DSP56800

16-bit Digital Signal Processor Family Manual

Motorola, Incorporated
Semiconductor Products Sector
DSP Division
6501 William Cannon Drive West
Austin, TX 78735-8598

TABLE OF CONTENTS

LIST OF FIGURES

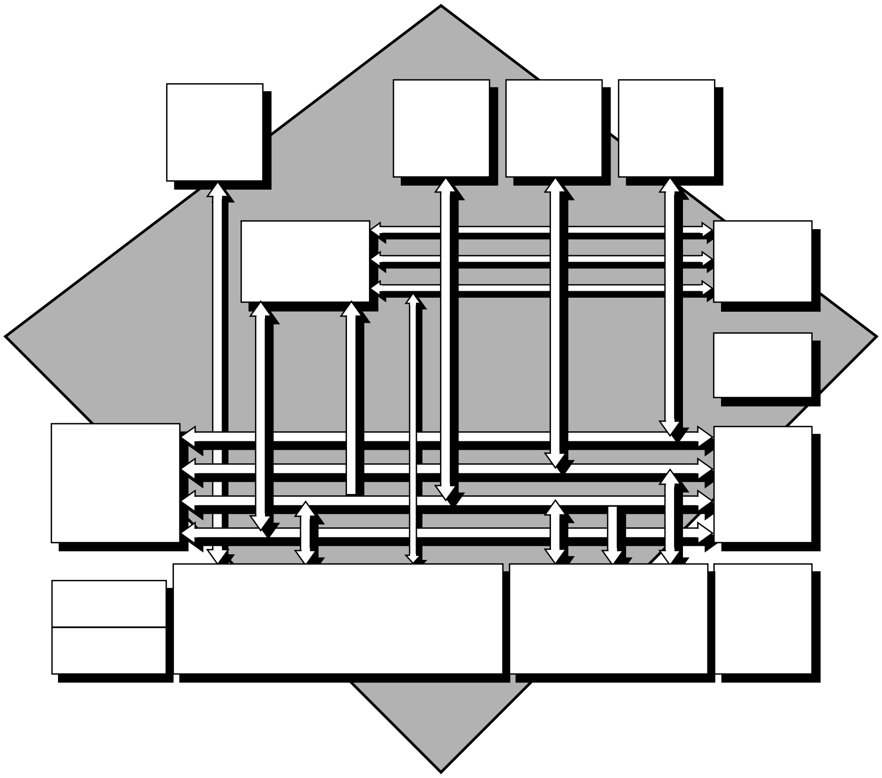
LIST OF FIGURES

LIST OF TABLES

LIST OF EXAMPLES

SECTION 1

INTRODUCTION



1.1 DSP56800 FAMILY ARCHITECTURE

The DSP56800 Family uses the DSP56800 16-bit DSP core. This core is a general-purpose central processing unit (CPU), designed for both efficient DSP and controller operations. Its instruction set efficiency as a DSP is superior to other low-cost DSP architectures and has been designed for efficient, straightforward coding of controller-type tasks.

The general-purpose MCU-style instruction set, with its powerful addressing modes and bit-manipulation instructions, enables a user to begin writing code immediately, without having to worry about the complexities previously associated with DSPs. A software stack allows for unlimited interrupt and subroutine nesting, as well as support for structured programming techniques such as parameter passing and use of local variables. The veteran DSP programmer sees a powerful DSP instruction set with many different arithmetic operations and flexible single- and dual-memory moves that can occur in parallel with an arithmetic operation. The general-purpose nature of the instruction set also allows for an efficient compiler implementation.

A variety of standard peripherals can be added around the DSP56800 core (see **Figure 1-1**) such as serial ports, general-purpose timers, real-time and watchdog timers, different memory configurations (RAM and/or ROM), and general-purpose I/O (GPIO) ports. Each peripheral interfaces to the DSP56800 core through a standard peripheral interface bus. This bus allows easy hookup of standard or custom designed peripherals.

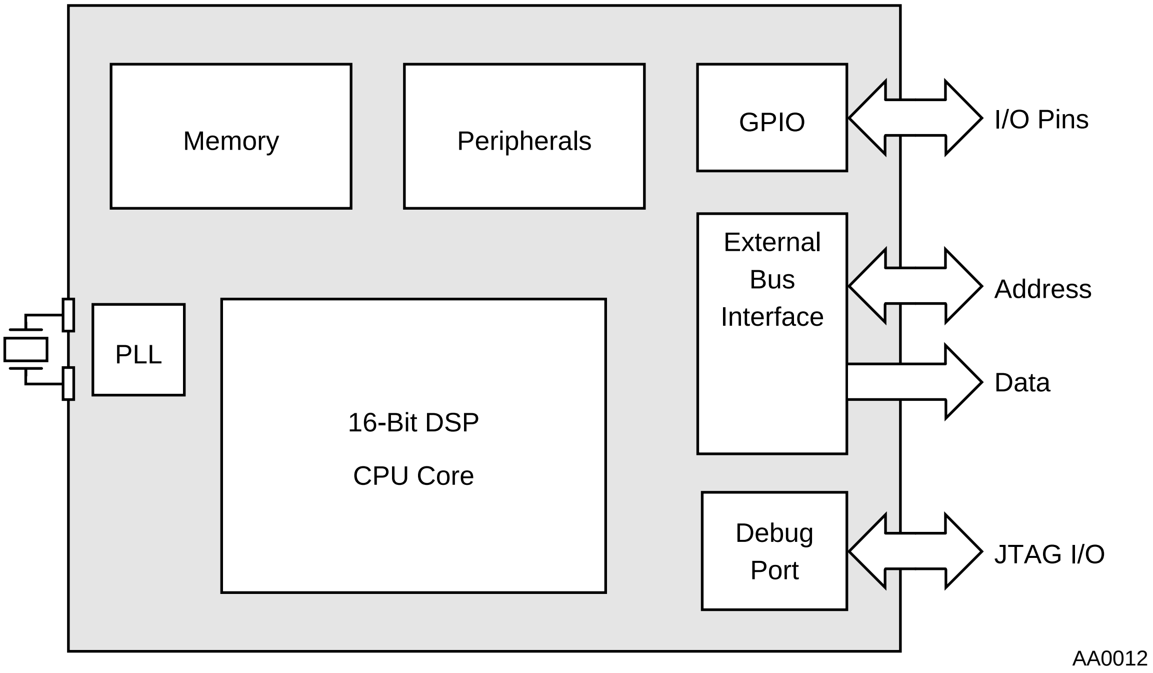


Figure 1-1 DSP56800-Based DSP Microcontroller Chip

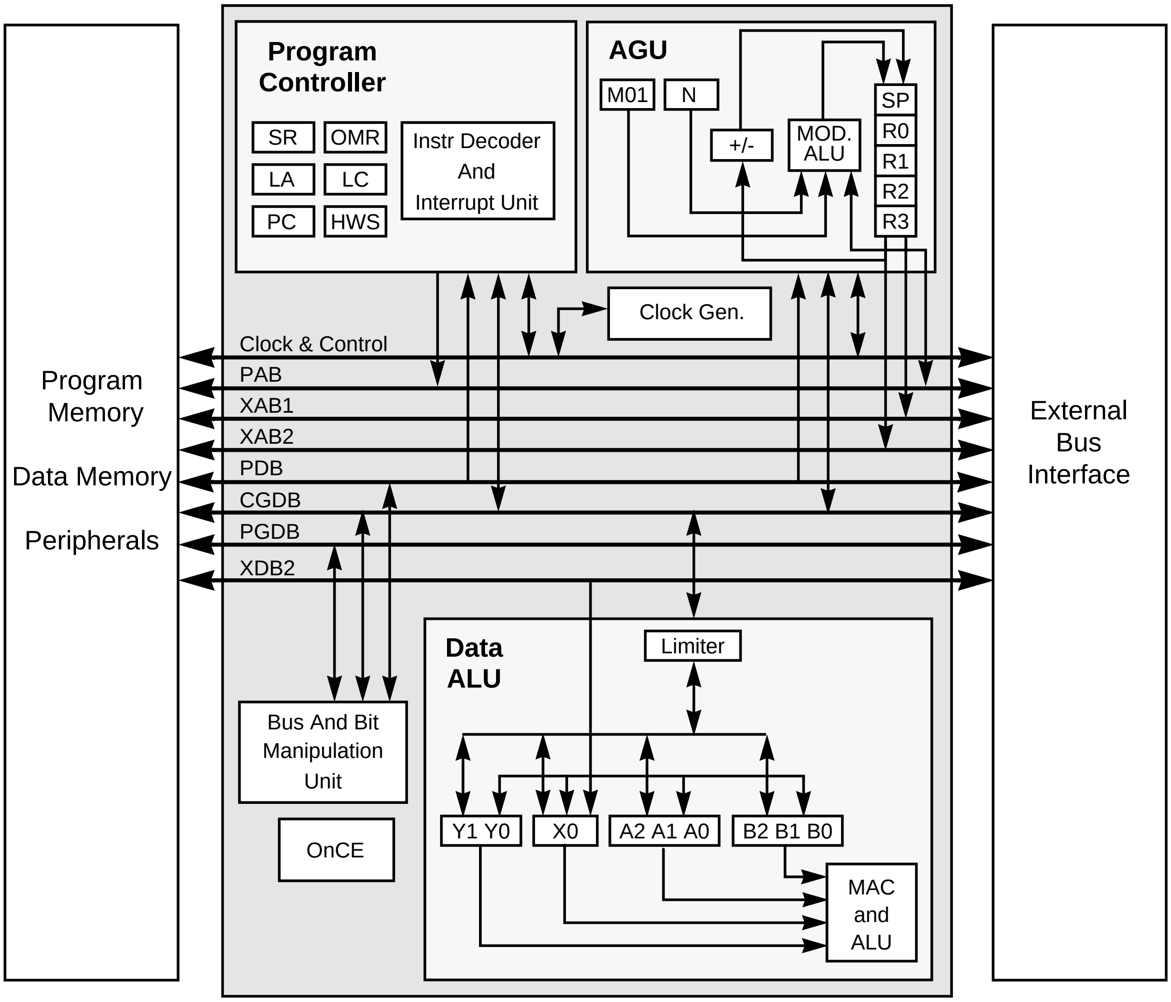
On-Chip Emulation (OnCE™) capability is provided through a debug port conforming to the JTAG standard. This provides real-time embedded system debug with on-chip emulation capability through the 5-pin JTAG interface. A user can set hardware and software breakpoints, display and change registers and memory locations, and single-step or step through multiple instructions in an application.

The DSP56800's efficient instruction set, multiple internal buses, on-chip program and data memories, external bus interface, standard peripherals, and IEEE-compliant Boundary Scan Architecture JTAG debug port make the DSP56800 Family an excellent solution for real-time embedded control tasks. It becomes an excellent fit for wireless or wireline DSP applications, digital control, and controller applications in need of more processing power.

1.1.1 Core Overview

The DSP56800 core is a programmable 16-bit CMOS digital signal processor that consists of a 16-bit data arithmetic logic unit (ALU), a 16-bit address generation unit (AGU), a program decoder, On-Chip Emulation (OnCE), associated buses, and an instruction set. **Figure 1-2** shows a block diagram of the DSP56800 core. The main features of the DSP56800 core include:

- Up to 25 million instructions per second (MIPS) at 40 MHz
  - 40 ns instruction cycle (50 MHz) at 4.5V to 5.5V
  - 50 ns instruction cycle (40 MHz) at 2.7V to 3.6V
- Single-instruction cycle 16 x 16-bit parallel multiply-accumulator
- Two 36-bit accumulators including extension bits
- Single-instruction 16-bit barrel shifter
- Parallel instruction set with unique DSP addressing modes
- Hardware DO and REP loops
- Two external interrupt request pins
- Three 16-bit internal core data buses
- Three 16-bit internal address buses
- One 16-bit data bus dedicated to peripherals
- Instruction set supports both DSP and controller functions
- Controller-style addressing modes and instructions for smaller code size
- Efficient C compiler and local variable support
- On-chip peripheral registers mapped in data memory space
- Software subroutine and interrupt stack with unlimited depth
- On-Chip Emulation for unobtrusive, processor speed-independent debugging
- Low power wait and stop modes
- Operating frequency down to DC
- Single power supply
- Low power (HCMOS)



AA0006

Figure 1-2 DSP56800 Core Block Diagram

1.1.2 Peripheral Blocks

The following peripheral blocks are available for members of the DSP56800 16-bit DSP family:

- Program ROM and RAM modules
- Bootstrap ROM for program RAM parts
- Data ROM and RAM modules

- Phase-locked loop (PLL) Module
    - Accepts 32.0 and 38.4 KHz crystals
    - Accepts crystal frequencies ≥ 1 MHz
    - Programmable multiplication factor
    - Three pins required (SXFC, V_{DDS}, and GNDS)
- 16-bit Timer Module
    - Contains three independent 16-bit timers
    - Each may be clocked from a pin, the oscillator clock, or the PLL output
    - Zero to two pins required
- Computer operating properly (COP) and real-time timer module
    - COP timer uses output of real-time timer chain
    - Programmable real-time timer
    - Count register readable
    - No pins required
- Synchronous serial interface module (SSI)
    - Synchronous serial interface for hooking up to codecs
    - Frame sync and gated clock modes
    - Independent transmit and receive channels
    - Up to 32-slot network mode available
    - Three to six pins required
- Serial peripheral interface (SPI)
    - Simple synchronous 8-bit serial interface for interfacing to MCUs and MCU-style peripherals
    - Master and slave modes
    - Four pins required
- Programmable general-purpose I/O
    - Pins can be individually programmed as input or output
    - Pins can be individually multiplexed between peripheral functionality and GPIO
    - Pins can have interrupt capability

More blocks will be defined in the future to meet customer needs.

1.1.3 Family Members

The DSP56800 core processor is designed as a core processor for a new family of Motorola DSPs. An example of a chip that can be built with this core is shown in **Figure 1-3**.

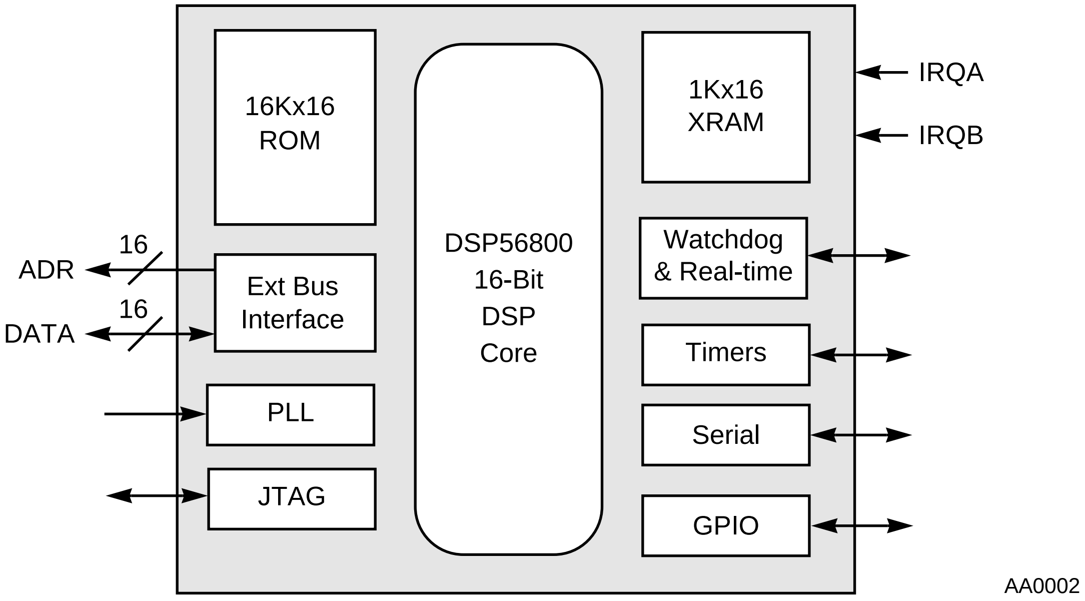


Figure 1-3 Example of Chip Built around the DSP56800 Core

1.2 INTRODUCTION TO DIGITAL SIGNAL PROCESSING

DSP is the arithmetic processing of real-time signals sampled at regular intervals and digitized. Examples of DSP processing include the following:

- Filtering
- Convolution (mixing two signals)
- Correlation (comparing two signals)
- Rectification, amplification, and / or transformation

Figure 1-4 shows an example of analog signal processing. The circuit in the illustration filters a signal from a sensor using an operational amplifier and controls an actuator with the result. Since the ideal filter is impossible to design, the engineer must design the filter for acceptable response considering variations in temperature, component aging, power supply variation, and component accuracy. The resulting circuit typically has low noise immunity, requires adjustments, and is difficult to modify.

