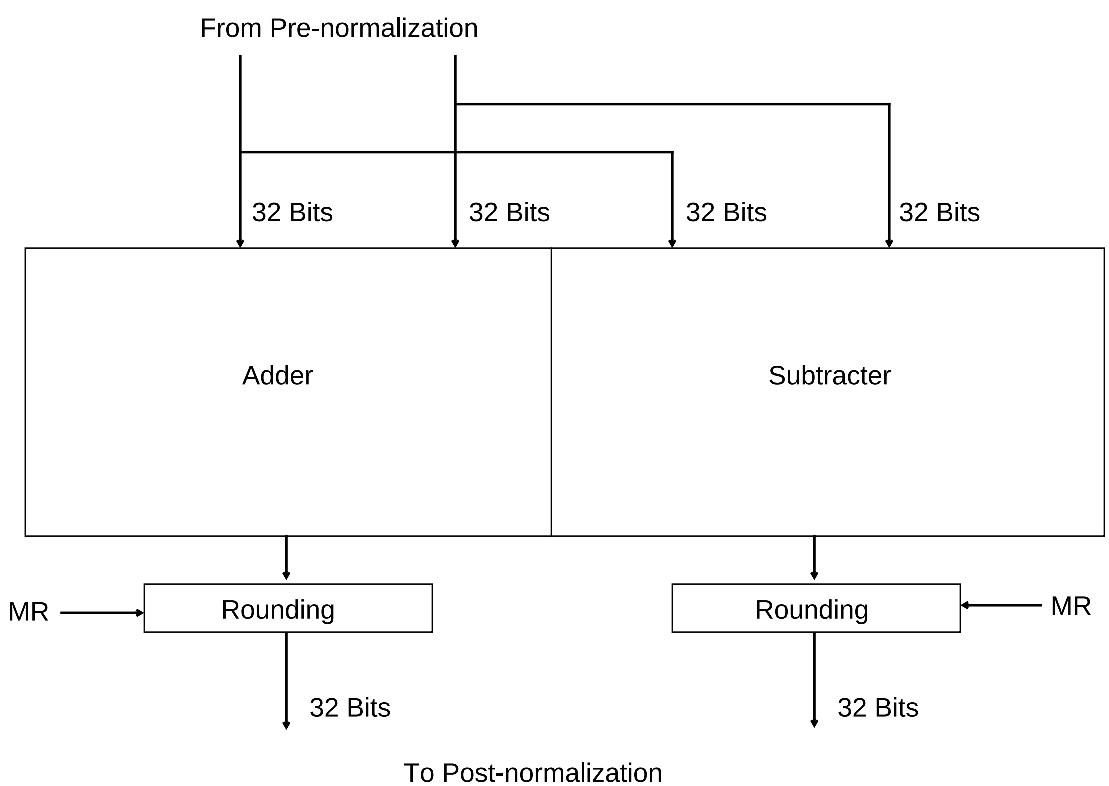
The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction.

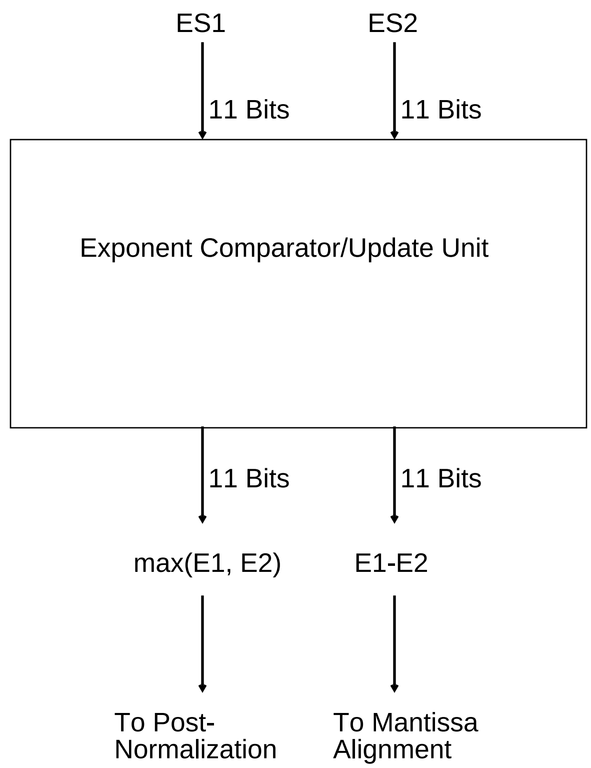
C.2.2 Integer Storage Format in Memory

The DSP96002 supports four integer memory data formats:

1. Signed word integer: 32 bits wide, two’s complement representation. This storage format can be used in either X and/or Y data memory space.
2. Signed Long Word Integer: 64 bits wide, two’s complement representation. This storage format


ES1
ES2
MS1
MS2
Exponent Comparator/
Update Unit
Barrel Shifter/
Normalization Unit
Adder
Subtracter
Round
ED1
MD1
ED2
MD2


From Pre-normalization
32 Bits
32 Bits
32 Bits
32 Bits
Adder
Subtracter
MR
Rounding
Rounding
MR
32 Bits
32 Bits
To Post-normalization



can only be used in long (L) data memory space.

3. Unsigned Word Integer: 32 bits wide with unsigned magnitude representation. This storage format can be used in either X and/or Y data memory space.
4. Unsigned Long Word Integer: 64 bits wide with unsigned magnitude representation. This storage format can only be used in long (L) data memory space.

Long type integers can be moved to and from the data ALU register file. However, long integers can not be directly used as input operands to data ALU operations. Long integers can however be results of data ALU operations.

C.2.3 Integer Storage Format in the Data ALU

There are thirty 32-bit registers in which can contain integer words. However, data ALU arithmetic operations use the low portion of the register files as word source and destination operands. Long word integers are only generated as results of integer arithmetic operations and are never used as source operands.

C.2.4 Integer Arithmetic

The integer arithmetic operations use the same arithmetic units in the data ALU as the floating point operations. These units consist of:

1. **Adder**: The subtract unit in the adder/subtracter unit described in paragraph C.1.5.3 is used for integer add and subtract operations. It accepts two 32-bit integer operands from the low portions of the data ALU source registers and delivers a 32-bit result in the low portion of the des-

tination register.

2. **Multiplier:** The multiplier in the multiply unit described in paragraph C.1.5.2 also performs the integer multiplications. It accepts two 32-bit operands in the low portion of the data ALU source registers, and delivers a 64-bit result in the low and middle portions of the destination register. Both signed and unsigned multiplications are supported.
3. **Logic Unit:** The logic unit is responsible for the logical operations AND, ANDC, OR, ORC, EOR, NOT, ROR. In addition, it performs the bit field manipulation instructions SPLIT, SPLITB, JOIN, JOINB, EXT, and EXTB. The logic unit operates on 32-bit operands located in the low portions of the data ALU registers. Results are also stored in the low portion of the destination.
4. **Barrel Shifter:** The barrel shifter in the normalization unit used for mantissa alignment in floating point additions is also available for performing multibit shifts on integer (fixed-point) data. Both single and multibit arithmetic shifts left and right and logical shifts left and right are supported.

APPENDIX D

D.1 FLOATING-POINT NUMBER STORAGE AND ARITHMETIC

D.1.1 General

The IEEE standard for binary floating point arithmetic provides for the compatibility of floating-point numbers across all implementations which use the standard by defining bit-level encoding of floating-point numbers. Maximum mathematical accuracy, with respect to roundoff errors, is achieved by optimally scaling floating-point numbers by using a normalized exponential notation. Error bounds are guaranteed by the standard for the basic mathematical operations (add, subtract, multiply, divide, square root, round to nearest integer, conversion to and from integers and conversion to and from decimal strings). The standard also defines error handling for five floating point exceptions: invalid operation, divide by zero, overflow, underflow and inexact result.

The standard defines two data storage formats which are identical across implementations (basic formats): Single Precision (SP) and Double Precision (DP). It also specifies the use of two implementation-dependent encodings (extended formats): Single Extended Precision (SEP) and Double Extended Precision (DEP), on which it only places some general constraints, and for which bit-level encodings are not defined. The extended formats are consequently implementation-dependent and should never be used for representation of numbers which are to be shared across different processors (i. e., stored).

Each format provides representation of the following elements:

1. **Floating-point numbers** of the form:

$$X = (-1)^S 2^E (b_0 \bullet b_1 b_2 \ldots_{p-1})$$

where:

$s = 0$ or 1

E = an integer between E_{min} and E_{max} , inclusive.

$b_i = 0$ or 1

2. **Infinities:** $+\infty$ and $-\infty$

3. **" Not-a-Numbers (NaNs) "**. NaNs are special symbolic elements, encoded in the floating point format. They can appear as operands and/or as results of arithmetic operations. The standard provides two types of NaNs:

   Quiet NaNs (QNaNs): are encodings of information regarding meaningless or invalid results. Examples of QNaNs are results of operations such as 0/0, $\infty-\infty$, ∞/∞, etc. Encodings of QNaNs

are intended to provide some kind of retrospective diagnostic information concerning the origin of the NaN. Since this information needs to remain available even after a large number of arithmetic operations, QNaNs "propagate" unchanged through arithmetic operations and format conversions. QNaNs can thus occur as operands of an arithmetic operation. If one or more QNaN occur as operands, the result is a quiet NaN, and no floating point exception is signaled. Hence the name "quiet" NaN. The standard specifies that at least one QNaN must be supported.

Signaling NaNs (SNaNs): Signaling NaNs are used only in systems with arithmetic-like enhancements that are not defined by the standard. As opposed to QNaNs, they are never generated by the DSP96002 arithmetic. They can, however, appear as operands in arithmetic operations (as generated by other processors, for instance). In this case, they always signal the "Invalid Operation" floating point exception. The returned result is a QNaN.

Floating point operands in the DSP96002 are either 32-bits long (Single Real), 64 bits long (Double Real) or 96 bits long (Register operand). The operand size is either explicitly encoded in the instruction or implicitly defined by the instruction operation. The following sections describe the details of each operand type.

D.1.2 DSP96002 Floating Point Storage Format in Memory

DP and SP are the only floating point formats for which the IEEE standard provides bit-level definitions. Since the DSP96002 is designed for multiprocessing applications, where data in memory can be shared among different processors, SP and DP are the only formats supported for memory storage of floating point numbers.

SP numbers are represented by 32-bits in memory, and can be located in either X: or Y: data spaces. DP numbers take up 64 bits in memory, and can thus only be stored in long (L:) memory space.

The basic formats (SP and DP) contain three fields in their binary representation, as shown in Figure D-1. These fields are described as:

1. Sign Bit (s): The sign bit denotes the sign of the number, in a signed magnitude notation. When s=0, the number is positive. When s=1, the number is negative. Note that floating-point numbers do not use a two's complement notation.
2. Exponent Field (e): The exponent of SP and DP numbers is stored as a positive (biased) integer:

   $$e = E + bias$$

   where E is the actual exponent of the floating point number as explained in Appendix D.1.1. e is also used in conjunction with the fractional field f to encode non-numerical values (infinities and NaNs).

   For SP, the exponent consists of 8 bits (bits 23 through 30) , and the bias equals 127. The biased exponent e can thus take on integer values between 0 (denoted by e_{min}-1) and 255 (denoted by e_{max}+1) inclusive.

   For DP, the exponent consists of 11 bits (bits 52 through 62) , and the bias equals 1023. Values for the biased exponent e thus fall between 0 (e_{min}-1) and 2047 (e_{max}+1), inclusive. Table D-1 summarizes these values for SP and DP.
3. Fraction (f): The fractional field consists of bits b_i:

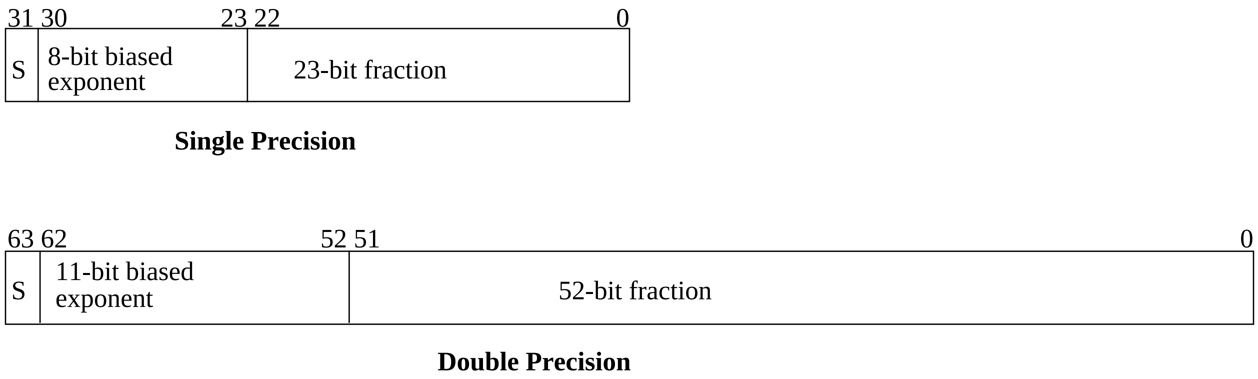


Figure D-1. SP and DP Formats

	p-1	bias	e_{min}	e_{max}
SP	23	127	+1	+254
DP	52	1023	+1	+2046

Table D-1. Parameters for Numerical Formats

$$f = \bullet b_1 b_2 \bullet\bullet\bullet b_{p-1}$$

There are 23 fractional bits (p=24) (bits 0 through 22) in the SP format, and 52 fractional bits (p=53) (bits 0 through 51) in the DP format.

The sign bit, exponent, and fraction fields encode the numerical values of floating-point numbers, as well as ± 0, ±∞, and NaNs as follows:

1. Normalized Numerical Values ($E_{min} \le E \le E_{max}$): For numerical values, the biased exponent e lies between e_{min} and e_{max}, inclusive. Equivalently, the exponent E takes on values between E_{min} and E_{max} inclusive. Table D-1 summarizes these values for SP and DP. If the biased exponent e is larger than e_{min} (E is larger than E_{min}), the number in question is called normalized, i.e. the implicit integer value b0 is equal to one. Note that this integer value is not stored in memory. Normalized numbers x are equal in value to:

$$x = (-1)^s \bullet 2^{e\ -\ bias}\ 1.f$$

where 1.f is a binary, fixed point number, i.e.:

$$1.f = 1+(o.5) \bullet b_1 + (0.25) \bullet b_2 + ... + \left(-\ \tfrac{1}{2}\right)^{p-1} \bullet b_{p-1}$$

Therefore, the smallest magnitude of any normalized number is equal to ($e=e_{min}$, f=0):

$$x_{min,n} = 1 \bullet 2^{e_{min}\ -\ bias}$$

Using the value from Table D-1, this equals approximately $1.18 \bullet 10^{-38}$ for SP numbers.

The largest (normalized) numerical value that can be represented equals (all b_i=1, $e=e_{max}$):

$$x_{max,n} = (2 - 0.5^{p-1})\ 2^{e_{max} - bias}$$

For SP this equals approximately (using the values in Table D-1) $3.4 \bullet 10^{38}$.

2. Denormalized Numerical Values ($e = e_{min}$-1, $f \neq 0$): When the exponent e equals the value e_{min}-1 and the fraction field is non-zero the floating point number is called denormalized, and the implicit integer bit b0 is equal to zero. The numerical value of a denormalized number y is given by:

$$y = (-1)^s \bullet 0.f \bullet 2^{e_{min}-bias}$$

The denormalization of the fractional part allows the representation of very small numbers near the underflow threshold. The smallest possible magnitude of any denormalized number equals ($f=f_{min}$) :

$$y_{min} = (0.5)^{p-1} \bullet 2^{e_{min} - bias}$$

For SP denormalized numbers, this results in a smallest magnitude of $1.4 \bullet 10^{-45}$.

3. Zeros ($e = e_{min}$-1,f=0): Floating point value(s) of zero are encoded by a biased exponent e equal to e_{min}-1, and a fractional field f of all zeros. Note that this encoding retains a significant sign bit: plus and minus zero are two separate entities. Figure D-2 shows the encoding of plus and minus zero in floating point format.

4. Infinities ($e = e_{max} + 1$, f = 0) Infinities are encoded in the floating point format by a biased exponent equal to e_{max}+1, and a fractional field f consisting of all zeros. The sign bit distinguishes between + and -∞. Figure D-3 shows the encodings for + and -∞ in SP and DP.

5. NaNs ($e = e_{max}$+1, f≠0): NaNs are encoded in the floating point format by a biased exponent equal to e_{max}+1, and a nonzero fractional field. The value of the sign bit is irrelevant in this encoding.

QNaNs (b_1=1) Quiet NaNs are represented by a fraction with MSB = 1 (and $e=e_{max}$+1). The DSP96002 only fully supports one QNaN, as required by the standard. This QNaN is encoded by a fractional field of all ones (all b_i = 1 in f) ("legal" QNaN). Other types of QNaNs ("illegal" NaNs) may occur in multiprocessing situations (as generated by other processors) however, and do deliver well-defined results in the DSP96002. When QNaNs other than the "legal" QNaN occur as operand(s) to floating point arithmetic, the delivered result is always a "legal" QNaN. Figure D-4 shows the encoding for QNaNs.

SNaNs (b_1=0) Signaling NaNs are never generated by the DSP96002 as arithmetic results, but may appear in the DSP96002 memory as passed along by other processors. SNaNs are characterized by a MSB of the fractional field equal to 0 (and $e = e_{max}$). When a SNaN appears as an operand of an arithmetic instruction, the invalid operation exception is signaled, and the result is returned as a "legal" QNaN.

The two basic formats, discussed in the previous paragraphs, are the only formats which are used for representation of floating point values in the DSP96002 memory (internal and/or external). As is shown in Appendix D.1.4, the SEP format, generated exclusively by the data ALU as a result of floating point arithmetic operations, is embedded in the DP format, and is thus stored implicitly as a DP number with zeros in the lower 21 bits of the fraction.

31	30 23	22 0
S	0	0

Single Precision

63	62 52	51 0
S	0	0

Double Precision

Figure D-2. Encodings for + and - Zero

31	30 23	22 0
S	11..................1	0

Single Precision

63	62 52	51 0
S	11..............................1	0

Double Precision

Figure D-3. Encodings for + and - Infinity

31	30 23	22 0
X	11..................1	1111....................................1

Single Precision

63	62 52	51 0
X	11..............................1	11111111..1

Double Precision

Figure D-4. Encodings for QNaNs

D.1.3 IEEE Floating Point Exceptions

The IEEE standard defines five types of exceptions which must be signaled when detected. The DSP96002 implements the default "trap disabled" way of signaling exceptions: when an exception occurs, a flag is set and program execution continues. The flag remains set until cleared by the user. The different exceptions are:

1. Invalid operation: The invalid operation exception is signaled when an operand is invalid for the specific operation to occur. The result of an invalid operation is a QNaN, as described above. Examples of invalid operations are 0/0, ∞/∞, $\infty-\infty$, $0\times\infty$, etc.
2. Division by zero: The result of a division by zero is an infinity (with the correct sign), and the operation is signaled as an exception.
3. Overflow: The overflow exception is signaled when the result of an operation exceeds the largest magnitude that the result precision can accommodate. The result is dependent upon the rounding mode. For round to nearest, an infinity with correct sign is generated. Round to zero results in the largest possible numerical value the result precision can accommodate, with correct sign (i. e., the result saturates). Round to $-\infty$ results in the largest possible numerical value the result precision can accommodate (i. e., the result saturates) when the overflow is positive. It results in $-\infty$ when the overflow is negative. Round to $+\infty$ results in $+\infty$ when the overflow is positive, and in the smallest negative numerical value the result precision can accommodate (i. e., the result saturates) when the overflow is negative.
4. Underflow: Underflow is signaled when both (1) a very small (tiny) number is detected as the result of a floating point operation (nonzero result with true exponent smaller than the minimum exponent, see Figure D-6) and (2) loss of accuracy is detected (delivered result differs from what would have been computed if the exponent range was unbounded – i. e., cannot be accurately represented as a denormalized number). Consider the case of floating point multiplication as an example. Let the first SP source operand have a mantissa of 1.01, with biased exponent e_{min}=1 (unbiased exponent of -126) and the second SP source operand have a mantissa of 1.0 with a biased exponent of 60 (unbiased exponent of -67). The result of a multiplication with infinite precision arithmetic would be a mantissa of 1.01 with actual (unbiased) exponent of -193 (=-126-67). Since this exponent is smaller than the smallest exponent possible in SP, the number is tiny, and since the number is so tiny that it cannot be accurately represented as a denormalized number, loss of accuracy also ocurs, underflows will be signaled. The delivered SP result would be a SP zero, and the underflow flag would be set. Note that the SEP format can accommodate this exponent, and thus the result of an SEP operation would not signal the underflow exception. In that case, the correct result is delivered. If the first operand of the SP multiplication has the same value as before, but the second operand has a biased exponent of 96 (actual exponent of -31), the result of an infinite-precision multiplication has a mantissa of 1.01 and an actual exponent of -157. The SP result consists of a denormalized number (i.e., tiny) with a mantissa of 0.0000000000000000000000000000001 and biased exponent of 0. Note that the denormalization process results in loss of accuracy, and therefore the the underflow flag will be set. Finally, if the second source operand has a biased exponent of 120 (actual exponent of -7), then the resulting mantissa with infinite precision would be 1.01 as before, with an actual exponent of -133. The SP result is again denormalized (tiny) with a mantissa of 0.000000101 and a biased exponent equal to 0. Note that there is no loss in accuracy due to the normalization (no lost significant bits), and thus the underflow flag will not be set. The de-

livered result is the correct SP denormalized number.

5. Inexact: The inexact exception is signaled if the delivered result differs from what would have been obtained with infinite-precision arithmetic. For instance, the examples of underflow shown above deliver numerically inexact results, and thus set the inexact flag. Another example is the case where floating point numbers are rounded up or down.

D.1.4 DSP96002 Floating Point Storage Format in the Data ALU

The data ALU is designed to accommodate mixed-precision operands in a common format. To this end, a common DP storage format is used internal to the data ALU. SP and DP numbers from memory are automatically converted to the internal format by means of a format conversion unit, the operation of which is transparent to the user.

The bit-level DP representation internal to the ALU is illustrated in Figure D-5. The internal floating point format is 96 bits wide and consists of the following fields:

1. Sign of the mantissa (S) bit 95.
2. SP Unnormalized tag (U) bit 94. The U-TAG is set when writing a floating-point register with a denormalized SP number. Cleared otherwise.
3. DP Unnormalized tag (V) bit 93. The V-TAG is set when writing a floating-point register with a denormalized DP number (denormalized SEP in the DSP96002). Cleared otherwise.
4. Unused bits (Z) bits 75 through 92 and bits 0 through 10. These bits read as zeros, and should be written with zeros for future compatibility. They are cleared by floating-point moves and operations.
5. Biased Exponent (e) bits 64 through 74. Since the internal ALU format is DP, there are 11 exponent bits, with an integer bias of 1023 ($3FF). The encodings of the exponent are identical to the ones explained in the section on memory storage formats (Appendix D.1.2) .
6. Integer bit (i or b_0) bit 63. The integer bit is explicitly presented in the internal representation as bit 63 and is the integer part of the mantissa.
7. Fraction – bits 11 through 62. This is a 52-bit field representing the fractional part of the mantissa (only 31 are used by the DSP96002 floating-point ALU). The remaining bits are set to zero by floating-point ALU operations or single-precision floating-point moves. Since the internal format is DP, the fraction consists of 52 bits. The data ALU arithmetic, however, only provides results in either SP or SEP. The SEP format is the same as the DP format, except for the size of the fraction. The SEP fraction consists of only 31 bits. Consequently, the lower 21 or 29 bits of the fraction will consist of zeros when representing SEP or SP arithmetic results, respectively. When DP values are moved from memory to the data ALU, the fraction contains all 52 significant bits. However, when using these DP values as operands in a floating-point arithmetic operation, only 31 bits of the 52-bit fraction are used; the remaining bits are simply truncated. The SEP format is shown in Figure D-7.

D.1.5 Data ALU Block Diagram

The block diagram of the data ALU is shown in Figure D-8. The data ALU consists of four main parts:

1. Register file and automatic conversion unit: All operations in the data ALU are register-based: operands as well as results of data ALU operations are read from and written to registers. A

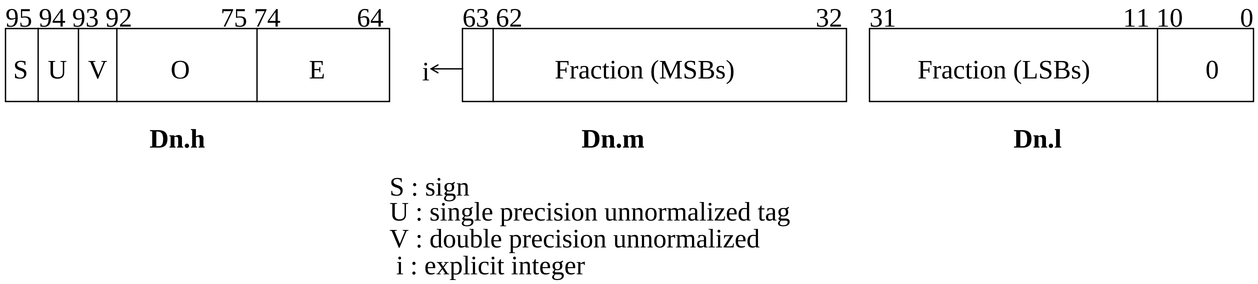


Figure D-5. DP Format in the Data ALU

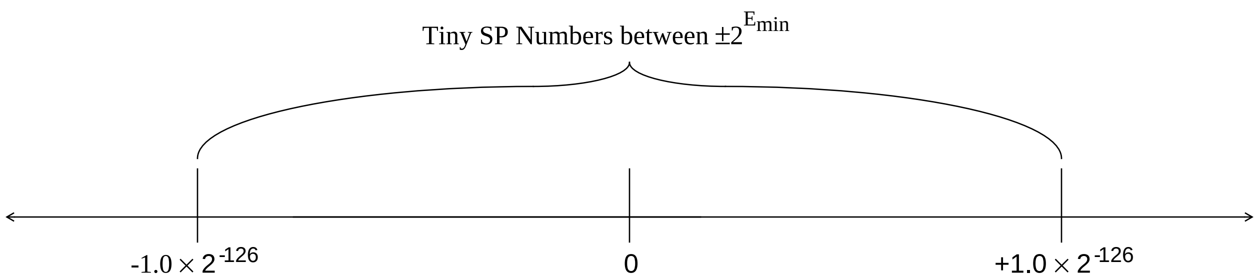


Figure D-6. Tiny Numbers

register file consisting of 10 96-bit registers for storage of floating-point numbers is available for that purpose. An automatic conversion unit converts the floating point storage format in memory to the internal DP format when moving operands and/or results from/to memory.

2. Multiply unit: A full IEEE floating-point multiply unit, delivering either a SP or SEP result in one instruction cycle.
3. Adder/Subtracter unit: A full IEEE floating-point adder/subtracter unit, which can deliver the sum as well as the difference of two operands in the same instruction cycle, to either SP or SEP.
4. Special function unit: A special function unit provides various logic functions, as well as support for divide and square root in terms of an initial seed for a fast convergent divide and square root algorithm.
5. Controller and arbitrator: A controller/arbitrator supplies all of the control signals necessary for the operation of the data ALU.

The data ALU uses the SEP format for all of its operations: the results are automatically rounded to either SP or SEP. All of the rounding modes specified by the IEEE standard are supported. These rounding modes are:

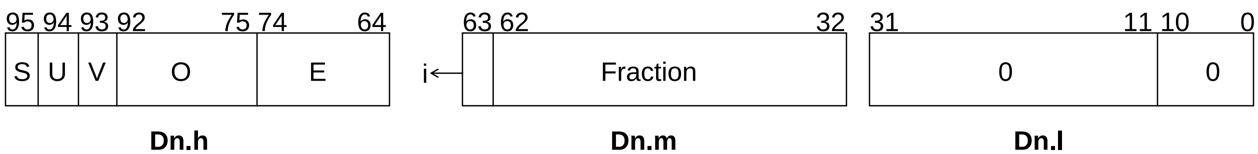


Figure D-7. SEP Format in the Data ALU

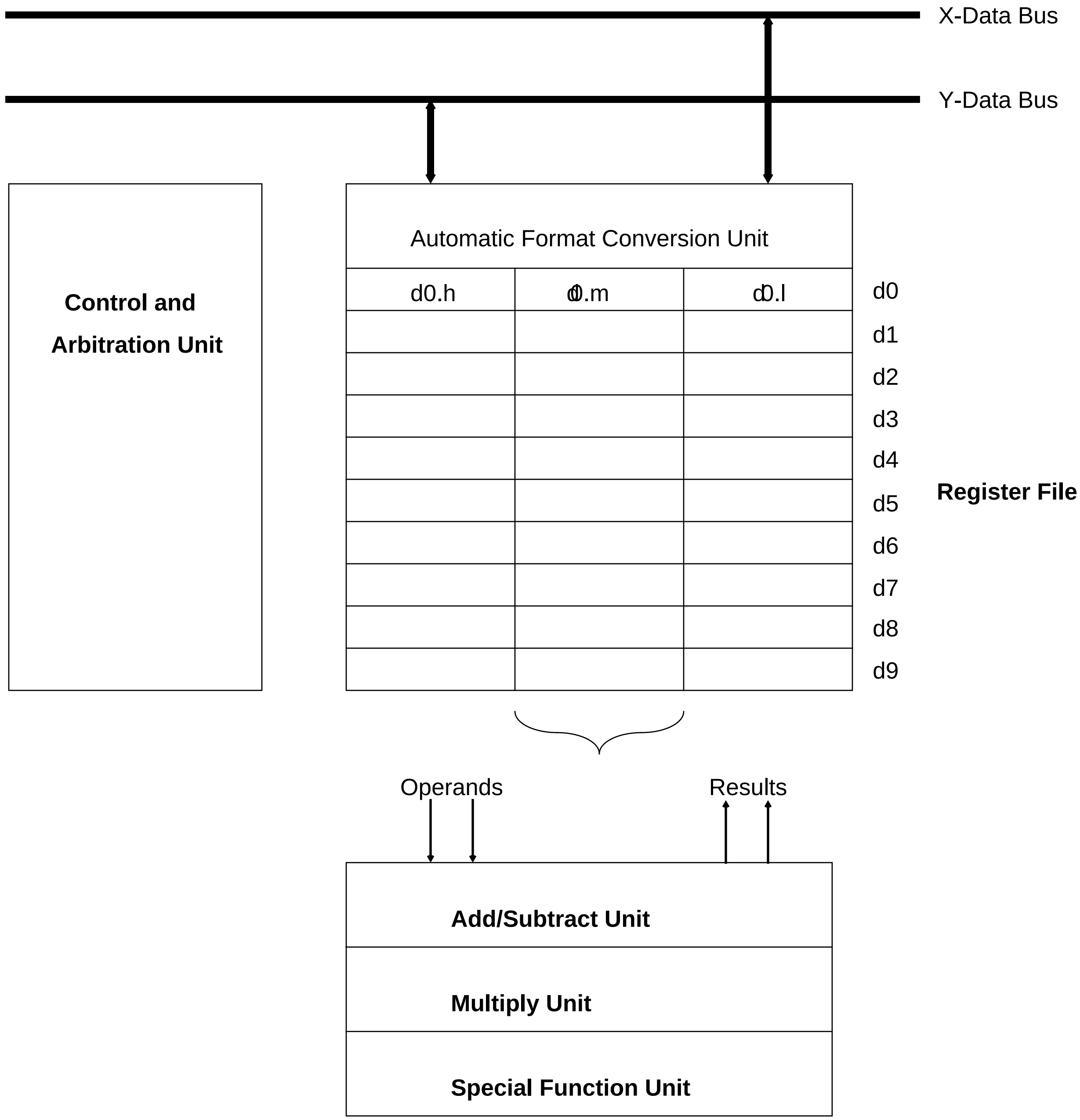


Figure D-8. The Data ALU

1. Round to nearest (even): a convergent rounding mode, designed to deliver results without a rounding bias. In this case the infinite-precision result is rounded to the finite-precision result which is closest. In the case of an absolute tie, the infinite-precision result is rounded to the "nearest even" finite precision result, as is illustrated in Table D-2.
2. finite precision result which is closest to zero. Clearly, results are rounded up in this mode when negative, and down when positive.
3. Round to plus infinity: results are always rounded in the direction of plus infinity, i.e. "up".