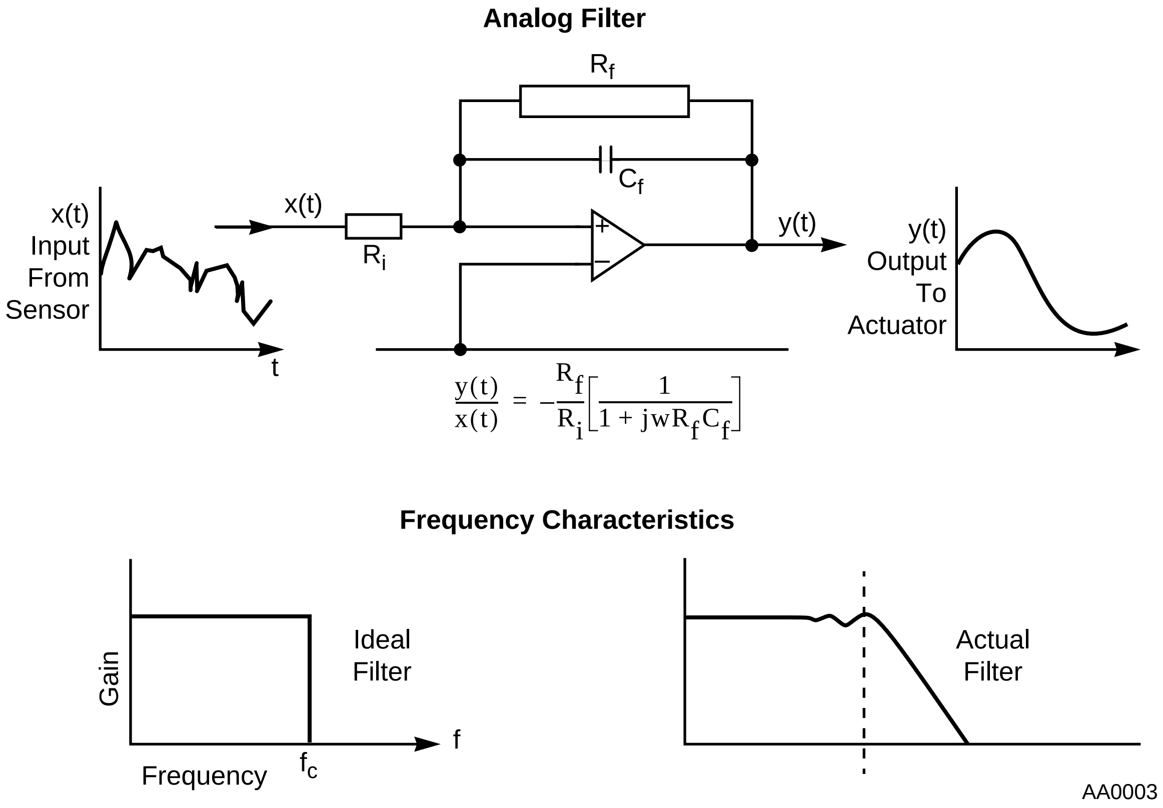


Figure 1-4 Analog Signal Processing

The equivalent circuit using a DSP is shown in **Figure 1-5**. This application requires an analog-to-digital (A / D) converter and digital-to-analog (D / A) converter in addition to the DSP. Even with these additional parts, the component count can be lower using a DSP due to the high integration available with current components.

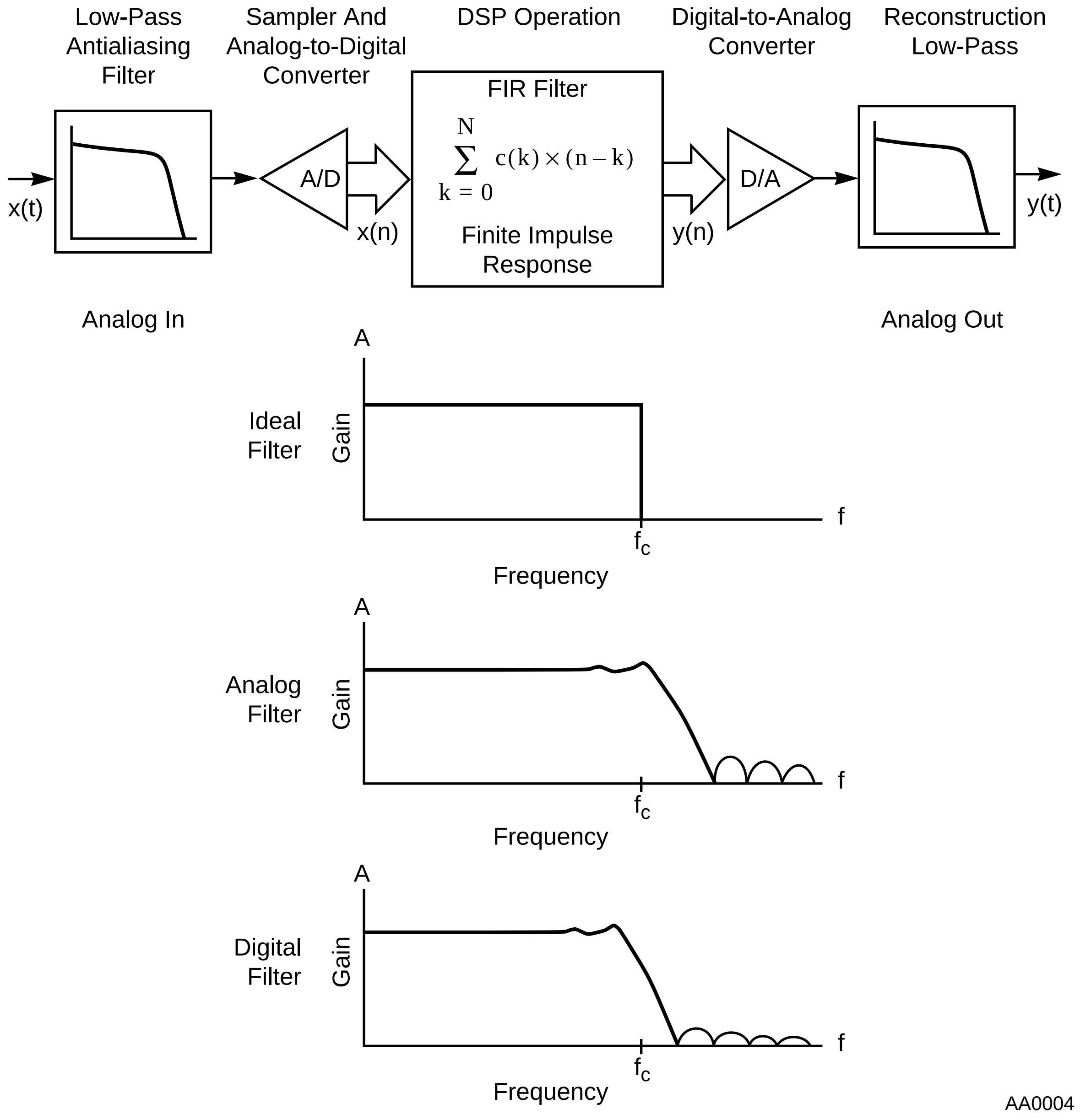


Figure 1-5 Digital Signal Processing

Processing in this circuit begins by band-limiting the input signal with an anti-alias filter, eliminating out-of-band signals that can be aliased back into the pass band due to the sampling process. The signal is then sampled, digitized with an A/D converter, and sent to the DSP.

The filter implemented by the DSP is strictly a matter of software. The DSP can directly employ any filter that can also be implemented using analog techniques. Also, adaptive filters can be easily put into practice using DSP, whereas these filters are extremely difficult to implement using analog techniques. (Similarly, compression can also be implemented on a DSP.)

The DSP output is processed by a D/A converter and is low-pass filtered to remove the effects of digitizing. In summary, the advantages of using the DSP include the following:

- Fewer components
- Stable, deterministic performance
- No filter adjustments
- Wide range of applications
- Filters with much closer tolerances
- High noise immunity
- Adaptive filters easily implemented
- Self-test can be built in
- Better power-supply rejection

The DSP56800 family is not a custom IC designed for a particular application; it is designed as a general-purpose DSP architecture to efficiently execute commonly-used DSP benchmarks and controller code in minimal time.

As shown in **Figure 1-6**, the keys attributes of a DSP are as follows:

- Multiply/accumulate (MAC) operation
- Fetching up to two operands per instruction cycle for the MAC
- Program control to provide versatile operation
- Input/output to move data in and out of the DSP

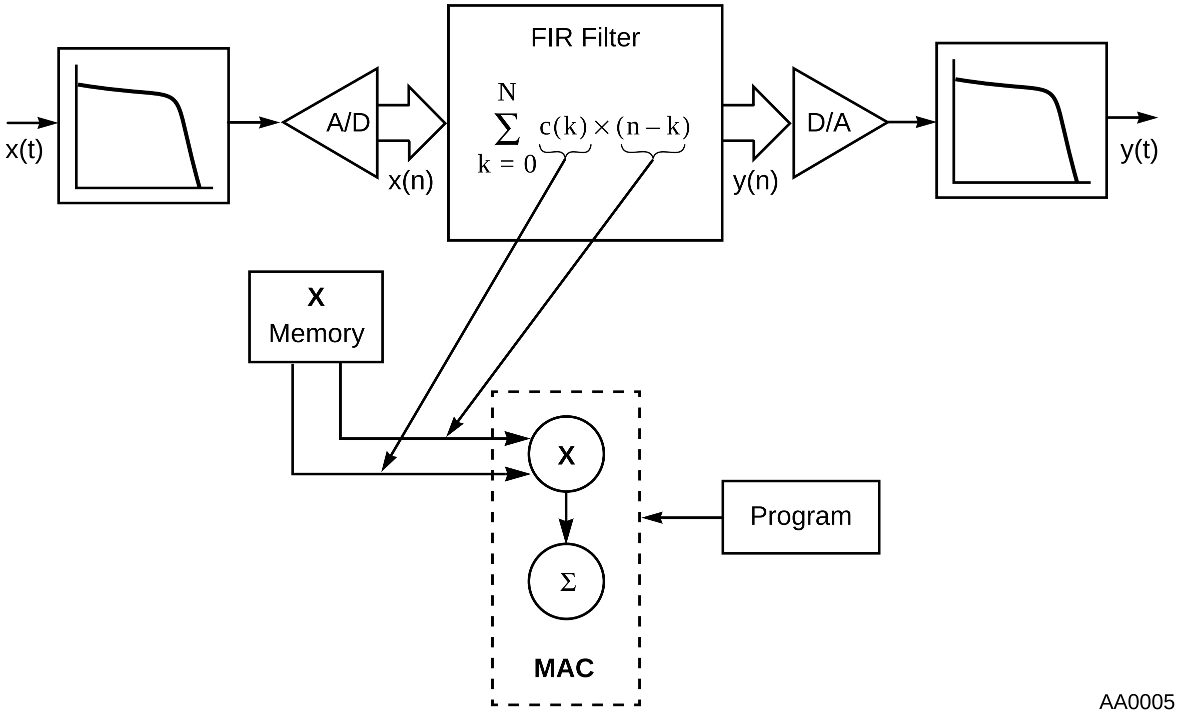


Figure 1-6 Mapping DSP Algorithms into Hardware

The multiply-accumulation (MAC) operation is the fundamental operation used in DSP. The DSP56800 family of processors has a dual Harvard architecture optimized for MAC operations. **Figure 1-6** shows how the DSP56800 architecture matches the shape of the MAC operation. The two operands, C() and X(), are directed to a multiply operation, and the result is summed. This process is built into the chip by allowing two separate data memory accesses to feed a single-cycle MAC. The entire process must occur under program control to direct the correct operands to the multiplier and save the accumulated result as needed. Since the memory and the MAC are independent, the DSP can perform two memory moves, a multiply and an accumulate, and two address updates in a single operation. As a result, many DSP benchmarks execute very efficiently for a single-multiplier architecture.

1.3 SUMMARY OF FEATURES

The high throughput of the DSP56800 family of processors makes them well-suited for wireless and wireline communication, high-speed control, low-cost voice processing, numeric processing, and computer and audio applications. The main features that contribute to this high throughput include the following:

- **Speed—**The DSP56800 supports most mid-performance DSP applications.
- **Precision—**The data paths are 16 bits wide, providing 96 dB of dynamic range; intermediate results held in the 36-bit accumulators can range over 216 dB.
- **Parallelism**—Each on-chip execution unit, memory, and peripheral operates independently and in parallel with the other units through a sophisticated bus system. The data ALU, AGU, and program controller operate in parallel so that the following can be executed in a single instruction:
  - An instruction pre-fetch
  - A 16-bit x 16-bit multiplication
  - A 36-bit addition
  - Two data moves
  - Two address-pointer updates using one of two types of arithmetic (linear or modulo)
  - Sending and receiving full-duplex data by the serial ports
  - Timers continuing to count in parallel
- **Flexibility**—While many other DSPs need external communications circuitry to interface with peripheral circuits (such as A/D converters, D/A converters, or host processors), the DSP56800 family provides on-chip serial and parallel interfaces that can support various configurations of memory and peripheral modules. The peripherals are interfaced to the DSP56800 core through a peripheral interface bus, designed to provide a common interface to many different peripherals.
- **Sophisticated Debugging**— Motorola’s on-chip emulation technology (OnCE) allows simple, inexpensive, and speed independent access to the internal registers for debugging. OnCE tells application programmers exactly what the status is within the registers, memory locations, and even the last instructions that were executed.
- **Phase-locked Loop (PLL)-Based Clocking**—The PLL allows the chip to use almost any available external system clock for full-speed operation while also supplying an output clock synchronized to a synthesized internal core clock. It improves the synchronous timing of the processors’ external memory port, eliminating the timing skew common on other processors.
- **Invisible Pipeline—**The three-stage instruction pipeline is essentially invisible to the programmer, allowing straightforward program development in either assembly language or high-level languages such as C or C++.

- **Instruction Set—**The instruction mnemonics are MCU-like, making the transition from programming microprocessors to programming the chip as easy as possible. New microcontroller instructions, addressing modes, and bit field instructions allow for significant decreases in program code size. The orthogonal syntax controls the parallel execution units. The hardware DO loop instruction and the repeat (REP) instruction make writing straight-line code obsolete.
- **Low Power—**Designed in CMOS, the DSP56800 family inherently consumes very low power. Two additional low power modes, stop and wait, further reduce power requirements. Wait is a low power mode where the DSP56800 core is shut down but the peripherals and interrupt controller continue to operate so that an interrupt can bring the chip out of wait mode. In stop mode, even more of the circuitry is shut down for the lowest power consumption mode. There are also several different ways to bring the chip out of stop mode.

1.4 MANUAL ORGANIZATION

This manual describes the central processing unit of the DSP56800 Family in detail. It is intended to be used with the appropriate DSP56800 Family member user’s manual, which describes the central processing unit, programming models, and details of the instruction set. The appropriate DSP56800 Family member technical data sheet provides timing, pinout, and packaging descriptions.

This manual provides practical information to help the user accomplish the following:

- Understand the operation and instruction set of the DSP56800 Family
- Write code for DSP algorithms
- Write code for general control tasks
- Write code for communication routines
- Write code for data manipulation algorithms

A list of notations can be found in **Notation** on page A-3.

Table 1-1 describes the contents of each section and each appendix:

Table 1-1 DSP56800 Family Manual Section Descriptions

Section / Appendix	Title and Description
2	**Core Architecture Overview—**The DSP56800 core architecture consists of the data arithmetic logic unit (ALU), address generation unit (AGU), program controller, bus and bit-manipulation unit, and a JTAG / On-Chip Emulation (OnCE) port. This section describes each subsystem and the buses interconnecting the major components in the DSP56800 central processing module.
3	**Data Arithmetic Logic Unit**—This section describes the data ALU architecture, its programming model, an introduction to fractional and integer arithmetic, and a discussion of other topics such as unsigned and multi-precision arithmetic on the DSP56800 Family.
4	**Address Generation Unit**—This section specifically describes the AGU architecture, its programming model, addressing modes, and address modifiers.
5	**Program Controller**—This section describes in detail the program controller architecture, its programming model, and hardware looping. Note, however, that the different processing states of the DSP56800 core including interrupt processing are described in **Section 7 Interrupts and the Processing States**.
6	**Instruction Set Introduction**—This section presents an introduction to parallel moves and a brief description of the syntax, instruction formats, operand / memory references, data organization, addressing modes, and instruction set. It also includes a summary of the instruction set, showing the registers and addressing modes available to each instruction. A detailed description of each instruction is given in **Appendix A Instruction Set Details**.
7	**Interrupts and the Processing States**—This section describes five of the six processing states (normal, exception, reset, wait, and stop). The sixth processing state (debug) is covered more completely in **Section 9 JTAG / On-Chip Emulation (OnCE)**.
8	**Software Techniques**—This section teaches the advanced user techniques for more efficient programming of the DSP56800 Family. It includes a description of useful instruction sequences and macros, optimal loop and interrupt programming, topics related to the stack of the DSP56800, and other useful software topics.
9	**JTAG /On-Chip Emulation (OnCE)**—This section describes the combined JTAG / OnCE port and its functions. These two are integrally related, sharing the same pins for I / O, and are presented together in this section.
10	**Development Tools**—This section describes the development tools that are available for the DSP56800 Family.