4. Round to minus infinity: results are always rounded in the direction of minus infinity, or "down".

D.1.5.1 Register file and automatic format conversion unit

The general-purpose register file consists of ten 96-bit registers named d0..d9, as shown in Figure D-9. Each 96-bit register accommodates the DP internal floating point storage format. Each 96-bit register is obtained by the concatenation of three 32-bit registers dn.h:dn.m:dn.l. The registers dn.h, dn.m, and dn.l can be accessed as individual registers by MOVE operations and integer and logic instructions, as is further described in Appendix D.2.

Infinite-precision result	**Rounded result (to p=4 bits for example)**
1.000 11100000....	1.001 (round up)
1.000 01100000....	1.000 (round down)
1.000 10000000....(absolute tie)	1.000 (round down)
1.001 10000000....	1.010 (round up)

Table D-2. Example of the Round to Nearest Mode.

The registers d0..d7 are general-purpose registers in the sense that MOVE instructions and data ALU operations do not differentiate between them. They are used for data ALU source and destination operands for most of the data ALU instructions. They can be used as operands for MOVE operations as well as for data ALU operations in the same instruction cycle: dual source operands are allowed. They can not be used as dual destinations in the same instruction cycle.

The registers d8 and d9 are auxiliary registers which can be used for temporary data storage. Their main purpose is to allow a fast, four-cycle radix-2, decimation in time FFT butterfly kernel, though their use is certainly not limited to this application. d8 and d9 can only be used as source operands in multiply operations and MOVE instructions, and can only be written as destinations of MOVE instructions.

The format conversion unit provides automatic format conversion from/to the SP and DP memory storage

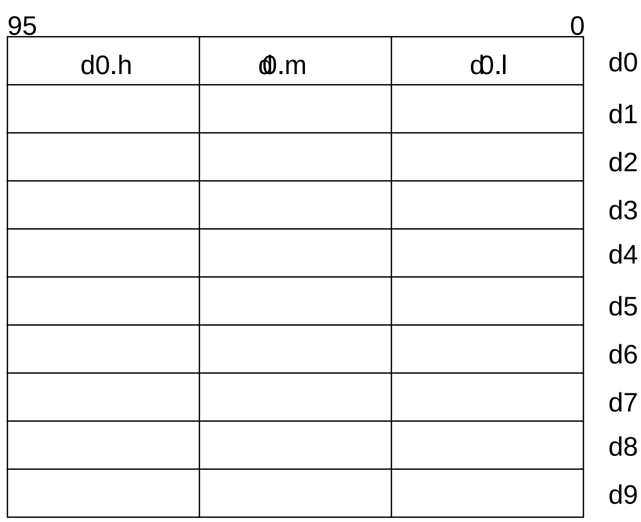
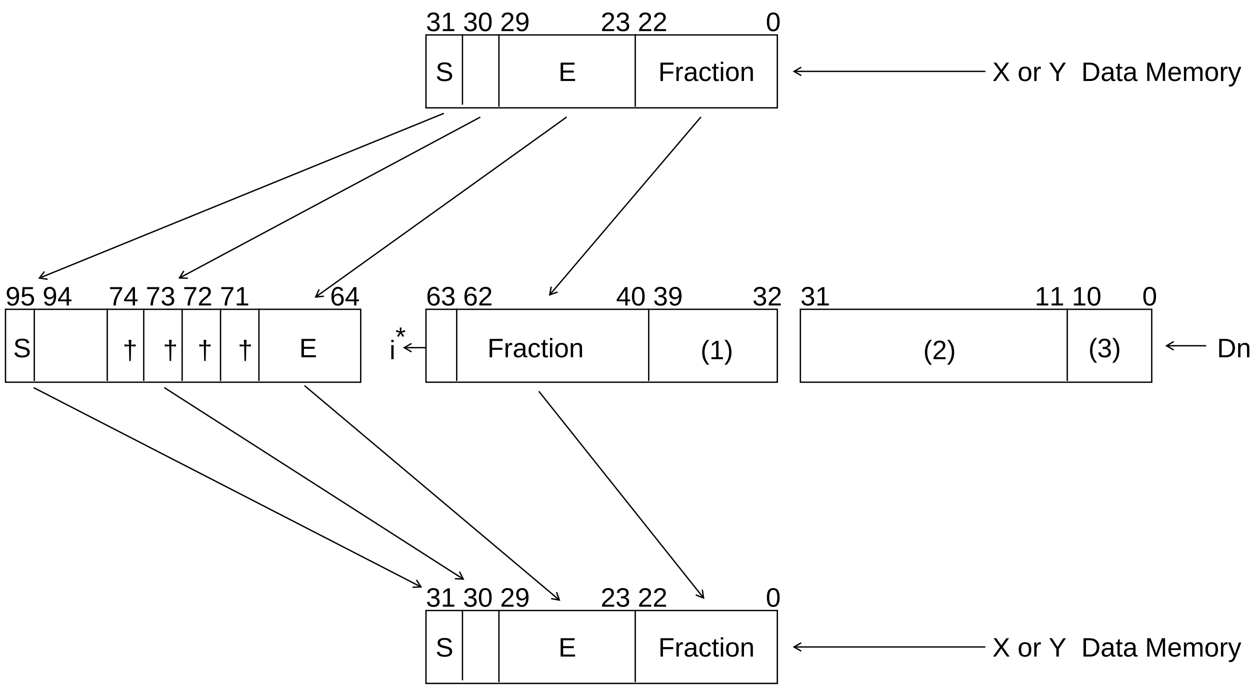


Figure D-9. The Data ALU’s Register File



Notes: * – i = 1 when normalized
i = 0 when unnormalized

† – When NaN bits 71, 72, 73 = 1
When not NaN Bit 74 ↔ Bit 30
Bits 73, 72, 71 are complement of Bit 74.

(1) – Bits 32-39 are nonzero when the register contains a SEP floating point result or a DP floating point number.
Bits 32-39 are zero when the register contains a SP floating point number.

(2) – Bits 11-31 are only nonzero when the register contains a DP floating point number.

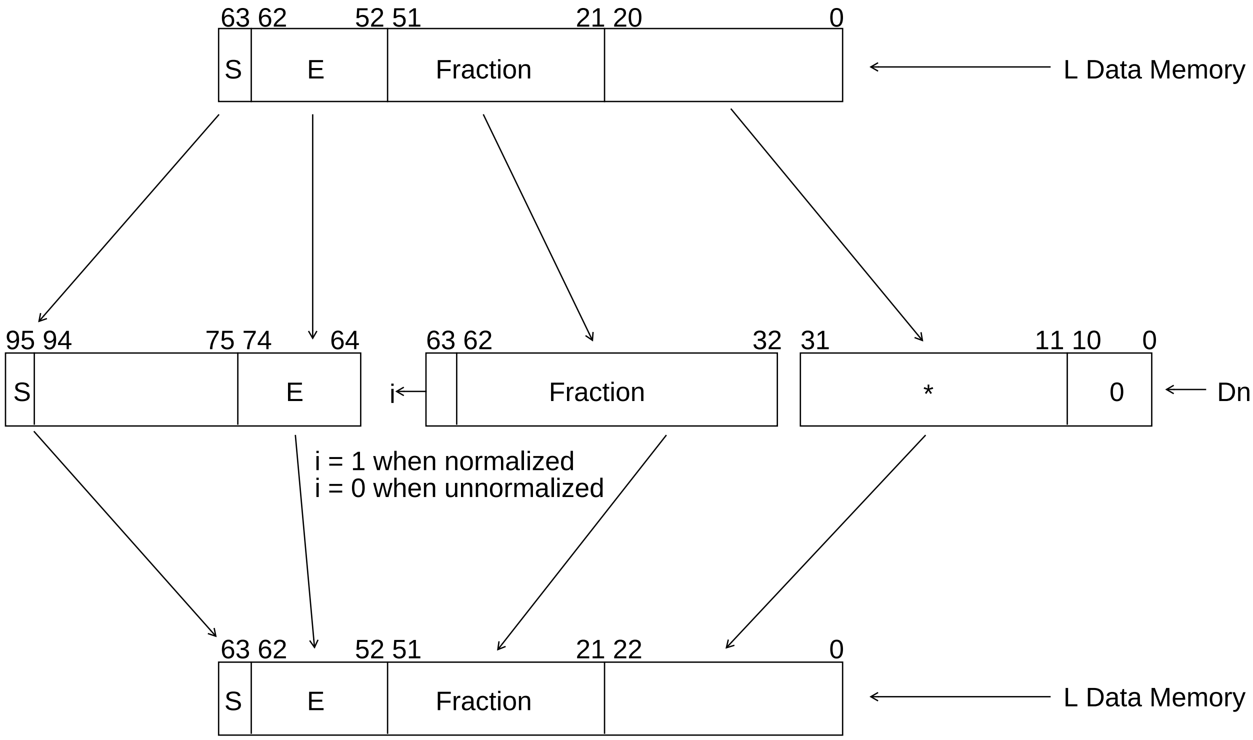
(3) – Bits 0-10 are always zero when representing a floating point number.

Figure D-10a. Automatic Format Conversion – Single Precision

formats to/from the DP storage format in the data ALUs register file. The conversion is depicted in Figure D-10 and is done in a transparent fashion.

When moving SP numbers into the data ALU, the 52-bit fraction of the DP internal format is written with the 23-bit fraction of the source in its most significant bits, and the implicit integer bit is made explicit. The remaining bits of the fraction are set equal to zero. If the number in question is denormalized (exponent = e_{min} and the first bit of the mantissa = 0), the U tag is set. In the non-IEEE "flush to zero" mode (indicated by the FZ bit in the Status Register), the number is considered zero when used as an operand for floating-point operations, although the contents of the register are not changed. In the IEEE mode, the number is "corrected" when used as an operand for floating point calculations, at the expense of extra cycles introduced for normalization.

The 8-bit exponent of the SP source is translated into an 11-bit exponent by copying the 7 least significant bits of the source exponent into the seven least significant bits of the destination. The most significant bit of the 8-bit exponent of the source is copied to the most significant bit of the exponent of the destination. The



* – Bits 11-31 (in Dn) or 0-20 (in L memory) are zero when the register contains an SEP result.

Figure D-10b. Automatic Format Conversion – Double Precision

remaining 3 bits of the destination's exponent are set if the number is an NaN or infinity, otherwise they are the inverted MSB of the source's exponent. Inverting the MSB effectively changes the bias from 127 to 1023.

When moving single precision numbers from the data ALU to memory, the above process is reversed, as shown in Figure 10-a. The 23 most significant bits of the fraction are moved to the 23 fraction bits of the destination. Note that the contents of the data ALU register may have more than 23 fractional bits if it was the result of a previous DP move or SEP arithmetic operation; in this case, the fraction is simply truncated.

The MSB of the 11-bit exponent of the source in the data ALU is moved to the MSB of the exponent of the destination. The 7 LSBs of the exponent of the source are copied to the seven LSBs of the exponent of the source. Note that if the source was not a SP number (result of a DP move or a SEP arithmetic operation), an incorrect exponent may be moved. Therefore, care must be taken to always round results to SP before moving them to memory as single precision numbers.

When moving DP numbers from memory, the 52 bit fraction of the source is moved to the 52 bit fraction of the destination, and the implicit integer bit is made explicit. If the number is denormalized, the V tag is set. Again, extra cycles may be required when a denormalized number is used as an operand, depending on the FZ bit in the SR. The 11-bit exponent of the source is copied to the 11-bit exponent of the destination.

When moving DP numbers from the data ALU to memory, the above process is reversed, as shown in Fig-

ure D-10b. Note that the 52-bit fraction may actually consist of zeros (21 or 29) if the number in question was the result of a SEP arithmetic or a SP move. SEP arithmetic result precision can only be retained in memory by using DP moves.

D.1.5.1.1 FLOATING-POINT MOVES TO/FROM DATA ALU REGISTERS

The following sections deal with the case where a write (move in) is followed by a read (move out) without any floating-point operation being actually performed on the Data ALU register (save-restore procedure). The only way to provide correct results for save-restore procedures is to perform the same type of moves when writing and then reading the register (SP write followed by SP read or DP write followed by DP read).

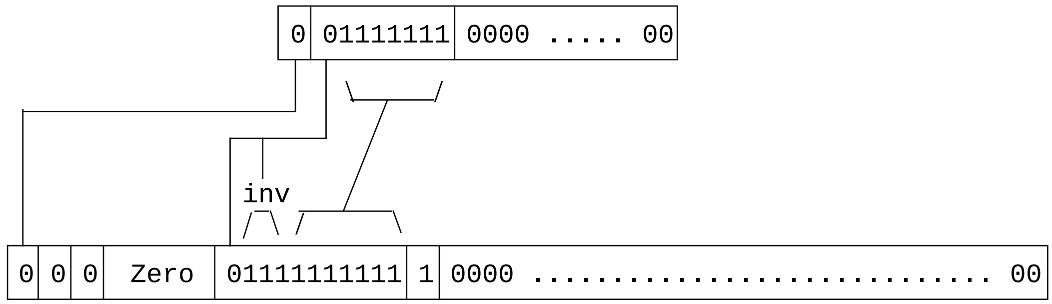
D.1.5.1.1.1 Single Precision (SP) Move Of A SP Normalized Number

This section describes what happens when a 32-bit source (normalized single precision) is writen by a single precision floating-point move and the data is stored in a Data ALU floating-point register D0-D9. Following the above operation, the Data ALU register will be read first by a single precision and then by a double precision floating-point move.

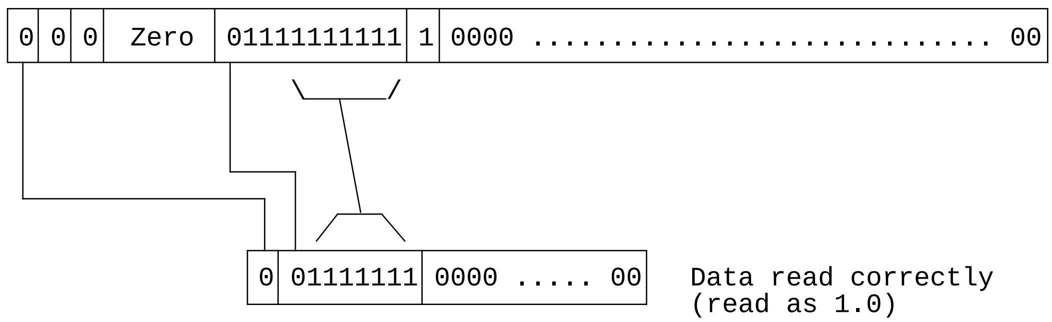
```
        - 32-bit data from source is 3F800000 (= +1.0)
             - exp      = 7F (8 bit bias)
             - mantissa = 000000 (the hidden bit is one)

        - data stored in the register
             - e     = 3FF   (correct representation with 11-bit bias)
             - I     = 1     (the number is normalized so hidden bit is 1)
             - U-TAG = 0     (cleared; the number can be used in computations
                              without adding extra cycles for normalization,
                            since it is a normalized number)        - fraction
     =   00...00         - mantissa = 1.00...00
```

One should notice that both single and double precision floating-point moves out of the register will produce correct results in this case.

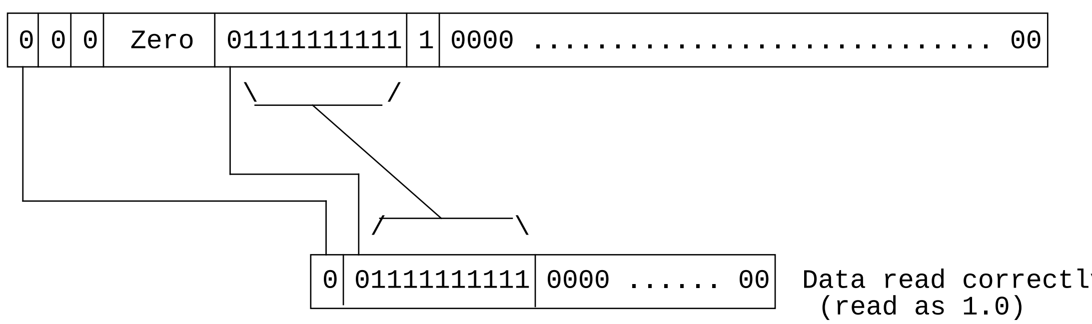
SP move into the register



SP read of the register



DP read of the register



D.1.5.1.1.2 SP Move Of A SP Denormalized Number

This section describes what happens when a 32-bit source (denormalized single precision) is writen by a single precision floating-point move and the data is stored in a Data ALU floating-point register D0-D9. Fol-

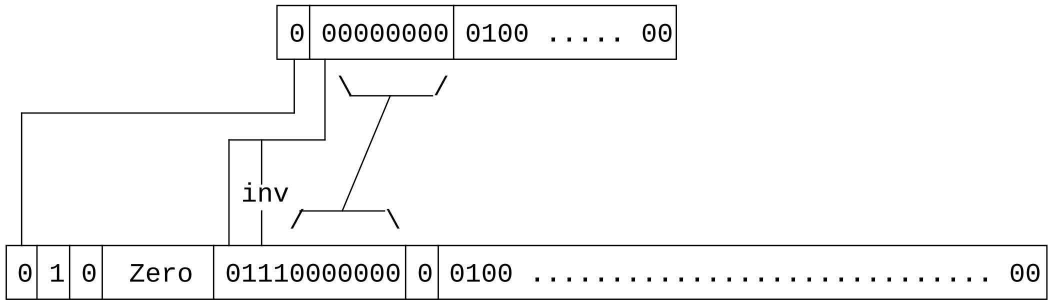
lowing the above operation, the Data ALU register will be read first by a single precision and then by a double precision floating-point move.

```
- 32-bit data from source is $00200000 (= +2**(-128))
     - exp      = $00     (8 bit bias)
     - mantissa = $200000 (the hidden bit is zero)

- data stored in the register
     - e     = 380   (incorrect representation with 11-bit bias; the
                      correct representation would be 37F)
     - I     = 0     (the number is unnormalized)
     - U-TAG = 1     (set; the number cannot be used in computations
                      without adding extra cycles for normalization,
                      since it is unnormalized)
     - fraction = 40000000
     - mantissa = 0.010...00
```

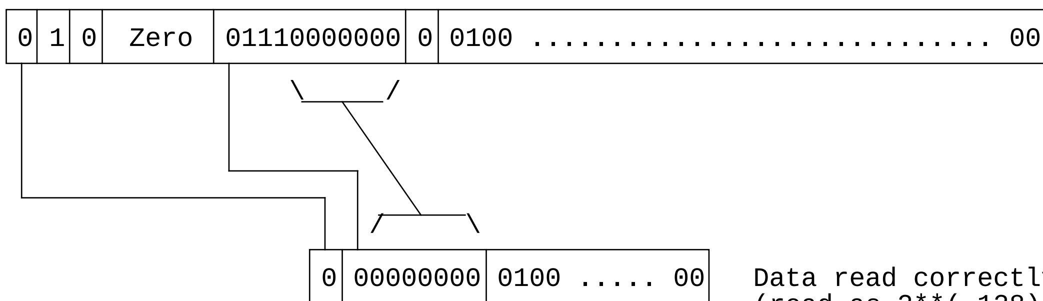
In this last case, the U-TAG tells us that an operation using this operand will first add extra cycles to normalize it. However, an SP move will render the correct result since the "formatting" scheme presented in Section 5.5 chooses the right bits. One should notice that a double precision floating-point move that reads

the register will yield the wrong data in this case.

SP move into the register

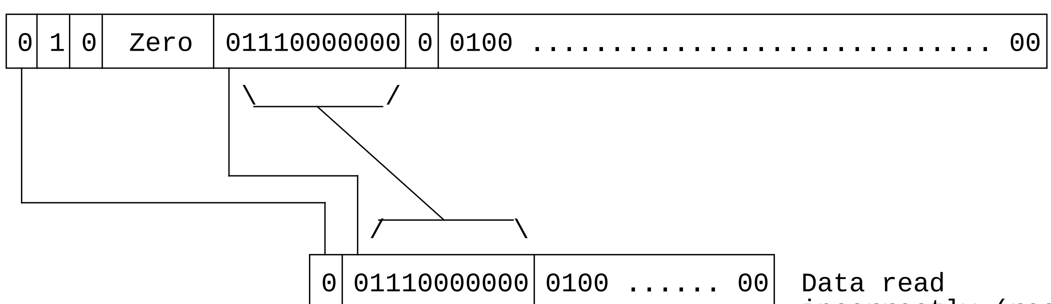


SP read of the register



Data read correctly
(read as 2**(-128))

DP read of the register



Data read
incorrectly (read
as 1.01x2**(127))

D.1.5.1.1.3 Denormalized Numbers In Double Precision (DP)

This section describes what happens when a 64-bit source (denormalized double precision) is writen by a double precision floating-point move and the data is stored in a Data ALU floating-point register D0-D9.

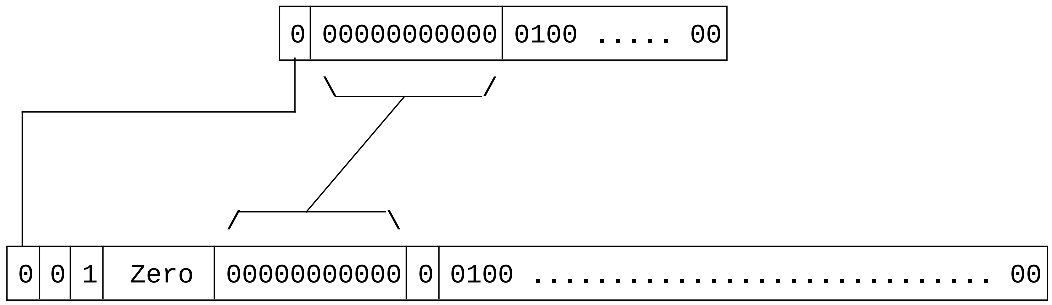
Following the above operation, the Data ALU register will be read first by a single precision and then by a double precision floating-point move.

The denormalized double precision data is stored in the Data ALU register with the V tag set and the exponent set to $000 (always). The V-TAG set indicates that floating-point multiply operations will require extra cycles to wrap it ("normalize") before using it as operand. Double precision moves will yield correct results when reading the denormalized DP from the register to memory (the V-TAG will also be set when single extended denormalized result is obtained from a Data ALU operation).

Here is an example of a double precision denormalized number:

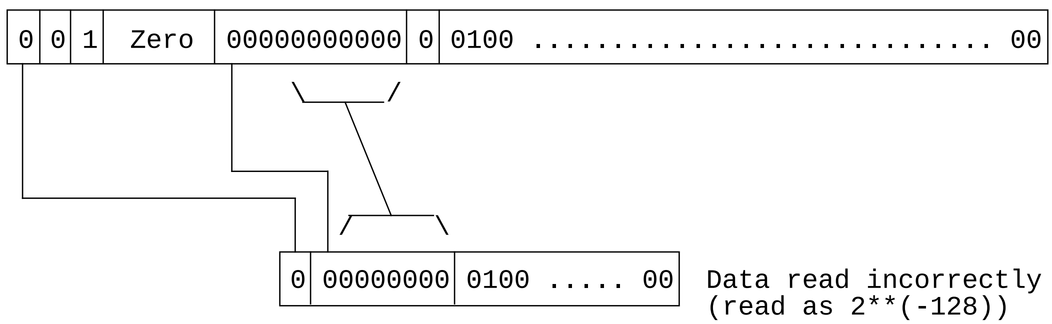
```
- 64 bit data from source is 0004000000000000 (= 2**(-1024))
     - exp      = $000 (11-bit bias)
     - mantissa = $4000000000000 (the hidden bit is zero)

- data stored in the register
     - e     = 000   (correct representation with 11-bit bias)
     - I     = 0     (the number is not normalized)
     - U-TAG = 0     (cleared; the number can be used in computations
                      as it is by the adder)

     - V-TAG = 1     (set; it indicates a denormalized number in DP,
                      requiring extra cycles for denormalization in
                      multiply operations)
     - fraction = 40000000
     - mantissa = 0.010...00
```

DP move into the register



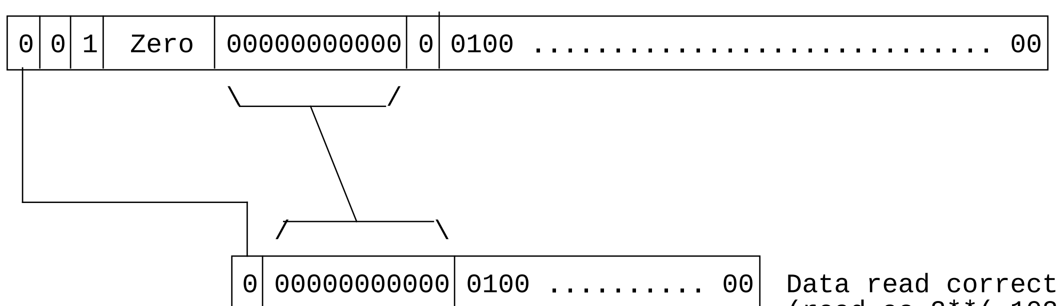
NOTE THAT THE V TAG IS SET IN THIS CASE

SP read of the register



Data read incorrectly
(read as 2**(-128))

DP read of the register



Data read correctly
(read as 2**(-1024))

D.1.5.1.1.4 Floating-Point Moves Summary

Figure C-1 summarizes what will be the result of a data move into an Data ALU register followed by a read of the same register, depending on the data range and the type of moves.

MOVE IN TYPE	EXPONENT RANGE (UNBIASED)	INPUT DATA	TAGS U	TAGS V	MOVE OUT TYPE	MOVE OUT RESULT
SP	e= 128 Fraction= .0xx...xx	signaling NaN (SNAN) written as DP SNAN read as SNAN (see Note 1)	0	0	SP	CORRECT
					DP	CORRECT
SP	e= 128 Fraction= .1xx...xx	non signaling NaN (QNAN) written as DP QNAN read as QNAN (see Note 2)	0	0	SP	CORRECT
					DP	CORRECT
SP	e= 128 Fraction= .000...00	infinity in SP written as DP infinity read as infinity (all formats)	0	0	SP	CORRECT
					DP	CORRECT
SP	-127<e< 128	normalized (all formats)	0	0	SP	CORRECT
					DP	CORRECT
SP	-150<e<-126	denormalized in SP	1	0	SP	CORRECT
					DP	WRONG
DP	e= 1024 Fraction= .0xx...xx	signaling NaN (SNAN) written as DP SNAN read as SNAN (see Notes 1,3)	0	0	SP	CORRECT
					DP	CORRECT
DP	e= 1024 Fraction= .1xx...xx	non signaling NaN (QNAN) written as DP QNAN read as QNAN (see Note 2)	0	0	SP	CORRECT
					DP	CORRECT
DP	e= 1024 Fraction= .000...00	infinity in SP written as DP infinity read as infinity (all formats)	0	0	SP	CORRECT
					DP	CORRECT
DP	-127<e< 1024	no SP representation normalized in DP/SEP	0	0	SP	WRONG
					DP	CORRECT
DP	-127<e< 128	normalized (all formats)	0	0	SP	TRUNC
					DP	CORRECT
DP	-150<e<-126	denormalized in SP normalized in DP/SEP	0	0	SP	WRONG
					DP	CORRECT
DP	-1023<e<-149	no SP representation normalized in DP/SEP	0	0	SP	WRONG
					DP	CORRECT
DP	-1054<e<-1022	denormalized (in DP/SEP)	0	1	SP	WRONG
					DP	CORRECT

Figure C- 1. Floating-Point Moves Summary

Note 1 The xx...xx pattern for the signaling NaNs indicates any NON-ZERO bit pattern.

Note 2 The xx...xx pattern for the non-signaling NaNs indicates any bit pattern.

Note 3 If a register is written with a SNAN using a double precision floating-point move and then the same register is read using single precision floating-point move the result will be a single precision SNAN (if the first 23 bits of the fraction are a non-zero pattern) or single precision infinity (if the first 23 bits of the fraction are a zero pattern).

Note 4 The case when both U-TAG = 1 and V-TAG = 1 is reserved for future use.

D.1.5.1.2 RESULTS OF DATA ALU FLOATING-POINT OPERATIONS

This section describes how the Data ALU floating-point operation results are stored in the Data ALU registers.

All DSP96002 Data ALU floating-point operations are executed in single extended precision, using single extended precision input operands, and return single extended or single precision results in double precision format. The results are formatted in double precision before being stored in the Data ALU registers. When performing a DP move into a register and then using that register in a DSP96002 SEP floating point operation, the mantissa of the operand will be first truncated to a SEP value, as the hardware is unable to operate on more than 32 mantissa bits. Figure C-2 explains how a DP register is used as operand for a SEP operating unit (adder/multiplier).

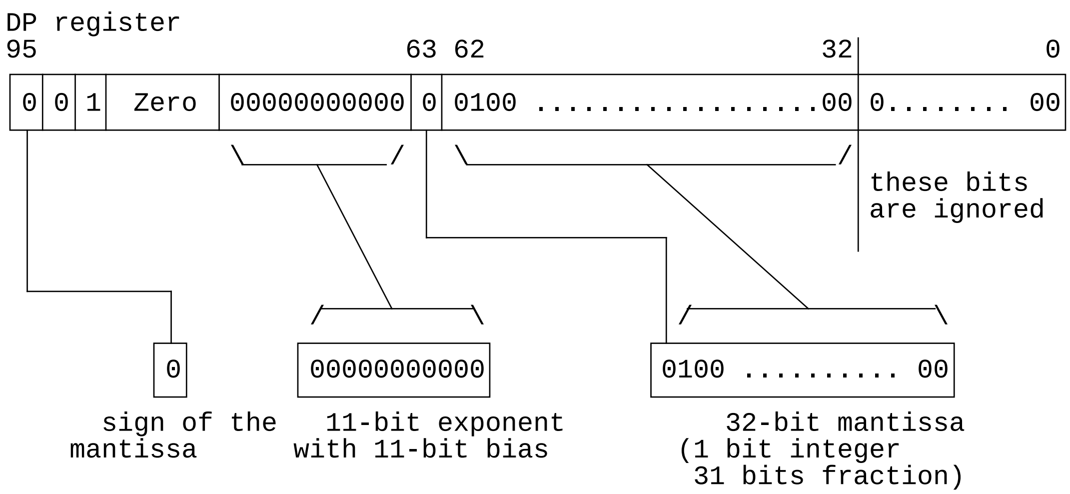


Figure C -2. DP operand in a SEP operation

The 11-bit exponent used by the SEP operating units is identical with the exponent of the original DP number loaded into the register (both have the same bias, namely $3FF). This means that the number can be used in computations directly, assuming that the least significant 21 mantissa bits are zero, otherwise a round towards zero occurs because the mantissa is truncated to 32 bits (21 bits of the 52-bit DP mantissa are ignored).

D.1.5.1.2.1 Results Rounded To SP

Data ALU results are rounded to SP when the instruction is specified with the .S suffix (FMPY.S, FADD.S, etc.).

D.1.5.1.2.2 Results Rounded To SP That Are Normalized

If the Data ALU operation result was rounded to SP and the rounded result may be represented as a normalized single precision floating-point number, the result will be stored in normalized DP format that may be read out by single and double precision moves without errors or truncation.

D.1.5.1.2.3 Results Rounded To SP That Are Denormalized

If the Data ALU operation result was rounded to SP and the rounded result must be represented as a denormalized single precision floating-point number, the result will be stored in unnormalized DP format with the U tag set and the I bit cleared, and it may be read out by single precision moves without errors or truncation. If the register is read by a double precision move, completely incorrect data will be obtained; see the discussion in Section C.3.2.

In this case, before the result is delivered, an additional Data ALU execution cycle is required in which the SEP mantissa is shifted right the required number of places for correct rounding to SP.

The presence of unnormalized numbers in DP format will add one dummy cycle followed by an additional cycle for each unnormalized DP operand to any Data ALU operation that uses them as input. During the additional cycle the unnormalized operand (U-TAG=1) is normalized, however the register itself will not be modified.

D.1.5.1.2.4 Results Rounded To SEP

Data ALU results are rounded to SEP when the instruction is specified with the .X suffix (FMPY.X, FADD.X, etc.).

D.1.5.1.2.5 Results Rounded To SEP That Are Normalized

If the Data ALU operation result was rounded to SEP and the rounded result may be represented as a normalized single extended precision floating-point number, the result will be stored in normalized DP format that may be read out by double precision moves without errors or truncation.

If the result stored in the register is read with a single precision move, two situations may occur:

1. The SEP exponent is in the range of the normalized SP exponent: the data read will be rounded to SP by truncating the SEP mantissa; this is equivalent to IEEE round towards zero.
2. The SEP exponent is not in the range of the normalized SP exponent: the data read will not have the right exponent. The correct value should have been infinity, zero or a denormalized SP, but the move instruction does not provide it.

D.1.5.1.2.6 Results Rounded To SEP That Are Denormalized

If the Data ALU operation result was rounded to SEP and the rounded result must be represented as a denormalized single extended precision floating-point number, the result will be stored in normalized DP format with the V tag set and I bit cleared, and it may be read out by double precision

moves without errors or truncation. If the register is read by a single precision move, completely incorrect data will be obtained; see the discussion in Section C.3.3 (double precision and single extended precision numbers have the same exponent bias).

D.1.5.1.2.7 Data ALU Results/Move Compatibility Summary

Figure C-3 summarizes what happens when Data ALU operation results of a certain range is stored in the destination register, and the register is read by a certain kind of move.

All cases where "move out type"=SP and "move out result"=WRONG can be corrected by rounding in the instruction (using the .S option). The case where "move out type"=SP and "move out result"=TRUNC can also be corrected by using the .S option.

ROUND TO	EXPONENT RANGE BEFORE ROUND (UNBIASED)	DATA ALU OPERATION RESULT	TAGS		MOVE OUT TYPE	MOVE OUT RESULT
			U	V		
SP	NaN operand or invalid op	non signaling NaN (QNAN) written as DP QNAN e=7FF mantissa=1.11...11	0	0	SP	CORRECT
					DP	CORRECT
SP	127<e	infinity (overflow) written as DP infinity e=7FF mantissa=1.00...00	0	0	SP	CORRECT
					DP	CORRECT
SP	127<e< 128	normalized (all formats)	0	0	SP	CORRECT
					DP	CORRECT
SP	-150 < e < –126	denormalized (in SP)	1	0	SP	CORRECT
					DP	WRONG
SP	e < –149	zero (underflow)	0	0	SP	CORRECT
					DP	CORRECT
SEP	NaN operand or invalid op	non signaling NaN (QNAN) written as DP QNAN e=7FF mantissa=1.11...11	0	0	SP	CORRECT
					DP	CORRECT
SEP	1023<e	infinity in SP and SEP written as DP infinity e=7FF mantissa=1.00...00	0	0	SP	CORRECT
					DP	CORRECT
SEP	127<e< 1024	infinity in SP normalized in SEP	0	0	SP	WRONG
					DP	CORRECT
SEP	-127<e< 128	normalized (all formats)	0	0	SP	TRUNC
					DP	CORRECT

Figure C -3. Data ALU Results/Move Compatibility Summary (Continued)

ROUND TO	EXPONENT RANGE BEFORE ROUND (UNBIASED)	DATA ALU OPERATION RESULT	TAGS		MOVE OUT TYPE	MOVE OUT RESULT
			U	V		
SEP	-150< e < -126	denormalized in SP normalized in SEP	0	0	SP	WRONG
					DP	CORRECT
SEP	-1023< e < -149	zero in SP normalized in SEP	0	0	SP	WRONG
					DP	CORRECT
SEP	-1054< e< -1022	zero in SP denormalized in SEP	0	1	SP	WRONG
					DP	CORRECT
SEP	e< -1053	zero in SP zero in SEP (underflow)	0	0	SP	CORRECT
					DP	CORRECT

Figure C- 4. Data ALU Results/Move Compatibility Summary

D.1.5.2 Multiply unit

The multiply unit consists of a hardware multiplier, an exponent adder, and a control unit, as shown in Figure D-11. The multiply unit accepts two 44 bit input operands for floating point multiplications, each consisting of a sign bit, eleven exponent bits, the explicit integer bit, and 31 fractional bits. Note that for full double precision operands, as obtained by double precision MOVEs, the least significant 8 bits of the fraction are simply truncated. Multiply operations occur in parallel with and independent of data moves over the X and Y data buses.

The hardware multiplier accepts the two 32-bit mantissas (integer bit + 31 bit fraction), and delivers a 64 bit result, as shown in Figure D-12. This result is automatically rounded to a 32-bit mantissa for SEP arithmetic or a 24 bit mantissa for SP arithmetic, as specified by the instruction opcode. The result is stored into the mantissa portion of the destination register.

The exponent adder takes the two unsigned (i. e., biased) operand exponents, adds them together, and subtracts the bias, resulting in an 11-bit biased exponent which is stored in the exponent part of the floating point format in the destination register, as depicted in Figure D-13.

D.1.5.3 Adder/Subtracter Unit

The adder unit is depicted in Figure D-14, and consists of a barrel shifter and normalization unit, an add unit, a subtract unit, an exponent comparator and update unit and a special function unit. The adder/subtracter unit accepts 44-bit floating point operands, and delivers 44-bit results. The adder/subtracter operations deliver the sum and the difference of the same two floating point operands in a single instruction cycle. In addition, the barrel shifter used for mantissa alignment in floating point additions and subtractions is used for executing multibit shifts. The adder/subtracter operates in parallel with and independent of data moves over the X and Y data buses.

The add unit is a high speed 32-bit adder, used in all floating-point non-multiply operations. For floating point operations, 32-bit mantissas (1 integer bit and 31 fractional bits) are first "aligned" for floating point addition

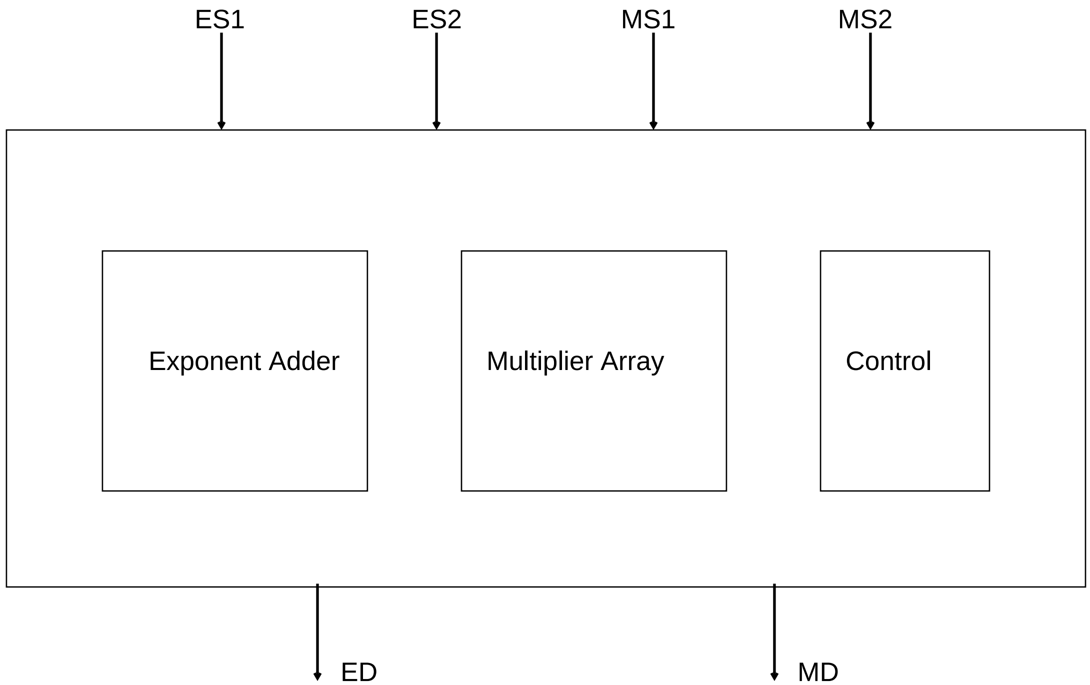


Figure D-11. The Multiply Unit

in the barrel shifter and normalization unit, after which they are added in the add unit. The result is then rounded to 32-bits for SEP results, and to 24 bits for SP results, as indicated by the instruction opcode. The type of rounding implemented depends on the rounding mode bits in the MR register. The rounded result is stored in the middle portion (mantissa) of the destination register.

The subtract unit is a high speed 32-bit adder/subtracter, used in all floating-point non-multiply operations and in all fixed point operations delivering a 32-bit result. For floating point operations, 32-bit mantissas (1 integer bit and 31 fractional bits) are first "aligned" for floating point subtraction in the barrel shifter and normalization unit, after which they are subtracted in the subtract unit. The result is then rounded to 32-bits for SEP results, and to 24 bits for SP results, as indicated by the instruction opcode. The type of rounding implemented depends on the rounding mode bits in the MR register. The rounded result is stored in the middle portion (mantissa) of the destination register for floating point operations, and in the low portion for fixed-point operations. This is shown in Figure D-15.

The barrel shifter/normalization unit is used for the alignment of the two operand mantissas, needed for addition of two floating point numbers. The barrel shifter is a 32-bit left-right multibit shifter, which is also used in fixed point arithmetic and logic shifting operations with a 32-bit result. For the addition of two floating point operands, the barrel shifter receives the exponent difference of the two operand exponents from the exponent comparator and update unit, and uses this difference to align the mantissas for addition. For example, if the biased exponent of the first floating point operand equals 10 and the biased exponent of the second floating point operand equals 13, the mantissa of the first operand will be right shifted by three positions (3 bit shift).

The exponent comparator and update unit consists of an 11 bit subtracter, which compares the two exponents of floating point operands, and delivers the difference to the barrel shifter for mantissa alignment. The largest of the two exponents is delivered to the exponent update unit. The exponent update unit may update

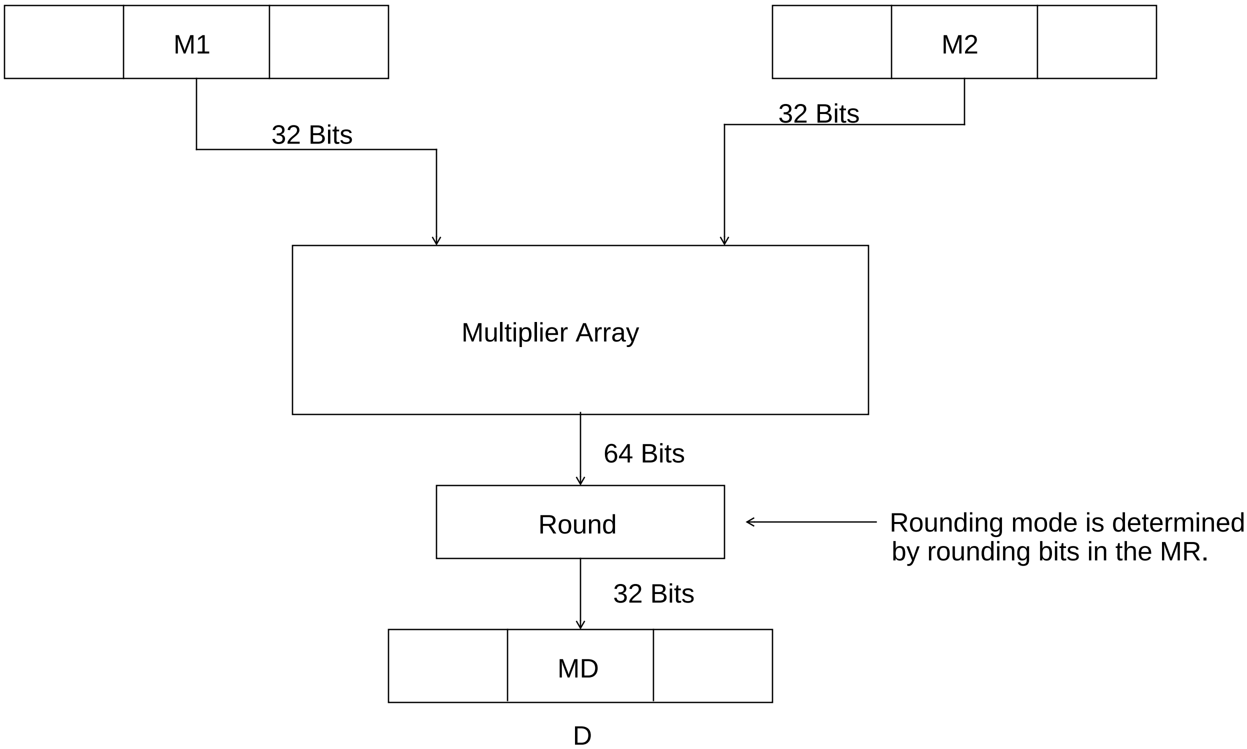


Figure D-12. The Multiply Unit

this exponent for normalization of the result, after which the exponent (biased) is stored in the high portion of the destination register. This is depicted in Figure D-16.

For example, if the mantissa of the first operand in a floating point addition is 1.010...0, with biased exponent of 10, and the mantissa of the second operand is 1.000...0000, with biased exponent of 13, the exponent comparator simply delivers the difference (=3) to the barrel shifter, the first operand's mantissa is aligned to 0.001010...0, the two mantissas are added to deliver 1.001010...0, and the result (biased) exponent equals 13. The postnormalization unit does not need to postnormalize the result in this case.

If the first operand's mantissa is 1.010...0 with biased exponent of 13 and the second operand's mantissa is 1.000...0 with biased exponent of 13, the exponent difference is zero and the barrel shifter does not need to realign the mantissas. The result after addition is now equal to 10.010...0, which needs to be postnormalized by adding one to the result exponent. The exponent update unit sets the result exponent (biased) equal to 14 and the result mantissa is 1.0010...0.

Finally, if the first operand's mantissa in a floating point subtraction is 1.010...0 with biased exponent of 10, and the second operand's mantissa is 1.00...0 with a biased exponent of 10, the result mantissa after subtraction is -0.010...0. This is not normalized, and the postnormalization unit subtracts two from the exponent. The result mantissa is -1.000...0 with a biased exponent equal to 8.

D.1.5.4 Special Function Unit

The special function unit consists of a logic unit and a divide and square root unit. The logic unit is further described under the fixed point (integer) operations.

The divide and square root unit supports execution of the divide and square root algorithms. These algorithms are iterative, and require an initial approximation or "seed". The FSEEDD and FSEEDR instructions

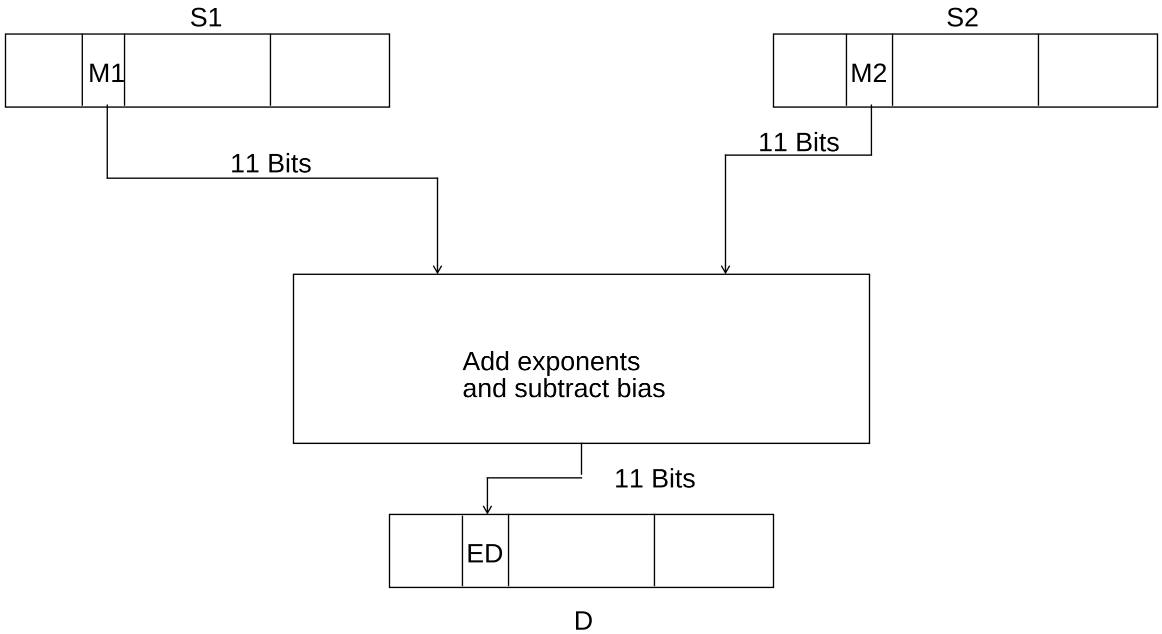


Figure D-13. The Exponent Adder

provide an initial approximation to 1/x and sqrt(1/x), as is described in Appendix A.

D.1.5.5 Controller and Arbtrator Unit

The controller and arbitrator (CA) unit supplies control signals to the processing units of the data ALU and register file, and is responsible for the full implementation of the IEEE standard. Its operation is determined by the flush-to-zero (FZ) bit in the status register (SR), which determines whether or not denormalized numbers are treated as defined by the standard. In the flush-to-zero mode, all denormalized input operands are treated as zeros (although their original contents are preserved), and denormalized results are set equal to zero ("flushed-to-zero"). In the flush-to-zero mode, no additional cycles are required for the normalization of denormalized numbers as they are treated as zeros. In the IEEE mode, the standard for treatment of denormalized numbers is correctly and fully implemented. However, operations on denormalized numbers can not be performed in a single instruction cycle, except for operations done in the floating point adder when the operand is a denormalized number in SEP. The controller and arbitrator is responsible for providing the correct sequence that deals with such situations.

When denormalized numbers are detected as input operands in IEEE mode, the CA unit adds one extra cycle for entering the IEEE mode procedure. Next, one additional cycle is added for each denormalized input operand. These cycles are used to normalize the input operand. The original value of the operand in the source register is not affected. During the IEEE mode procedure all activity of the chip is suspended until the input operands have been normalized. When denormalized output results are detected, the IEEE mode procedure is entered (one additional instruction cycle) and each result is again normalized (another cycle).

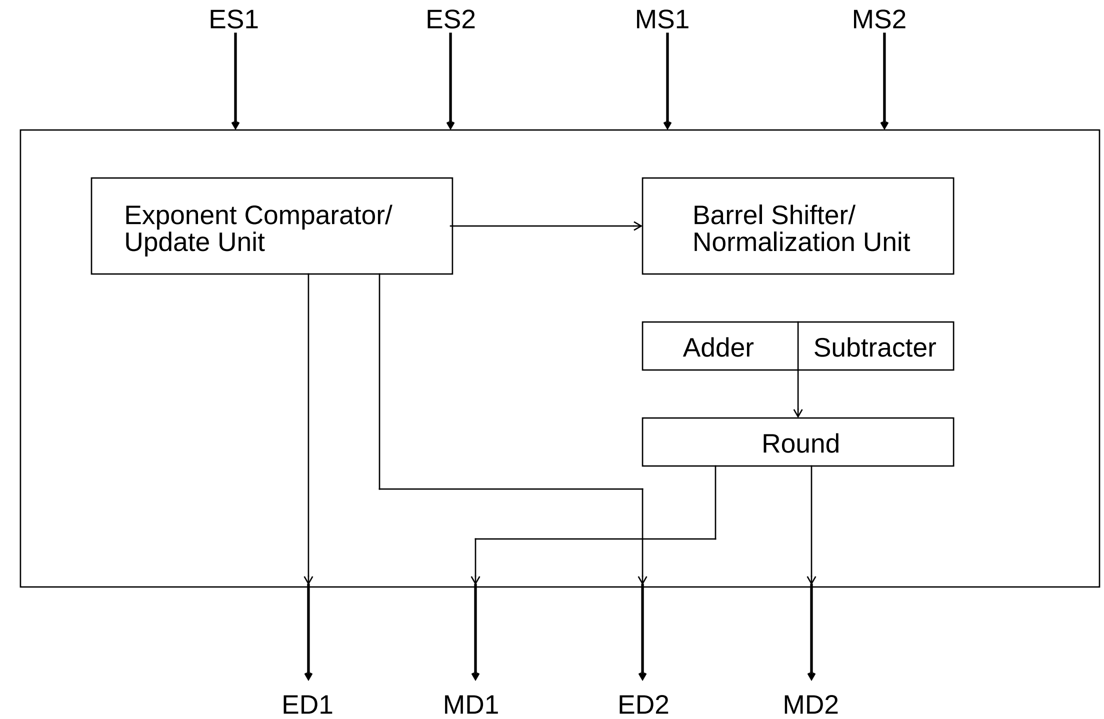


Figure D-14. The Adder/Subtracter

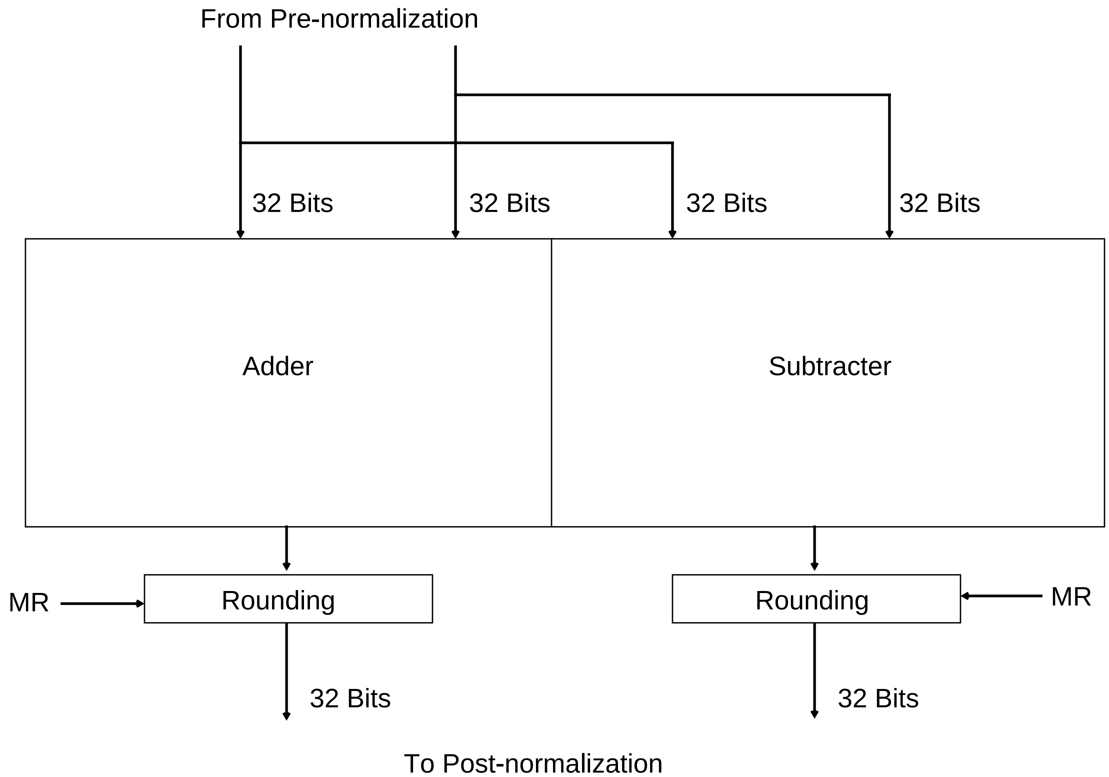


Figure D-15. The Adder/Subtracter Unit

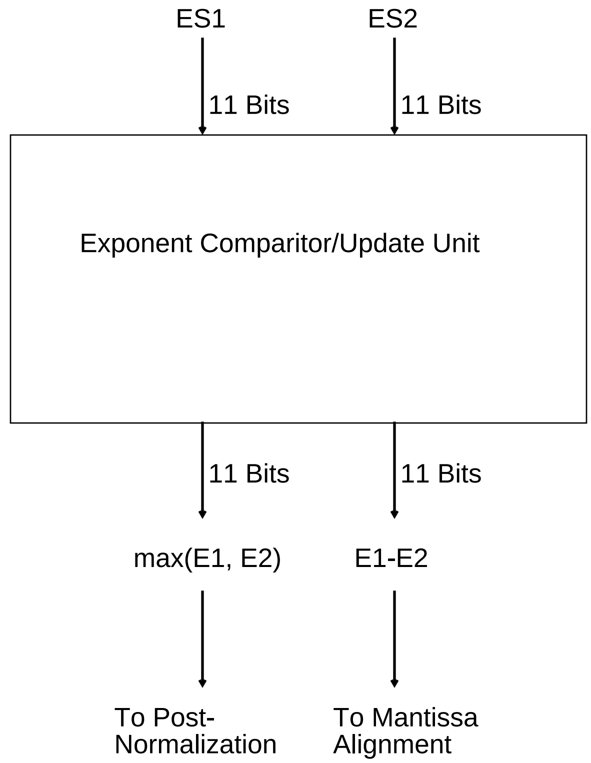


Figure D-16. Exponent Comparator/Update Unit.

D.2 FIXED-POINT NUMBER STORAGE AND ARITHMETIC

D.2.1 General

Integer operand sizes are defined as follows:

1. **Byte:** 8 bits long
2. **Short word:** 16 bits long
3. **Word:** 32-bits long
4. **Long word:** 64 bits long

The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction.

D.2.2 Integer Storage Format in Memory

The DSP96002 supports four integer memory data formats:

1. Signed word integer: 32-bits wide, two’s complement representation. This storage format can be used in either X and/or Y data memory space.
2. Signed Long Word Integer: 64 bits wide, two’s complement representation. This storage format can only be used in long (L) data memory space.
3. Unsigned Word Integer: 32-bits wide with unsigned magnitude representation. This storage format can be used in either X and/or Y data memory space.

4. Unsigned Long Word Integer: 64 bits wide with unsigned magnitude representation. This storage format can only be used in long (L) data memory space.

Long type integers can be moved to and from the data ALU register file. However, long integers can not be directly used as input operands to data ALU operations. Long integers can however be results of data ALU operations.

D.2.3 Integer Storage Format in the Data ALU

There are thirty 32-bit registers in which can contain integer words. However, data ALU arithmetic operations use the low portion of the register files as word source and destination operands. Long word integers are only generated as results of integer arithmetic operations and are never used as source operands.

D.2.4 Integer Arithmetic

The integer arithmetic operations use the same arithmetic units in the data ALU as the floating point operations. These units consist of:

1. **Adder**: The subtract unit in the adder/subtracter unit described above is used for integer add and subtract operations. It accepts two 32-bit integer operands from the low portions of the data ALU source registers and delivers a 32-bit result in the low portion of the destination register.
2. **Multiplier:** The multiplier in the multiply unit described above also performs the integer multiplications. It accepts two 32-bit operands in the low portion of the data ALU source registers, and delivers a 64-bit result in the low and middle portions of the destination register. Both signed and unsigned multiplications are supported.
3. **Logic Unit:** The logic unit is responsible for the logical operations AND, ANDC, OR, ORC, EOR, NOT, ROR. In addition, it performs the bit field manipulation instructions SPLIT, SPLITB, JOIN, JOINB, EXT, and EXTB. The logic unit operates on 32-bit operands located in the low portions of the data ALU registers. Results are also stored in the low portion of the destination.
4. **Barrel Shifter:** The barrel shifter in the normalization unit used for mantissa alignment in floating point additions is also available for performing multibit shifts on integer (fixed-point) data. Both single and multibit arithmetic shifts left and right and logical shifts left and right are supported.

Order this document by DSP96002UM/AD

Order this document by
DSP96002UMAD/AD

DSP96002

Addendum to

DSP96002 Digital Signal Processor User Manual

THE DSP96002 INSTRUCTION CACHE and 32-BIT TIMER/EVENT COUNTER

FOREWORD

This document is an addendum to the DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual (DSP96002UM/AD). It describes significant new features added to the DSP96002 functionality, including an instruction cache, a new integer mode of operation and new parallel integer instructions to support it, data ALU register file decoupling, enhancements to the OnCE, and a new timer/event counter.

The revised DSP96002 is fully compatible with its predecessor. Special mode bits in various registers allow the user to access the new features.

This addendum describes each of the features in detail. Section 2 introduces the Instruction Cache. Section 3 describes the new integer mode and its associated parallel integer instructions. Section 4 presents presents the single precision mode. Section 5 introduces enhancements to the On Chip Emulation (OnCE) module. Section 6 describes the new timer/event counter modules, Section 7 discusses some additional changes to support the timer operation, and APPENDIX A gives the details of additions to the DSP96002 instruction set.

1 SUMMARY OF NEW DSP96002 FEATURES

Instruction Cache

The functionality of the 1K internal Program Memory (PRAM) has been extended by allowing it to operate as a 4K byte (1K word) "real-time" Instruction Cache. The term "real-time" emphasizes the high degree of controllability available within the Instruction Cache permitting deterministic results. After reset the cache is disabled and the Program Memory functionality is identical to the DSP96002 described in the DSP96002 User's Manual.