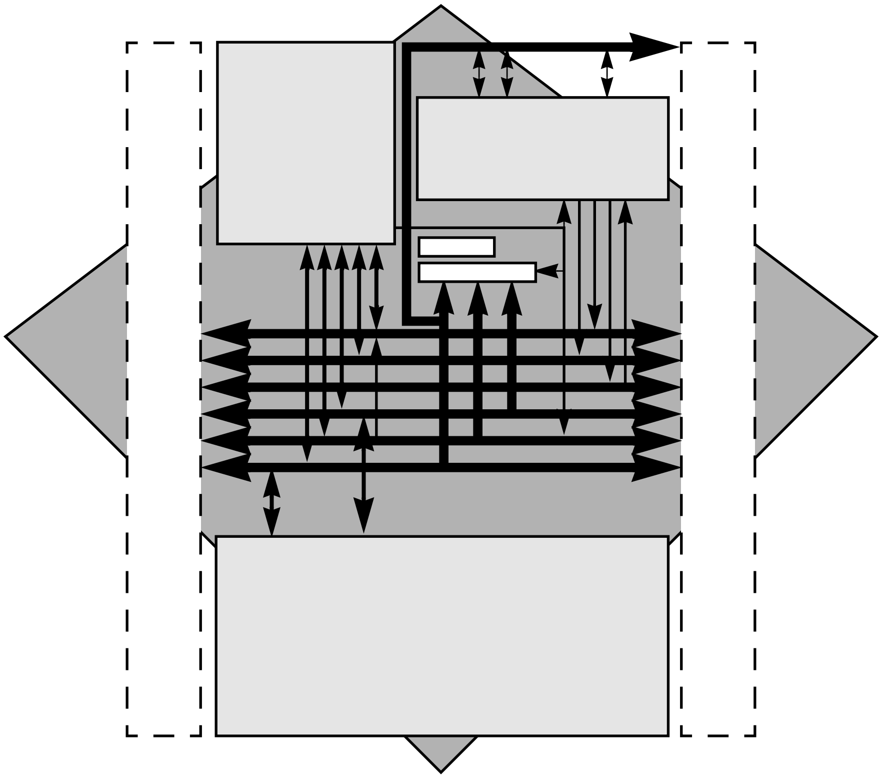
Table 1-1 DSP56800 Family Manual Section Descriptions (Continued)

Section / Appendix	Title and Description
11	**Additional Support**—This section presents the various ways to receive support for the DSP56800 Family.
A	**Instruction Set Details**—This section presents a detailed description of each DSP56800 Family instruction, its use, and its effect on the processor are presented.
B	**DSP Benchmarks**—DSP56800 Family benchmark example programs and results are listed in this appendix.

Note: The latest electronic version of this document as well as other DSP documentation (including user's manuals, product briefs, data sheets, and errata) can be found on the Motorola DSP World Wide Web site. See **Section 11 Additional Support** for more information.

SECTION 2

CORE ARCHITECTURE OVERVIEW



2.1 INTRODUCTION

The DSP56800 core architecture is a 16-bit multiple-bus processor designed for efficient real-time digital signal processing and general purpose computing. The architecture is designed as a standard programmable core from which various DSP integrated circuit family members can be designed with different on-chip and off-chip memory sizes and on-chip peripheral requirements. This section presents the overall core architecture and the general programming model. More detailed information on the data ALU, AGU, program controller, and JTAG / OnCE blocks within the architecture are found in later sections.

2.2 CORE BLOCK DIAGRAM

The DSP56800 core is composed of functional units that operate in parallel to increase the throughput of the machine. The program controller, AGU, and data ALU each contain their own register set and control logic, so each may operate independently and in parallel with the other two. Likewise, each functional unit interfaces with other units, with memory, and with memory-mapped peripherals over the core's internal address and data buses. The architecture is pipelined to take advantage of the parallel units and significantly decrease the execution time of each instruction.

For example, it is possible for the data ALU to perform a multiply in a first instruction, the AGU to generate up to two addresses for a second instruction, and the program controller to be fetching a third instruction. In a similar manner, it is possible for the bit-manipulation unit to perform an operation of the third instruction described above in place of the multiplication in the data ALU.

The major components of the core are the following:

- Data ALU
- AGU
- Program controller and hardware looping unit
- Bus and bit-manipulation unit
- OnCE debug port
- Address buses
- Data buses

Core Block Diagram

Figure 2-1 shows an overall block diagram of a DSP56800-based DSP processor including the DSP56800 core, on-chip peripherals, on-chip memory, and expansion buses. **Figure 2-2** shows a block diagram of the CPU architecture.

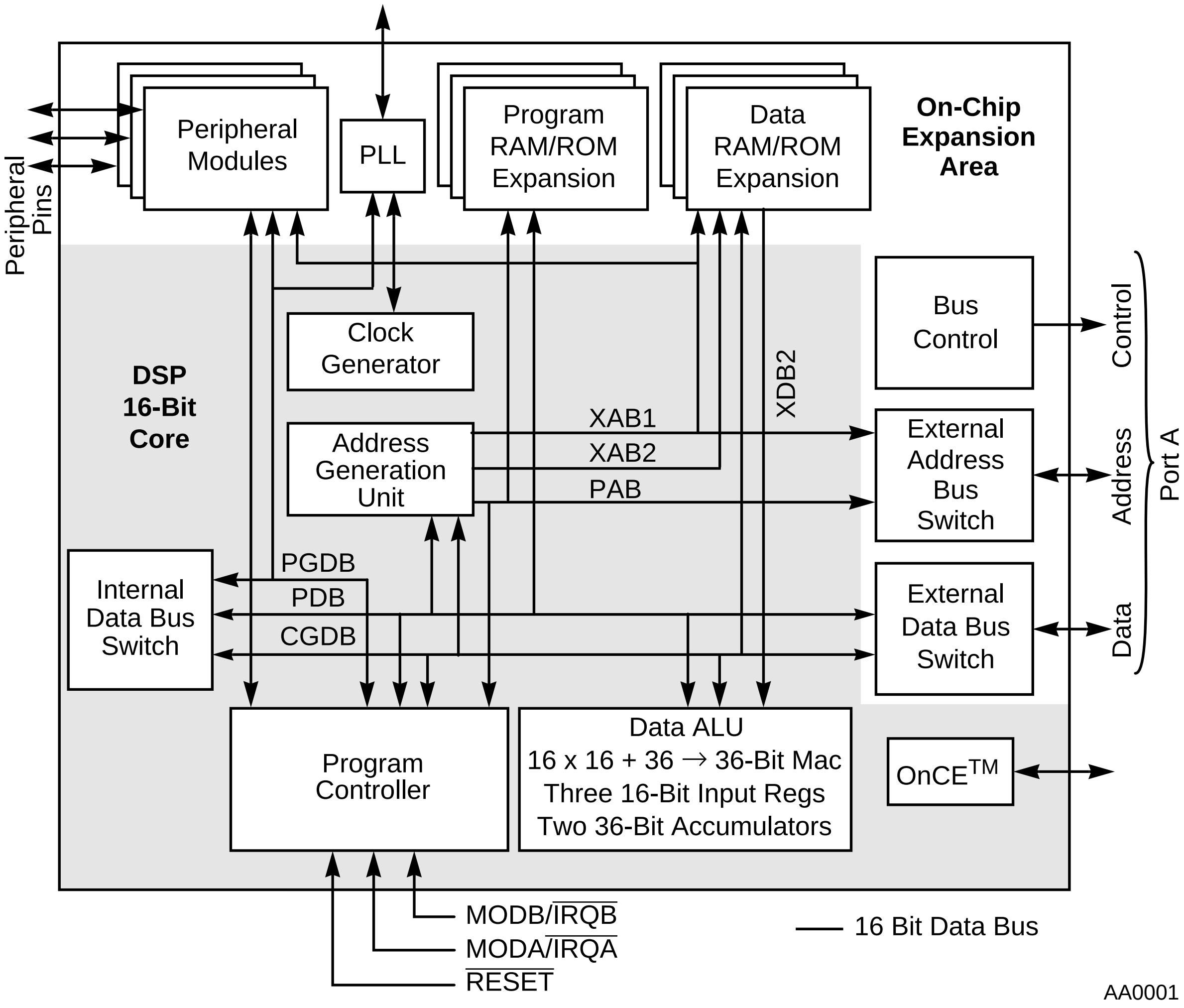


Figure 2-1 DSP56800 Family Chip Block Diagram

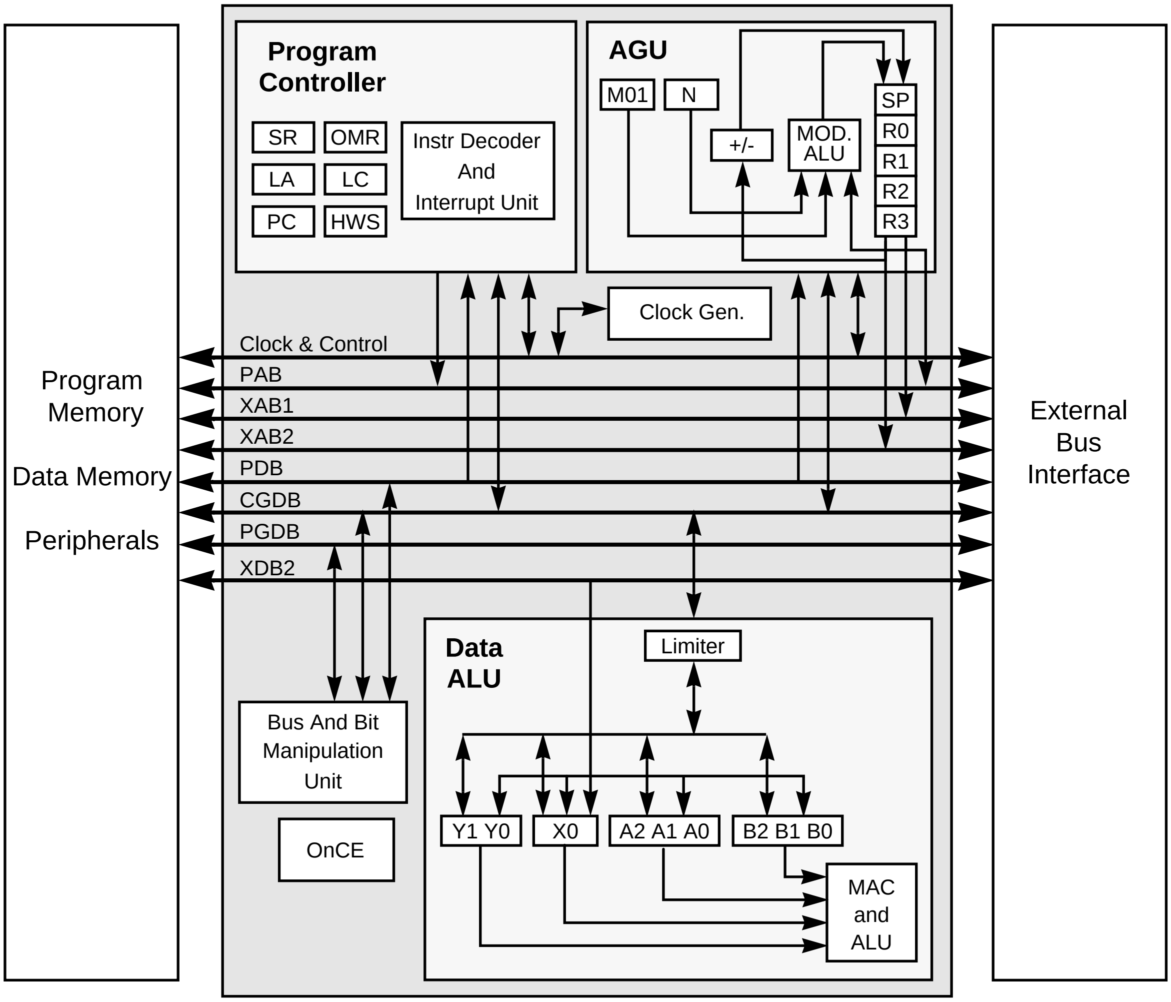


Figure 2-2 DSP56800 Core Block Diagram

2.2.1 Data Arithmetic Logic Unit (ALU)

The data arithmetic logic unit (ALU) performs all of the arithmetic and logical operations on data operands. It consists of the following:

- Three 16-bit input registers (X0, Y0, and Y1)
- Two 32-bit accumulator registers (A and B)
- Two 4-bit accumulator extension registers (A2 and B2)
- An accumulator shifter (AS)

- One data limiter
- One 16-bit barrel shifter
- One parallel (single cycle, non-pipelined) multiply-accumulator (MAC) unit

The data ALU is capable of multiplication, multiply-accumulate (with positive or negative accumulation), addition, subtraction, shifting, and logical operations in one instruction cycle. Arithmetic operations are done using 2's complement fractional or integer arithmetic. Support is also provided for unsigned and multi-precision arithmetic.

Data ALU source operands may be 16, 32 or 36 bits and may individually originate from input registers, memory locations, immediate data, or accumulators. ALU results are stored in one of the accumulators. In addition, some arithmetic instructions store their 16-bit results either in one of the three data ALU input registers or directly in memory. Arithmetic operations and shifts can have a 16-bit or a 36-bit result. Logical operations are performed on 16-bit operands and always yield 16-bit results.

Data ALU register values can be transferred (read or write) across the core global data bus (CGDB) as 16-bit operands. The X0 register value can also be written by the external data bus (XDB2) as a 16-bit operand. Refer to **Section 3 Data Arithmetic Logic Unit** for a detailed description of the data ALU.

2.2.2 Address Generation Unit (AGU)

The address generation unit (AGU) performs all of the effective address calculations and address storage necessary to address data operands in memory. The AGU operates in parallel with other chip resources to minimize address generation overhead. It contains two ALUs, allowing the generation of up to two 16-bit addresses every instruction cycle: one for either the external address bus one (XAB1) or program address bus (PAB) and one for the external address bus two (XAB2). The ALU can directly address 65,536 locations on the XAB1 or XAB2 and 65,536 locations on the PAB, 131,072 16-bit data words total. It supports a complete set of addressing modes. Its arithmetic unit can both linear and modulo arithmetic.

The AGU contains the following registers:

- Four address registers (R0-R3)
- A stack pointer register (SP)
- An offset register (N)

- A modifier register (M01)
- A modulo arithmetic unit
- An incrementer/decrementer unit

The address registers are 16-bit registers that may contain an address or data. Each address register can provide an address for the XAB1 and PAB address buses. For instructions that read two values from X data memory, R3 provides an address for the XAB2, and R0 or R1 provides an address for the XAB1. The modifier and offset registers are 16-bit registers that control updating of the address registers. The offset register can also be used to store 16-bit data. AGU registers may be read or written by the CGDB as 16-bit operands. Refer to **Section 4 Address Generation Unit** for a detailed description of the AGU.

2.2.3 Program Controller and Hardware Looping Unit

The program controller performs the following:

- Instruction prefetch
- Instruction decoding
- Hardware loop control
- Interrupt (exception) processing

Instruction execution is carried out in other core units such as the data ALU, AGU, or bit-manipulation unit. The program controller consists of the following:

- A program counter unit
- Instruction latch and decoder
- Hardware looping control logic
- Interrupt control logic
- Status and control registers

Located within the program controller are the following:

- Four user-accessible registers:
  – Loop address register (LA)
  – Loop count register (LC)
  – Status register (SR)

  - Operating mode register (OMR)
- A program counter (PC)
- A hardware stack (HWS)

The 16-bit program counter register is combined with three extra bits in the status register to form a 19-bit program counter capable of accessing 524,288 16-bit locations in program memory space.