May, 1993

Integer Mode

The integer performance on the DSP96002 has been doubled with the introduction of the Integer Mode (IM). The Integer Mode of operation significantly improves the performance of integer algorithms and supports four new parallel arithmetic operations:

- integer signed multiply and add (MPYS//ADD)
- integer signed multiply and subtract (MPYS//SUB)
- integer unsigned multiply and add (MPYU//ADD)
- integer unsigned multiply and subtract (MPYU//SUB)

Single Precision Mode

The newly added Single Precision Mode (SPM) of operation improves the efficiency of the Data ALU register file. This new operating mode gives the user access to two Data-ALU register files: a 10 floating-point register file (d0.h..d9.h, d0.m..d9.m) and a 10 integer register file (d0.l..d9.l). If the program uses only single-precision MOVE operations and floating-point operations that yield single-precision results, then the two register files are completely decoupled - thus effectively doubling the amount of registers available to the data ALU.

OnCE Enhancement

The support for development and debugging of multiprocessor systems has been improved by the addition of a new OnCE[1] feature that permits simultaneous start of the program execution for any number of processors, regardless of the code they are executing. Different processors may be stopped at different points in the code they are executing, and then their activity may be restarted synchronously and simultaneously.

Timer/Event Counter Modules

This addendum also describes the two identical and independent timer/event counter modules newly featured on the DSP96002. The timers can use internal or external clocking and can interrupt the processor after a number of events (clocks) specified by a user program, or it can signal an external device after counting internal events. Figure 1 shows the DSP96002 block diagram revised to include the timers.

New a$\overline{\text{WR}}$ and b$\overline{\text{WR}}$ (Write Strobe) Pins

The DSP96002 features two new outputs, aW/$\overline{\text{R}}$ and bW/$\overline{\text{R}}$, which support a glueless interface to external SRAMs. They are active-low when the DSP96002 is the bus master, and three-stated when the DSP96002 is not the bus master. They are asserted during external memory write cycles to indicate that the address lines A0-A32, S1, S0, $\overline{\text{BS}}$, $\overline{\text{BL}}$, and R/$\overline{\text{W}}$ are stable. The output data goes to the data bus after $\overline{\text{WR}}$ is asserted. $\overline{\text{WR}}$ is three-

1. OnCE is a trademark of Motorola Inc.

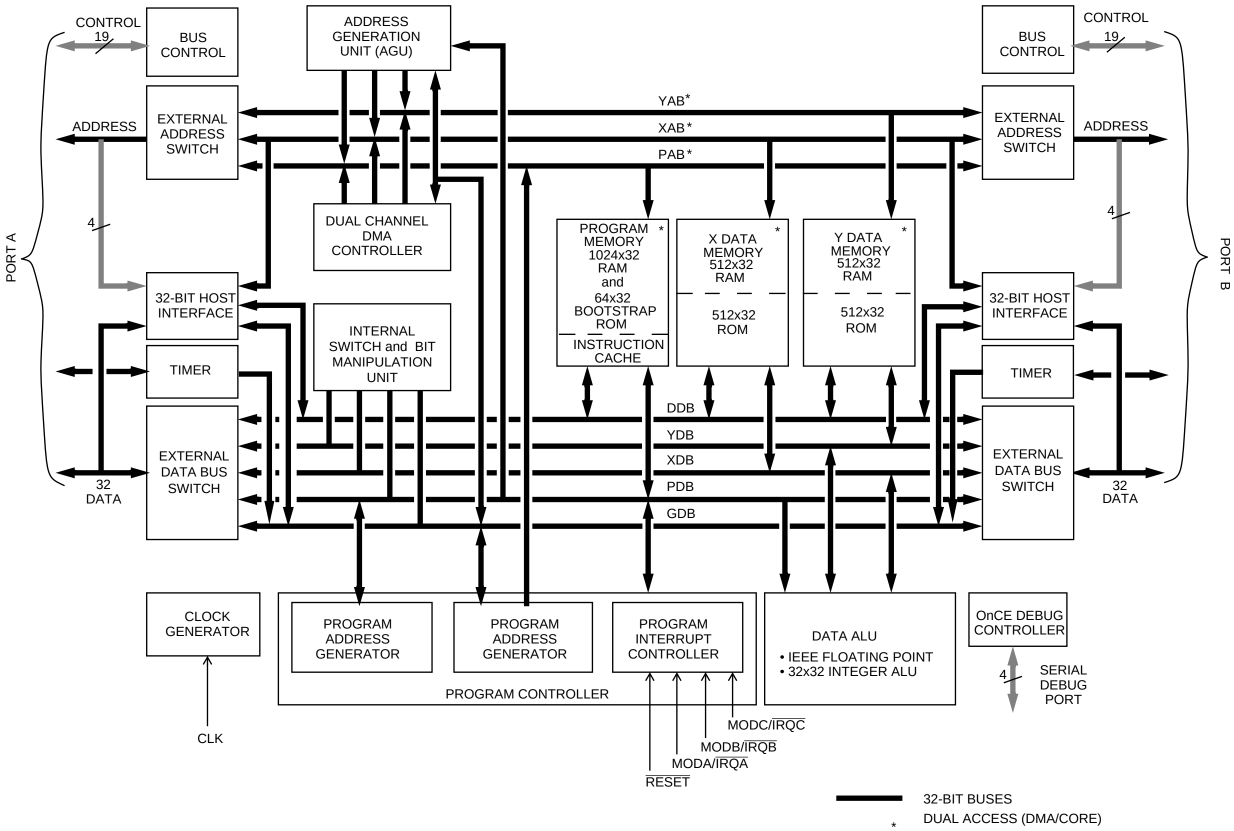


Figure 1 - DSP96002 Block Diagram

stated during hardware reset, requires a weak external pull-up resistor, and can be connected directly to the $\overline{WE}$ pin of a static RAM. The DSP96002 diagram shown in Figure 6 on page 29 includes the new write strobe pins.

The timings and functionality of $\overline{TS}$ and R/$\overline{W}$ remain unchanged, so that existing configurations may still be used. From a logical standpoint, $\overline{WR}$ = ($\overline{TS}$ or R/$\overline{W}$).

2 INSTRUCTION CACHE

2.1 INTRODUCTION

The instruction cache may be viewed as a buffer memory between the main (external and probably slow) memory, and the fast CPU. The cache is used to store frequently used program instructions and it offers an increase in throughput by eliminating the time required to access the instruction words on the external bus.

Reduced external bus activity maintains single-cycle program memory access, while allowing the use of a low cost, slow external program memory. It also frees the processor's memory expansion port for other tasks such as data moves, DMA transfers, Host Interface data moves, etc.

The DSP96002 instruction cache is a “real-time” cache and therefore it has no inherent penalty on a cache miss. In other words, if there is a cache hit, it takes exactly one bus cycle to fetch the instruction from the on-chip cache - like fetching any other data from an on-chip memory. If there is a cache miss, it behaves exactly as a “normal” instruction fetch, as if it were fetching any other data from that external memory.

Furthermore, a “real-time” instruction cache allows the user to declare some code areas as time-critical and therefore “non-replaceable”. Six new instructions have been added to the instruction set, allowing the user to lock sectors of the cache, and to flush the cache contents under software control.

The following list is a summary of the instruction cache features:

- 1K, 32-bit wide, on-chip instruction cache
- Switching from PRAM mode to cache mode is software controlled
- Fully compatible with the DSP96002 PRAM mode when cache is disabled
- 8-way, fully associative, sectored cache
- One-word transfer granularity
- Least recently used (LRU) sector replacement algorithm
- User transparent - no user management required
- No additional wait states on cache miss
- Global cache mode, allowing normal cache operation
- Individual sector locking, preventing replacement of sector contents, but allowing updating of new entries within sector
- Global cache flush in software, allowing immediate clearing of the contents of the Instruction Cache
- Global PRAM mode, allowing compatibility with original architecture (including PRAM disabled and DMA to/from program memory)
- Full cache observability (tags, valid-bits, LRU, locked sectors) with OnCE commands in debug mode.

2.2 INSTRUCTION CACHE STRUCTURE

A cache controller has been added to the existing Internal Program RAM. Figure 2 shows a block diagram of the instruction cache controller.

The internal program RAM contains 1024 32-bit words, logically divided into eight 128-word cache sectors. In a similar way, the external program memory is virtually divided into 128-word sectors. The term “sector” is used, rather than “block”, since a sectored-cache distinguishes between “sectors” which are the basic replacement units, and “blocks” which are the basic transfer units. In our case a “block” is a 32 bit word so that one can use the terms “block” and “word” interchangeably.

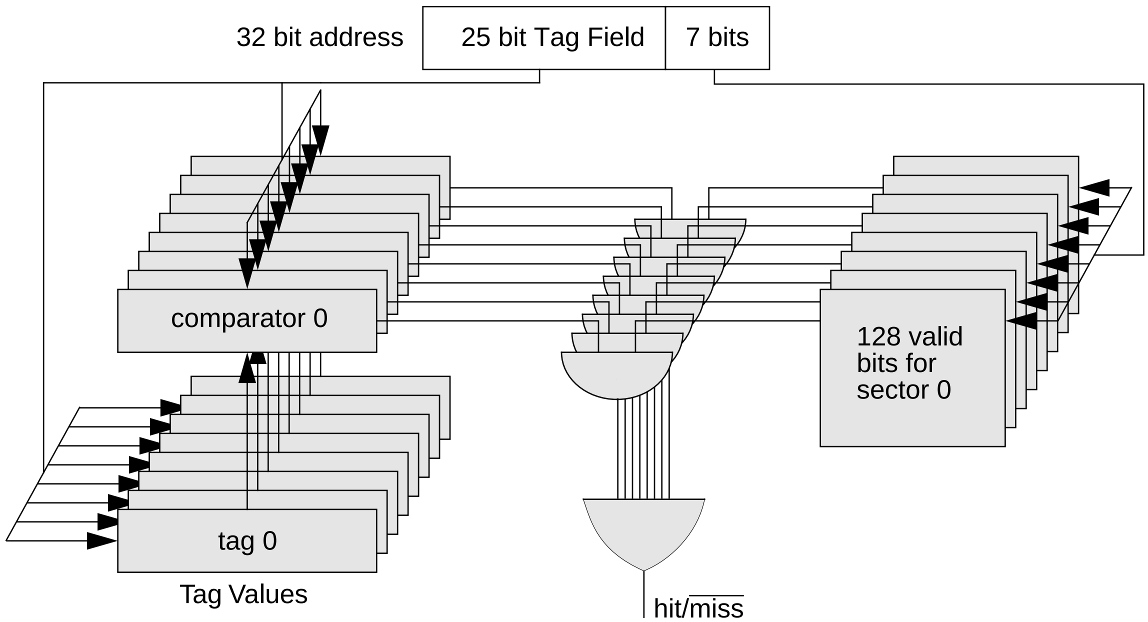


Figure 2 - Cache Controller Block Diagram

Since there are 8 sectors of 128 words each, in the internal program RAM, the 32 bit address is divided into the following two fields:

- 7 LSBs for the word displacement or offset in the sector
- 25 MSBs for the tag

The sector placement algorithm is fully associative so that each external program memory sector could be placed in any of the 8 internal program RAM sectors, essentially making it an eight-way fully associative cache.

A 25-bit tag is associated with every one of the eight internal program memory RAM sectors. When the cache controller searches for a tag equal to the tag field of the current address, it compares it to the eight tags in parallel using the eight comparators.

Each word in each cache sector is associated with a cache-word-valid-bit (or valid-bit), that specifies whether the data in that word has already been fetched from external memory and is therefore valid. There are a total of 1024 valid-bits, arranged as eight banks of 128 valid-bits each, one bank for every sector. Note that the valid-bits are not available to the user for direct use. They are cleared by the processor RESET to indicate that the PRAM context has not been initialized.

2.3 CACHE OPERATION

During cache operation each instruction is fetched on demand, only when it is needed. When the core generates an address for an instruction fetch, the cache controller compares the tag field portion of the physical address to the tag values currently stored in the tag register file. The tag values are the memory sector's 25 upper bits currently mapped into the cache.

When a tag match occurs (i.e. sector hit), then the valid-bit of the corresponding word in that cache sector is checked. If the valid-bit is set, meaning the word in the cache has already been brought to the cache and is valid, then that word is fetched from the cache location corresponding to the desired address. This event is called a cache hit, meaning that both the sector and its corresponding instruction word are present and valid in the instruction cache. The sector replacement unit (SRU) updates the used sector state according to the LRU algorithm.

When a tag match occurs, but the desired word is not valid in the cache (corresponding valid-bit cleared, indicating a word miss), then the cache initiates a read cycle from the external program memory. The fetched instruction is both sent to the core and copied to the relevant sector location. Then the valid-bit of that word is set. All of this is done in parallel with normal execution and does not require any additional clock or memory cycles. The SRU updates the used sector state according to the LRU algorithm.

If no match occurs between the tag field and all sector tag registers, meaning that the memory sector containing the requested word is not present in the cache, the situation is called a sector miss, which is another form of a cache miss. When a sector miss occurs, the cache's SRU selects the sector to be replaced. The cache controller then flushes the selected cache sector by resetting all corresponding valid-bits, loads the corresponding tag register with the new tag field, and at the same time initiates an external instruction read cycle from the physical address requested by the core. When the data arrives from external memory, it is transferred to the core, and at the same time the cache controller copies it to the word location in the cache sector, specified by the 7 LSBs of the address, and sets the corresponding valid-bit. The SRU now updates the new situation in the sector replacement control unit.

In PRAM mode, when the cache is disabled, fetches are done internally or externally as in the first revision of the DSP96002.

2.4 INSTRUCTION CACHE PROGRAMMING MODEL

2.4.1 Operating Mode Register (OMR)

To support the cache operation, the Operating Mode Register (OMR) now features a new

Cache Enable (CE) bit. When the CE bit is cleared (0) the DSP96002 is in PRAM mode. When the CE bit is set, the processor is in cache mode. The CE bit is cleared during reset.

31–6	5	4	3	2	1	0
reserved	SPM	CE	DE	MC	MB	MA

Cache Enable

2.5 NEW INSTRUCTIONS

The DSP96002 instruction set features six new instructions discussed in the following paragraphs to support the instruction cache operation. APPENDIX A, starting on page 54, presents a full description for each of the new instructions.

2.5.1 PLOCK ea

The PLOCK instruction locks the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, then the instruction will load the least recently used cache sector tag with the 25 most significant bits of the specified address and then lock that cache sector. The instruction will update the LRU stack accordingly.

All memory-alterable addressing modes may be used for the effective address, but a short absolute address may not.

The PLOCK instruction is enabled only in cache mode. In PRAM mode it will cause an illegal instruction trap to be taken.

2.5.2 PUNLOCK ea

The PUNLOCK instruction unlocks the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, the instruction will load the least recently used cache sector tag with the 25 most significant bits of the specified address. The instruction will then update the LRU stack accordingly.

All memory-alterable addressing modes may be used for the effective address, but a short absolute address may not.

The PUNLOCK instruction is enabled only in cache mode. In PRAM mode it will cause an illegal instruction trap to be taken.

2.5.3 PLOCKR label or PLOCKR Rn

The PLOCKR instruction locks the cache sector to which the sum (PC + specified displacement) belongs. If the sum does not belong to any cache sector, then the instruction

will load the least recently used cache sector tag with the 25 most significant bits of the sum and then lock that cache sector. The instruction will update the LRU stack accordingly.

The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the address to be locked. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the 32-bit PC Relative Displacement.

The PLOCKR instruction is enabled only in cache mode. In PRAM mode it will cause an illegal instruction trap to occur.

2.5.4 PUNLOCKR label or PUNLOCKR Rn

The PUNLOCKR instruction unlocks the cache sector to which the sum (PC + specified displacement) belongs. If the sum does not belong to any cache sector, and is therefore definitely unlocked, nevertheless, the instruction will load the least recently used cache sector tag with the 25 most significant bits of the sum. The instruction will then update the LRU stack accordingly.

The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the address to be locked. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the 32-bit PC Relative Displacement.

The PUNLOCKR instruction is enabled only in cache mode. In PRAM mode it will cause an illegal instruction trap to occur.

2.5.5 PFREE

The PFREE instruction unlocks all the locked cache sectors.

The PFREE instruction is enabled in both the cache mode and the PRAM mode.

2.5.6 PFLUSH

The PFLUSH instruction will flush the whole cache, unlock all cache sectors, set the LRU stack, and tag registers to their default values.

The PFLUSH instruction is enabled both in cache mode and PRAM mode.

2.6 CACHE OPERATING MODES

There are two main operating modes for the DSP96002: cache mode and PRAM mode. They are both global, as they affect the internal program memory as a whole. When the processor is in cache mode, each separate sector could be in one of two operating modes: sector unlocked mode or sector locked mode. When the processor is in PRAM mode the PRAM itself could be in one of two modes: PRAM enabled or PRAM disabled. Both in cache mode and PRAM mode, the whole cache can be flushed by a software instruction.

The following list summarizes the DSP96002's operating modes:

Cache Mode (global):

- Sector Unlocked Mode (per sector)
- Sector Locked Mode (per sector)
- Cache flush (global)

PRAM Mode (global):

- PRAM Enabled (global)
- PRAM Disabled (global)
- Cache flush (global)

2.6.1 Cache Mode

In the cache mode, accesses to the storage area of the sectors are done implicitly by instruction fetches or by MOVEM instructions. DMA references to and from program memory space (in the cache or external) are disabled in hardware.

2.6.1.1 Sector Unlocked Mode

When the processor is in the sector unlocked mode, the program memory sector is configured as a regular cache sector. Sector replacement from that cache sector is allowed. The cache controller will decide when to replace an external memory sector that resides in a certain cache sector (sector miss), according to the cache controller LRU algorithm.

Unlocking a sector could happen in four different situations. In the first situation, the user unlocks a specific cache sector by using the PUNLOCK instruction. In the second situation, the user unlocks all the cache sectors in the internal program memory by using the PFREE instruction. In the third situation, the user unlocks all the cache sectors in the internal program memory as part of a cache flush by using the PFLUSH instruction. In the forth situation, a hardware reset unlocks all the cache sectors.

A locked sector can be unlocked by the new special instructions PUNLOCK and PUNLOCKR. Their operand is an effective memory address. The memory sector containing this address is allocated into a cache sector (if it is not already in a cache sector) and the

cache sector is unlocked. As a result of this sequence, the unlocked cache sector is placed at the top of the LRU stack, as it is the most recently used.

Unlocking a locked cache sector using the PUNLOCK or PUNLOCKR instructions does not affect the sector’s contents, its tag, or its valid-bits. If the specified effective address does not belong to one of the current cache sectors, a memory sector containing the specified address will be allocated into the cache, thereby flushing the least recently used cache sector. The unlocked cache sector will be placed at the top of the LRU stack and it will be readied for replacement by the LRU algorithm.

All of the locked sectors can be unlocked simultaneously using PFREE instruction, which provides a software reset of the locking mechanism. Unlocking the sectors using PFREE does not affect the sectors’ contents (instructions already fetched into the sector storage area), their valid-bits, their tag register contents or the LRU stack status.

The locked sectors could also be unlocked by the PFLUSH instruction as part of a total cache flush. Unlocking the sectors using PFLUSH clears all the sector’s valid-bits and sets the LRU stack and tag registers to their default values.

2.6.1.2 Sector Locked Mode

The sector locked mode is useful for latching some time critical code parts in the cache memory. The sector locked mode is set by the user to lock the memory sector that currently resides in the cache sector. When a cache sector is in sector locked mode the sector replacement unit (SRU) cannot replace it even if it is the least recently used sector (bottom of LRU stack).

The sector locked mode allows the processor to fetch instructions from addresses contained in the current memory sector and it will either update the storage area (during a word miss), or it will be read directly from the sector area (during a cache hit). On the other hand, replacement of the current sector by the SRU is disabled. When a sector is locked, its LRU status continues to be updated, but when choosing the cache sector to be replaced, this sector is ignored and will never be the destination for the new memory sector.

The PLOCK and the PLOCKR instructions can lock a sector. The instructions’ operand is an effective memory address. The cache sector to which the address belongs (if there is one) is locked. If the specified effective address does not belong to one of the current cache sectors, a memory sector containing the address will be allocated into the cache, thereby replacing the least recently used cache sector. This cache sector will be locked but empty. As a result, the locked cache sector is placed at the top of the LRU stack indicating that it is the most recently used sector.

Locking a sector does not affect the contents of the cache sector (instructions already fetched into the cache sector storage area), the valid-bits or the tag register contents of that particular sector.

2.6.2 PRAM Mode

In the PRAM mode the DSP96002 is fully compatible with the original DSP96002. The internal program RAM is either enabled or disabled, according to the OMR. DMA references to/from program memory, and the MOVEM instruction are fully enabled.

Nevertheless, when writing a word into the internal PRAM in PRAM mode, the corresponding valid-bit is set, indicating that, when the user switches into cache mode, the word has been initialized and is therefore valid.

In the PRAM mode, the processor does not update the tag registers in any way, it does not update the SRU, it does not test the valid-bits, and it ignores the HIT/$\overline{\text{MISS}}$ signal.

The PFLUSH and PFREE instructions can be issued when the chip is in PRAM mode. For further information on PFLUSH usage, refer to the next section.

2.6.3 Cache Flush

Cache flush is a cache operation rather than a cache operating mode. It is performed by executing the PFLUSH instruction, which causes a global cache flush that brings the cache to a reset condition. All valid-bits will be cleared. The tag registers' values will form a contiguous 1K segment of memory and therefore hold the values 0,1,2,...,7 that correspond to the PRAM addresses 0, 128, 256,... etc. The LRU stack will hold a default descending order of sectors. All locked cache sectors will be unlocked.

PFLUSH works in either PRAM or cache mode.

When switching from PRAM mode to cache mode, the PFLUSH instruction will allow the user to flush the old data stored in the internal Program Memory. But if the user has brought valid data into the internal program memory while in PRAM mode and would prefer to leave the data untouched, it is not necessary to execute the PFLUSH instruction in connection with changing modes.

However, when switching from cache mode to PRAM mode the cache <u>is not flushed automatically</u> and it is highly recommended that the PFLUSH instruction be executed. If the cache is not flushed, the tag register could contain values different than the 0 to 1K address mapping. In such a case, a write into the internal PRAM could set a valid-bit that corresponds, from the tag value point of view, to an address outside the 0 to 1K address range. This will be transparent to the user while in PRAM mode, but it could be harmful when switching back to cache mode (again if no PFLUSH had been executed).

The PFLUSH instruction is not performed automatically when switching from cache mode to PRAM mode to give the user full control of the cache.

2.7 SECTOR REPLACEMENT POLICY

When a sector miss occurs, a cache sector must be selected to contain the new desired memory sector. The selected cache sector typically contains another memory sector. The sector replacement policy determines which sector would be flushed from the cache, and thus frees the cache sector for the new memory sector. In order to determine which sector should be replaced during a sector miss, the SRU constantly monitors the use of requested addresses and sectors and uses the information as input to the sector replacement algorithm.

The sector replacement policy dictates the replacement of the Least Recently Used (LRU) sector.

The LRU stack status is effected only in cache mode by fetch operations and by PLOCK and PUNLOCK instructions. Locked cache sectors continue to “move” up and down the LRU stack. This implies that when picking the <u>least</u> recently used sector (the one at the bottom of the LRU stack), locked sectors that can’t be flushed from the cache should be skipped.

When the processor is in cache mode, MOVEM instructions do not affect the LRU stack status. When the processor is in PRAM mode, fetches, MOVEM instructions, or DMA transfers do not effect the LRU stack status either.

2.8 DMA TRANSFERS TO/FROM PROGRAM MEMORY

DMA transfers to and from the program memory space (internal and external) are only possible while the cache is in PRAM mode because, while the processor is in cache mode, cache misses update the internal program memory using the DMA time slot. Therefore, DMA transfers to/from program memory are disabled in hardware by blocking the DMA strobes so that such DMA sequences will run without actually accessing the program memory.

While the processor is in PRAM mode a DMA move into the internal PRAM should set the corresponding valid-bit to indicate that the location has been initialized. This feature could be useful is the user wishes to load the cache while the processor is still in PRAM mode.

Note that transferring code from external program memory addresses higher than 1K to internal program memory address (0 to 1K), and then switching into cache mode would cause non-consistency because the cache content for the first 1K addresses is different from the external program memory for these addresses. Since the DMA transfer into internal program memory is usually used for time critical routines and interrupt vectors, and

since these will be usually locked, all further accesses to these locations would not cause a miss and therefore the external Program Memory would not be read. In this case the non-consistency would have no affect. On the other hand, a user that switches from PRAM mode to cache mode and doesn't want the content to be kept should issue the PFLUSH instruction and therefore prevent this situation altogether.

Before switching from PRAM mode to cache mode, or before issuing a PFLUSH instruction while in PRAM mode, it is the user's responsibility to check that any previously started DMA transfers to/from Program Memory have been completed.

2.9 MOVEM/MOVEP/MOVES INSTRUCTIONS

The MOVE(M) instruction (Move Program Memory) performs a move from a register to program memory or from program memory to a register. For simplicity, this discussion will use the term "MOVEM-in" to indicate a MOVEM into the program memory and the term "MOVEM-out" to indicate a MOVEM from program memory. Furthermore, MOVE(P) instruction (Move Peripheral Data) and MOVE(S) instruction (Move Absolute Short) perform similarly when the source or destination is a program memory location, and therefore will not be mentioned separately.

The MOVE(M) instruction is widely used by the OnCE. For example, MOVEM-out is used for program memory display and disassembler while MOVEM-in is used by in-line assembler and software breakpoints.

For compatibility reasons, all of these capabilities are available in cache mode. Therefore, when performing a MOVEM-out instruction, the program memory location has to be read from the cache if it resides in the cache (hit), and from the external program memory if it does not (miss). When preforming a MOVEM-in, the program memory location has to be written both inside the cache and in the external program memory if there was a hit (to maintain cache coherency) but only in the external program memory if there was a cache miss.

In cache mode, neither MOVEM-out nor MOVEM-in updates the valid-bit or the LRU status, nor do they write back the missed word into the cache if there was a miss! This is because MOVEM instruction is NOT an instruction fetch. Furthermore, it allows the user to use the MOVEM with OnCE in a non-intrusive manner.

While in PRAM mode a MOVEM-in to the internal PRAM should set the corresponding valid-bit, to indicate that the location has been initialized. This feature could be useful for a user that wises to load his cache while still in PRAM mode.

Note: For implementation reasons, when a MOVEM-in in cache mode causes a word miss, but a sector hit (i.e. the specified word is not in the cache but the sector it belongs

to does), the content of that word is changed in the internal Program Memory. This should be transparent to the user since, although the word content had been changed, it's valid-bit remains cleared as it was, and therefore the content is meaningless. Nevertheless, if the user switches to PRAM mode without flushing the cache the new word content could be meaningful.

2.10 DEFAULT MODE ON HARDWARE RESET

After reset, the DSP96002 configuration acts just as if there were no instruction cache feature available, and the three MOD pins determine the processor's operating mode. All valid-bits are cleared. All cache sectors are in unlocked state. The tag registers values form a contiguous 1K segment of memory and therefore hold the values 0,1,2,...,7 that correspond to the PRAM addresses 0, 128, 256,... etc. The LRU stack holds a default descending order of sectors, so that sector number 0 is the most recently used and sector number 7 is the least recently used.

2.11 CACHE OBSERVABILITY THROUGH THE OnCE

The DSP96002 OnCE supports a fully non-intrusive system debug capability when the processor is in cache mode. It allows the user to observe the cache status, showing which memory sectors are currently mapped into cache sectors, which cache sectors are locked, and which cache sector is the least recently used. Furthermore, the user can observe the values of the valid-bits for any cache location while the chip is in debug mode by reading the tag registers' contents, lock bits, LRU bits, and valid-bits serially through the OnCE.

For more information, refer to Section 5 - OnCE ENHANCEMENTS.

2.12 RESTRICTIONS AND REMARKS

2.12.1 Change of OMR Bit 4 (Cache Enable bit)

The instruction which changes the value of OMR bit 4 should be followed by three NOPs prior to the first instruction whose fetch will be executed in the new cache operating mode. The use of NOPs is highly recommended. Although other instructions could be used, note that the delay in the switch of cache operating mode will be three decoding cycles. For example, a MOVE with predecrement addressing mode, followed by a single NOP will suffice.

It is recommended that OMR bits 0, 1, and 2 not be changed in parallel with a change in OMR bit 4 since they affect the bootstrap mode, which should not be used while the processor is in cache mode. Therefore, it is recommended that the ORI and ANDI instructions

be executed to set or clear OMR bit 4 without affecting other OMR bits, which could be changed safely three cycles later.

2.12.2 Change of OMR Bit 4 Relative to PLOCK/PUNLOCK

The instruction that sets OMR bit 4 should appear at least three instruction cycles prior to a PLOCK or PUNLOCK instruction, otherwise an illegal instruction trap will be executed.

2.12.3 Fetches Following a PFLUSH Instruction

When the processor is in cache mode, the first two words following a PFLUSH instruction are not cached. The first word of the two words will be cached at first, but then flushed. The second of the two words will be fetched from external program memory but will not be written into the internal program memory. The tag registers, valid-bits, and LRU stack will not be updated by this last fetch.

2.12.4 Bootstrap in Cache Mode

The user may select a bootstrap mode by writing into the OMR, thereby mapping the bootstrap ROM into addresses 0 to 64 of the program address space. A jump to address 0 will begin the bootstrap program that is coded in the bootstrap ROM. But, if the processor is in cache mode, the result could be unpredictable. From these 64 words, a word that is in the cache will be fetched from the internal bootstrap ROM, but a word that is not contained in the cache will be fetched from external program memory. Therefore, it is strongly recommended that the user switch the processor into PRAM mode and flush the cache before mapping the bootstrap ROM into the program address space.

2.12.5 Change of Port Select Register (PSR) in Cache Mode

A change in the PSR while the processor is in cache mode could change the program memory mapping from one port to another, causing an inconsistency problem since the cache data brought from one port could differ from the external memory content at the same addresses in the other port.

2.12.6 JCC Instructions in Cache Mode

When the processor performs JCC (Jump Conditionally) instructions, it fetches both the next code word and the memory location to which the effective address (“target”) points before the condition is resolved. Therefore, both the “next” and “target” code words may cause a miss, or even a sector miss, thereby replacing the current LRU sector with a new sector that is not necessarily needed.

2.13 CACHE USE SCENARIO

This section demonstrates a possible scenario of cache use in a real time system.

1. The DSP96002 leaves the hardware reset in PRAM mode as determined by the mode bits in the OMR.

2. To achieve “hit on first access” (especially important for the fast interrupt vectors), the user, while still in PRAM mode and using DMA, transfers the interrupt vectors and some critical routines into the lower PRAM addresses. The DMA transfers set the corresponding valid-bits. Presume that the code uses 200 PRAM words and therefore it will be contained in 2 cache sectors. Since these routines are time critical, the user will wish to lock the sectors. A possible code may look like this:

LABEL	ADDRESS	CODE
	$00000000	reset vector
	. . .	
	$0000003e	host b write p memory vector
user_code	$00000040	user critical routines
	. . .	
	$0000007f	end of sector 1
	$00000080	beginning of sector 2
	. . .	
	$000000c8	end of user critical routines

3. To enter cache mode, the user sets OMR bit 4. To lock address 0 to 200 in the cache the user issues the PLOCK instruction twice, each time with an effective address that belongs to the corresponding memory sector. Please notice that three cycles should separate the change of OMR bit 4 from the PLOCK instruction.

The code may look like this:

```
ORI #$10, OMR        ; set CE bit in OMR
NOP                  ; pipeline delay
NOP                  ; pipeline delay
NOP                  ; pipeline delay
PLOCK #0             ; lock sector containing address 0
MOVE #128, R0        ; load effective address to r0
NOP                  ; pipeline delay for move
PLOCK R0             ; lock sector containing address 128
```

Notice that the code doesn't fall within the critical sectors, but rather in the initialization code.

PLOCK is the first instruction fetched in cache mode.

4. Now the cache is ready for normal operation with 2 sectors locked and 6 sectors in unlocked mode. Notice that a fetch from one of the locked sectors (addresses 0 to 200) will not cause a miss since the code for these sectors was brought into the cache while in the processor is in PRAM mode.

5. The user can lock an additional sector dynamically. The sequence is similar to the one shown in steps 2 and 3, but a dynamically locked cache sector will not necessarily contain the valid data and would therefore be filled by word misses each time a new word is fetched.

6. It would be wise to place time critical routines on sector boundaries. This would give optimal cache sector utilization. The compiler could certainly obey this constraint.

7. To unlock the cache sector containing addresses 128 to 255, for example, all the user has to do is:

```
MOVE #140, R0        ; load effective address to r0
NOP                  ; pipeline delay
PUNLOCK R0           ; unlock sector containing address 128
```

Notice that address 140 was used as an example since it belongs to the range 128 to 255.

8. To unlock all the locked cache sectors the code should be:

```
PFREE
```

This instruction is useful in case the user forgets which sectors or addresses were previously locked, or as a software reset to the locking mechanism.

9. When debugging the software or the system, the user can enter the debug mode at any time and observe the tags, the valid-bits, the lock bits, and the least recently used sector to be replaced next.

10. To execute the bootstrap program the user switches to PRAM mode, executes the 3 NOPs needed for pipeline delay, performs a PFLUSH, and only then switches to bootstrap mode:

```
ANDI #$ef, OMR          ; clear CE bit in OMR
NOP                     ; pipeline delay
NOP                     ; pipeline delay
NOP                     ; pipeline delay
PFLUSH
MOVEI #$04, OMR         ; bootstrap from Port A
NOP                     ; pipeline delay
JMP #0                  ; jump to bootstrap ROM
```

Notice that PFLUSH was fetched and executed in PRAM mode. It could have appeared one cycle earlier, in which case it would have been fetched in cache mode but executed in PRAM mode.

3 INTEGER MODE

The DSP96002's integer performance has been doubled with the definition of the new integer mode. The integer mode improves the performance of integer algorithms and supports four new parallel integer operations that are enabled while the processor is in integer mode:

- MPYS//ADD (integer signed multiply and add)
- MPYS//SUB (integer signed multiply and subtract)
- MPYU//ADD (integer unsigned multiply and add)
- MPYU//SUB (integer unsigned multiply and subtract).

A full description of these instructions appears in APPENDIX A on page 54. Since they use the opcodes of the parallel floating-point instructions, the following four instructions are disabled while the processor is in integer mode:

- FMPY//FADD.S
- FMPY//FSUB.S
- FMPY//FADD.X
- FMPY//FSUB.X

3.1 CHANGE TO THE PROGRAMMING MODEL (INTEGER MODE)

To support the integer mode, bit 25 of the status register now features a new integer mode (IM) bit as shown in Figure 3.

When the IM bit is cleared (0) the integer mode is disabled. When the IM bit is set, the processor is in integer mode. The IM bit is cleared during reset.

31	30	29	28	27	26	25	24	
LF	*	I1	I0	FZ	MP	IM	*	**MR**

- 24: Reserved
- 25: Integer Mode
- 26: Multiply
- 27: Flush to Zero
- 28, 29: Interrupt Mask
- 30: Reserved
- 31: Loop Flag

23	22	21	20	19	18	17	16	
*	R1	R0	SIOP	SOVF	SUNF	SDZ	SINX	**IER**

15	14	13	12	11	10	9	8	
UN CC	NAN	S NAN	OP ERR	OVF	UNF	DZ	INX	**ER**

7	6	5	4	3	2	1	0	
A	R	LR	I	N	Z	V	C	**CCR**

Figure 3 - DSP96002 Programming Model

3.1.1 Switching Into Integer Mode

The correct sequence for switching from the floating-point mode to integer mode is:

```
ORI  #2,mr    ; set the IM bit in MR register
NOP           ; pipeline delay
NOP           ; pipeline delay
parallel integer operation
```

4 SINGLE PRECISION MODE

The efficiency of the data ALU register file has been improved with the definition of the new single precision mode (SPM), where the user has access to two data ALU register files: a 10 floating-point register file (d0.h..d9.h, d0.m..d9.m) and a 10 integer register file (d0.l..d9.l). If the program uses only single-precision MOVE operations and floating-point

operations that yield single-precision results, then the two register files are completely decoupled - thus effectively doubling the amount of registers available for the data ALU.

4.1 CHANGE TO THE PROGRAMMING MODEL (SINGLE PRECISION MODE)

To support the single precision mode, bit 5 of the OMR supports a new single precision mode (SPM) bit. When OMR bit 5 is clear, the single precision mode is disabled. When OMR bit 5 is set, the processor is in the single precision mode. The SPM bit is cleared during reset.

31 … 6	5	4	3	2	1	0
reserved	SPM	CE	DE	MC	MB	MA

Single Precision Mode Bit

4.2 SINGLE PRECISION MODE DETAILS

The processor supports the following three measures to achieve the Data-ALU Register File decoupling when it is in single precision mode:

1. Single-precision MOVE operations affect the high and middle portion of the destination register. They DO NOT clear the low portion of the destination register.
2. Data-ALU floating-point operations that yield single-precision results affect the high and middle portion of the destination register. They DO NOT clear the low portion of the destination register.
3. Integer multiply operations (MPYS and MPYU) yield 64-bit results (from the condition code's point of view) of which only the 32 least significant bits are written into the low portion of the destination register. The middle portion of the destination register is not affected. Thus, the implication is that the largest two integers that can be multiplied in this mode without a loss of significant digits is 16. If you are using the integer multiply operation MPYS for the multiplication of 16-bit numbers, you must sign-extend the upper 16 bits of the multiplicand and the multiplier to get a valid integer result.

These measures assure that a single-precision floating-point operation or a MOVE does not overwrite an integer variable stored in the low portion of the destination register. Furthermore these measures assure that an integer multiply does not overwrite a single-precision floating-point number stored in the high and middle portions of the destination register. Thereby full decoupling is achieved.

Single Precision Mode does not affect double-precision MOVE operations, long integer MOVE operations or the single-extended-precision floating-point operations.

5 OnCE ENHANCEMENTS

The OnCE has been enhanced to provide the user with fully non-intrusive system debug capability when the processor is in cache mode. When the processor is in debug mode, the OnCE offers the ability to observe the cache status, such as which memory sectors are currently mapped into cache sectors, which cache sectors are locked, and which cache sector is the least recently used by reading the tag registers contents, lock bits, and LRU bits serially.

After the user has determined which memory sectors are in the cache, it is still necessary to find out which words in each sector are actually valid. Performing a loop for every sector that accesses the corresponding addresses using MOVEM instruction and testing a status bit that indicates HIT/$\overline{\text{MISS}}$ will make the determination, which shows again that MOVEM does not effect the cache status in any way.

5.1 Change to OnCE Status and Control Register (OSCR)

The OnCE status and control register has been changed to support cache mode debug with the addition of the read-only cache hit (HIT) at bit 20. Bit 20 is set when a cache hit has occurred when the processor is in cache mode and in debug mode. When the processor is in PRAM mode, bit 20 will read as zero. Hardware reset clears the HIT bit.

5.2 Change to Register Select Bits (RS4-RS0) of the OnCE Command Format

The Register Select Bits (RS4-RS0) now support a new register address destination to accommodate writes to the tags buffer. The RS4-RS0 configuration 10010 refers to the tags buffer (8 tags + locks/lru).

The configuration was previously noted as the Program Address Bus Latch for Decode (OPABD) in the table on page 10-17 of the DSP96002 User's Manual (DSP96002UM/AD). The following table replaces the table currently in the manual.

Table 1 Register Select Bits 4-0 (RS4-RS0)

RS4-RS0	Register Selected
00000	Debug Status/Control (OSCR)
00001	Breakpoint Counter Program (OPBC)
00010	Breakpoint Counter Data (ODBC)
00011	Trace Counter (OTC)
00100	Breakpoint Data Memory Higher-Equal (ODULR)
00101	Breakpoint Data Memory Lower-Equal (ODLLR)
00110	Breakpoint Program Memory Higher-Equal (OPULR)

Table 1 Register Select Bits 4-0 (RS4-RS0)

RS4-RS0	Register Selected
00111	Breakpoint Program Memory Lower-Equal (OPLLR)
01000	Transfer Register (OGDBR)
01001	Program Data Bus Latch (OPDBR)
01010	Program Address Bus Latch for Fetch (OPABF)
01011	Program Instruction Latch (OPILR)
01100	Clear Program Breakpoint Counter
01101	Clear Data Breakpoint Counter
01110	Clear Trace Counter
01111	Reserved
10000	Reserved
10001	Program Address Bus FIFO and Increment Counter
10010	**Tags Buffer**
10011	**Program Address Bus Latch for Decode (OPABD)**
101xx	Reserved
11xx0	Reserved
11x0x	Reserved
110xx	Reserved
11111	No Register Selected

5.3 Obtaining Cache Information Through the OnCE

The OnCE allows the user to keep track of the eight tag values, tags lock/unlock status, and LRU status. In the OnCE, nine 32-bit registers are implemented as a circular buffer with a 4-bit counter. All these registers have the same address but any access to the tags buffer in the cache controller will cause the counter to increment, and thus point to the next register in the circular buffer. When the processor leaves the debug mode, the counter is cleared. When the processor enters debug mode again, the first read from the tags buffer address will always start from the first of the nine registers (tag number 0) and will continue circularly among them.

The registers mapped in the circular tags buffer are shown in Figure 4.

At any point in time at least one lru bit in the “LRU/LOCK status” register will be set. But it is possible for more than one of the lru bits to be set simultaneously because locked sec-

tors could be “least recently used” although they can not be replaced. Therefore, the “next to be replaced sector” is the only sector whose lru bit is set and lock bit cleared. The exception to this rule is the case where all of the eight sectors are locked and designated as “least recently used”, in which case there is no “next to be replaced sector” because no sector will be replaced until at least one sector is unlocked.

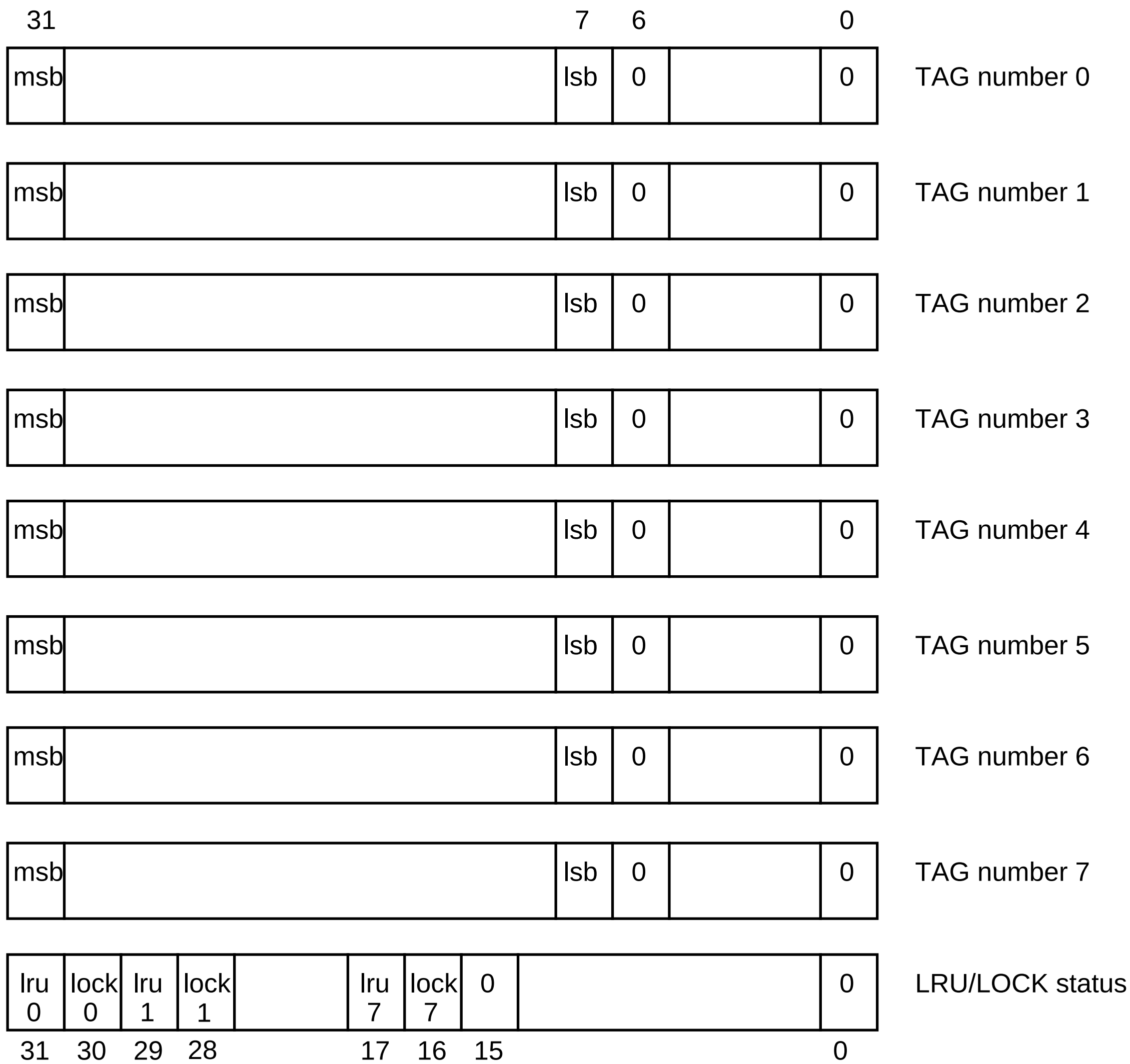


Figure 4 - Circular Tags Buffer

5.4 USING THE OnCE FOR CACHE OBSERVABILITY

5.4.1 Displaying the tags, locks and LRU status

1. ACK
2. Save pipeline information:
    1. Send command READ PDB REGISTER
    2. ACK
    3. CLK
    4. Send command READ PIL REGISTER (instruction latch).
    5. ACK
    6. CLK
3. Read the 9 registers from the tags buffer:
    1. Send command READ TAGS BUFFER (read tag 0 and increment pointer).
    2. ACK
    3. CLK
    4. Send command READ TAGS BUFFER (read tag 1 and increment pointer).
    5. ACK
    6. CLK
    7. Send command READ TAGS BUFFER (read tag 2 and increment pointer).
    8. ACK
    9. CLK
    10. Send command READ TAGS BUFFER (read tag 3 and increment pointer).
    11. ACK
    12. CLK
    13. Send command READ TAGS BUFFER (read tag 4 and increment pointer).
    14. ACK
    15. CLK
    16. Send command READ TAGS BUFFER (read tag 5 and increment pointer).
    17. ACK
    18. CLK
    19. Send command READ TAGS BUFFER (read tag 6 and increment pointer).
    20. ACK
    21. CLK
    22. Send command READ TAGS BUFFER (read tag 7 and increment pointer).
    23. ACK
    24. CLK
    25. Send command READ TAGS BUFFER (read locks/lru register and increment pointer).
    26. ACK
    27. CLK

5.4.2 Displaying the Valid-bits of Specific Cache Locations Starting From Address xxx

This routine uses R0 as pointer to cache addresses. Therefore this register has to be read before the routine, and has to be loaded with the value xxx. At the end of the routine, the values of R0 must be restored. See Section 10.12.3 in the DSP96002 User's Manual (DSP96002UM/AD) for an example.

1. Send command WRITE PDB REGISTER and GO (no EX).

   (ODEC selects PDB as destination for serial data.)

2. ACK
3. Send the 32-bit opcode: "MOVEP P:(R0)+, x:OGDB"

   (After the 32 bits have been received, the PDB register drives the PDB. ODEC releases the chip from "halt" state and the MOVEM instruction is executed. This instruction does not change the cache status in any way but the hit/miss mechanism is activated. The value of HIT/MISS signal is sampled in bit 20 in the OSCR register. The signal that marks the end of the instruction returns the chip to the "halt" state and an acknowledge is issued to the command controller.)

4. ACK
5. Send command READ OSCR REGISTER

   (ODEC selects OSCR as the source for the serial data and an acknowledge is issued to the command controller.)

6. ACK
7. CLK
8. Send command NO SELECTION and GO (no EX).

   (ODEC releases the chip from the "halt" state and the instruction is executed again (in a "REPEAT-like "fashion). The signal that marks the end of the instruction returns the chip to the "halt" state and an acknowledge is issued to the command controller.)

9. ACK
10. Send command READ OSCR REGISTER

    (ODEC selects OSCR as source for serial data and an acknowledge is issued to the command controller.)

11. ACK
12. CLK
13. Repeat from step 8 until the entire cache area is examined. At the end of the process R0 should be restored.

5.4.3 Displaying the Valid-bits of Specific Cache Locations Starting From Address xxx, When in PRAM Mode

When in PRAM mode the MOVEM instruction would not activate the HIT/$\overline{\text{MISS}}$ mechanism and therefore the value of the valid-bit would not be reflected in the HIT/$\overline{\text{MISS}}$ status bit. Therefore, it is necessary to switch to cache mode before reading the valid-bits. Use the following sequence to switch to cache mode:

1. Send command WRITE PDB REGISTER and GO (no EX).

   (ODEC selects PDB as destination for serial data.)

2. ACK

3. Send the 32-bit opcode: “ORI #$10, OMR”

   (After the 32 bits have been received, the PDB register drives the PDB. ODEC releases the chip from “halt” state and the ORI instruction is executed. This instruction sets the “CE” bit in the OMR register. The signal that marks the end of the instruction returns the chip to the “halt” state and an acknowledge is issued to the command controller.)

4. ACK

Only now can we read the valid-bits using the HIT/$\overline{\text{MISS}}$ mechanism as described in section 5.2.

To switch back to the PRAM mode, the same sequence is preformed, but this time using “ANDI #$ef, OMR”.

5.4.4 Synchronous Start of the Execution of Multiple Chips

This routine will load each processor with the information necessary for starting the execution of its program and in the end will synchronously release all the processors from the Debug Mode.

1. The command controller selects the first processor.

2. Send command WRITE PDB REGISTER (no GO, no EX).

   (ODEC selects PDB as destination for serial data.)

3. ACK

4. Send 32 bits of the opcode of a two word jump instruction ($030c3f80).

   (After all the 32-bits have been received the PDB register drives the PDB. ODEC causes the core to load the opcode. An acknowledge is issued to the command controller.)

5. ACK

6. Send command WRITE PDB REGISTER (no GO, no EX).

   (ODEC selects PDB as destination for serial data.)

7. ACK

8. Send 32 bits of the target absolute address for the first processor ($xxxxxxxx)

9. ACK

10. The command controller selects the second processor.

11. Send command WRITE PDB REGISTER (no GO, no EX).

    (ODEC selects PDB as destination for serial data.)

12. ACK

13. Send 32 bits of the opcode of a two word jump instruction ($030c3f80).

    (After all the 32-bits have been received the PDB register drives the PDB. ODEC causes the core to load the opcode. An acknowledge is issued to the command controller.)

14. ACK

15. Send command WRITE PDB REGISTER (no GO, no EX).

    (ODEC selects PDB as destination for serial data.)

16. ACK
17. Send 32 bits of the target absolute address for the second processor ($xxxxxxxx).
18. ACK

The sequence of instructions described above will be repeated for the remaining processors in the system. Finally the command controller will select ALL the processors in the system and will issue in a broadcast manner the synchronous GO command.

19. Send command GO and EX with no register select.

    (All the chips will resume fetching from their target addresses synchronously. Note that the trace counter will count this instruction so the current trace counter may need to be corrected if the trace mode enable bit in the OSCR has been set.)

6 INTRODUCTION TO THE TIMER/EVENT COUNTER

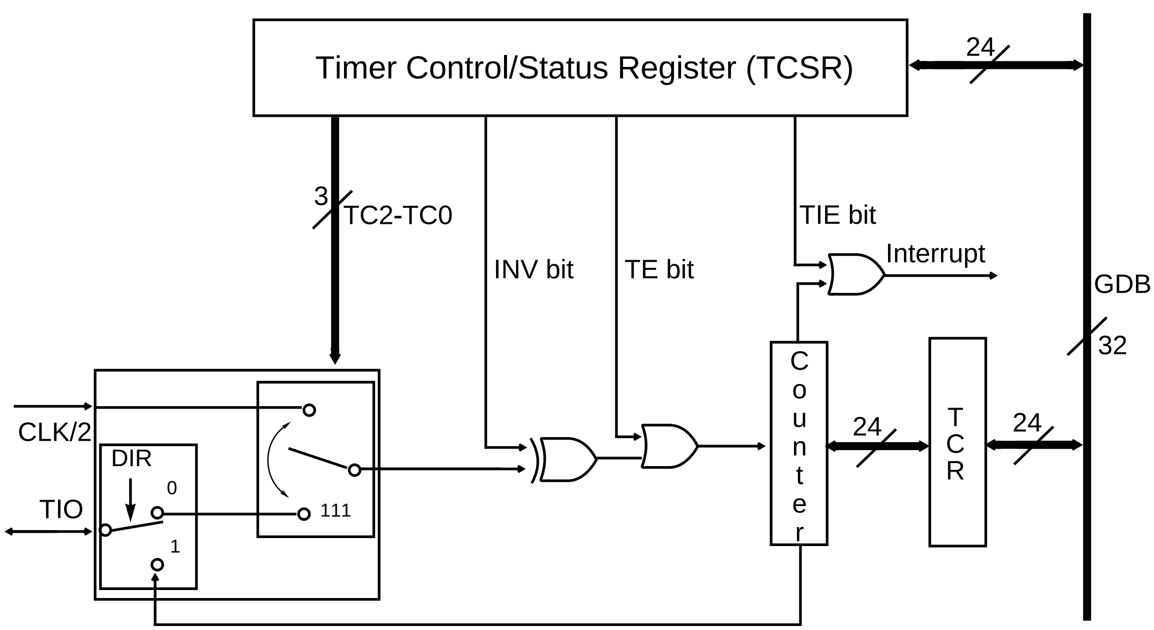
This section describes the two identical and independent timer/event counter modules now featured on the DSP96002. The timer can use internal or external clocking and can interrupt the processor after a number of events specified by a user program, or it can signal an external device after counting internal events. The timer can also be used to trigger DMA transfers after a specified number of events (clocks) occurs.

Each timer connects to the external world through its own bidirectional TIO pin. When TIO is used as input, the module is functioning as an external event counter or is measuring external pulse width/signal period. When TIO is used as output, the module is functioning as a timer and TIO becomes the timer pulse. When the TIO pin is not used by the timer module it can be used as a general purpose I/O (GPIO) pin.

Note: When the timer is disabled, the TIO pin becomes three-stated. To prevent undesired spikes from occurring, the TIO pin should be pulled up or down when it is not in use.

6.1 TIMER BLOCK DIAGRAM

Figure 5 shows a block diagram of the timer module. It includes a 32-bit read-write Timer Control and Status Register (TCSR), a 32-bit read-write Timer Count Register (TCR), a 32-bit counter, and logic for clock selection and interrupt generation.



Register addresses are shown in Figure 5 on page 28.

Figure 5 - Single Timer Module Block Diagram

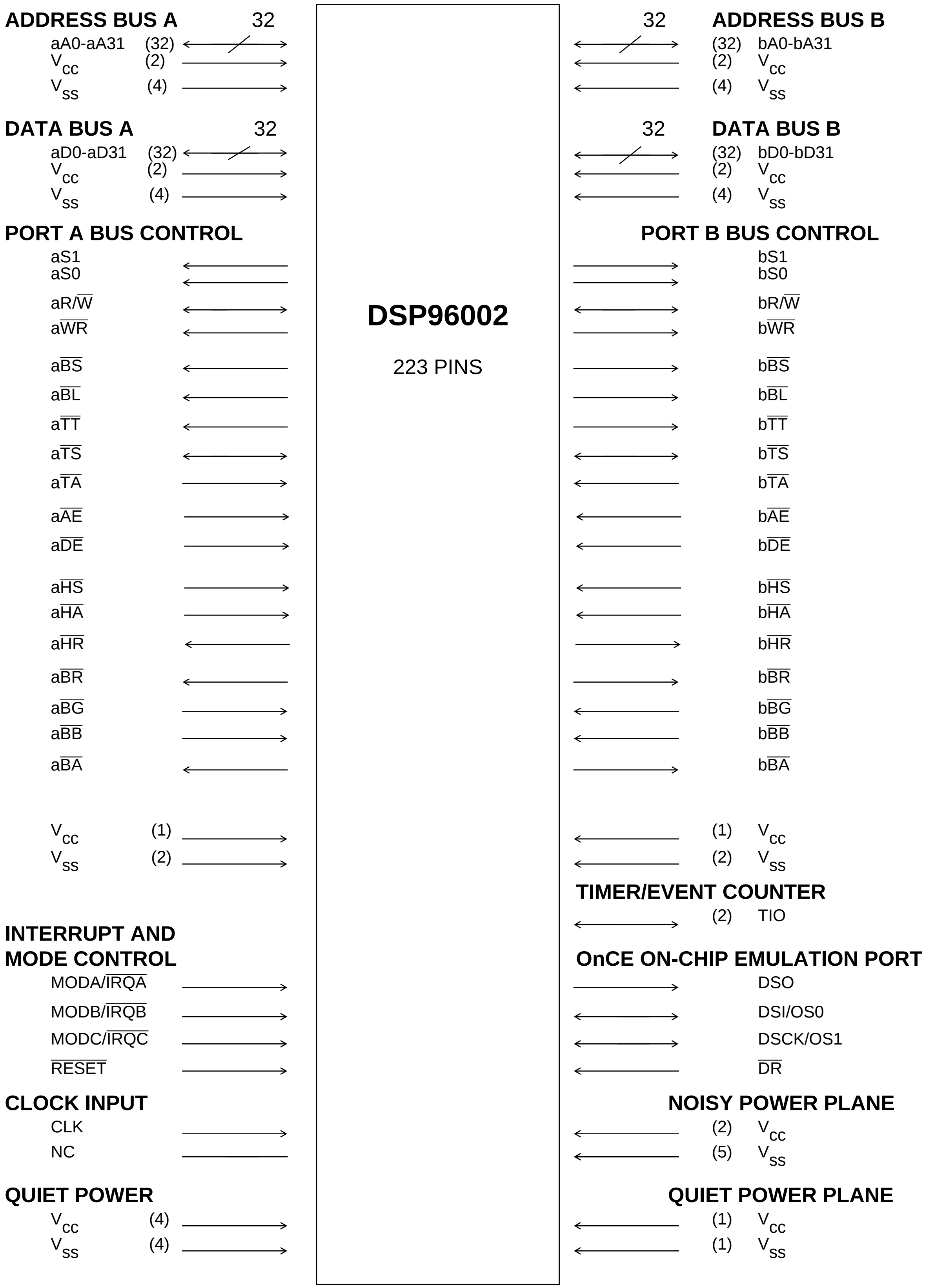


Figure 6 - DSP96002 Signal Functional Groups

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
A	BA23	BA27	BA29	BA31	$\overline{IRQA}$	A$\overline{BB}$	A$\overline{BR}$	**TIO0**	AR/$\overline{W}$	AS0	A$\overline{TS}$	A$\overline{AE}$	AA02	AA04	AA07	AA10	AA13	AA16	**A**
B	BA20	BA25	BA28	BA30	$\overline{IRQB}$	A$\overline{BG}$	A$\overline{BA}$	B$\overline{TT}$	AS1	A$\overline{BS}$	AA00	AA03	AA06	AA09	AA11	AA14	AA18	AA20	**B**
C	BA17	BA21	BA26	GNDN	$\overline{IRQC}$	$\overline{RES}$	A$\overline{BL}$	A$\overline{TT}$	**A$\overline{WR}$**	AA01	AA05	AA08	AA12	AA15	AA17	AA19	AA21	AA23	**C**
D	BA15	BA18	BA24		GNDN	GNDN	GNDN	VCCN	VCCN	VCCQ	GNDQ	VCCN	GNDN	GNDN	GNDN	AA22	AA25	AA26	**D**
E	BA13	BA16	BA22	GNDN											GNDN	AA24	AA28	AA29	**E**
F	BA12	BA14	BA19	GNDN											GNDN	AA27	AA30	AD31	**F**
G	BA09	BA10	VCCN	VCCN					**DSP96002**						GNDN	AA31	AD30	AD29	**G**
H	BA08	CLK	BA11	VCCQ					**223 PIN**						VCCN	AD28	AD27	AD26	**H**
J	A$\overline{TA}$	B$\overline{TA}$	BA07	GNDQ					**PGA**						GNDQ	AD24	AD25	AD23	**J**
K	BA04	BA05	BA06	VCCN					**TOP VIEW**						GNDQ	AD20	AD21	AD22	**K**
L	BA03	BA01	BA02	VCCN											VCCQ	AD16	AD18	AD19	**L**
M	BA00	BS1	BS0	GNDN											VCCN	VCCN	A$\overline{DE}$	AD17	**M**
N	B$\overline{AE}$	**TIO1**	**B$\overline{WR}$**	GNDN											GNDN	AD11	AD14	AD15	**N**
P	BR/$\overline{W}$	B$\overline{TS}$	B$\overline{BL}$	GNDN											GNDN	AD07	AD12	AD13	**P**
R	B$\overline{BS}$	B$\overline{BR}$	B$\overline{BB}$	GNDN	GNDN	GNDN	VCCN	GNDQ	VCCQ	VCCQ	VCCN	GNDN	GNDN	GNDN	GNDN	AD05	AD09	AD10	**R**
T	B$\overline{BG}$	B$\overline{BA}$	A$\overline{HR}$	$\overline{DR}$	A$\overline{HS}$	BD31	GNDN	BD26	BD22	BD17	BD14	BD11	BD07	BD04	BD01	AD02	AD06	AD08	**T**
U	B$\overline{HR}$	DSCK	NC(1)	A$\overline{HA}$	B$\overline{DE}$	BD29	BD27	BD24	BD21	BD18	BD15	BD12	BD09	BD06	BD03	BD00	AD03	AD04	**U**
V	DSO	DSI	B$\overline{HA}$	B$\overline{HS}$	BD30	BD28	BD25	BD23	BD20	BD19	BD16	BD13	BD10	BD08	BD05	BD02	AD00	AD01	**V**
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	

Figure 7 - DSP96002 Pin Assignment

The DSP96002 views each timer as a memory-mapped peripheral occupying two 32-bit words in the X data memory space, and may use each timer as a normal memory-mapped peripheral by using standard polled or interrupt programming techniques.The programming model is shown in Figure 5.