There are two mode and interrupt control pins that provide external interrupts to the interrupt controller. The mode select A / external interrupt request A (MODA / $\overline{\text{IRQA}}$) and mode select B / external interrupt request B (MODB / $\overline{\text{IRQB}}$) pins select the chip operating mode out of reset and receive interrupt requests from external sources during normal operation.

The reset pin resets the chip. When it is asserted, it initializes the chip and places it in the reset state. When it is deasserted, the chip assumes the operating mode indicated by the MODA and MODB pins.

The HWS is a separate internal last-in-first-out (LIFO) buffer of two 16-bit words that stores the address of the first instruction in a hardware DO loop. When a new hardware loop is begun by executing the DO instruction, the address of the first instruction in the loop is stored (pushed) on the “top” location of the HWS, and the LF bit in the SR is set. The previous value of the loop flag (LF) bit is copied to the OMR’s NL bit. When an ENDDO instruction is encountered or a hardware loop terminates naturally, the 16-bit address in the “top” location of the HWS is discarded, and the LF bit is updated with the value in the OMR’s nested looping (NL) bit.

The program controller is described in detail in **Section 5 Program Controller**. For more details on program looping, refer to **Program Looping** on page 5-19 and **Loops** on page 8-25. For information on reset and interrupts, refer to **Section 7 Interrupts and the Processing States**.

2.2.4 Bus and Bit-manipulation Unit

Transfers between internal buses are accomplished in the bus unit. The bus unit is similar to a switch matrix and can connect any two of the three data buses together without adding any pipeline delays. This is required for transferring a core register to a peripheral register, for example, because the core register is connected to the CGDB and the peripheral register is connected to the PGDB.

The bit-manipulation unit performs bit-field manipulations on X memory words, peripheral registers, and all registers within the DSP56800 core. It is capable of testing, setting, clearing, or inverting any bits specified in a 16-bit mask. For branch-on-bit-field instructions, this unit tests bits on the upper or lower byte of a 16-bit word (i.e., the mask can only test up to eight bits at a time).

2.2.5 On-Chip Emulation (OnCE) Unit

The On-Chip Emulation (OnCE) unit allows the user to interact in a debug environment with the DSP56800 core and its peripherals non-intrusively. Its capabilities include examining registers, on-chip peripheral registers, or memory, setting breakpoints on program or data memory, and stepping or tracing instructions. It provides simple, inexpensive, and speed-independent access to the internal DSP56800 core by interacting with a user interface program running on a host workstation for sophisticated debugging and economical system development.

Dedicated pins through the JTAG port allow the user access to the DSP in a target system, retaining debug control without sacrificing other user accessible on-chip resources. This technique eliminates the costly cabling and the access to processor pins required by traditional emulator systems. Refer to **Section 9 JTAG /On-Chip Emulation (OnCE)** for a detailed description of the JTAG / OnCE port, and to **Section 10 Hardware Development Environment** for a description of the debugging environment.

2.2.6 Address Buses

Addresses are provided to the internal X data memory on two unidirectional 16-bit buses, external address bus one (XAB1) and external address bus two (XAB2). Program memory addresses are provided on the 19-bit program address bus (PAB). Note that XAB1 can provide addresses for accessing both internal and external memory, whereas XAB2 can only provide addresses for accessing internal memory.

2.2.7 Data Buses

Inside the chip, data is transferred using the following:

- Three bidirectional 16-bit buses

  - Core global data bus (CGDB)
  - Program data bus (PDB)
  - Peripheral global data bus (PGDB)
- One unidirectional 16-bit bus: external data bus two (XDB2)

Data transfer between the data ALU and the X data memory uses the CGDB when one memory access is performed. When two simultaneous memory reads are performed, the transfers use the CGDB and the XDB2. All other data transfers to core blocks occur using the CGDB. All transfers to and from peripherals occur over the PGDB. Instruction word fetches occur simultaneously over the PDB. The bus structure supports general register-to-register moves, register-to-memory moves, and memory-to-register moves, and can transfer up to three 16-bit words in the same instruction cycle. Transfers between buses are accomplished in the bus and bit-manipulation unit. As a general rule, when reading any register less than 16 bits wide, the unused bits are read as zeros. Reserved and unused bits should always be written with zeros to insure future compatibility.

2.3 BLOCKS OUTSIDE THE DSP56800 CORE

The following blocks are optionally found on DSP56800-based DSP chips, and are considered peripheral and memory blocks, not part of the DSP56800 core. These and other blocks are described in greater complexity in the appropriate chip-specific user’s manual.

2.3.1 External Data Memory

External data memory (data RAM and / or data ROM) can be added around the core on a chip. Addresses are received from the XAB1 and XAB2. Data transfers occur on the CGDB and XDB2. One read, one write, or two reads can be performed during one instruction cycle using the internal data memory. The top 128 locations in the X data memory space (X:$FF80-X:$FFFF) are reserved for on-chip peripherals. The next 128 locations in the X data memory space (X:$FF00-X:$FF7F) are reserved for off-chip peripherals. X memory may be expanded off chip. A total of 65,536 memory locations can be addressed.

2.3.2 Program Memory

Program memory (program RAM and/or ROM) can be added around the core on a chip. Addresses are received from the PAB and data transfers occur on the PDB. The first 128 locations of the program memory are available for interrupt vectors, although it is not necessary to use all 128 locations for interrupt vectors. Some can be used for the user program if desired. The number of locations required for an application depends on what peripherals on the chip are used by an application and the locations of their corresponding interrupt vectors. The program memory may be expanded off-chip, and up to 65,536 locations can be addressed.

2.3.3 Bootstrap Memory

A program bootstrap ROM is usually found on a chip that has on-chip program RAM instead of on-chip program ROM. The bootstrap ROM is used for initially loading the on-chip program RAM so that an application can be run from the program RAM. During bootstrapping (operating mode 0 or 1), the bootstrap ROM is read by the program controller when fetching instructions and the on-chip program RAM is configured for write-only. All writes during the bootstrap operation are done to the on-chip program RAM. Refer to **Operating Mode Bits (MB and MA)—Bits 0-1** on page 5-15 and to the user's manual of the particular DSP chip for a description of the different bootstrapping modes.

2.3.4 External Bus Interface

External 16-bit address and data buses allow the DSP chip to access external off-chip data memory, program memory or I/O devices when required. Separate select lines indicate an access to external program or data memory spaces.

The external bus interface asynchronously interfaces with a wide variety of memory and peripheral devices. These devices include high-speed static RAMs, slower memory devices, and other DSPs and MPUs in master/slave configurations. This variety is possible because the expansion bus timing is programmable and can be tailored to match the speed requirements of the different memory spaces.

Note: The second read of a dual read instruction always accesses internal memory. In the case where the second access of a dual read needs to access external memory, it is necessary to break the instruction into two instructions, each of which performs a single memory access.

2.3.5 Phase-Locked Loop (PLL)

The phase-locked loop (PLL) allows the DSP chip to use an external clock different from the internal system clock, while optionally supplying an output clock synchronized to a synthesized internal clock. This PLL allows full-speed operation using an external clock running at a different speed. The PLL performs frequency multiplication, skew elimination, and reduces overall system power by reducing the frequency on the input reference clock.

2.4 DSP56800 CORE PROGRAMMING MODEL

The registers in the DSP56800 core that are considered part of the DSP56800 core programming model are shown in **Figure 2-3**. There are also other important registers typically found on a DSP chip that are not considered part of the DSP56800 core, but rather are memory-mapped peripheral registers. These include the interrupt priority register (IPR) described in **Interrupt Priority Register (IPR)** on page 7-13, as well as any peripheral registers, such as the bus control register (BCR), that are typically described in the user’s manual of a specific chip within the DSP56800 core family. (See **Figure 2-4** for an I/O and on-chip peripheral memory map.)

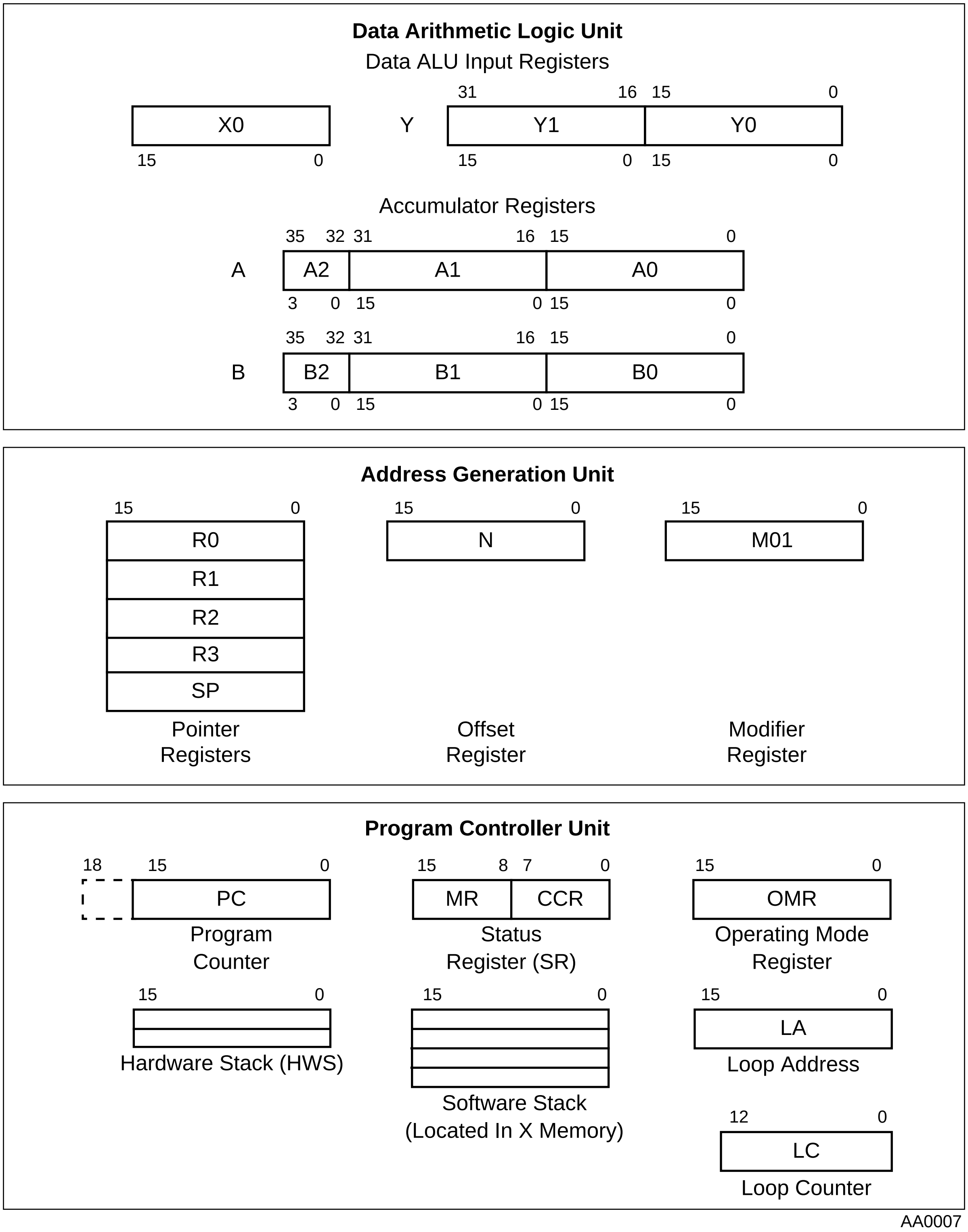


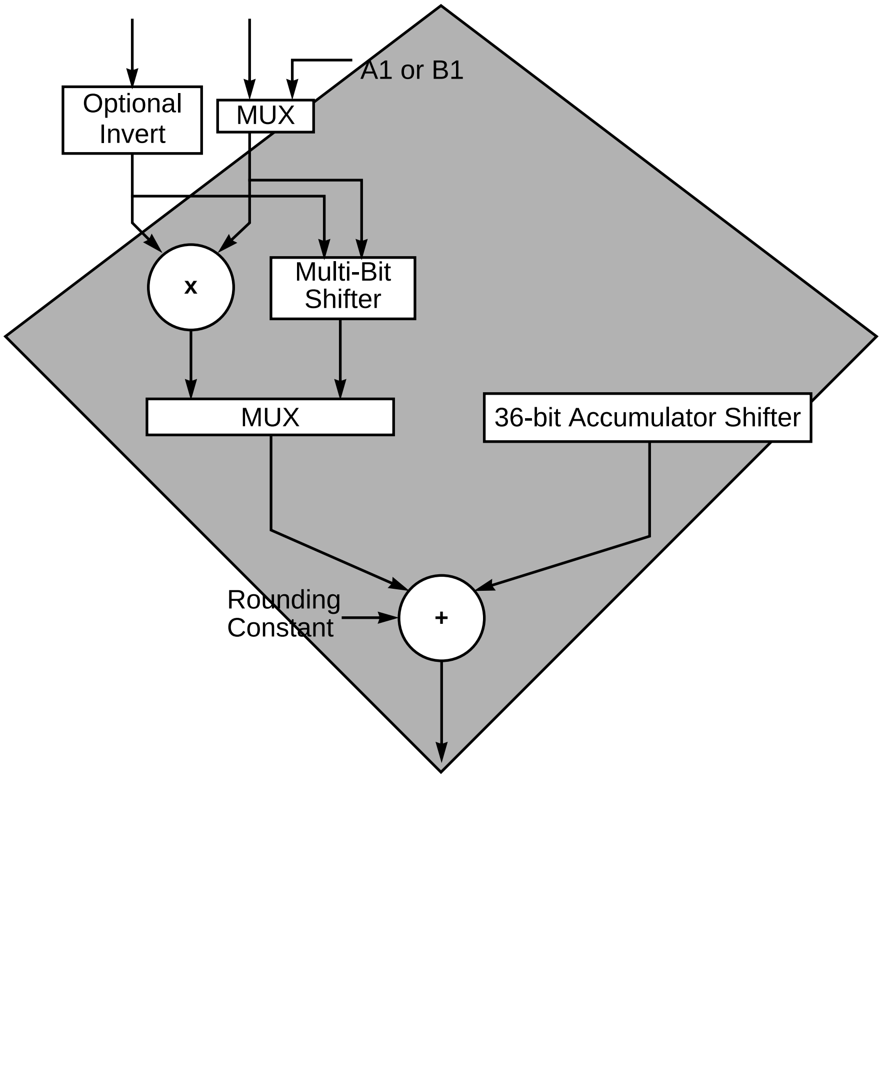
Figure 2-3 DSP56800 Core Programming Model

X:$FFFF	Reserved for On-chip Emulation	X:$FFDF	(Available for Peripherals)
X:$FFFE	(Reserved)	X:$FFDE	(Available for Peripherals)
X:$FFFD	(Reserved)	X:$FFDD	(Available for Peripherals)
X:$FFFC	(Reserved)	X:$FFDC	(Available for Peripherals)
X:$FFFB	IPR: Interrupt Priority Register	X:$FFDB	(Available for Peripherals)
X:$FFFA	(Reserved)	X:$FFDA	(Available for Peripherals)
X:$FFF9	BCR: Bus Control Register	X:$FFD9	(Available for Peripherals)
X:$FFF8	(Reserved)	X:$FFD8	(Available for Peripherals)
X:$FFF7	(Reserved)	X:$FFD7	(Available for Peripherals)
X:$FFF6	(Reserved)	X:$FFD6	(Available for Peripherals)
X:$FFF5	(Reserved)	X:$FFD5	(Available for Peripherals)
X:$FFF4	(Reserved)	X:$FFD4	(Available for Peripherals)
X:$FFF3	(Available for Peripherals)	X:$FFD3	(Available for Peripherals)
X:$FFF2	(Available for Peripherals)	X:$FFD2	(Available for Peripherals)
X:$FFF1	(Available for Peripherals)	X:$FFD1	(Available for Peripherals)
X:$FFF0	(Available for Peripherals)	X:$FFD0	(Available for Peripherals)
X:$FFEF	(Available for Peripherals)	X:$FFCF	(Available for Peripherals)
X:$FFEE	(Available for Peripherals)	X:$FFCE	(Available for Peripherals)
X:$FFED	(Available for Peripherals)	X:$FFCD	(Available for Peripherals)
X:$FFEC	(Available for Peripherals)	X:$FFCC	(Available for Peripherals)
X:$FFEB	(Available for Peripherals)	X:$FFCB	(Available for Peripherals)
X:$FFEA	(Available for Peripherals)	X:$FFCA	(Available for Peripherals)
X:$FFE9	(Available for Peripherals)	X:$FFC9	(Available for Peripherals)
X:$FFE8	(Available for Peripherals)	X:$FFC8	(Available for Peripherals)
X:$FFE7	(Available for Peripherals)	X:$FFC7	(Available for Peripherals)
X:$FFE6	(Available for Peripherals)	X:$FFC6	(Available for Peripherals)
X:$FFE5	(Available for Peripherals)	X:$FFC5	(Available for Peripherals)
X:$FFE4	(Available for Peripherals)	X:$FFC4	(Available for Peripherals)
X:$FFE3	(Available for Peripherals)	X:$FFC3	(Available for Peripherals)
X:$FFE2	(Available for Peripherals)	X:$FFC2	(Available for Peripherals)
X:$FFE1	(Available for Peripherals)	X:$FFC1	(Available for Peripherals)
X:$FFE0	(Available for Peripherals)	X:$FFC0	(Available for Peripherals)

Figure 2-4 DSP56800 Chip I/O and On-Chip Peripheral Memory Map

SECTION 3

DATA ARITHMETIC LOGIC UNIT



3.1 INTRODUCTION

This section describes the architecture and the operation of the data arithmetic logic unit (ALU), the block where the multiplication, logical operations, and arithmetic operations are performed. (Addition can also be performed in the address generation unit, and the bit-manipulation unit can perform logical operations.) The data ALU contains the following:

- Three 16-bit input registers (X0, Y0, and Y1)
- Two 32-bit accumulator registers (A and B)
- Two 4-bit accumulator extension registers (A2 and B2)
- An accumulator shifter (AS)
- One data limiter
- One 16-bit barrel shifter
- One parallel (single cycle, non-pipelined) multiply-accumulator (MAC) unit

Multiple buses in the data ALU perform complex arithmetic operations (such as a multiply-accumulate) in parallel with up to two memory transfers. A discussion of fractional and integer data representations, signed, unsigned, and multi-precision arithmetic, condition code generation, and the rounding modes used in the data ALU are also described in this section.

The data ALU can perform the following operations in a single instruction cycle:

- Multiplication (with or without rounding)
- Multiplication with inverted product (with or without rounding)
- Multiplication and accumulation (with or without rounding)
- Multiplication and accumulation with inverted product (with or without rounding)
- Addition and subtraction
- Compares
- Increments and decrements
- Logical operations (AND, OR, and EOR)
- One’s-complement
- Two’s-complement (negation)

- Arithmetic and logical shifts
- Rotates
- Multi-bit shifts on 16-bit values
- Rounding
- Absolute value
- Division iteration
- Normalization iteration
- Conditional register moves (Tcc)
- Saturation (limiting)

3.2 OVERVIEW AND ARCHITECTURE

The major components of the data ALU are:

- Three 16-bit input registers (X0, Y0, and Y1)
- Two 32-bit accumulator registers (A and B)
- Two 4-bit accumulator extension registers (A2 and B2)
- An accumulator shifter (AS)
- One data limiter
- One 16-bit barrel shifter
- One parallel (single cycle, non-pipelined) multiply-accumulator (MAC) unit

A block diagram of the data ALU unit is shown in **Figure 3-1**, and its corresponding programming model is shown in **Figure 3-2**. In the programming model, accumulator “A” refers to the entire 36-bit accumulator register, whereas “A2,” “A1,” and “A0” refer to the directly-accessible extension, most significant, and least significant portions of the 36-bit accumulator, respectively. The instruction has the ability to access the entire register as a whole or by these individual portions (see **Data ALU Accumulator Registers** on page 3-6). The blocks and registers within the data ALU are explained in the following sections.

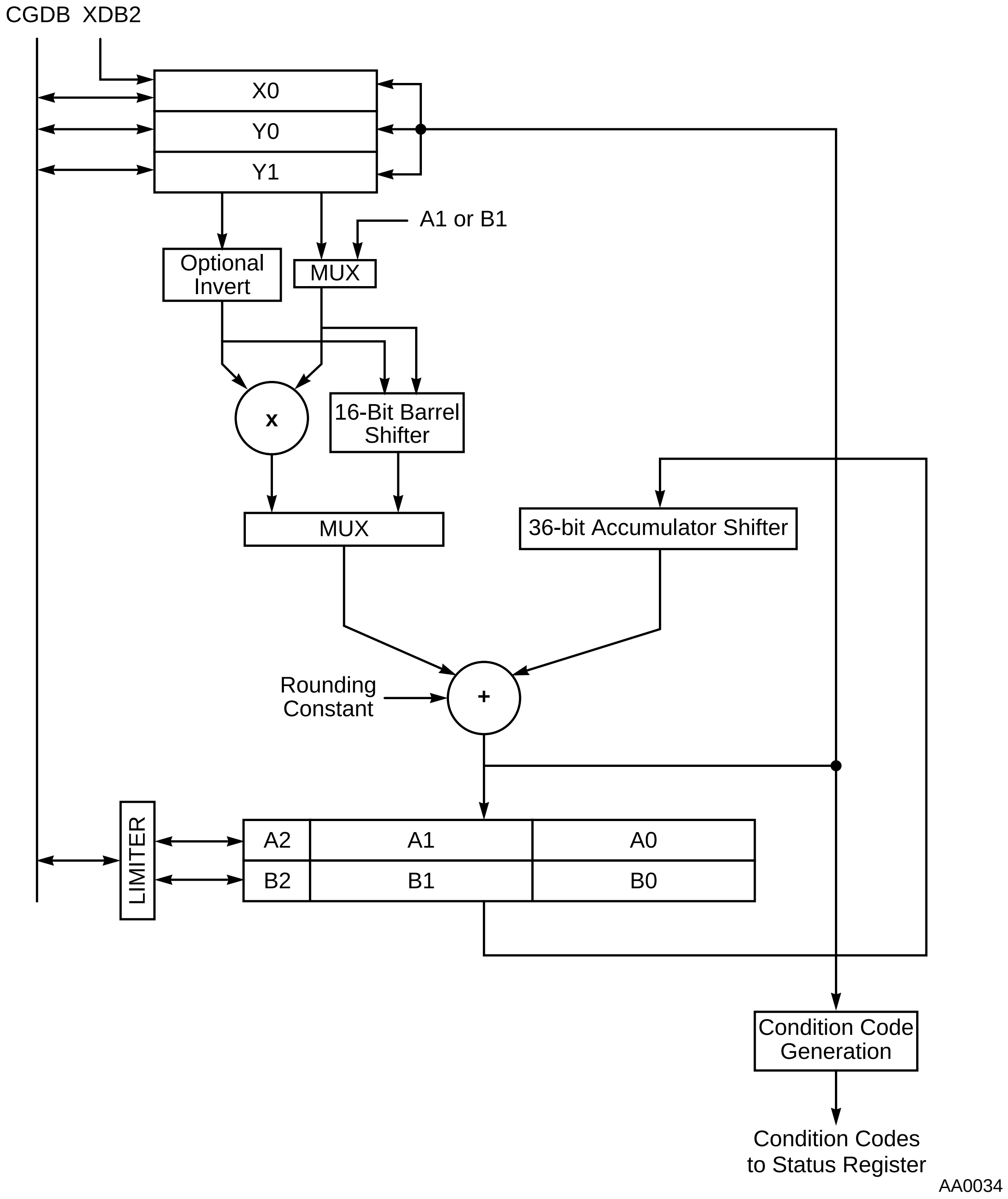


Figure 3-1 Data ALU Block Diagram

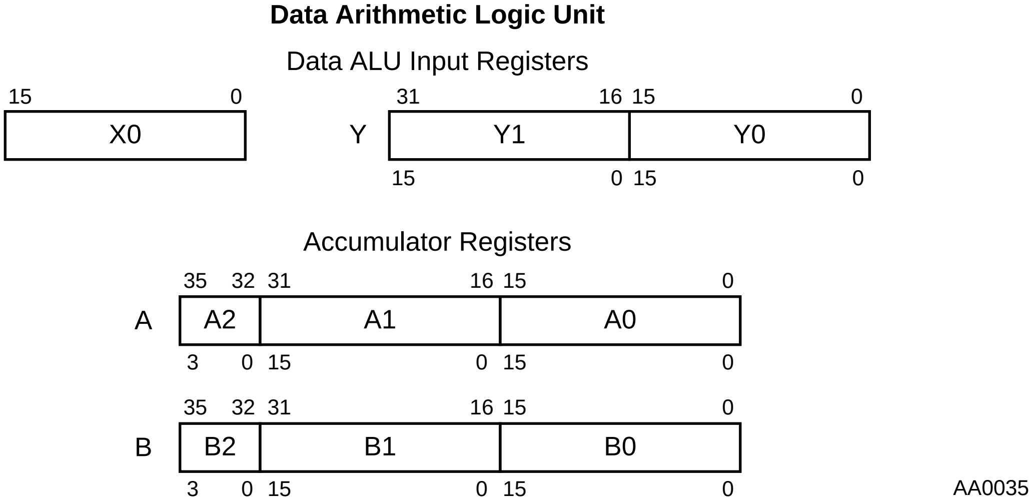


Figure 3-2 Data ALU Programming Model

3.2.1 Data ALU Input Registers (X0, Y1, Y0)

The data ALU input registers (X0, Y1, and Y0) are 16-bit registers that serve as input registers for the data ALU. Each register may be read or written by the CGDB as a word operand. They may be treated as three independent 16-bit registers. In addition, two of these can be treated as a 32-bit register called Y, which is developed by the concatenation of Y1:Y0 (Y1 is the most significant word in Y).

These data ALU input registers are used as source operands for most data ALU operations and allow new operands to be loaded from the memory for the next instruction while the register contents are used by the current instruction. X0 may also be written by the XDB2 during the dual read instruction. Certain arithmetic operations also allow these registers to be specified as destinations.

3.2.2 Data ALU Accumulator Registers

The two 36-bit data ALU accumulator registers can be accessed either as a 36-bit register (A or B) or as by the individual portions of the register listed below:

- 4-bit extension register (A2 or B2)
- 16-bit MSP (A1 or B1)
- 16-bit LSP (A0 or B0)

The three individual portions make up the entire accumulator register, as shown in **Figure 3-2**.

These two techniques for accessing the accumulator registers provide important flexibility for both DSP algorithms as well as general-purpose computing tasks. Accessing these registers as entire accumulators (A or B) is particularly useful for DSP tasks because this preserves the entire precision of multiplication and other ALU operations and also gives limiting or saturation capability in cases when moving a final result of a computation that has overflowed (see **Data Limiter** on page 3-13).

Saturation is especially important when running data through a digital filter whose output goes to a D/A, since it “clips” the output data instead of allowing arithmetic overflow. Without saturation, the output data incorrectly switches from a large positive number to a large negative value, which can cause problems on a D/A output in an embedded application.

The second technique of accessing the accumulator through its individual portions (A2, A1, A0, B2, B1, or B0) is useful for systems and control programming. For example, if a DSP algorithm is in progress and an interrupt is received, it may be necessary to save the accumulator value so it becomes usable by the interrupt service routine. Since the interrupt occurs before the DSP task has completed, it is important that no saturation takes place. Thus, an interrupt service routine can store the individual portions of the accumulator on the stack, effectively storing the entire 36-bit value without any limiting. Upon completion of the interrupt routine, the contents of the accumulator are then restored from the stack.

The accumulator registers may be accessed in both of these manners through the instruction set of the DSP, depending whether an instruction specifies an entire accumulator (A or B), or a portion of it (A2, A1, A0, B2, B1, or B0). **Table 3-1** summarizes the different accesses possible.

Table 3-1 Accessing the Accumulator Registers

Register	Read of the Accumulator Register	Write to the Accumulator Register
A or B	When used in a MOVE instruction, it follows these rules: If the extension bits are not in use for the accumulator to be read, then the 16-bit contents of A1 or B1 of the accumulator are read onto the CGDB bus. If the extension bits are in use, then a 16-bit “limited” value is instead read onto the CGDB (see **Data Limiter** on page 3-13). When used in an arithmetic operation, all 36 bits are sent to the computation unit without limiting.	When used in a MOVE instruction, the 16-bit CGDB is written into the 16-bit A1 or B1 register. A2 or B2 are filled with sign extension, and A0 or B0 are set to zero.
A2 or B2	The 4-bit register is read onto the 4 LSBs of the CGDB bus, and the upper 12 bits of the bus are sign extended. Together, this forms a 16-bit source. They can only be used in a MOVE instruction (see **Figure 3-3**).	The 4 LSBs of the CGDB are written into the 4-bit register, and the upper 12 bits are ignored. A1, A0, B1, and B0 are not modified. They can only be used in a MOVE instruction (see **Figure 3-3**).
A1 or B1	The 16-bit A1 or B1 register is read onto the 16-bit CGDB or used as a 16-bit operand for an arithmetic operation. They can be used in a MOVE, parallel MOVE, and several different arithmetic instructions.	The contents of the 16-bit CGDB are written into the 16-bit A1 or B1 register. A2, A0, B2, and B0 are not modified. They can only be used in a MOVE or in a parallel move instruction.
A0 or B0	The 16-bit A0 or B0 register is read onto the 16-bit CGDB bus. They can only be used in a MOVE instruction.	The contents of the 16-bit CGDB are written into the 16-bit A0 or B0 register. A2, A1, B2, and B1 are not modified. They can only be used in a MOVE instruction.

3.2.2.1 Accessing an Accumulator by its Individual Portions

A1, A0, B1 and B0 are 16-bit registers that serve as data ALU accumulator registers. A2 and B2 are 4-bit latches that serve as accumulator extension registers. Each register may be read or written by the CGDB as a word operand. When A2 or B2 is read, the register contents occupy the low-order portion (bits 3-0) of the word; the high-order portion (bits 15-4) is sign-extended. When A2 or B2 is written, the register receives the low-order portion of the word; the high-order portion is not used. (See **Figure 3-3**.)

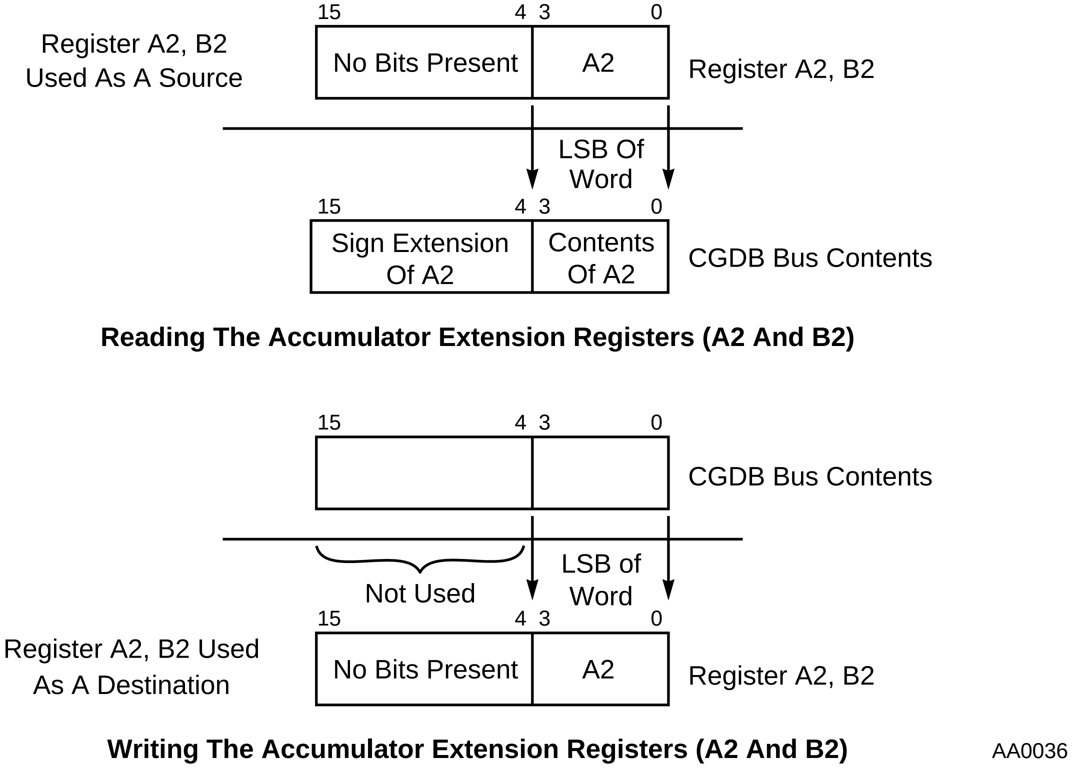


Figure 3-3 Accessing the Accumulator Extension Registers (A2, B2)

3.2.2.2 Accessing an Accumulator as a Whole

The accumulator registers are written with the result of an ALU or multiplication operation. In this case, they are written as an entire 36-bit accumulator (A or B), and not as an individual register (A2, A1, A0, B2, B1, or B0). The accumulator registers receive the EXT:MSP:LSP of the multiply-accumulator unit output and supply a source accumulator of the same form. Most data ALU operations specify the 36-bit accumulator registers as source and/or destination operands.