6.2 TIMER CONTROL/STATUS REGISTER (TCSR)

The 32-bit read/write TCSR controls the timer and verifies its status. The TCSR can be accessed by normal move instructions and by bit manipulation instructions. The control and status bits are described in the following paragraphs.

6.2.1 Timer Enable (TE) Bit 31

The TE bit enables or disables the timer. Setting the TE bit (TE=1) will enable the timer, and the counter will be loaded with the value contained in the TCR and will start decrementing at each incoming event. Clearing the TE bit will disable the timer. Hardware $\overline{\text{RESET}}$ and software RESET (RESET instruction) clear TE.

6.2.2 Timer Interrupt Enable (TIE) Bit 30

The TIE bit enables the timer interrupts after the counter reaches zero and a new event occurs. If TCR is loaded with n, an interrupt will occur after (n+1) events.

Setting TIE (TIE=1) will enable the interrupts.When the bit is cleared (TIE=0) the interrupts are disabled. Hardware and software resets clear TIE.

6.2.3 Inverter (INV) Bit 29

The INV bit affects the polarity of the external signal coming in on the TIO input and the polarity of the output pulse generated on the TIO output.

If TIO is programmed as an input and INV=0, the 0-to-1 transitions on the TIO input pin will decrement the counter. If INV=1, the 1-to-0 transitions on the TIO input pin will decrement the counter.

If TIO is programmed as output and INV=1, the pulse generated by the timer will be inverted before it goes to the TIO output pin. If INV=0, the pulse is unaffected.

In Timer Mode 4 (see Section **6.4.4 - Timer Mode 4 (Pulse Width Measurement Mode)**), the INV bit determines whether the high pulse or the low pulse is measured to determine input pulse width. In Timer Mode 5 (see Section **6.4.5 - Timer Mode 5 (Period Measurement Mode)**), the INV bit determines whether the period is measured between leading or trailing edges.

In GPIO mode, the INV bit determines whether the data read from or written to the TIO pin shall be inverted (INV=1) or not (INV=0).

INV is cleared by hardware and software resets.

31	30	29	28	27	26	25	24
TE (0)	TIE (0)	INV (0)	TC2 (0)	TC1 (0)	TC0 (0)	GPIO (0)	TS (0)

23	22	21	20	19	18	17	16
DIR (0)	DI (1)	DO (0)	**	**	**	**	**

15	14	13	12	11	10	9	8
**	**	**	**	**	**	**	**

7	6	5	4	3	2	1	0
**	**	**	**	**	**	**	**

READ/WRITE
TIMER CONTROL/STATUS
REGISTER (TCSR0)
ADDRESS X:$FFFFFFE0

** - reserved, read as zero, should be written with zero for future compatibility
The numbers in parentheses represent the bits' reset values

31	0

READ/WRITE
TIMER COUNT
REGISTER (TCR0)
ADDRESS X:$FFFFFFE1

31	30	29	28	27	26	25	24
TE (0)	TIE (0)	INV (0)	TC2 (0)	TC1 (0)	TC0 (0)	GPIO (0)	TS (0)

23	22	21	20	19	18	17	16
DIR (0)	DI (1)	DO (0)	**	**	**	**	**

15	14	13	12	11	10	9	8
**	**	**	**	**	**	**	**

7	6	5	4	3	2	1	0
**	**	**	**	**	**	**	**

READ/WRITE
TIMER CONTROL/STATUS
REGISTER (TCSR1)
ADDRESS X:$FFFFFFE8

** - reserved, read as zero, should be written with zero for future compatibility
The numbers in parentheses represent the bits' reset values

31	0

READ/WRITE
TIMER COUNT
REGISTER (TCR1)
ADDRESS X:$FFFFFFE9

Figure 8 - Timer Module Programming Model

Note: Because of its affect on signal polarity, and on how GPIO data is read and written,

the status of the INV bit is crucial to the timer's function. Change it only when the timer is disabled (TE=0).

6.2.4 Timer Control (TC2-TC0) Bits 28-26

The three TC bits control the source of the timer clock, the behavior of the TIO pin, and the timer mode of operation. Table 2 summarizes the functionality of the TC bits.

A detailed description of the timer operating modes is given in Section 6.4 on page 35.

The timer control bits are cleared by hardware $\overline{\text{RESET}}$ and software RESET (RESET instruction).

Note 1: If the clock is external, the counter will be decremented by the transitions on the TIO pin. The DSP synchronizes the external clock to its own internal clock. The external clock's frequency should be lower than the maximum internal frequency divided by 4 (CLK/4).

Note 2: The TC2-TC0 bits should be changed only when the timer is disabled (TE=0) to ensure proper functionality.

Table 2 TC Bit Functionality

TC2	TC1	TC0	TIO	CLOCK	MODE
0	0	0	GPIO*	Internal	Timer (Mode 0)
0	0	1	Output	Internal	Timer Pulse (Mode 1)
0	1	0	Output	Internal	Timer Toggle (Mode 2)
0	1	1	—	—	Reserved - Do Not Use
1	0	0	Input	Internal	Input Width (Mode 4)
1	0	1	Input	Internal	Input Period (Mode 5)
1	1	0	—	—	Undefined
1	1	1	Input	External	Event Counter (Mode 7)

* - the GPIO function is enabled only if TC2-TC0 are all 0 (zero) and the GPIO bit is set.

6.2.5 General Purpose IO (GPIO) Bit 25

If the GPIO bit is set (GPIO=1) and if TC2-TC0 are all zeros, the TIO pin operates as a general purpose IO pin, whose direction is determined by the DIR bit. If GPIO=0 the general purpose IO function is disabled. GPIO is cleared by hardware and software resets.

Note: The case where TC2-TC0 are not all zero and GPIO=1 is undefined and should not be used

6.2.6 Timer Status (TS) Bit 24

When the TS bit is set, it indicates that the counter has been decremented to zero.

The TS bit is cleared when the TCSR is read. The bit is also cleared when the timer interrupt is serviced (timer interrupt acknowledge). TS is cleared by hardware and software resets.

6.2.7 Direction (DIR) Bit 23

The DIR bit determines the behavior of the TIO pin when TIO acts as general purpose IO. When DIR=0, the TIO pin acts as an input. When DIR=1, the TIO pin acts as an output. DIR is cleared by hardware and software resets.

Note: The TIO pin can act as a general purpose IO pin only when TC2-TC0 are all zero and the GPIO bit is set. If one of TC2, TC1, or TC0 is not 0, the GPIO function is disabled and the DIR bit has no effect.

6.2.8 Data Input (DI) Bit 22

When the TIO pin acts as a general purpose IO input pin (TC2-TC0 are all zero and DIR=0), the contents of the DI bit will reflect the value the TIO pin. However, if the INV bit is set, the data in DI will be inverted. When GPIO mode is disabled or it is enabled in output mode (DIR=1), the DI bit reflects the value of the TIO pin, again depending on the status of the INV bit. DI is set by hardware and software resets.

6.2.9 Data Output (DO) Bit 21

When the TIO pin acts as a general purpose IO output pin (TC2-TC0 are all zero and DIR=1), writing to the DO bit writes the data to the TIO pin. However, if the INV bit is set, the data written to the TIO pin will be inverted. When GPIO mode is disabled, writing to the DO bit will have no effect. DO is cleared by hardware and software resets.

6.2.10 TCSR Reserved bits (Bits 20-0)

These reserved bits are read as zero and should be written with zero for future compatibility.

6.3 TIMER COUNT REGISTER (TCR)

The 32-bit read-write TCR contains the value (specified by the user program) to be loaded into the counter when the timer is enabled (TE=1), or when the counter has been decremented to zero and a new event occurs. If the TCR is loaded with n, the counter will be reloaded after (n+1) events.

If the timer is disabled (TE=0) and the user program writes to the TCR, the value is stored there but will not be loaded into the counter until the timer becomes enabled. When the timer is enabled (TE=1) and the user program writes to the TCR, the value is stored there and will be loaded into the counter after the counter has been decremented to zero and a new event occurs.

In Timer Modes 4 and 5, however, the TCR will be loaded with the current value of the counter on the appropriate edge of the TIO input signal (rather than with a value specified by the user program). The value loaded to the TCR represents the width or the period of the signal coming in on the TIO pin, depending on the timer mode. See Sections **6.4.4** and **6.4.5** for detailed descriptions of Timer Modes 4 and 5.

6.4 TIMER MODES OF OPERATION

This section gives the details of each of the timer modes of operation. Table 2 on page 33 summarizes the items which determine the timer mode, including the configuration of the timer control bits, the function of the TIO pin, and the clock source.

6.4.1 Timer Mode 0 (Standard Timer Mode, Internal Clock, No Timer Output)

Timer Mode 0 is defined by TCSR bits TC2-TC0 equal to 000.

With the timer enabled (TE=1), the counter is loaded with the value contained by the TCR. The counter is decremented by a clock derived from the internal DSP clock, divided by two (CLK/2). During the clock cycle following the point where the counter reaches 0, the TS bit is set and the timer generates an interrupt. The counter is reloaded with the value contained by the TCR, and the entire process is repeated until the timer is disabled (TE=0). Figure 9 illustrates Mode 0 with the timer enabled. Figure 10 illustrates the events with the timer disabled.

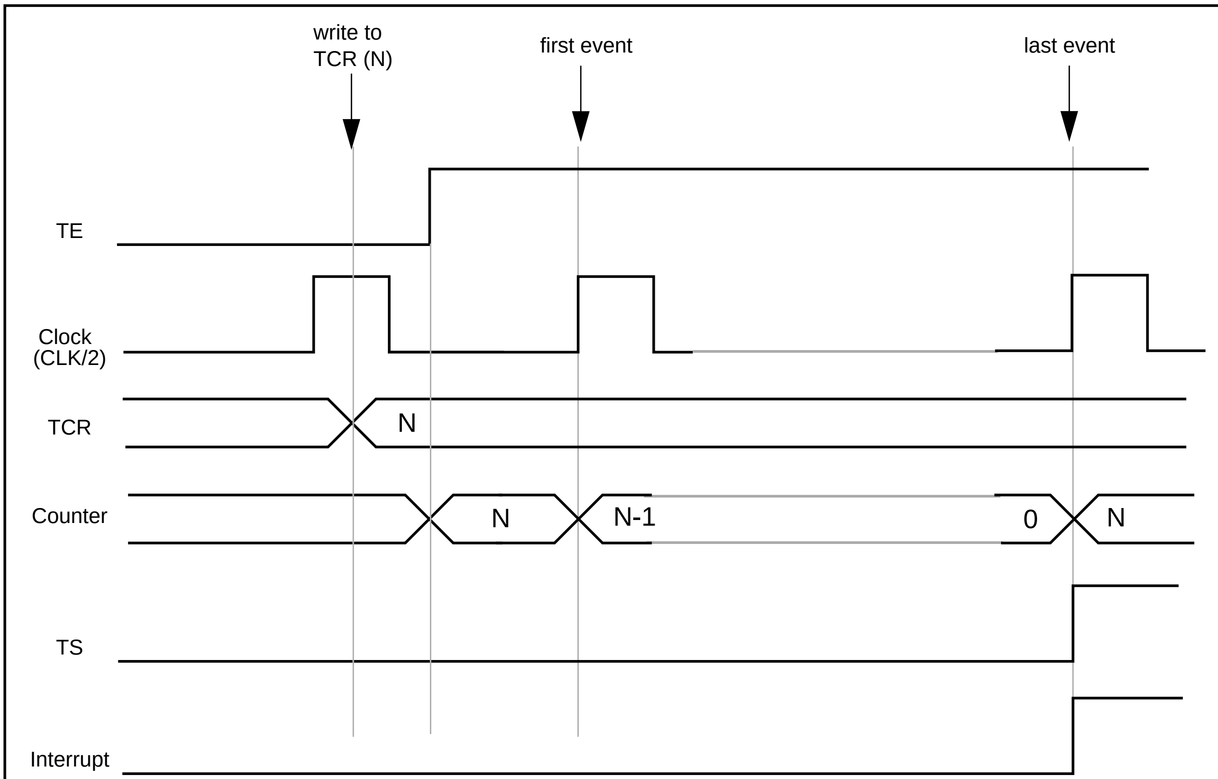


Figure 9 - Standard Timer Mode (Mode 0)

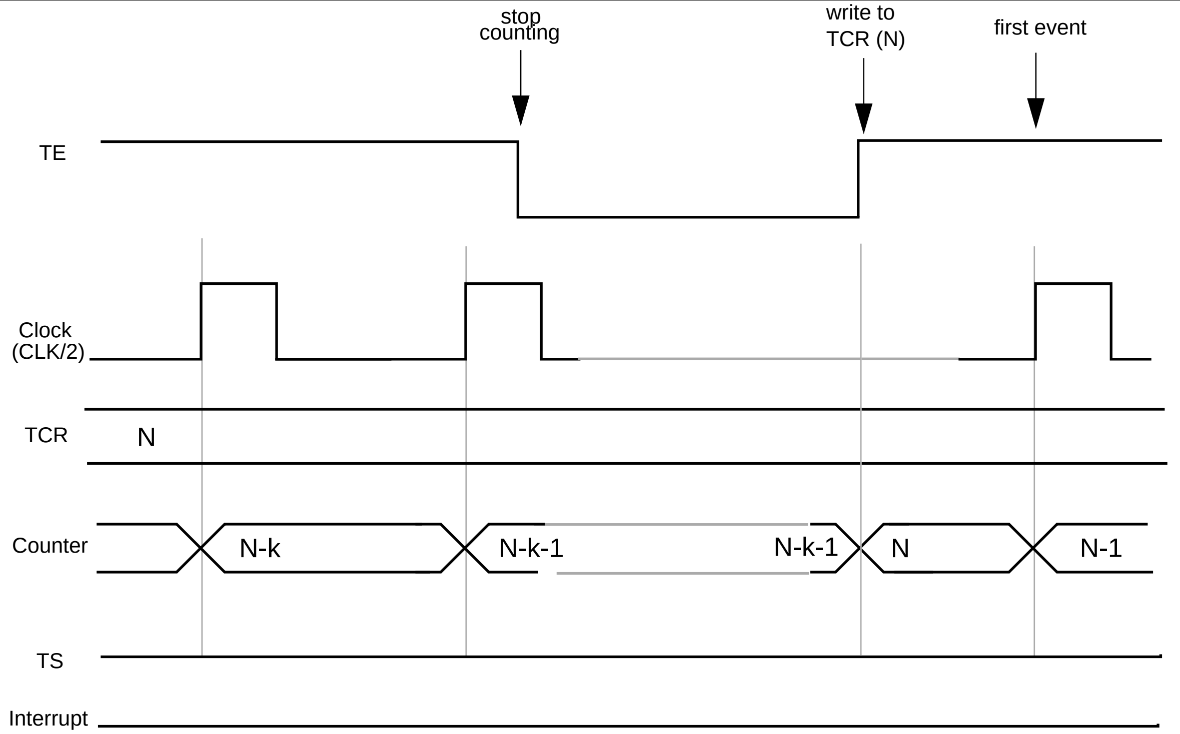


Figure 10 - Timer Disabled

Note: It is recommended that the GPIO input function of Mode 0 only be activated with the timer disabled. If the processor attempts to read the DI bit, it must read the entire TCSR register, which would clear the TS bit and, thus, clear a pending timer interrupt.

6.4.2 Timer Mode 1 (Standard Timer Mode, Internal Clock, Output Pulse Enabled)

Timer Mode 1 is defined by TC2-TC0 equal to 001.

With the timer enabled (TE=1), the counter is loaded with the value contained by the TCR. The counter is decremented by a clock derived from the DSP's internal clock, divided by two (CLK/2). During the clock cycle following the point where the counter reaches 0, the TS bit is set and the timer generates an interrupt. A pulse with a width equal to two clock cycles, and whose polarity is determined by the INV bit, will be put out on the TIO pin. The counter is reloaded with the value contained by the TCR and the entire process is repeated until the timer is disabled (TE=0). Figure 11 illustrates Timer Mode 1 when INV=0, and Figure 12 illustrates Timer Mode 1 when INV=1.

6.4.3 Timer Mode 2 (Standard Timer Mode, Internal Clock, Output Toggle Enabled)

Timer Mode 2 is defined by TC2-TC0 equal to 010.

With the timer enabled (TE=1), the counter is loaded with the value contained by the TCR. The counter is decremented by a clock derived from the DSP's internal clock, divided by

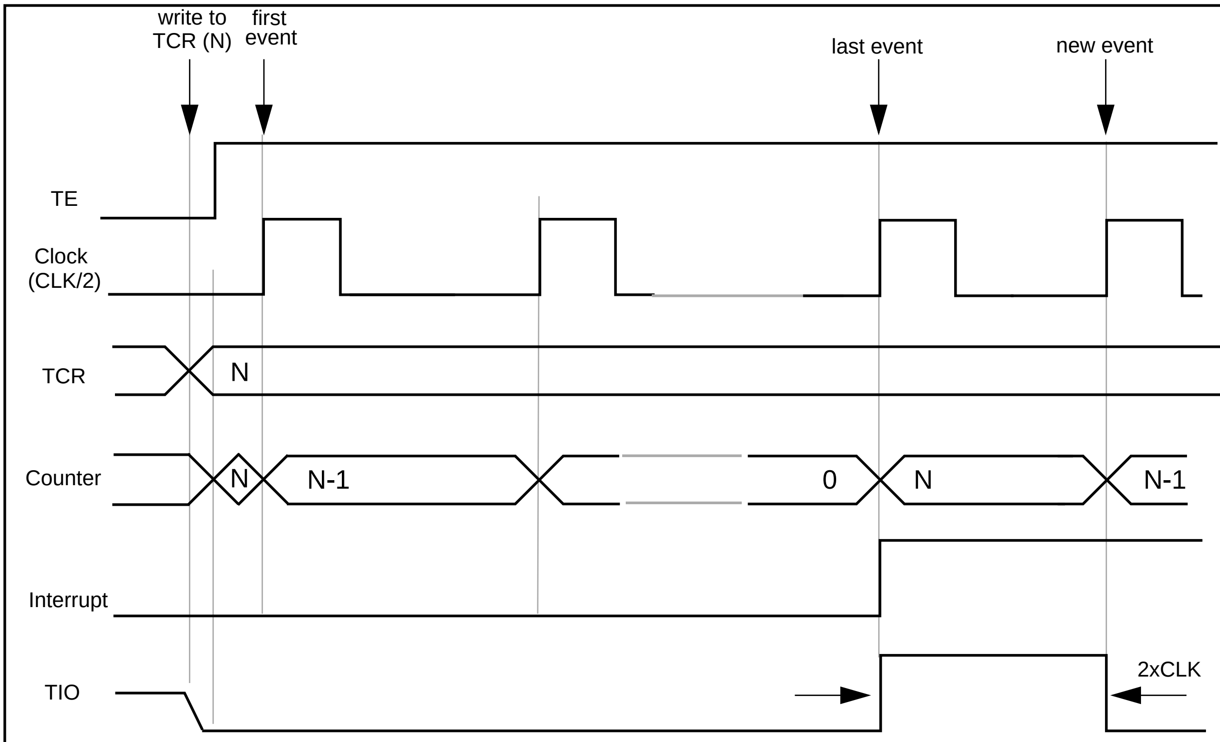


Figure 11 - Standard Timer Mode, Internal Clock, Output Pulse Enabled (INV=0)

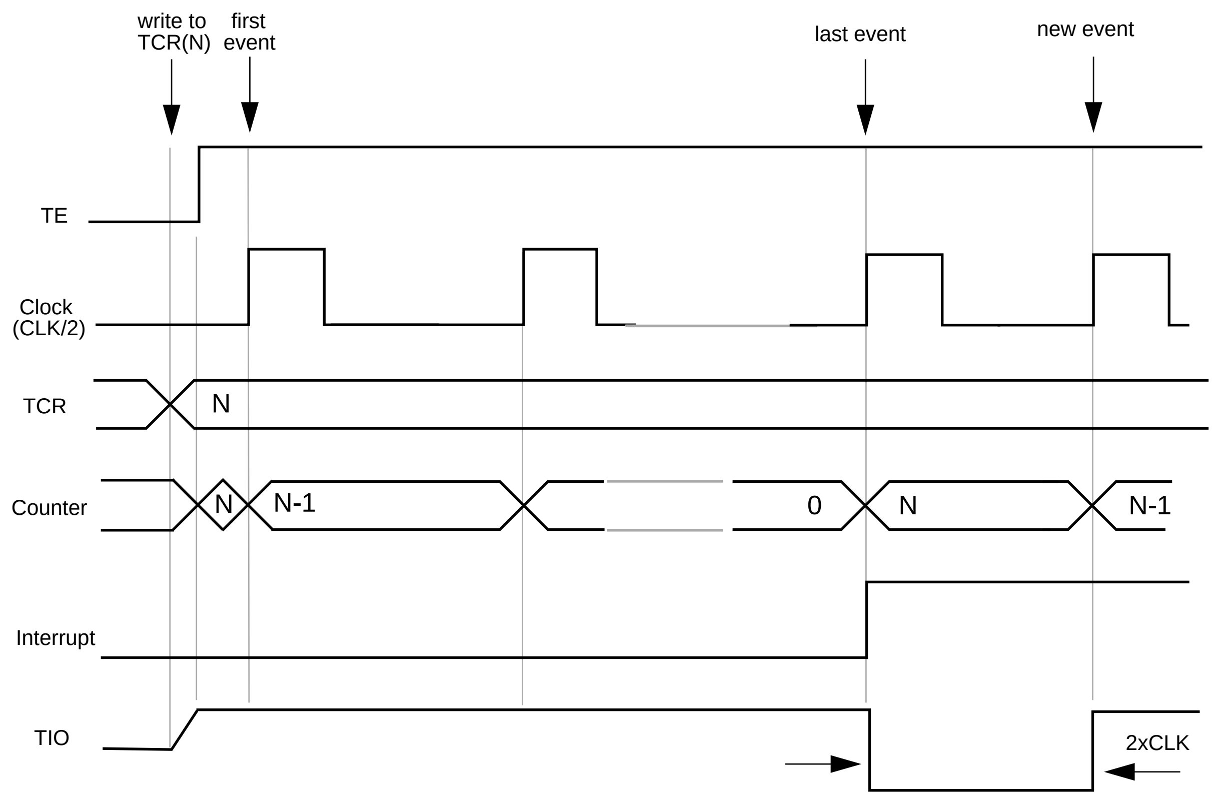


Figure 12 - Standard Timer Mode, Internal Clock, Output Pulse Enabled (INV=1)

two (CLK/2). During the clock cycle following the point where the counter reaches 0, the

TS bit in TCSR is set and, if the TIE is set, an interrupt is generated.The counter is reloaded with the value contained by the TCR and the entire process is repeated until the timer is disabled (TE=0). Each time the counter reaches 0, the TIO output pin will be toggled. The INV bit determines the polarity of the TIO output. Figure 13 illustrates Timer Mode 2.

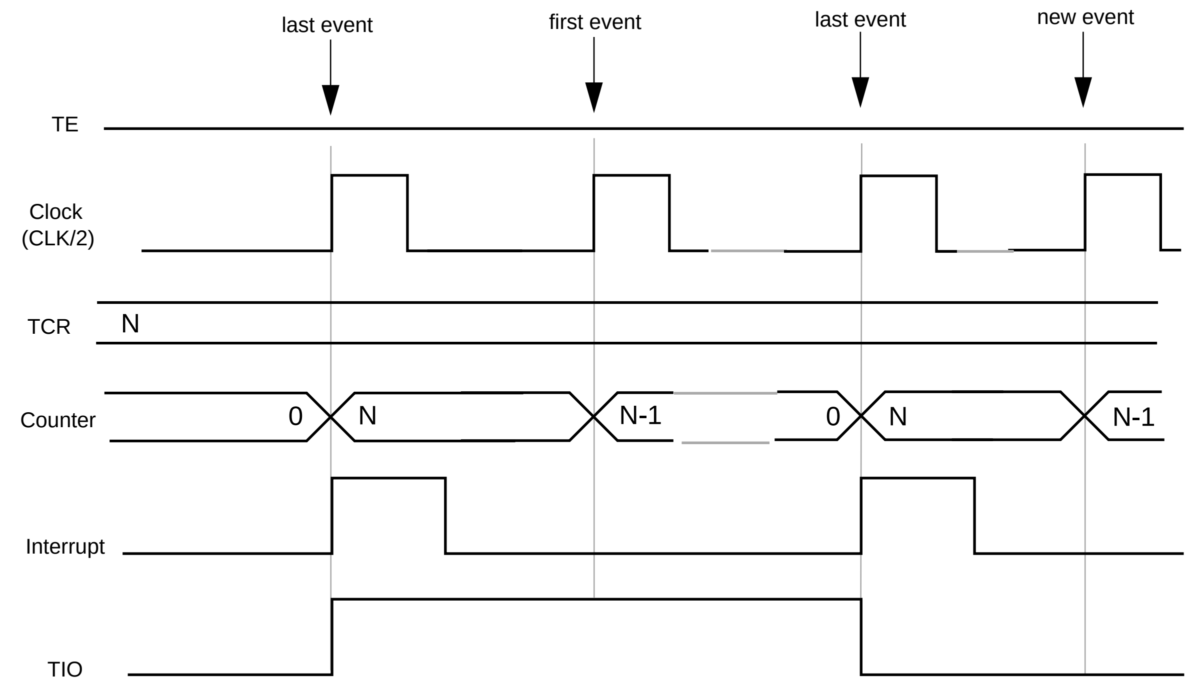


Figure 13 - Standard Timer Mode, Internal Clock, Output Toggle Enable

6.4.4 Timer Mode 4 (Pulse Width Measurement Mode)

Timer Mode 4 is defined by TC2-TC0 equal to 100.

In this mode, TIO acts as a gating signal for the DSP’s internal clock. With the timer enabled (TE=1), the counter is driven by a clock derived from the DSP’s internal clock divided by two (CLK/2). The counter is loaded with 0 by the first transition occurring on the TIO input pin and starts incrementing. When the first edge of opposite polarity occurs on TIO, the counter stops, the TS bit in TCSR is set and, if TIE is set, an interrupt is generated. The contents of the counter is loaded into the TCR. The user’s program can read the TCR, which now represents the widths of the TIO pulse. The process is repeated until the timer is disabled (TE=0).The INV bit determines whether the counting is enabled when TIO is high (INV=0) or when TIO is low (INV=1). Figure 14 illustrates Timer Mode 4 when INV=0 and Figure 15 illustrates Timer Mode 4 with INV=1.

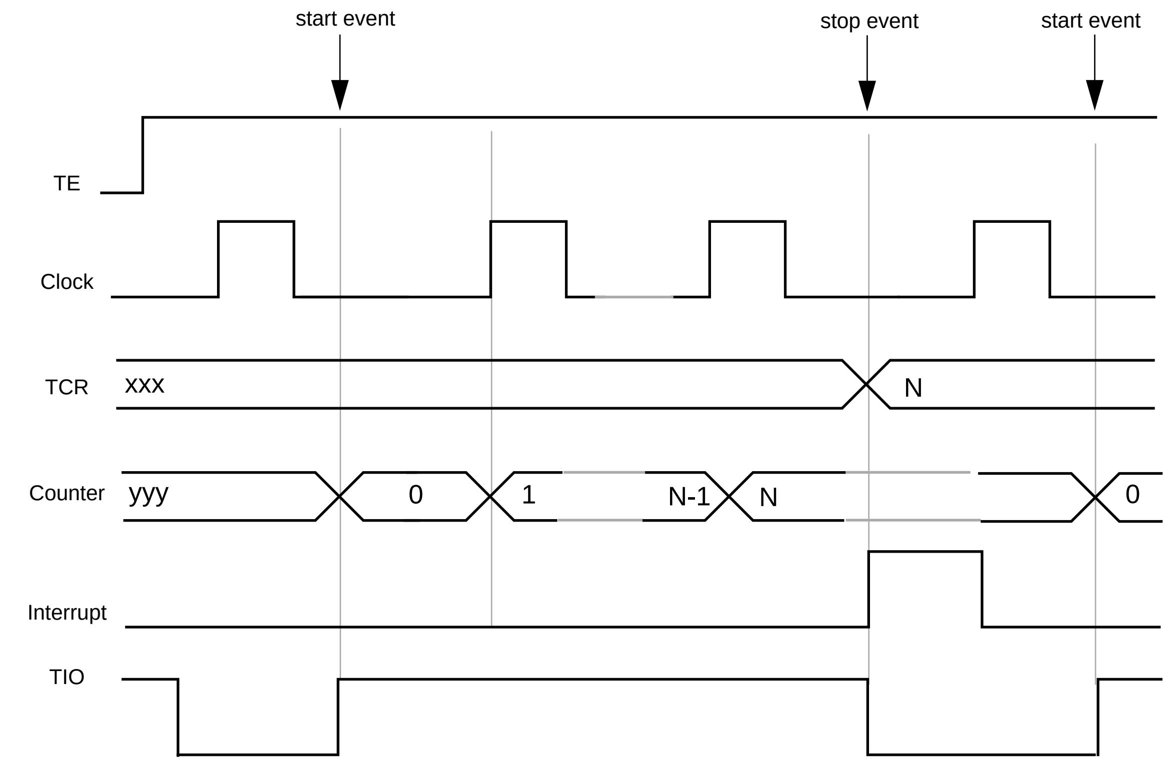


Figure 14 - Pulse Width Measurement Mode (INV=0)

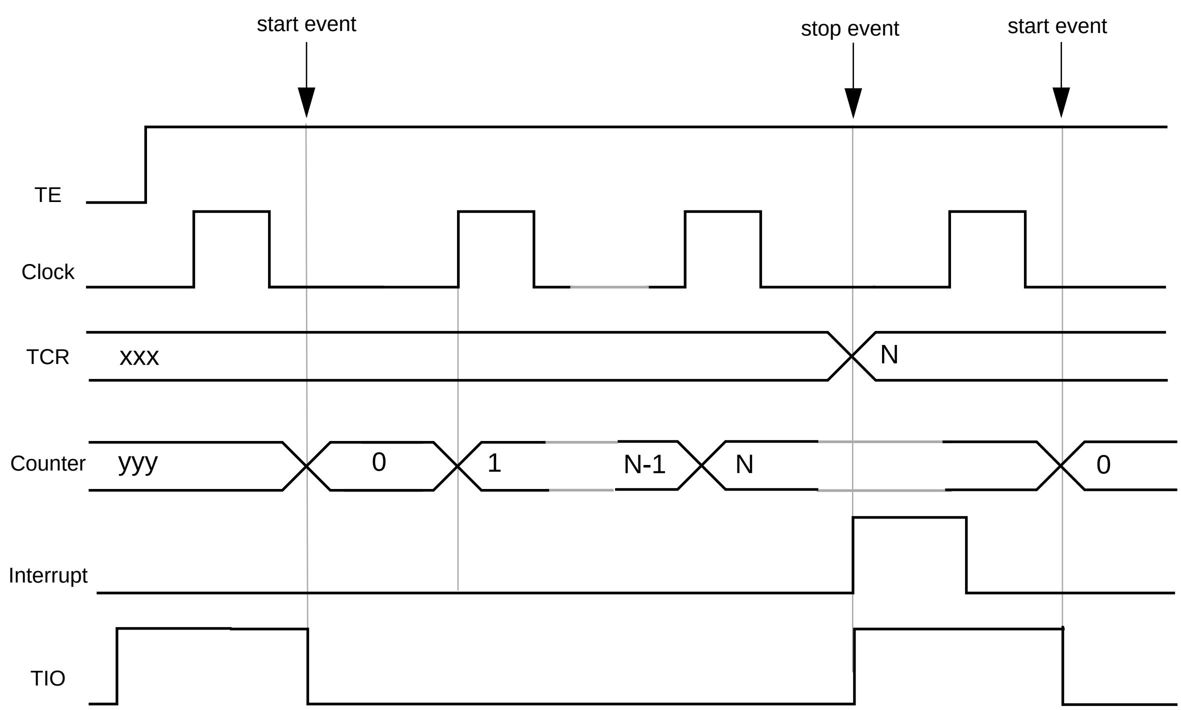


Figure 15 - Pulse Width Measurement Mode (INV=1)

6.4.5 Timer Mode 5 (Period Measurement Mode)

Timer Mode 5 is defined by TC2-TC0 equal to 101.