Automatic sign extension of the 36-bit accumulators is provided when the A or B register is written with a smaller size operand. This can occur when writing A or B from the CGDB (MOVE instruction) or with the results of certain data ALU operations (TFR instruction). If a word operand is to be written to an accumulator register (A or B), the MSP of the accumulator is written with the word operand, the LSP is zeroed, and the EXT portion is sign-extended from the MSP. This is also the case for a MOVE instruction that moves one accumulator to another, but is not the case for a TFR instruction that moves one accumulator to another. No sign extension is performed if an individual 16-bit register is written (A1, A0, B1, or B0).

Note: A read of the A1 or B1 register in a MOVE instruction is identical to a read of the A or B register when the extension bits of that accumulator only contain sign extension information. In this case there is no need for saturation or limiting, so reading the A or B register produces the same result as reading the A1 or B1 register.

The extension registers A2 and B2 offer protection against 32-bit overflow. When the result of an accumulation crosses the MSB of MSP (bit 31 of A or B), the extension bit of the status register (E) is set. Up to 15 overflows or underflows are possible using these extension bits, after which the sign is lost beyond the MSB of the extension register. When this occurs, the overflow bit (V) in the status register is set. Having an extension register allows overflow during intermediate calculations without losing this information. This is useful because sometimes in DSP algorithms, there will be overflow in intermediate calculations, but there is no overflow in the final result that is written to memory or to a peripheral.

The logic detection of the extension register in use is also used to saturate the value of an accumulator as it passes through the limiter while reading A or B accumulators over the CGDB or transferring them to any data ALU register. The content of A or B is not affected in that case (except if the same accumulator is specified as both source and destination); only the value transferred over the CGDB is limited to a full-scale positive or negative 16-bit value ($7FFF or $8000). When limiting occurs, a flag is set and latched in the status register (L). The limiting block is explained in more detail in **Data Limiter** on page 3-13.

Note: Limiting will be performed only when the entire 36-bit accumulator register (A or B) is specified as the source for a parallel data move over the CGDB or register transfer. It is not performed when A0, A1, A2, B0, B1, or B2 is specified.

3.2.2.3 Converting from 36-bit Accumulator to 16-bit Portion

There are two instructions that are useful for converting the 36-bit contents of an accumulator to a 16-bit value, which can then be stored to memory or used for further computations. This is useful for processing word-sized operands (16 bits), since it guarantees an accumulator contains correct sign extension and that the least significant 16 bits are all zeros. The two techniques are shown in **Example 3-1**.

Example 3-1 Converting a 36-bit Accumulator to a 16-bit Value

```
            ;Converting with No Limiting
MOVE  A1,A  ;Sign Extend A2, A0 set to $0000
MOVE  B1,B  ;Sign Extend B2, B0 set to $0000
```

Example 3-1 Converting a 36-bit Accumulator to a 16-bit Value (Continued)

```
            ;Converting with Limiting Performed
            ;if Extension is in Use
MOVE  A,A   ;Sign Extend A2, A0 set to $0000, Limit if
            ;Necessary
MOVE  B,B   ;Sign Extend B2, B0 set to $0000, Limit if
            ;Necessary
```

3.2.3 Multiply-accumulator (MAC) and Logic Unit

The multiply-accumulator (MAC) and logic unit is the main arithmetic processing unit of the DSP. This is the block that performs all multiplications, additions, subtractions, logical, and other arithmetic operations except shifting. It accepts up to three input operands and outputs one 36-bit result of the form extension:most significant product:least significant product (EXT:MSP:LSP). Arithmetic operations in the MAC unit occur independently and in parallel with memory accesses on the CGDB, XDB2, and PDB. The data ALU registers provide pipelining for both data ALU inputs and outputs. An input register may be written by memory in the same instruction where it is used as the source for a data ALU operation. The MAC unit is completely asynchronous logic and all ALU operations occur in one instruction cycle. The inputs of the MAC and logic unit can come from the X and Y registers (X0, Y1, Y0), the accumulators (A1, B1, A, B), and also directly from memory for common instructions such as ADD and SUB.

The multiplier executes 16 x 16-bit parallel signed / unsigned fractional and 16 x 16-bit parallel signed integer multiplications. The 32-bit product is added to the 36-bit contents of either the A or B accumulator or the 16-bit contents of the X0, Y0, or Y1 registers and then stored in the same register. This multiply / accumulate is a single cycle operation (no pipeline). For integer multiplication, the 16 LSBs of the product are stored in the MSP of the accumulator, and the extension register is filled with sign extension.

If a multiply without accumulation is specified by a MPY or MPYR instruction, the unit clears the accumulator and then adds the contents to the product. The results of all arithmetic instructions are valid (sign extended) 36-bit operands in the form EXT:MSP:LSP (A2:A1:A0 or B2:B1:B0).

When a 36-bit result is to be stored as a 16-bit operand, the LSP can simply be truncated or it can be rounded into the MSP. The rounding performed is either the convergent rounding (round to the nearest even) or two’s-complement rounding.

The type of rounding is specified by the rounding bit in the operating mode register. See **Rounding** on page 3-37 for a more detailed discussion of rounding.

The logic unit performs the logical operations AND, OR, EOR, and NOT on data ALU registers. It is 16 bits wide and operates on data in the MSP of the accumulator. The least significant and EXT portions of the accumulator are not affected. Logical operations can also be performed in the bit-manipulation unit. The bit-manipulation unit is used when performing logical operations with immediate values and can be performed on any register or memory location.

3.2.4 Barrel Shifter

The 16-bit barrel shifter performs single-cycle 0- to 15-bit arithmetic or logical shifts of 16-bit data. Since both the amount to be shifted as well as the value to shift come from registers, it is possible to shift data by a variable amount. (See **Figure 3-4**.) It is possible to use this unit to right-shift 32-bit values as well using the ASRAC and LSRAC instructions, as demonstrated in Section 8.4.

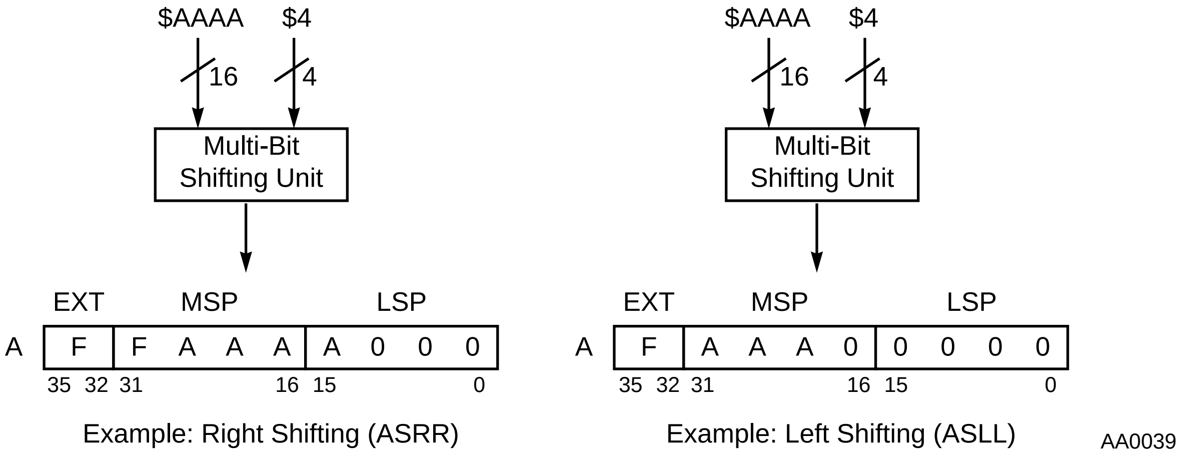


Figure 3-4 Right and Left Shifts Through the Multi-bit Shifting Unit

After shifting, the extension register is always loaded with zero extension or sign extension for logical or arithmetic shifts, respectively. For right shifts, the LSP is set to zero except for the ASRAC and LSRAC instructions, where the lower bits are shifted into the LSP. For left shifts, the upper bits are NOT shifted into the extension register, and the LSP is always set to zero.

3.2.5 Accumulator Shifter (AS)

The accumulator shifter (AS) is an asynchronous parallel shifter with a 36-bit input and a 36-bit output. The operations performed by this unit are as follows:

- No shift performed—ADD, SUB, MAC, etc.
- 1-bit left shift—ASL, LSL, ROL
- 1-bit right shift—ASR, LSR, ROR
- Force to zero—MPY, IMPY(16)

The output of the shifter goes directly to the MAC unit as an input.

3.2.6 Data Limiter

The data limiter provides a saturation or limiting capability when reading a 36-bit accumulator register out to the CGDB or to another register.

Saturation arithmetic protects against overflow by selectively limiting when reading a data ALU accumulator register. When a MOVE instruction specifies an accumulator (A or B) as a source, and if the contents of the selected source accumulator can be represented in the destination operand size without overflow (i.e., the accumulator extension register not in use), the data limiter is disabled and the register contents are placed onto the CGDB unmodified. If the contents of the selected source accumulator cannot be represented without overflow in the destination operand size, the data limiter will substitute a “limited” data value onto the CGDB that has maximum magnitude and the same sign as the source accumulator, as shown in **Table 3-2**.

Test logic exists in each accumulator register to support the operation of the limiter circuit; the logic detects overflows so that the limiter can substitute one of two constants to minimize errors due to overflow. This process is called “saturation arithmetic.” When limiting does occur, a flag is set and latched in the status register. The value of the accumulator is not changed.

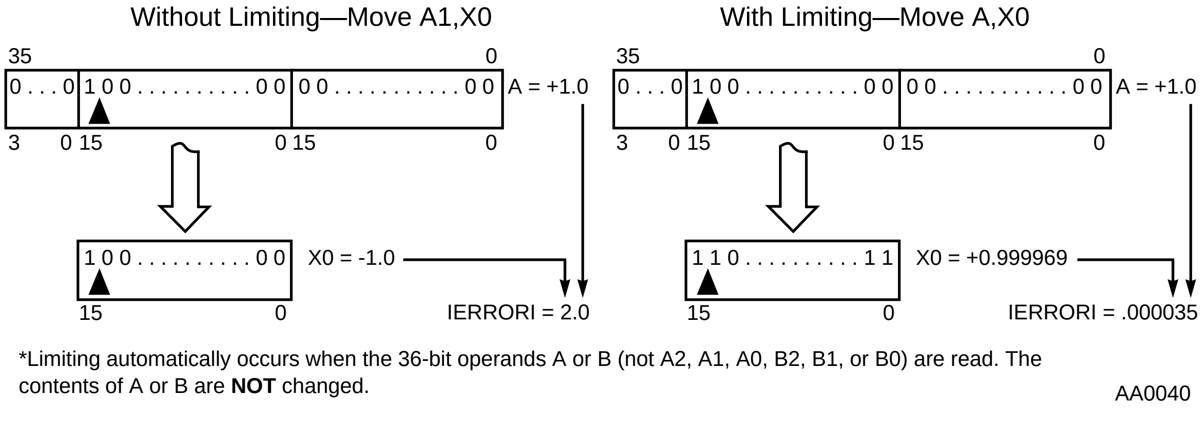
Table 3-2 Saturation by the Limiter

Extension bits in use in selected accumulator?	MSB of A2, B2	Output of Limiter onto the CGDB Bus
No	(don't care)	Same as Input—Unmodified MSP
Yes	0	$7FFF—Maximum Positive Value
Yes	1	$8000—Maximum Negative Value

It is possible to bypass this limiting feature when reading an accumulator by reading it out through its individual portions (A1 or B1).

For example, if the 36-bit source accumulator to be read to a 16-bit destination was 0000 1.000 0000 0000 0000 0000 0000 0000 0000 in a binary representation (+ 1.0 in decimal, $0 8000 0000 in hexadecimal) and the destination register was a 16-bit register, the destination register would contain 1.000 0000 0000 0000 (- 1.0 decimal, $8000 in hexadecimal) after the transfer, assuming signed fractional arithmetic. This is clearly in error because overflow has occurred. To minimize the error due to overflow, it is preferable to write the maximum ("limited") value the destination can assume. In the example, the limited value would be 0.111 1111 1111 1111 (+ 0.999969 decimal, $7FFF in hexadecimal), which is clearly closer to + 1.0 than - 1.0 and, therefore, introduces less error. Saturation is equally applicable to both integer and fractional arithmetic.

Figure 3-5 shows the effects of saturation arithmetic on a move from register A1 to register X0. The instruction MOVE A1,X0 performs a move without limiting, and the instruction MOVE A,X0 performs a move of the same 16 bits with limiting. The magnitude of the error without limiting is 2.0; with limiting it is 0.000035.



*Limiting automatically occurs when the 36-bit operands A or B (not A2, A1, A0, B2, B1, or B0) are read. The contents of A or B are **NOT** changed.

Figure 3-5 Example of Saturation Arithmetic

3.3 FRACTIONAL AND INTEGER DATA ALU ARITHMETIC

The section that follows give an introduction to fractional and integer arithmetic followed by sections on interpreting data and data formats, simple mathematics (addition, subtraction, multiplication, and division), unsigned arithmetic, and multi-precision operands.

3.3.1 Introduction

The ability to perform both integer and fractional arithmetic is one of the strengths of the DSP56800 architecture; there is a need for both types of arithmetic.

Fractional arithmetic is typically required for computation-intensive algorithms such as digital filters, speech coders, vector and array processing, digital control, or other signal processing tasks. In this mode the data is interpreted as fractional values, and the computations are performed interpreting the data as fractional. Often, saturation is used when performing calculations in this mode to prevent the severe distortion that occurs in an output signal generated from a result where a computation overflows without saturation (see **Figure 3-5**). Saturation can be selectively enabled or disabled so that intermediate calculations can be performed without limiting, and limiting is only done on final results (see **Example 3-2**).

Example 3-2 Fractional Arithmetic Examples

```
0.5 x 0.25 = 0.125
0.625 + 0.25 = 0.875
0.125 / 0.5 = 0.25
0.5 >> 1 = 0.25
```

Integer arithmetic, on the other hand, is invaluable for controller code, for array indexing and address computations, compilers, peripheral setup and handling, bit manipulation, bit-exact algorithms, and other general purpose tasks. Typically, saturation is not used in this mode, but is available if desired. (See **Example 3-3**.)

Example 3-3 Integer Arithmetic Examples

```
4 x 3 = 12
1201 + 79 = 1280
63 / 9 = 7
100 << 1 = 200
```

The main difference between fractional and integer representations is the location of the decimal (or binary) point. For fractional arithmetic, the decimal (or binary) point is always located immediately to the right of the MSP's most significant bit; for integer values, it is always located immediately to the right of the value's LSB. **Figure 3-6** shows the location of the decimal point (binary point), bit weightings, and operands alignment for different fractional and integer representations supported on the DSP56800 architecture.

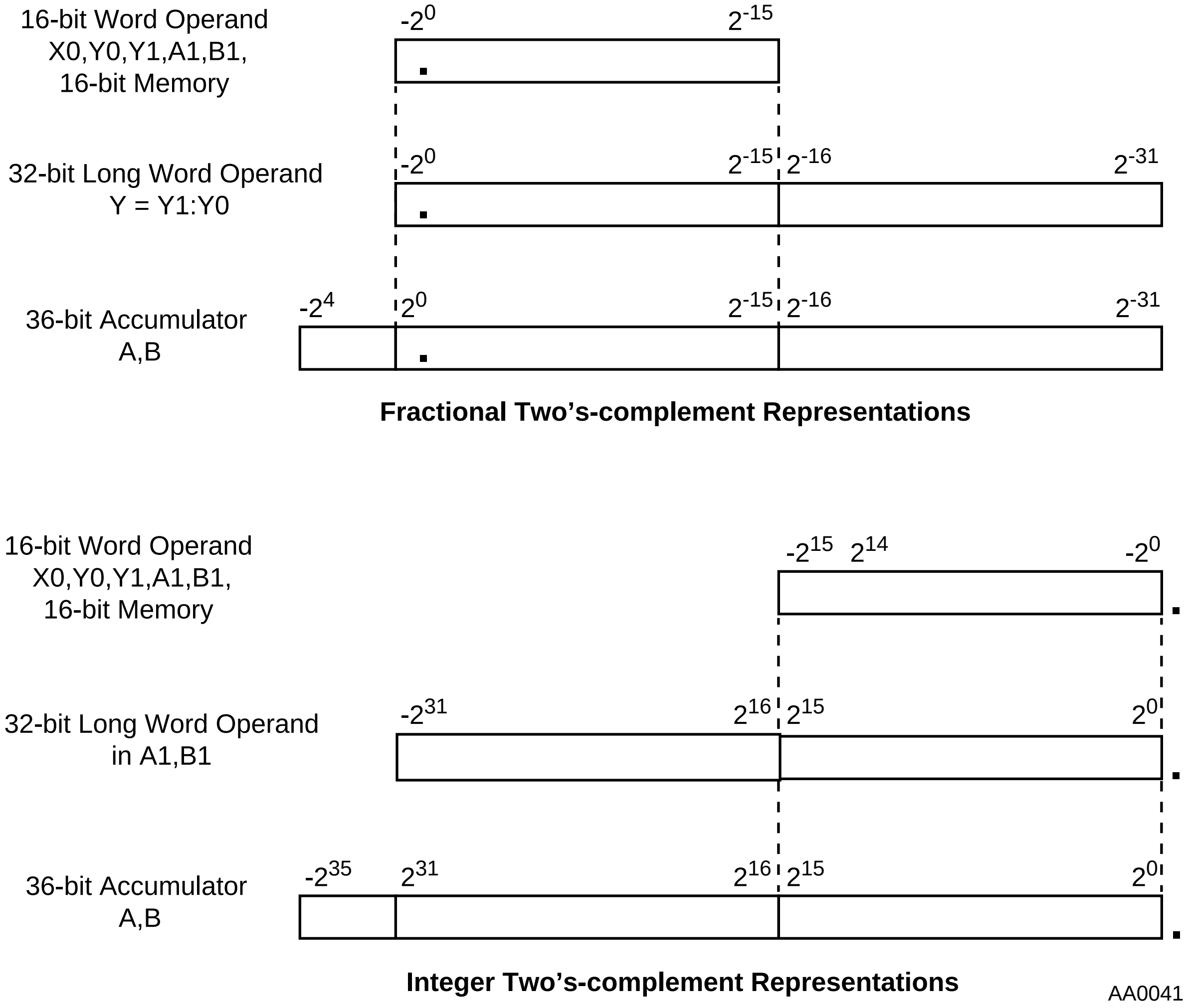


Figure 3-6 Bit Weighting and Alignments for Operands

The representation of numbers allowed on the DSP56800 architecture are:

- Two's-complement values
- Fractional or integer values

- Signed or unsigned values
- Word (16-bit), long word (32-bit), or accumulator (36-bit)

The different representations not only affect the arithmetic operations, but the condition code generation as well. These numbers can be represented as decimal, hexadecimal, or binary numbers.

To maintain alignments of the binary point when a word operand is written to an accumulator A or B, the operand is written to the most significant accumulator register (A1 and B1) and its most significant bit is automatically sign-extended through the accumulator extension register. The least significant accumulator register is automatically cleared.

Some of the advantages of fractional data representation are as follows:

- The MSP (left half) has the same format as the input data.
- The LSP (right half) can be rounded into the MSP without shifting or updating the exponent.
- Conversion to floating-point representation is easier because the industry-standard floating-point formats use fractional mantissas.
- Coefficients for most digital filters are derived as fractions by DSP digital-filter design software packages. The results from the DSP design tools can be used without the extensive data conversions that other formats require.
- A significant bit is not lost through sign extension.

3.3.2 Interpreting Data

Data in a memory location or register can be interpreted as fractional or integer, depending on the needs of a user’s program. **Table 3-3** shows how a 16-bit value can be interpreted as either a fractional or integer value, depending on the location of the binary point.

Table 3-3 Interpretation of 16-bit Data Values

Binary Representation[1]	Hexadecimal Representation	Integer Value (decimal)	Fractional Value (decimal)
0.100 0000 0000 0000	$4000	16384	0.5
0.010 0000 0000 0000	$2000	8192	0.25
0.001 0000 0000 0000	$1000	4096	0.125
0.111 0000 0000 0000	$7000	28672	0.875
0.000 0000 0000 0000	$0000	0	0.0
1.100 0000 0000 0000	$C000	- 16384	- 0.5
1.110 0000 0000 0000	$E000	- 8192	- 0.25
1.111 0000 0000 0000	$F000	- 4096	- 0.125
1.001 0000 0000 0000	$9000	- 28672	- 0.875

Note: 1. This corresponds to the location of the binary point when interpreting the data as fractional. If the data is interpreted as integer, the binary point is located immediately to the right of the LSB.

The following equation shows the relationship between a 16-bit integer and a fractional value:

$$\text{Fractional Value} = \text{Integer Value} \ / \ (2^{15})$$

There is a similar equation relating 36-bit integers and fractional values:

$$\text{Fractional Value} = \text{Integer Value} \ / \ (2^{31})$$

Table 3-4 shows how a 36-bit value can be interpreted as either an integer or fractional value, depending on the location of the binary point.

Table 3-4 Interpretation of 36-bit Data Values

Hexadecimal Representation	36-bit Integer in Entire Accumulator (decimal)	16-bit Integer in MSP (decimal)	Fractional Value (decimal)
$0 4000 0000	1073741824	16384	0.5
$0 2000 0000	536870912	8192	0.25
$0 0000 0000	0	0	0.0
$F C000 0000	- 1073741824	- 16384	- 0.5
$F E000 0000	- 536870912	- 8192	- 0.25

3.3.3 Data Formats

Four types of two’s-complement data formats are supported by the 16-bit DSP core:

- Signed fractional (SF)
- Unsigned fractional (UF)
- Signed integer (SI)
- Unsigned integer (UI)

The ranges for each of these formats (discussed below) apply to all data stored in memory and to data stored in the data ALU registers. The extension registers associated with the accumulators allow word growth so that the most positive number that can be represented in an accumulator is approximately 16.0 and the most negative number is -16.0. When the accumulator extension registers are in use, the data contained in the accumulators cannot be stored exactly in memory or other registers. In these cases the data must be limited to the most positive or most negative number consistent with the size of the destination and the sign of the accumulator, the MSB of the extension register.

3.3.3.1 Signed Fractional

In this format the N bit operand is represented using the 1.[N-1] format (1 sign bit, N-1 fractional bits). Signed fractional numbers lie in the following range:

$$-1.0 \leq SF \leq +1.0 - 2^{-[N-1]}$$

For words and long word signed fractions, the most negative number that can be represented is -1.0, whose internal representation is $8000 and $80000000, respectively. The most positive word is \$7FFF or 1.0 - 2^{-15}, and the most positive long word is \$7FFFFFFF or 1.0 - 2^{-31}.

3.3.3.2 Unsigned Fractional

Unsigned fractional numbers may be thought of as positive only. The unsigned numbers have nearly twice the magnitude of a signed number with the same number of bits. Unsigned fractional numbers lie in the following range:

$$0.0 \le UF \le 2.0 - 2^{-[N-1]}$$

Examples of unsigned fractional numbers are 0.25, 1.25, and 1.999. The binary word is interpreted as having a binary point after the MSB. The most positive 16-bit unsigned number is \$FFFF or $\{1.0 + (1.0 - 2^{-[N-1]})\} = 1.99996948$. The smallest unsigned number is zero (\$0000).

3.3.3.3 Signed Integer

This format is used when processing data as integers. Using this format, the N-bit operand is represented using the N.0 format (N integer bits). Signed integer numbers lie in the following range:

$$-2^{-[N-1]} \le SI \le [2^{[N-1]}-1]$$

For words and long word signed integers the most negative word that can be represented is -32768 (\$8000), and the most negative long word is -2147483648 (\$80000000). The most positive word is 32767 (\$7FFF), and the most positive long word is 2147483647 (\$7FFFFFFF).

3.3.3.4 Unsigned Integer

Unsigned integer numbers may be thought of as positive only. The unsigned numbers have nearly twice the magnitude of a signed number of the same length. Unsigned integer numbers lie in the following range:

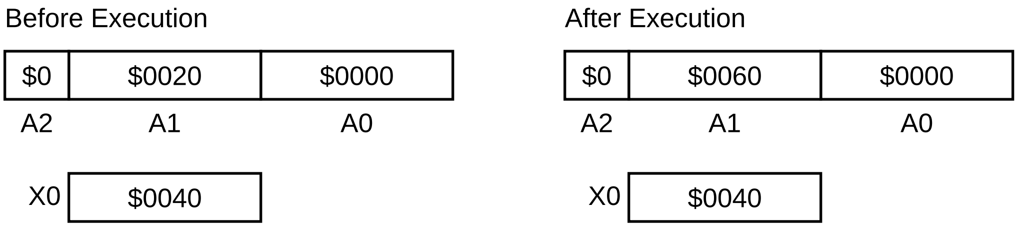
$$0 \le UI \le [2^{N}-1]$$

Examples of unsigned integer numbers are 25, 125, and 1999. The binary word is interpreted as having a binary point immediately to the right of the LSB. The most positive 16-bit unsigned integer is 65536 (\$FFFF). The smallest unsigned number is zero (\$0000).

3.3.4 Addition and Subtraction

For fractional and integer arithmetic, the operations are performed identically for addition, subtraction, or comparing two values. This means that any add, subtract, or compare instruction can be used for both fractional and integer values.

To perform fractional or integer arithmetic operations with word-sized data, the data is loaded into the MSP (A1 or B1) of the accumulator as shown in **Figure 3-7**.



```
MOVE  #64,X0      ; Load integer value “64” ($40) into X0
MOVE  #32,A       ; Load integer value “32” ($20) into the A Accumulator
                  ; (correctly sign extends into A2 and zeros A0)
ADD   X0,A        ; Perform Integer Word Addition
MOVE  A1,X:RESULT ; Save Result (without saturating) to Memory
```

AA0045

Figure 3-7 Word-Sized Integer Addition Example

Fractional word-sized arithmetic would be performed in a similar manner. For arithmetic operations where the destination is a 16-bit register or memory location, the fractional or integer operation is correctly calculated and stored in its 16-bit destination.

3.3.5 Logical Operations

For fractional and integer arithmetic, the logical operations (AND, OR, EOR, and bit-manipulation instructions) are performed identically. This means that any DSP56800 logical or bit-field instruction can be used for both fractional and integer values. Typically, logical operations are only performed on integer values, but there is no inherent reason why they cannot be performed on fractional values as well.

Likewise, shifting can be done on both integer and fractional data values. For both of these, an arithmetic left shift of one bit corresponds to a multiplication by two. An arithmetic right shift of one bit corresponds to a division of a signed value by two, and a logical right shift of one-bit corresponds to a division of an unsigned value by two.

3.3.6 Multiplication

The multiplication operation is not the same for integer and fractional arithmetic. The result of a fractional multiplication differs in a simple manner from the result of an integer multiplication. This difference amounts to a 1-bit shift of the final result, as illustrated in **Figure 3-8**. Any binary multiplication of two N-bit signed numbers gives a signed result that is 2N-1 bits in length. This 2N-1 bit result must then be correctly placed into a field of 2N bits to correctly fit into the on-chip registers. For correct fractional multiplication, an extra 0 bit is placed at the LSB to give a 2N bit result. For correct integer multiplication, an extra sign bit is placed at the MSB to give a 2N bit result.

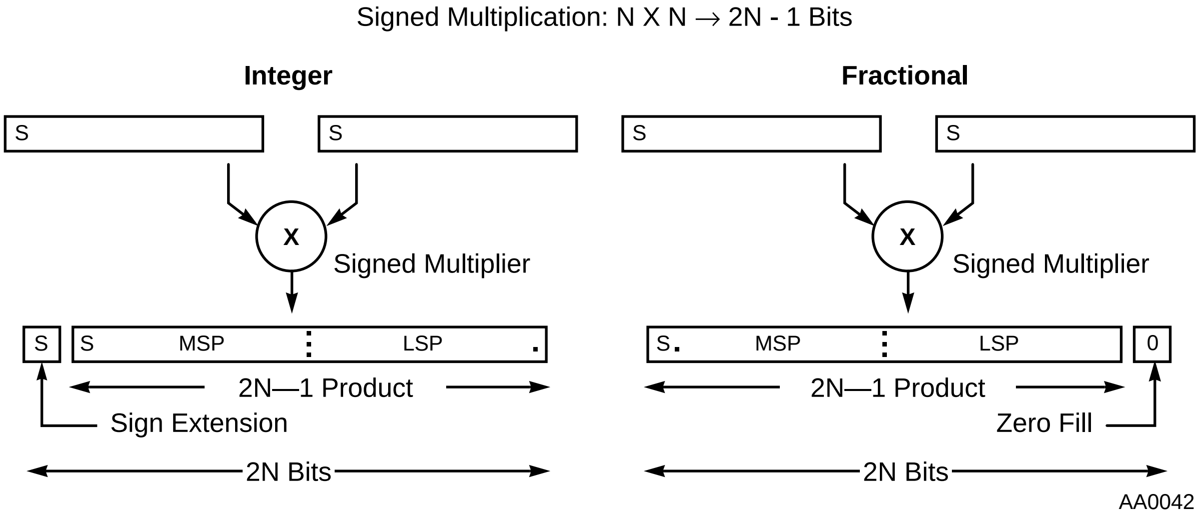


Figure 3-8 Comparison of Integer and Fractional Multiplication

The MPY, MAC, MPYR, and MACR instructions perform fractional multiplication and fractional multiply-accumulation. The IMPY(16) instruction performs integer multiplication. **Integer Multiplication** on page 3-23 explains how to perform integer multiplication.

3.3.6.1 Fractional Multiplication

Figure 3-9 shows the multiply-accumulation implementation for fractional arithmetic. The multiplication of two 16-bit signed fractional operands gives an intermediate 32-bit signed fractional result with the LSB always set to zero. This intermediate result is added to one of the 36-bit accumulators. If rounding is specified in the MPY or MAC instruction (MACR or MPYR), the intermediate results will be rounded to 16 bits before being stored back to the destination accumulator and the LSP will be set to zero.

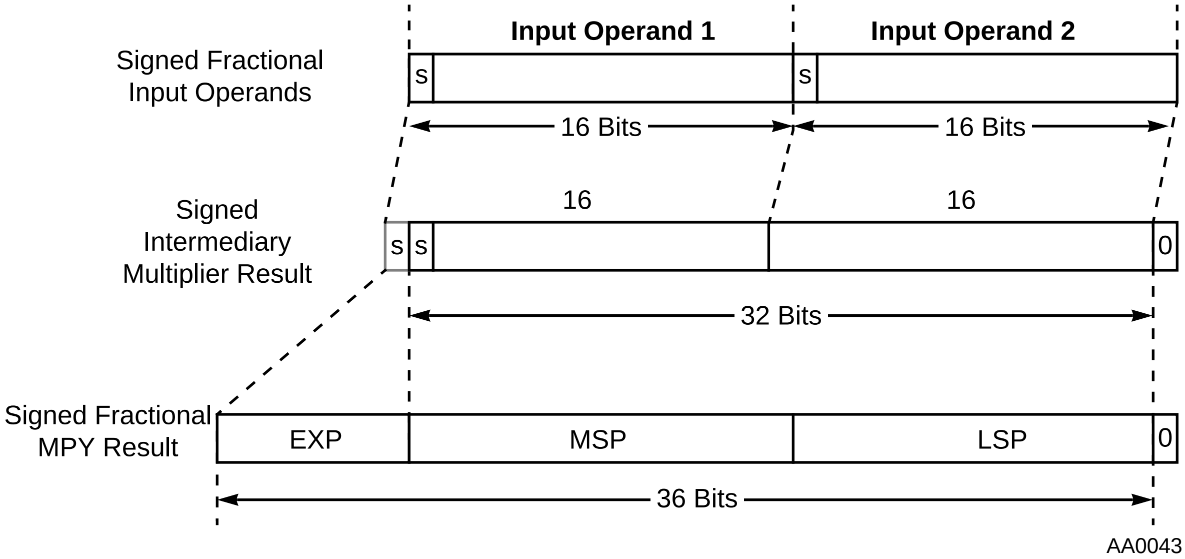

Input Operand 1
Input Operand 2
Signed Fractional Input Operands
s
s
16 Bits
16 Bits
Signed Intermediary Multiplier Result
16
16
s
s
0
32 Bits
Signed Fractional MPY Result
EXP
MSP
LSP
0
36 Bits
AA0043

Figure 3-9 MPY Operation—Fractional Arithmetic

3.3.6.2 Integer Multiplication

Two techniques for performing integer multiplication on the DSP core are listed below:

- Using the IMPY16 instruction to generate a 16-bit result in the MSP of an accumulator
- Using the MPY and MAC instructions to generate a 36-bit full precision result

Each technique has its advantages for different types of computations.