In Timer Mode 5, the counter is driven by a clock derived from the DSP's internal clock divided by 2 (CLK/2). With the timer enabled (TE=1), the counter is loaded with the value contained by the TCR and starts incrementing. On each transition of the same polarity that occurs on TIO, the TS bit in TCSR is set and, if TIE is set, an interrupt is generated. The contents of the counter is loaded in the TCR. The user's program can read the TCR and subtract consecutive values of the counter to determine the distance between TIO edges. The counter is not stopped and it continues to increment. The INV bit determines whether the period is measured between 0-to-1 transitions of TIO (INV=0), or between 1-to-0 transitions of TIO (INV=1). Figure 16 illustrates Timer Mode 5 when INV=0, and Figure 17 illustrates Mode 5 with INV=1.

6.4.6 Timer Mode 7 (Event Counter Mode, External Clock)

Timer Mode 7 is defined by TC2-TC0 equal to 111.

With the timer enabled (TE=1), the counter is loaded with the value contained by the TCR. The counter is decremented by the transitions of the signal coming in on the TIO input pin. At the transition that occurs after the counter has reached 0, the TS bit in TCSR is set and, if the TIE is set, the timer generates an interrupt. The counter is reloaded with the value contained by the TCR, and the entire process is repeated until the timer is disabled (TE=0). The INV bit determines whether 0-to-1 transitions (INV=0) or 1-to-0 transitions (INV=1) will decrement the counter. Figure 18 illustrates Timer Mode 7 when INV=0, and Figure 19 illustrates Timer Mode 7 when INV=1.

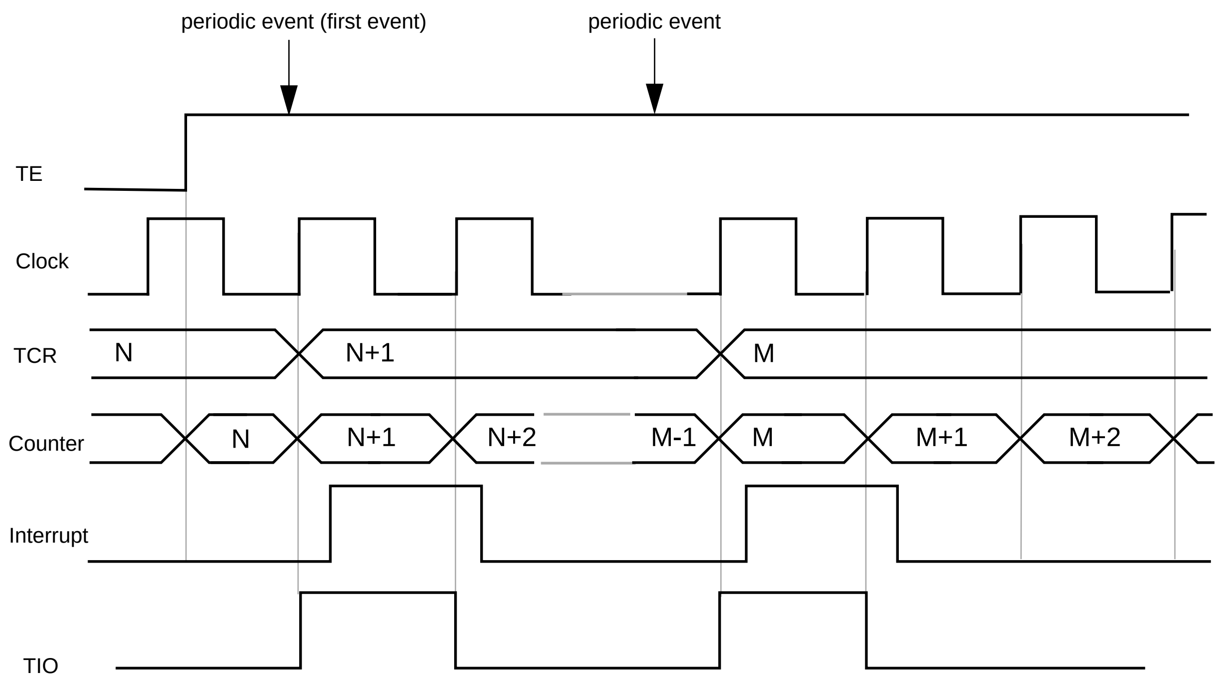


Figure 16 - Period Measurement Mode (INV=0)

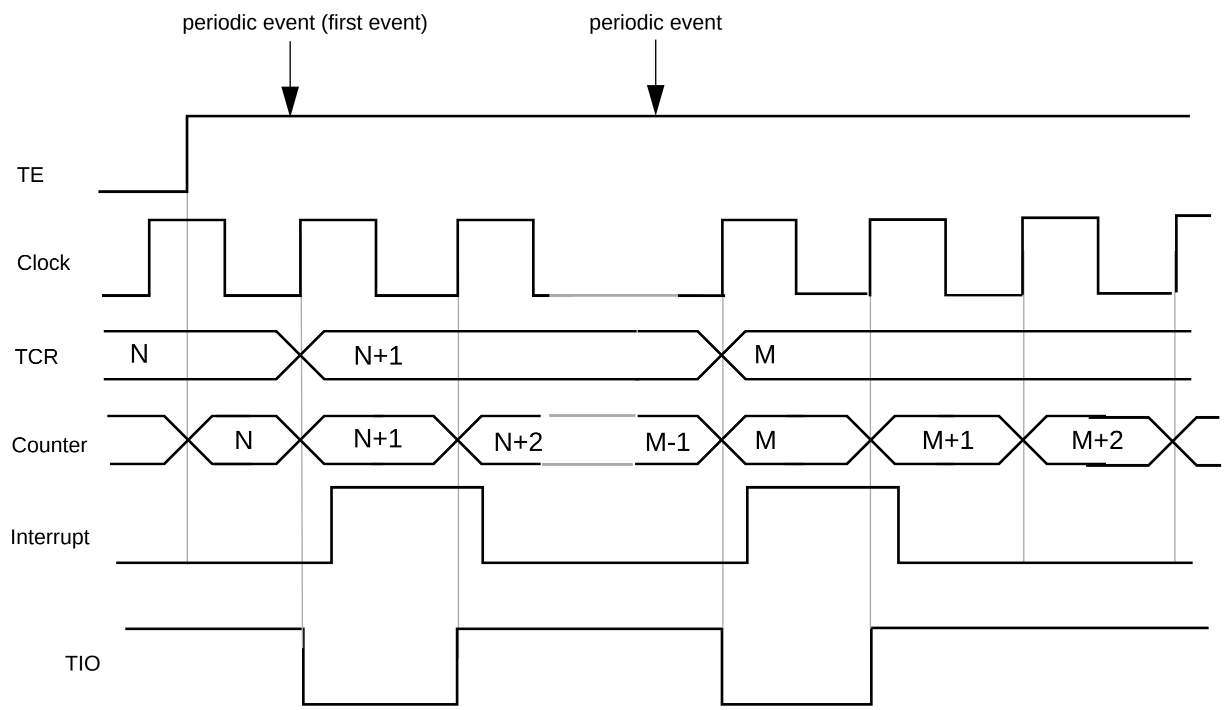


Figure 17 - Period Measurement Mode (INV=1)

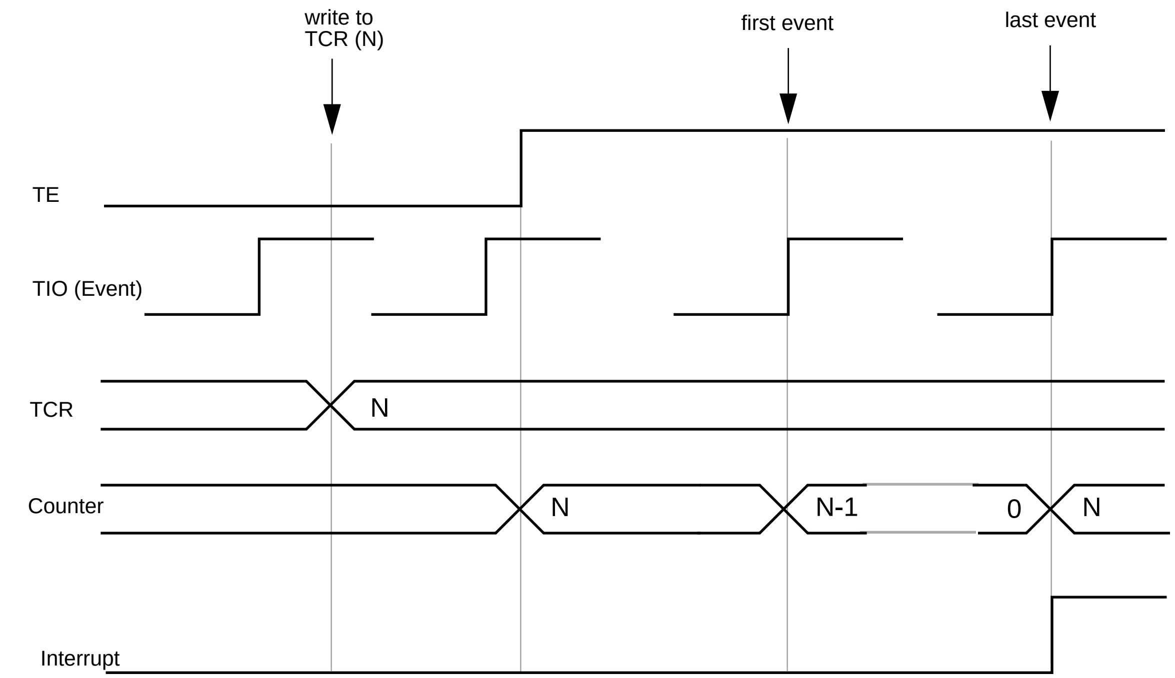


Figure 18 - Event Counter Mode, External Clock (INV=0)

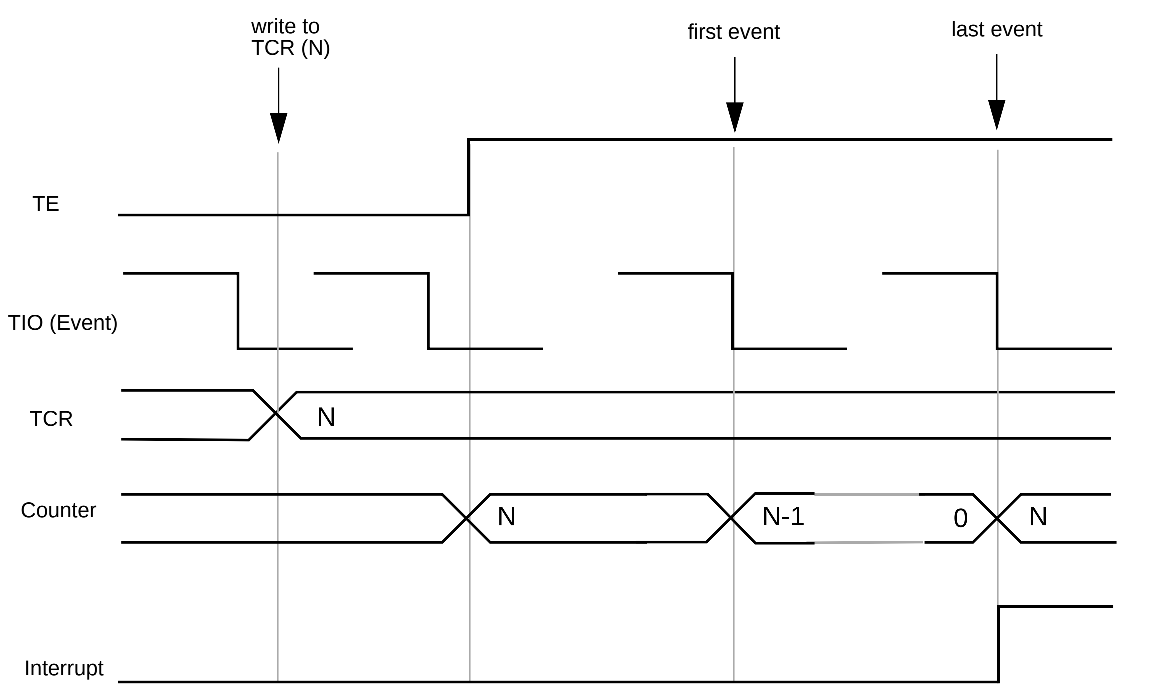


Figure 19 - Event Counter Mode, External Clock (INV=1)

6.5 TIMER BEHAVIOR DURING WAIT and STOP

During the execution of the WAIT instruction, the timer clocks are active and the timer activity continues undisturbed. If the timer interrupt is enabled when the final event occurs, an interrupt will be generated and serviced.

It is recommended that the timer be disabled before executing the STOP instruction because, during the execution of the STOP instruction, the timer clocks are disabled and the timer activity will be stopped. If, for example, the TIO pin is used as input, the changes that occur while the chip is in STOP mode will be ignored.

6.6 OPERATING CONSIDERATIONS

The value 0 for the Timer Count Register (TCR) is considered a boundary case and affects the behavior of the timer under the following conditions:

- If the TCR is loaded with 0, and the counter contained a non-zero value before the TCR was loaded, then after the timer is enabled, it will count 2^{32} events, generate an interrupt, and then generate an interrupt for every new event.
- If the TCR is loaded with 0, and the counter contained a zero value prior to loading, then after the timer is enabled, it will generate an interrupt for every event.
- If the TCR is loaded with 0 after the timer has been enabled, the timer will be loaded with 0 when the current count is completed and then generate an interrupt for every new event.

6.7 SOFTWARE EXAMPLES

6.7.1 General purpose IO input

The following routine can be used to read the TIO0 input pin:

```
            movep #$02000000,X:TCSR0      ;clear TC2-TC0, set GPIO
                                          ;and clear INV for GPIO input
here
            jset #22,x:TCSR0,here         ; spin here until TIO0 is set
......
```

6.7.2 General purpose IO output

The following routine can be used to write the TIO1 output pin:

```
movep #$02800000,x:TCSR1      ;clear TC2-TC0, set GPIO
                              ;and set DIR for GPIO output, set TIO1 to 0
movep #$02a00000,x:TCSR1      ; set TIO1 to 1
movep #$02800000,x:TCSR1      ; set TIO1 to 0
```

This routine generates a pulse on the TIO1 pin with the duration equal to 8 CLK (assuming no wait states, no external bus conflict etc.)

6.7.3 Standard timer mode (mode 0), input clock, no output and GPIO output

The following program illustrates the standard timer mode with simultaneous GPIO. The timer is used to activate an internal task after 65536 clocks; at the end of the task the TIO0 pin is toggled to signal end of task.

```
        org p:$14                 ; this is timer 0 interrupt vector address
        jsr task                  ; go and execute task (long interrupt)
....
        org p:main_body
.....
        movep #$42000000,x:TCSR0  ; enable timer interrupts and enable GPIO
                                  ; (input!) and set DO =0 to have stable data
        movep #$42800000,x:TCSR0  ; change DIR to output (clean 0, no spikes)
        movep #$0000ffff,x:TCR0   ; load 64k -1 into the counter
        bset #24,x:IPR            ; enable IPL for timer 0
        andi #$cf,mr              ; remove interrupt masking in status register
        bset #31,x:TCSR0          ; timer enable
......
; application program
.....
task
.....
; task instructions
....
end_of_task
        bset #22,x:TCSR0          ; set TIO0 to signal end of task
        bclr #22,x:TCSR0          ; clear TIO0
        rti                       ; return to main program
```

6.7.4 Pulse width measurement mode (mode 4)

The following program illustrates the use of the timer module for input pulse width measurement. The width is measured in this example for the low active period of the input pulse on the TIO1 pin and is stored in a table (in multiples of the chip operating clock divided by 2).

```
                org x:$100                          ; define buffer in X memory internal
pulse_width     ds              $100                ; measure up to 256 pulses

                org p:$16
; this is timer1 interrupt vector address
                movep x:TCR1,x:(r0)+                ; store width value in table
                nop                                 ; second word of the short interrupt
....
                org p:main_body
.....
                move #pulse_width,r0                ; r0 points to start of table
                move #$ff,m0                        ; modulo 100 to wrap around on end of table
                movep #$70000000,x:TCSR1            ; enable timer interrupts, mode 4 and set INV
                                                    ; to measure the low active pulse
                bset #26,x:IPR                      ; enable IPL for timer 1
                andi #$cf,mr                        ; remove interrupt masking in status register
                bset #31,x:TCSR1                    ; timer enable
......
; do other tasks
.....
```

6.7.5 Period measurement mode (mode 5)

The following program illustrates the usage of the timer module for input period measurement. The period is measured in this example between 0 to 1 transitions of the input signal on TIO0 and is stored in a table (in multiples of the chip operating clock divided by 2).

```
              org x:$100                            ; define buffer in X memory internal
period         ds                $100               ; measure up to 256 pulses
temp           ds                $1                 ; temporary storage

              org p:$14                             ; this is timer0 interrupt vector address
              jsr measure                           ; long interrupt to measure period
....
              org p:main_body
.....
              move #0,x:temp                        ; clear temporary storage
              move #period,r0                       ; r0 points to start of table
              move #$ff,m0                          ; modulo 100 to wrap around on end of table
              movep #$54000000,x:TCSR0              ; enable timer interrupts, mode5
              bset #26,x:IPR                        ; enable IPL for timer 1
              andi #$cf,mr                          ; remove interrupt masking in status register
              bset #31,x:TCSR0                      ; timer enable
......
; do other tasks
.....
measure
              movep x:TCR0,d0.l                     ; read new counter value
              move x:temp,d1.l                      ; retrieve former read value (initially zero)
              sub d1.l,d0.l     d0.l,x:temp         ; compute delta (i.e. new -old) and store the
                                                    ; new read value in temp
              move d0.l,x:(r0)+                     ; store period value in table
              rti
```

7 ADDITIONAL CHANGES

This section presents various other changes to the DSP96002 to support the addition of the Timer/Event Counter modules. Specifically, two new DMA mask bits (M7 and M8) were added to the DMA Control/Status Register. Figure 20 and Figure 20 indicate the changed DMA Controller Programming Models. Table 3 indicates the DMA Request Mask Bits functions. The DMA Controller Programming Model is discussed on Section 7 of the DSP96002 User's Manual (DSP96002UM/AD)

This section also presents the X Memory map, interrupt vector addresses, the list of priorities within an IPL, and the interrupt priority register for the DSP96002, all of which have been changed in support of the timer modules.

31 ... 0	Register
	DMA Source Modifier Register DSM0 addr X:$FFFFFFDF
	DMA Source Address Register DSR0 addr X:$FFFFFFDE
	DMA Source Offset Register DSN0 addr X:$FFFFFFDD
	DMA Destination Modifier Register DDM0 addr X:$FFFFFFDB
	DMA Destination Address Register DDR0 addr X:$FFFFFFDA
	DMA Destination Offset Register DDN0 addr X:$FFFFFFD9
	DMA Counter DCO0 addr X:$FFFFFFDC

DMA Control/Status Register
DCS0
addr X:$FFFFFFD8

31	30	29	28	27	26	25	24
DE	DIE	*	DTD	*	DTM1	DTM0	DMAP
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
DCP	*	*	*	*	*	*	**M8**
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
M7	M6	M5	M4	M3	M2	M1	M0
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
*	*	DSS2	DSS1	DSS0	DDS2	DDS1	DDS0

Figure 20 - DMA Controller Programming Model - Channel 0

31	0	
		DMA Source Modifier Register DSM1 addr X:$FFFFFFD7
		DMA Source Address Register DSR1 addr X:$FFFFFFD6
		DMA Source Offset Register DSN1 addr X:$FFFFFFD5
		DMA Destination Modifier Register DDM1 addr X:$FFFFFFD3
		DMA Destination Address Register DDR1 addr X:$FFFFFFD2
		DMA Destination Offset Register DDN1 addr X:$FFFFFFD1
		DMA Counter DCO1 addr X:$FFFFFFD4

DMA Control/Status Register
DCS1
addr X:$FFFFFFD0

31	30	29	28	27	26	25	24
DE	DIE	*	DTD	*	DTM1	DTM0	DMAP
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
DCP	*	*	*	*	*	*	**M8**
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
M7	M6	M5	M4	M3	M2	M1	M0
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
*	*	DSS2	DSS1	DSS0	DDS2	DDS1	DDS0

Figure 21 - DMA Controller Programming Model - Channel 1

7.1 DCS Reserved Bits (Bits 6, 7, 17-22, 27, 29)

These bits read as zero and should be written with zero for future compatibility.

7.2 DCS DMA Request Masks (M0-M8) Bits 8-16

The DMA Request mask bits select the source of DMA requests used to trigger DMA transfers. If a mask bit is set, the corresponding device is selected as the DMA request source. If the mask bit is cleared, the device is ignored. The DMA request sources may be the internal peripherals or external devices requesting service through the $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$ and $\overline{\text{IRQC}}$ pins. The external inputs behave as edge-triggered synchronous inputs. The mask bits are cleared by hardware and software reset. The internal DMA request sources are produced by ANDing the internal peripheral status bits with DE.

Each requesting device input is first individually ANDed with its respective mask bit (M0,M1,etc) and then all AND outputs are ORed together. The OR output goes to the edge-triggered latch whose output initiates the DMA transfer. If an input is unmasked, asserting that input will set the latch and initiate a DMA transfer. The DMA state machine clears the latch when accessing the DMA source address. If more than one requesting device input is enabled, the first edge on any input is latched and triggers a DMA transfer, and any other edge that appears before the latch is cleared will be ignored.

Table 3 DMA Request Mask Bits

DMA Request Mask Bit	Requesting Device
M0	External ($\overline{\text{IRQA}}$ pin)
M1	External ($\overline{\text{IRQB}}$ pin)
M2	External ($\overline{\text{IRQC}}$ pin)
M3	Port A Host Receive Data (HRDF=1)
M4	Port A Host Transmit Data (HTDE=1)
M5	Port B Host Receive Data (HRDF=1)
M6	Port B Host Transmit Data (HTDE=1)
M7	**Timer 0 (TS=1)**
M8	**Timer 1 (TS=1)**

Table 4 Internal I/O Memory Map of the X Data Memory Space

ADDRESS	REGISTER	ADDRESS	REGISTER
$FFFFFFFF	IPR - Interrupt Priority Register	**$FFFFFFE0**	**TCSR0 - Timer Control Status Register 0**
$FFFFFFFE	BCRA - Port A Bus Control Register	$FFFFFFDF	DSM0 -DMA CH0 Source Modifier Register
$FFFFFFFD	BCRB - Port B Bus Control Register	$FFFFFFDE	DSR0 -DMA CH0 Source Address Register
$FFFFFFFC	PSR - Port Select Register	$FFFFFFDD	DSN0 -DMA CH0 Source Offset Register
	: RESERVED :	$FFFFFFDC	DCO0 -DMA CH0 Counter Register
$FFFFFFF0	Reserved for OnCE Operation (OGDBR)	$FFFFFFDB	DDM0 -DMA CH0 Destination Modifier Register
$FFFFFFEF	HTXA/HRXA - HOSTA HTX/HRX Register	$FFFFFFDA	DDR0 -DMA CH0 Destination Address Register
$FFFFFFEE	HTXCA - HOSTA HTX Reg. and HMRC Clear	$FFFFFFD9	DDN0 -DMA CH0 Destination Offset Register
$FFFFFFED	HSRA - HOSTA Status Register	$FFFFFFD8	DCS0 -DMA CH0 Control/Status Register
$FFFFFFEC	HCRA - HOSTA Control Register	$FFFFFFD7	DSM1 -DMA CH1 Source Modifier Register
:	RESERVED :	$FFFFFFD6	DSR1 -DMA CH1 Source Address Register
$FFFFFFE9	**TCR1 - Timer Count Register 1**	$FFFFFFD5	DSN1 -DMA CH1 Source Offset Register
$FFFFFFE8	**TCSR1 - Timer Control Status Register 1**	$FFFFFFD4	DCO1 -DMA CH1 Counter Register
$FFFFFFE7	HTXB/HRXB - HOSTB HTX/HRX Register	$FFFFFFD3	DDM1 -DMA CH1 Destination Modifier Register
$FFFFFFE6	HTXCB - HOSTB HTX Reg. and HMRC Clear	$FFFFFFD2	DDR1 -DMA CH1 Destination Address Register
$FFFFFFE5	HSRB - HOSTB Status Register	$FFFFFFD1	DDN1 -DMA CH1 Destination Offset Register
$FFFFFFE4	HCRB - HOSTB Control Register	$FFFFFFD0	DCS0 -DMA CH1 Control/Status Register
$FFFFFFE3	RESERVED :	$FFFFFFCF	RESERVED
$FFFFFFE2	RESERVED	:	RESERVED :
$FFFFFFE1	**TCR0 - Timer Count Register 0**	$FFFFFF80	RESERVED

Table 5 Interrupt Vector Addresses

Interrupt Starting Address	Interrupt Source
$FFFFFFFE	Hardware RESET
$00000000	Hardware RESET
$00000002	Stack Error
$00000004	Illegal Instruction
$00000006	(F)TRAPcc (default)
$00000008	IRQA
$0000000A	IRQB
$0000000C	IRQC
$0000000E	Reserved
$00000010	DMA Channel 1
$00000012	DMA Channel 2
$00000014	**Timer 0**
$00000016	**Timer 1**
$00000018	Reserved
$0000001A	Reserved
$0000001C	Host A Command (default)
$0000001E	Host B Command (default)
$00000020	Host A Receive Data
$00000022	Host A Transmit Data
$00000024	Host A Read X Memory
$00000026	Host A Read Y Memory

Interrupt Starting Address	Interrupt Source
$00000028	Host A Read P Memory
$0000002A	Host A Write X Memory
$0000002C	Host A Write Y Memory
$0000002E	Host A Write P Memory
$00000030	Host B Receive Data
$00000032	Host B Transmit Data
$00000034	Host B Read X Memory
$00000036	Host B Read Y Memory
$00000038	Host B Read P Memory
$0000003A	Host B Write X Memory
$0000003C	Host B Write Y Memory
$0000003E	Host B Write P Memory
$00000040	Reserved
:	:
$000000FE	Reserved
$00000100	User interrupt vector
:	:
$000001FE	User interrupt vector

7.3 Exception Priorities within an IPL

If more than one exception is pending when an instruction is executed, the interrupt with the highest priority level is serviced first. Within a given interrupt priority level, a second priority structure determines which interrupt is serviced when multiple interrupt requests with the same IPL are pending.

Table 6 DSP96002 Exception Priorities within an IPL

Priority	Exception	Enabled by
highest	Hardware RESET	-
	Illegal Instruction	-
	Stack Error	-
	(F)TRAPcc	-
	IRQA (External Interrupt)	(IPR) IAL1-IAL0
	IRQB (External Interrupt)	(IPR) IBL1-IBL0
	IRQC (External Interrupt)	(IPR) ICL1-ICL0
	Host A Command Interrupt	(HCR) HCIE
	Host A Receive Data Interrupt	(HCR) HRIE
	Host A Read X Memory Interrupt	(HCR) HXRE
	Host A Read Y Memory Interrupt	(HCR) HYRE
	Host A Read P Memory Interrupt	(HCR) HPRE
	Host A Write X Memory Interrupt	(HCR) HXWE
	Host A Write Y Memory Interrupt	(HCR) HYWE
	Host A Write P Memory Interrupt	(HCR) HPWE
	Host A Transmit Data Interrupt	(HCR) HTIE
	Host B Command Interrupt	(HCR) HCIE
	Host B Receive Data Interrupt	(HCR) HRIE
	Host B Read X Memory Interrupt	(HCR) HXRE
	Host B Read Y Memory Interrupt	(HCR) HYRE
	Host B Read P Memory Interrupt	(HCR) HPRE
	Host B Write X Memory Interrupt	(HCR) HXWE
	Host B Write Y Memory Interrupt	(HCR) HYWE
	Host B Write P Memory Interrupt	(HCR) HPWE
	Host B Transmit Data Interrupt	(HCR) HTIE
	DMA Channel 0 Interrupt	(DCS0) DIE0
	DMA Channel 1 Interrupt	(DCS1) DIE1
	Timer0 Interrupt	**(TCSR0) TIE0**
lowest	**Timer1Interrupt**	**(TCSR1) TIE1**

7.4 Interrupt Priority Register (IPR)

The Interrupt Priority Register supports the timer module with the addition of the Timer0 and Timer1 priority level bits. Figure 21 shows the revised IPR with the new bits indicated in bold characters.

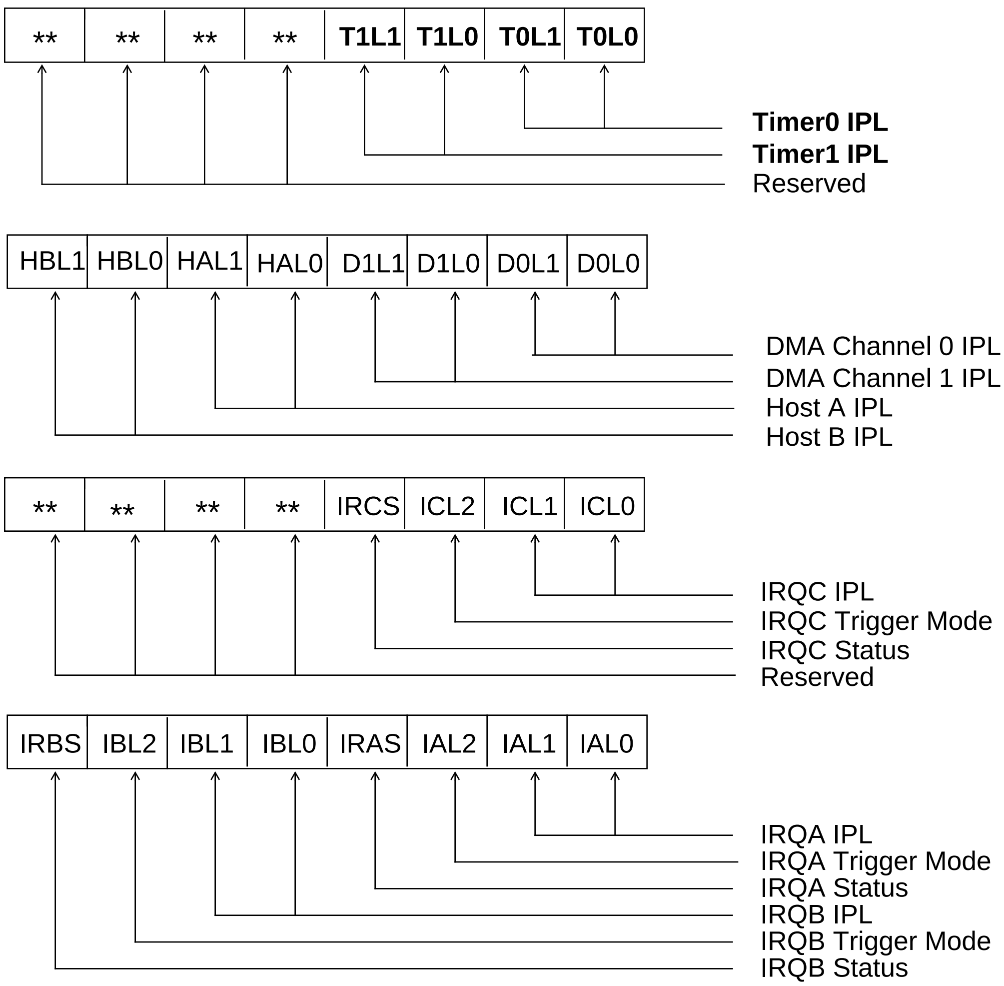


Figure 21 - Interrupt Priority Register (Address X:$FFFFFFFF)

7.4.1 Reserved bits (Bits 12-15, 28-31)

These reserved bits read as zero and should be written with zero for future compatibility.

7.4.2 Timer 0 Interrupt Priority Level - T0L1-T0L0 (Bits 24-25)

The Timer 0 Interrupt Priority Level (T0L1-T0L0) bits are used to enable and specify the priority level of the Timer 0 interrupt.

T0L1	T0L0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

7.4.3 Timer 1 Interrupt Priority Level - T1L1-T1L0 (Bits 26-27)

The Timer 1 Interrupt Priority Level (T1L1-T1L0) bits are used to enable and specify the priority level of the Timer 1 interrupt.

T1L1	T1L0	Enabled	Int. Priority Level (IPL)
0	0	no	-
0	1	yes	0
1	0	yes	1
1	1	yes	2

APPENDIX A – INSTRUCTION SET ADDENDUM DETAILS

The following pages present a detailed description of the new instructions added to the DSP96002 instruction set.

Integer Signed Multiply and Add

MPYS//ADD

Operation:

$S1.L * S2.L \rightarrow D1.M{:}D1.L$

(parallel data bus move)

$S3.L + D2.L \rightarrow D2.L$

Assembler Syntax:

MPYS S1,S2,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

MPYS S2,S1,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two signed operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit signed integer stored in the middle and low portions of D1.

Simultaneously, add the low portion of the two operands S3 and D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Addition Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if carry is generated from the MSB of the addition result. Cleared otherwise.
V	- Set if the addition result overflows. Cleared otherwise.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYS S1,S2,D1 ADD S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	00 1sss ddQQ QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYS//SUB Integer Signed Multiply and Subtract MPYS//SUB

Operation:

S1.L * S2.L → D1.M:D1.L

(parallel data bus move)

D2.L - S3.L → D2.L

Assembler Syntax:

MPYS S1,S2,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

MPYS S2,S1,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two signed operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit signed integer stored in the middle and low portions of D1.

Simultaneously, subtract the low portion of the specified source operand S3 from the low portion of the destination operand D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Subtraction Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if borrow is generated from the MSB of the subtraction result. Cleared **otherwise.**
V	- Set if the subtraction result overflows. Cleared otherwise.
Z	- Set if result of the subtraction is zero. Cleared otherwise.
N	- Set if result of the subtraction is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYS S1,S2,D1 SUB S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	01	1sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**QQQQ**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYU//ADD Integer Unsigned Multiply and Add MPYU//ADD

Operation:

$S1.L * S2.L \rightarrow D1.M:D1.L$

(parallel data bus move)

$S3.L + D2.L \rightarrow D2.L$

Assembler Syntax:

MPYU S1,S2,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

MPYU S2,S1,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two unsigned operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit unsigned integer stored in the middle and low portions of D1.

Simultaneously, add the low portion of the two operands S3 and D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Addition Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if carry is generated from the MSB of the addition result. Cleared otherwise.
V	- Set if the addition result overflows. Cleared otherwise.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYU S1,S2,D1 ADD S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	00	0sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYU//SUB Integer Unsigned Multiply and Subtract MPYU//SUB

Operation:

$S1.L * S2.L \rightarrow D1.M:D1.L$

(parallel data bus move)

$D2.L - S3.L \rightarrow D2.L$

Assembler Syntax:

MPYU S1,S2,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

MPYU S2,S1,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two unsigned operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit unsigned integer stored in the middle and low portions of D1.

Simultaneously, subtract the low portion of the specified source operand S3 from the low portion of the destination operand D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Subtraction Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if borrow is generated from the MSB of the subtraction result. Cleared **otherwise.**
V	- Set if the subtraction result overflows. Cleared otherwise.
Z	- Set if result of the subtraction is zero. Cleared otherwise.
N	- Set if result of the subtraction is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYU S1,S2,D1 SUB S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	01	0sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:	**Assembler Syntax:**
Flush instruction cache	PFLUSH

Description:

Flush the whole instruction cache, unlock all cache sectors, set the LRU stack and tag registers to their default values.

The PFLUSH instruction is enabled both in Cache Mode and PRAM Mode.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PFLUSH

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0011

Instruction Fields:

None

Timing: 2 oscillator clock cycles

Memory: 1 program words

PFREE Program-Cache Global Unlock PFREE

Operation:
Unlock all locked sectors

Assembler Syntax:
PFREE

Description:

Unlock all the locked cache sectors in the instruction cache.

The PFREE instruction is enabled both in Cache Mode and PRAM Mode.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PFREE

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0010

Instruction Fields:
None

Timing: 2 oscillator clock cycles

Memory: 1 program words

PLOCK Program-Cache-Sector Lock PLOCK

Operation:
Lock sector by ea

Assembler Syntax:
PLOCK ea

Description:

Lock the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, then load the least recently used cache sector tag with the 25 most significant bits of the specified address and then lock that cache sector. Update the LRU stack accordingly. All memory alterable addressing modes may be used for the effective address, but not a short absolute address.

The PLOCK instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PLOCK ea

31				14	13			0
0000	0011	0000	MMMR	RR	00	0000	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PLOCKR Program-Cache-Sector Relative Lock PLOCKR

Operation:

Lock sector by PC + xx

Lock sector by PC + xxxx

Lock sector by PC + Rn

Assembler Syntax:

PLOCKR label

PLOCKR Rn

Description:

Lock the cache sector to which the sum PC + specified displacement belongs. If the sum does not belong to any cache sector, then load the least recently used cache sector tag with the 25 most significant bits of the sum and then lock that cache sector. Update the LRU stack accordingly.

The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the address to be locked. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the 32-bit PC Relative Displacement.

The PLOCKR instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PLOCKR label (short)

31				14	13			0
0000	0011	10aa	aaaa	aa	00	0000	0aaa	aaaa

Instruction Format: PLOCKR label

31				14	13			0
0000	0011	0000	1000	00	00	0000	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: PLOCKR Rn

31				14	13			0
0000	0011	0000	001R	RR	00	0000	0000	0000

Instruction Fields:

Rn - R0-R7

Long PC Relative Displacement - 32 bits

Short PC Relative Displacement - aaaaaaaaaaaaaaa (15 bits)

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PUNLOCK Program-Cache-Sector Unlock PUNLOCK

Operation:
Unlock sector by ea

Assembler Syntax:
PUNLOCK ea

Description:

Unlock the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, and is therefore definitely unlocked, nevertheless, load the least recently used cache sector tag with the 25 most significant bits of the specified address. Update the LRU stack accordingly. All memory alterable addressing modes may be used for the effective address, but not a short absolute address.

The PUNLOCK instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PUNLOCK ea

31				14	13			0
0000	0011	0100	MMMR	RR	00	0000	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PUNLOCKR Program-Cache-Sector Relative Unlock PUNLOCKR

Operation:

Unlock sector by PC + xx

Unlock sector by PC + xxxx

Unlock sector by PC + Rn

Assembler Syntax:

PUNLOCKR label

PUNLOCKR Rn

Description:

Unlock the cache sector to which the sum PC + specified displacement belongs. If the sum does not belong to any cache sector, and is therefore definitely unlocked, nevertheless, load the least recently used cache sector tag with the 25 most significant bits of the sum. Update the LRU stack accordingly.

The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the address to be locked. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the 32-bit PC Relative Displacement.

The PUNLOCKR instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PUNLOCKR label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	00	0000	0aaa	aaaa

Instruction Format: PUNLOCKR label

31				14	13			0
0000	0011	0100	1000	00	00	0000	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: PUNLOCKR Rn

31				14	13			0
0000	0011	0100	001R	RR	00	0000	0000	0000

Instruction Fields:

Rn - R0-R7

Long PC Relative Displacement - 32 bits

Short PC Relative Displacement - aaaaaaaaaaaaaaa (15 bits)

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

MOTOROLA
SEMICONDUCTOR
TECHNICAL DATA

DSP96002

Addendum to the
DSP96002 Digital Signal Processor Instruction Set
found in the
DSP96002 Digital Signal Processor User's Manual
and the
DSP96002 CLAS Documentation

FOREWORD

The following ten instructions have been added to the DSP96002 instruction set. These instructions are available only on versions of the DSP96002 that have an instruction cache. The silicon mask numbers for the DSP96002s that **do not have** these instructions available are:

- C15T
- D12C
- D91G
- D35G

All later mask numbers have these instructions available. This mask number can be found on the top of the chip along with the chip designation and other numbers.

The descriptions of these new instructions can also be found in the addendum to the DSP96002 Digital Signal Processor User's Manual — The DSP96002 Instruction Cache and 32-bit Timer/event Counter (order number DSP96002UMAD/AD).

MPYS//ADD Integer Signed Multiply and Add MPYS//ADD

Operation:

$S1.L * S2.L \rightarrow D1.M:D1.L$

(parallel data bus move)

$S3.L + D2.L \rightarrow D2.L$

Assembler Syntax:

MPYS S1,S2,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

MPYS S2,S1,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two signed operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit signed integer stored in the middle and low portions of D1.

Simultaneously, add the low portion of the two operands S3 and D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Addition Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if carry is generated from the MSB of the addition result. Cleared otherwise.
V	- Set if the addition result overflows. Cleared otherwise.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYS S1,S2,D1 ADD S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	00 1sss ddQQ QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYS//SUB Integer Signed Multiply and Subtract MPYS//SUB

Operation:

$S1.L * S2.L \rightarrow D1.M:D1.L$

(parallel data bus move)

$D2.L - S3.L \rightarrow D2.L$

Assembler Syntax:

MPYS S1,S2,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

MPYS S2,S1,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two signed operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit signed integer stored in the middle and low portions of D1.

Simultaneously, subtract the low portion of the specified source operand S3 from the low portion of the destination operand D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Subtraction Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if borrow is generated from the MSB of the subtraction result. Cleared **otherwise.**
V	- Set if the subtraction result overflows. Cleared otherwise.
Z	- Set if result of the subtraction is zero. Cleared otherwise.
N	- Set if result of the subtraction is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYS S1,S2,D1 SUB S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	01	1sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYU//ADD — Integer Unsigned Multiply and Add — MPYU//ADD

Operation:

$S1.L * S2.L \rightarrow D1.M:D1.L$

(parallel data bus move)

$S3.L + D2.L \rightarrow D2.L$

Assembler Syntax:

MPYU S1,S2,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

MPYU S2,S1,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two unsigned operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit unsigned integer stored in the middle and low portions of D1.

Simultaneously, add the low portion of the two operands S3 and D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Addition Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if carry is generated from the MSB of the addition result. Cleared otherwise.
V	- Set if the addition result overflows. Cleared otherwise.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYU S1,S2,D1 ADD S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	00	0sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYU//SUB   Integer Unsigned Multiply and Subtract   MPYU//SUB

Operation:

S1.L $*$ S2.L $\rightarrow$ D1.M:D1.L

(parallel data bus move)

D2.L - S3.L $\rightarrow$ D2.L

Assembler Syntax:

MPYU S1,S2,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

MPYU S2,S1,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two unsigned operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit unsigned integer stored in the middle and low portions of D1.

Simultaneously, subtract the low portion of the specified source operand S3 from the low portion of the destination operand D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Subtraction Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if borrow is generated from the MSB of the subtraction result. Cleared **otherwise.**
V	- Set if the subtraction result overflows. Cleared otherwise.
Z	- Set if result of the subtraction is zero. Cleared otherwise.
N	- Set if result of the subtraction is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYU S1,S2,D1 SUB S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	01 0sss ddQQ QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

Operation:
Flush instruction cache

Assembler Syntax:
PFLUSH

Description:

Flush the whole instruction cache, unlock all cache sectors, set the LRU stack and tag registers to their default values.

The PFLUSH instruction is enabled both in Cache Mode and PRAM Mode.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PFLUSH

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0011

Instruction Fields:
None

Timing: 2 oscillator clock cycles

Memory: 1 program words

PFREE Program-Cache Global Unlock PFREE

Operation:
Unlock all locked sectors

Assembler Syntax:
PFREE

Description:

Unlock all the locked cache sectors in the instruction cache.

The PFREE instruction is enabled both in Cache Mode and PRAM Mode.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PFREE

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0010

Instruction Fields:

None

Timing: 2 oscillator clock cycles

Memory: 1 program words

PLOCK Program-Cache-Sector Lock PLOCK

Operation:
Lock sector by ea

Assembler Syntax:
PLOCK ea

Description:

Lock the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, then load the least recently used cache sector tag with the 25 most significant bits of the specified address and then lock that cache sector. Update the LRU stack accordingly. All memory alterable addressing modes may be used for the effective address, but not a short absolute address.

The PLOCK instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PLOCK ea

31				14	13			0
0000	0011	0000	MMMR	RR	00	0000	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PLOCKR   Program-Cache-Sector Relative Lock   PLOCKR

Operation:

Lock sector by PC + xx

Lock sector by PC + xxxx

Lock sector by PC + Rn

Assembler Syntax:

PLOCKR label

PLOCKR Rn

Description:

Lock the cache sector to which the sum PC + specified displacement belongs. If the sum does not belong to any cache sector, then load the least recently used cache sector tag with the 25 most significant bits of the sum and then lock that cache sector. Update the LRU stack accordingly.

The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the address to be locked. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the 32-bit PC Relative Displacement.

The PLOCKR instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PLOCKR label (short)

31				14	13			0
0000	0011	10aa	aaaa	aa	00	0000	0aaa	aaaa

Instruction Format: PLOCKR label

31				14	13			0
0000	0011	0000	1000	00	00	0000	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: PLOCKR Rn

31				14	13			0
0000	0011	0000	001R	RR	00	0000	0000	0000

Instruction Fields:

Rn - R0-R7

Long PC Relative Displacement - 32 bits

Short PC Relative Displacement - aaaaaaaaaaaaaaa (15 bits)

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PUNLOCK Program-Cache-Sector Unlock PUNLOCK

Operation:

Unlock sector by ea

Assembler Syntax:

PUNLOCK ea

Description:

Unlock the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, and is therefore definitely unlocked, nevertheless, load the least recently used cache sector tag with the 25 most significant bits of the specified address. Update the LRU stack accordingly. All memory alterable addressing modes may be used for the effective address, but not a short absolute address.

The PUNLOCK instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PUNLOCK ea

31				14	13			0
0000	0011	0100	MMMR	RR	00	0000	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PUNLOCKR Program-Cache-Sector Relative Unlock PUNLOCKR

Operation:

Unlock sector by PC + xx

Unlock sector by PC + xxxx

Unlock sector by PC + Rn

Assembler Syntax:

PUNLOCKR label

PUNLOCKR Rn

Description:

Unlock the cache sector to which the sum PC + specified displacement belongs. If the sum does not belong to any cache sector, and is therefore definitely unlocked, nevertheless, load the least recently used cache sector tag with the 25 most significant bits of the sum. Update the LRU stack accordingly.

The displacement is a 2’s complement 32-bit integer that represents the relative distance from the current PC to the address to be locked. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the 32-bit PC Relative Displacement.

The PUNLOCKR instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PUNLOCKR label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	00	0000	0aaa	aaaa

Instruction Format: PUNLOCKR label

31				14	13			0
0000	0011	0100	1000	00	00	0000	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: PUNLOCKR Rn

31				14	13			0
0000	0011	0100	001R	RR	00	0000	0000	0000

Instruction Fields:

Rn - R0-R7

Long PC Relative Displacement - 32 bits

Short PC Relative Displacement - aaaaaaaaaaaaaaa (15 bits)

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

Literature Distribution Centers:

USA: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.

EUROPE: Motorola Ltd.; European Literature Center; 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.

JAPAN: Nippon Motorola Ltd.; 4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141 Japan.

ASIA-PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbor Center, No. 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong.

Addendum

DSP96002

Addendum to the

DSP96002 Digital Signal Processor Instruction Set found in the DSP96002 Digital Signal Processor User's Manual and the DSP96002 CLAS Documentation

FOREWORD

The following ten instructions have been added to the DSP96002 instruction set. These instructions are available only on versions of the DSP96002 that have an instruction cache. The silicon mask numbers for the DSP96002s that **do not have** these instructions available are:

- C15T
- D12C
- D91G
- D35G

All later mask numbers have these instructions available. This mask number can be found on the top of the chip along with the chip designation and other numbers.

The descriptions of these new instructions can also be found in the addendum to the DSP96002 Digital Signal Processor User's Manual — The DSP96002 Instruction Cache and 32-bit Timer/event Counter (order number DSP96002UMAD/AD).

MPYS//ADD Integer Signed Multiply and Add MPYS//ADD

Operation:

$S1.L * S2.L \rightarrow D1.M{:}D1.L$

(parallel data bus move)

$S3.L + D2.L \rightarrow D2.L$

Assembler Syntax:

MPYS S1,S2,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

MPYS S2,S1,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two signed operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit signed integer stored in the middle and low portions of D1.

Simultaneously, add the low portion of the two operands S3 and D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Addition Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if carry is generated from the MSB of the addition result. Cleared otherwise.
V	- Set if the addition result overflows. Cleared otherwise.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYS S1,S2,D1 ADD S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	00 1sss ddQQ QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA	

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYS//SUB Integer Signed Multiply and Subtract MPYS//SUB

Operation:

S1.L * S2.L → D1.M:D1.L

(parallel data bus move)

D2.L - S3.L → D2.L

Assembler Syntax:

MPYS S1,S2,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

MPYS S2,S1,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two signed operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit signed integer stored in the middle and low portions of D1.

Simultaneously, subtract the low portion of the specified source operand S3 from the low portion of the destination operand D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Subtraction Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if borrow is generated from the MSB of the subtraction result. Cleared **otherwise.**
V	- Set if the subtraction result overflows. Cleared otherwise.
Z	- Set if result of the subtraction is zero. Cleared otherwise.
N	- Set if result of the subtraction is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYS S1,S2,D1 SUB S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	01	1sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYU//ADD Integer Unsigned Multiply and Add MPYU//ADD

Operation:

$S1.L * S2.L \rightarrow D1.M:D1.L$

(parallel data bus move)

$S3.L + D2.L \rightarrow D2.L$

Assembler Syntax:

MPYU S1,S2,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

MPYU S2,S1,D1 ADD S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two unsigned operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit unsigned integer stored in the middle and low portions of D1.

Simultaneously, add the low portion of the two operands S3 and D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Addition Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if carry is generated from the MSB of the addition result. Cleared otherwise.
V	- Set if the addition result overflows. Cleared otherwise.
Z	- Set if result of the addition is zero. Cleared otherwise.
N	- Set if result of the addition is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYU S1,S2,D1 ADD S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	00	0sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

MPYU//SUB Integer Unsigned Multiply and Subtract MPYU//SUB

Operation:

$S1.L * S2.L \rightarrow D1.M:D1.L$

(parallel data bus move)

$D2.L - S3.L \rightarrow D2.L$

Assembler Syntax:

MPYU S1,S2,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

MPYU S2,S1,D1 SUB S3,D2

(move syntax - see the MOVE instruction description.)

Description:

Multiply the two unsigned operands S1 and S2 and store the product in the specified destination register D1. The two source operands S1and S2 are 32-bit integers and are taken from the low portion of S1 and S2. The result is a 64-bit unsigned integer stored in the middle and low portions of D1.

Simultaneously, subtract the low portion of the specified source operand S3 from the low portion of the destination operand D2 and store the result in the low portion of the destination operand D2.

This instruction is enabled only in Integer Mode.

Input Operand(s) Precision: 32-bit integer.

Subtraction Output Operand Precision: 32-bit integer.

Multiplication Output Operand Precision: 64-bit integer.

CCR Condition Codes:

C	- Set if borrow is generated from the MSB of the subtraction result. Cleared **otherwise.**
V	- Set if the subtraction result overflows. Cleared otherwise.
Z	- Set if result of the subtraction is zero. Cleared otherwise.
N	- Set if result of the subtraction is negative. Cleared otherwise.
I	- Not affected.
LR	- Not affected.
$\overline{R}$	- Not affected.
A	- Not affected.

ER Status Bits: Not affected

IER Flags: Not affected

Instruction Format: MPYU S1,S2,D1 SUB S3,D2 (move syntax - see the MOVE instruction description.)

DATA BUS MOVE FIELD	01	0sss	ddQQ	QQDD
OPTIONAL EFFECTIVE ADDRESS EXTENSION OR IMMEDIATE LONG DATA				

Instruction Fields:

D1	**D D**	
Dn	n n	where nn = 0-3

D2	**d d**	
Dn	n n	where nn = 0-3

S3	**s s s**	
Dn	n n n	where nnn = 0-7

S1*S2	**Q Q Q Q**
D0*D4	0 0 0 0
D4*D4	0 0 0 1
D4*D5	0 0 1 0
D4*D6	0 0 1 1
D5*D6	0 1 0 0
D4*D7	0 1 0 1
D5*D7	0 1 1 0
D6*D7	0 1 1 1
D4*D8	1 0 0 0
D5*D8	1 0 0 1
D6*D8	1 0 1 0
D7*D8	1 0 1 1
D4*D9	1 1 0 0
D5*D9	1 1 0 1
D6*D9	1 1 1 0
D7*D9	1 1 1 1

Timing: 2 + mv oscillator clock cycles

Memory: 1 + mv program words

PFLUSH — Program-Cache Flush

Operation:
Flush instruction cache

Assembler Syntax:
PFLUSH

Description:

Flush the whole instruction cache, unlock all cache sectors, set the LRU stack and tag registers to their default values.

The PFLUSH instruction is enabled both in Cache Mode and PRAM Mode.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PFLUSH

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0011

Instruction Fields:
None

Timing: 2 oscillator clock cycles

Memory: 1 program words

PFREE Program-Cache Global Unlock PFREE

Operation:
Unlock all locked sectors

Assembler Syntax:
PFREE

Description:

Unlock all the locked cache sectors in the instruction cache.

The PFREE instruction is enabled both in Cache Mode and PRAM Mode.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PFREE

31				14	13			0
0000	0000	0000	0000	00	00	0000	0000	0010

Instruction Fields:

None

Timing: 2 oscillator clock cycles

Memory: 1 program words

PLOCK Program-Cache-Sector Lock PLOCK

Operation:

Lock sector by ea

Assembler Syntax:

PLOCK ea

Description:

Lock the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, then load the least recently used cache sector tag with the 25 most significant bits of the specified address and then lock that cache sector. Update the LRU stack accordingly. All memory alterable addressing modes may be used for the effective address, but not a short absolute address.

The PLOCK instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PLOCK ea

31				14	13			0
0000	0011	0000	MMMR	RR	00	0000	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PLOCKR Program-Cache-Sector Relative Lock PLOCKR

Operation:
Lock sector by PC + xx
Lock sector by PC + xxxx
Lock sector by PC + Rn

Assembler Syntax:
PLOCKR label
PLOCKR Rn

Description:

Lock the cache sector to which the sum PC + specified displacement belongs. If the sum does not belong to any cache sector, then load the least recently used cache sector tag with the 25 most significant bits of the sum and then lock that cache sector. Update the LRU stack accordingly.

The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the address to be locked. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the 32-bit PC Relative Displacement.

The PLOCKR instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PLOCKR label (short)

31				14	13			0
0000	0011	10aa	aaaa	aa	00	0000	0aaa	aaaa

Instruction Format: PLOCKR label

31				14	13			0
0000	0011	0000	1000	00	00	0000	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: PLOCKR Rn

31				14	13			0
0000	0011	0000	001R	RR	00	0000	0000	0000

Instruction Fields:

Rn - R0-R7

Long PC Relative Displacement - 32 bits

Short PC Relative Displacement - aaaaaaaaaaaaaaa (15 bits)

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PUNLOCK Program-Cache-Sector Unlock PUNLOCK

Operation:

Unlock sector by ea

Assembler Syntax:

PUNLOCK ea

Description:

Unlock the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, and is therefore definitely unlocked, nevertheless, load the least recently used cache sector tag with the 25 most significant bits of the specified address. Update the LRU stack accordingly. All memory alterable addressing modes may be used for the effective address, but not a short absolute address.

The PUNLOCK instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PUNLOCK ea

31				14	13			0
0000	0011	0100	MMMR	RR	00	0000	1000	0000
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

ea Rn - R0-R7 (Memory alterable addressing modes only)

Absolute Address - 32 bits

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

PUNLOCKR Program-Cache-Sector Relative Unlock PUNLOCKR

Operation:

Unlock sector by PC + xx

Unlock sector by PC + xxxx

Unlock sector by PC + Rn

Assembler Syntax:

PUNLOCKR label

PUNLOCKR Rn

Description:

Unlock the cache sector to which the sum PC + specified displacement belongs. If the sum does not belong to any cache sector, and is therefore definitely unlocked, nevertheless, load the least recently used cache sector tag with the 25 most significant bits of the sum. Update the LRU stack accordingly.

The displacement is a 2's complement 32-bit integer that represents the relative distance from the current PC to the address to be locked. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 15-bit data is sign extended to form the 32-bit PC Relative Displacement.

The PUNLOCKR instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap to be taken.

CCR Condition Codes: Not affected.

ER Status Bits: Not affected.

IER Flags: Not affected.

Instruction Format: PUNLOCKR label (short)

31				14	13			0
0000	0011	11aa	aaaa	aa	00	0000	0aaa	aaaa

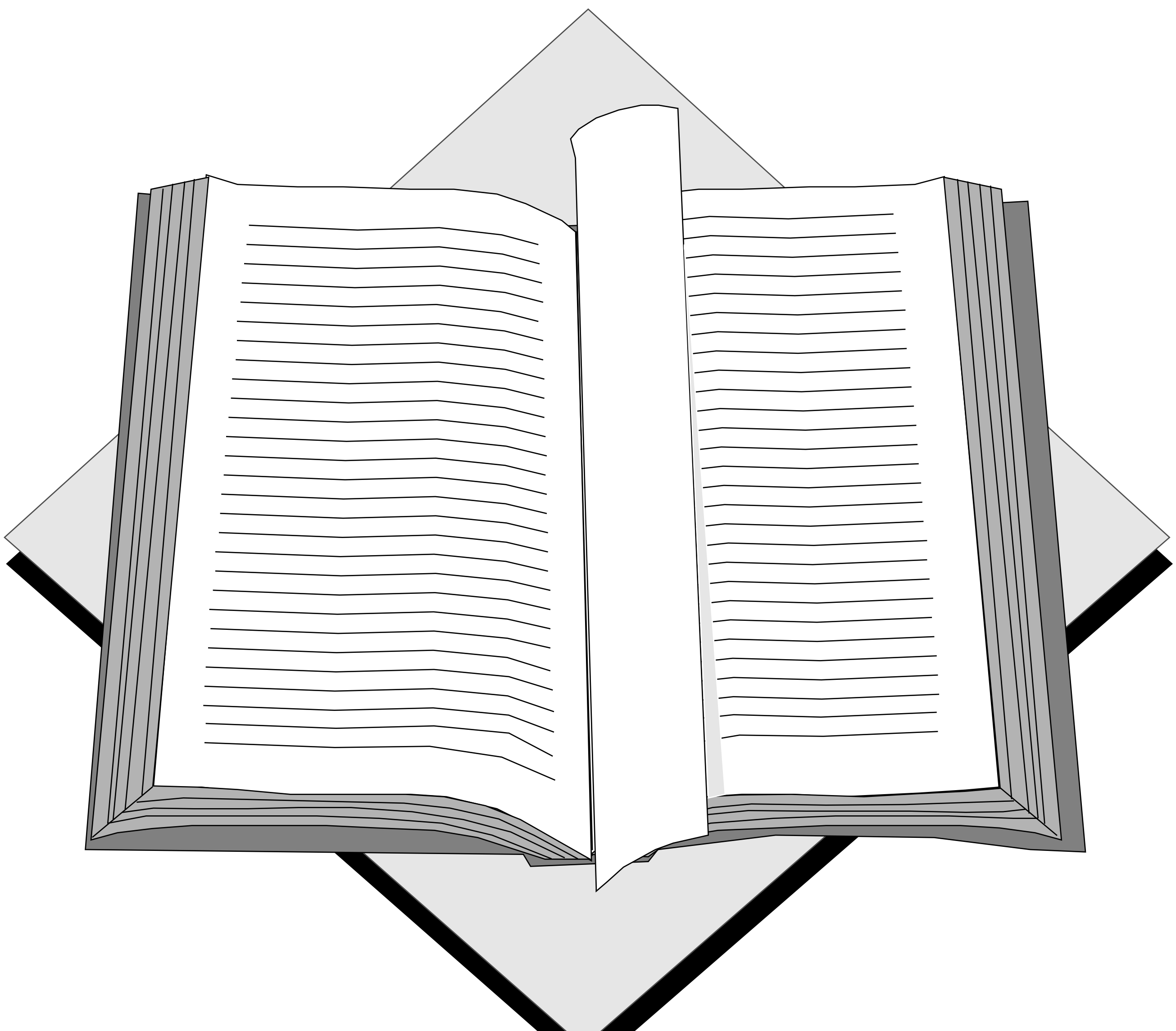
Instruction Format: PUNLOCKR label

31				14	13			0
0000	0011	0100	1000	00	00	0000	0000	0000
PC RELATIVE DISPLACEMENT								

Instruction Format: PUNLOCKR Rn

31				14	13			0
0000	0011	0100	001R	RR	00	0000	0000	0000

INDEX



INDEX

—D—

Instruction Fields:

Rn - R0-R7

Long PC Relative Displacement - 32 bits

Short PC Relative Displacement - aaaaaaaaaaaaaaa (15 bits)

Timing: 4 + ea oscillator clock cycles

Memory: 1 + ea program words

Literature Distribution Centers:

USA: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.

EUROPE: Motorola Ltd.; European Literature Center; 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.

JAPAN: Nippon Motorola Ltd.; 4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141 Japan.

ASIA-PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbor Center, No. 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong.

MOTOROLA