An examination of the instruction set shows that for execution of single precision operations, most often the instructions operate on the MSP (bits 31-16) of the accumulator instead of the LSP (bits 15-0). This is true for the LSL, LSR, ROL, ROR, NOT, INCW, DECW, etc. instructions. Likewise, for the parallel MOVE instructions, it is possible to move data to and from the MSP of an accumulator, but this is not true for the LSP. Thus, an integer multiplication instruction that places its result in the MSP of an accumulator allows for more efficient computing. This is the reason why the IMPY16 instruction places its results in bits 31-16 of an accumulator. The limitation with the IMPY16 instruction is that the result must fit within 16 bits or there is an overflow.

Figure 3-10 shows the multiply operation for integer arithmetic. The multiplication of two 16-bit signed integer operands using the IMPY16 instruction gives a 16-bit signed integer result that is placed in the MSP (A1 or B1) of the accumulator. The corresponding extension register (A2 or B2) is filled with sign extension and the LSP (A0 or B0) remains unchanged.

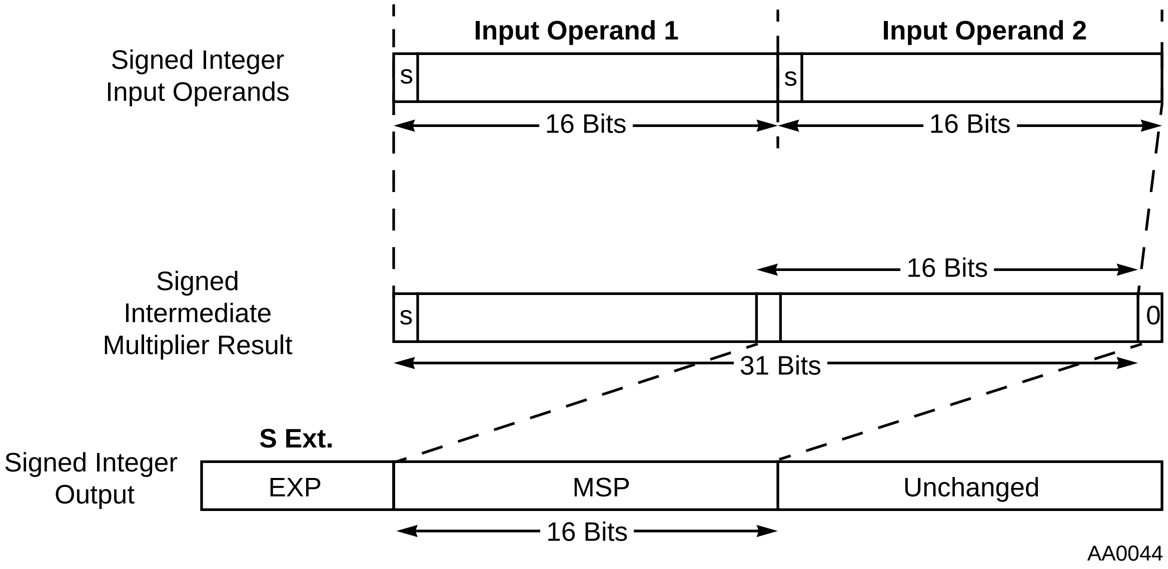


Figure 3-10 Integer Multiplication (IMPY)

At other times it is necessary to maintain the full 32-bit precision of an integer multiplication. To obtain integer results, a MPY instruction is used, immediately followed by an ASR instruction. The 32-bit long integer result is then correctly located into the MSP and LSP of an accumulator with correct sign extension in the extension register of the same accumulator (see **Example 3-4**).

Example 3-4 Multiplying Two Signed Integer Values with Full Precision

```
        MPY   X0,Y0,A       ; Generates correct answer shifted one
                            ; bit to the left
        ASR   A             ; Leaves Correct 32-bit Integer Result
                            ; in the A Accumulator
                            ; and the A2 register contains
                            ; correct sign extension
```

When performing a multiply-accumulate on a set of integer numbers, there is a faster way for generating the result than performing an “ASR” instruction after each multiply. The technique is to use fractional multiply-accumulates for the bulk of the computation and to then convert the final result back to integer.

Example 3-5 Fast Integer MACs using Fractional Arithmetic

```
      MOVE           X:(R0)+,Y0  X:(R3)+,X0
      DO   #N,LABEL
      MAC  X0,Y0,A   X:(R0)+,Y0  X:(R3)+,X0
LABEL
      ASR  A     ; Convert to Integer only after MACs are
                 ; completed
```

3.3.7 Division

Fractional and integer division of both positive and signed values is supported using the DIV instruction. The dividend (numerator) is a 32-bit fractional or 31-bit integer value, and the divisor (denominator) is a 16-bit fractional or integer value, respectively. See **Division** on page 8-17 for a complete discussion of division.

3.3.8 Unsigned Arithmetic

Unsigned arithmetic can be performed on the DSP56800 architecture. The addition, subtraction, and compare instructions work for both signed and unsigned values, but the condition code computation is different. Likewise, there is a difference for unsigned multiplication.

3.3.8.1 Condition Codes for Unsigned Operations

Unsigned arithmetic can be supported on operations such as addition, subtraction, compares, and logical operations using the same ADD, SUB, CMP, etc. instructions used for signed computations. The operations are performed the same for both representations. The difference lies in how the data is interpreted (**Data Formats** on page 3-19) and which status bits are used when comparing signed versus unsigned numbers.

When comparing unsigned numbers, there are four new condition codes:

- HS (High or same)—unsigned greater than or equal to
- LS (Low or same)—unsigned less than or equal to
- HI (High)—unsigned greater than
- LO (Low)—unsigned less than

The four condition codes used for comparing unsigned numbers, HS, LS, HI, LO, are used in place of GE, LE, GT, and LT respectively, which are used for comparing signed numbers. Note that the HS condition code is exactly the same as the carry clear (CC), and that LO is exactly the same as carry set (CS).

Unsigned comparisons are supported when the CC bit in the OMR register is set. When this bit is set, the value in the extension register is ignored when generating the C, V, N, and Z condition codes, and the condition codes are set using only the 32 LSBs of the result. Typically, this mode is very useful for controller and compiled code.

Note: The unsigned condition codes (HS, LS, HI, and LO) may only be used when the CC bit is set in the program controller’s OMR register. If this bit is not set, then these condition codes should NOT be used.

In cases where it is necessary to maintain all 36 bits of the result and the extension register is required, any unsigned numbers must first be converted to signed when loaded into the accumulator using the technique in **Unsigned Load of an Accumulator** on page 8-10, which converts unsigned numbers to signed numbers. In these cases, the extension register will contain the correct value, and since values are now signed, it is possible to use the signed branch conditions: GE, LE, GT, or LT. Typically, this mode is more useful for DSP code.

3.3.8.2 Unsigned Multiplication

Unsigned multiplications are supported with the MACSU and MPYSU instructions. If only one operand is unsigned, then these instructions can be used directly. If both operands are unsigned, an unsigned x unsigned multiplication is performed using the technique demonstrated in **Example 3-6**.

Example 3-6 Multiplying Two Unsigned Fractional Values

```
      MOVE  X:FIRST,X0  ; Get first operand from memory
      ANDC  #$7FFF,X0   ; Force first operand to be positive
      MOVE  X:SECOND,Y0 ; Get second operand from memory
      MPYSU X0,Y0,A
      TSTW  X:FIRST     ; Perform final addition if MSB of
                        ; first operand was a “1”
      BGE   OVER
      ADD   Y0,A
OVER
      (ASR  A)          ; Optionally convert to integer result
```

3.3.9 Multi-precision Operations

3.3.9.1 Multi-precision Addition and Subtraction

Two instructions, ADC and SBC, assist in performing multi-precision addition (**Example 3-7**) and subtraction (**Example 3-8**), such as 64-bit or 96-bit operations.

Example 3-7 64-bit Addition

X:$1:X:$0:Y1:Y0 + A2:A1:A0:B1:B0 = A2:A1:A0:B1:B0
(B2 must contain only sign extension before addition begins,
i.e. bits 35-31 are all 1s or 0s)

```
MOVE  X:$21, B    ; correct sign extension
MOVE  X:$20, B0
ADD   Y,B         ; First 32-bit addition,
MOVE  X:$0,Y0     ; Get second 32-bit operand from memory
MOVE  X:$1,Y1
ADC   Y,A         ; Second 32-bit addition
```

Example 3-8 64-bit Subtraction

A2:A1:A0:B1:B0 - X:$1:X:$0:Y1:Y0 = A2:A1:A0:B1:B0
(B2 must contain only sign extension before addition begins,
i.e. bits 35-31 are all 1s or 0s)

```
MOVE  X:$21, B    ; correct sign extension
MOVE  X:$20, B0
SUB   Y,B         ; First 32-bit subtraction
MOVE  X:$0,Y0     ; Get second 32-bit operand from memory
MOVE  X:$1,Y1
SBC   Y,A         ; Second 32-bit subtraction
```

3.3.9.2 Multi-precision Multiplication

Two instructions are provided to assist with multi-precision multiplications. When these instructions are used, the multiplier accepts one signed two's-complement operand and one unsigned two's-complement operand. The instructions are:

- MPYSU—multiplication with one signed and one unsigned operand
- MACSU—multiply-accumulate with one signed and one unsigned operand

The use of these instructions in multi-precision multiplications is demonstrated in **Figure 3-11** and **Figure 3-12** on page 3-30, with corresponding examples shown in **Example 3-9** through **Example 3-11** on page 3-31.

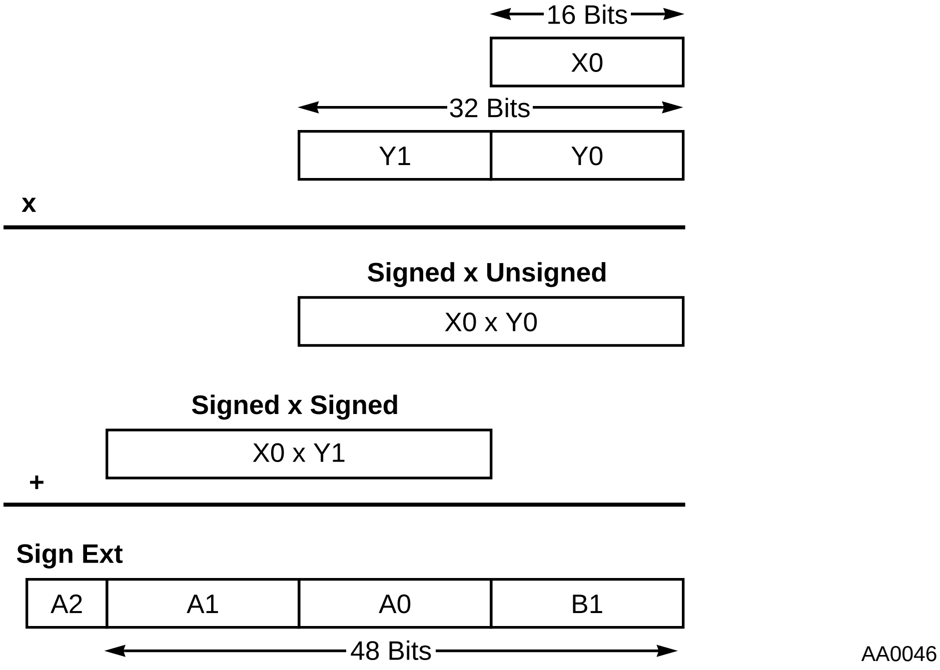


Figure 3-11 Single- Times Double-precision Signed Multiplication

Example 3-9 Fractional Single- Times Double-precision Value—Both Signed

(5 Icyc, 5 Instruction Words)

```
      MPYSU  X0,Y0,A     ; Single Precision times Lower Portion
      MOVE   A0,B

      MOVE   A1,A0       ; 16-bit Arithmetic Right Shift
      MOVE   A2,A1       ; (note that A2 contains only sign
                         ; extension)

      MAC    X0,Y1,A     ; Single Precision times Upper Portion
                         ; and added to Previous
```

Example 3-10 Integer Single- Times a Double-precision Value—Both Signed

(7 Icyc, 7 Instruction Words)

```
MPYSU X0,Y0,A     ; Single Precision times Lower Portion
MOVE  A0,B

MOVE  A1,A0       ; 16-bit Arithmetic Right Shift
MOVE  A2,A1       ; (note that A2 contains only sign
                  ; extension)

MAC   X0,Y1,A     ; Single Precision times Upper Portion
                  ; and added to Previous
ASR   A           ; Convert Result to Integer, A2
                  ; contains sign extension
ROR   B           ; (52-bit shift of A2:A1:A0:B1)
```

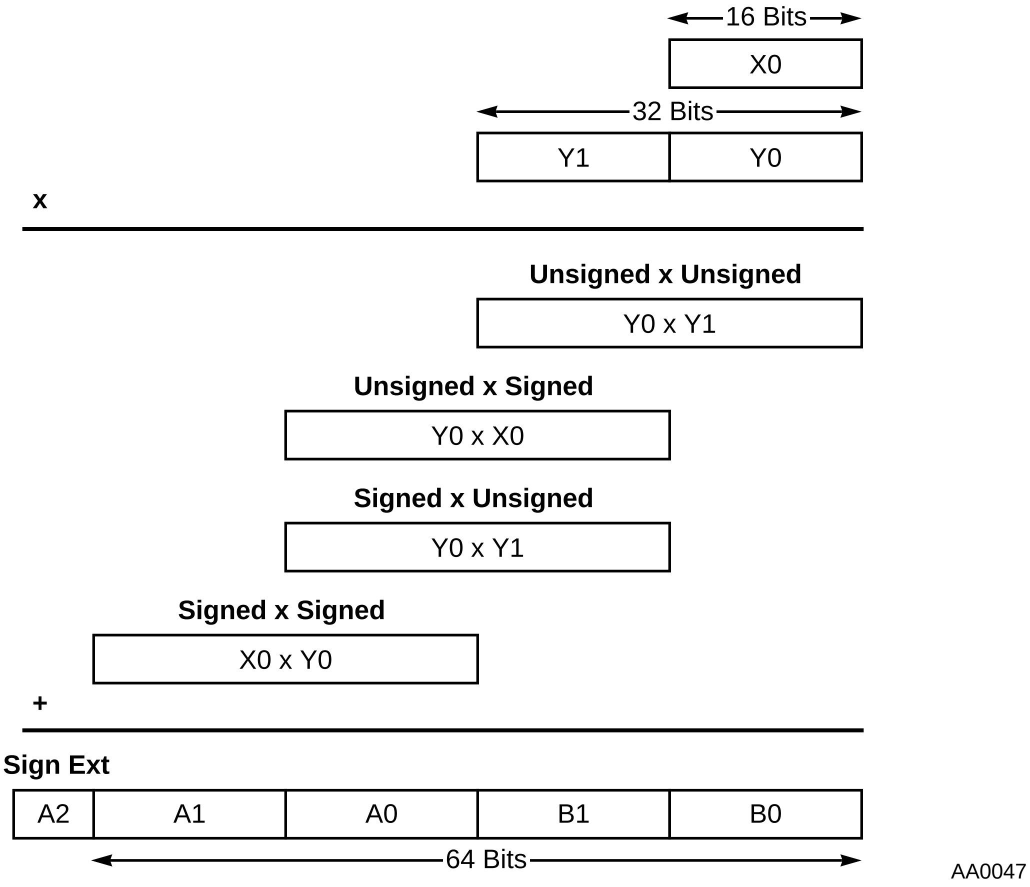


Figure 3-12 Double- Times Double-precision Signed Multiplication

Example 3-11 Multiplying Two Fractional Double-precision Values

X:OP1UPR:X:OP1LWR x X:OP2UPR:X:OP2LWR
(Both 32-Bit Operands are Signed)

```
;Unsigned x Unsigned Fractional Multiplication
      MOVE  X:OP1LWR,Y0 ; Get first operand from memory
      ANDC  #$7FFF,Y0   ; Force first operand to be positive
      MOVE  X:OP2LWR,Y1 ; Get second operand from memory
      MPYSU Y0,Y1,A
      TSTW  X:OP1LWR    ; Perform final addition if MSB of
                        ; first operand was a “1”
      BPL   OVER
      ADD   Y1,A
OVER

;Save Lowest 16 bits of Result
      MOVE  A0,X:RES0

;Unsigned x Signed Fractional Multiplication
      MOVE  A1,A0       ; 16-bit arithmetic right shift
      MOVE  A2,A1
      MOVE  X:OP1UPR,X0
      MACSU X0,Y0,A

;Signed x Unsigned Fractional Multiplication
      MOVE  Y0,Y1
      MOVE  X:OP2UPR,Y0
      MACSU Y0,Y1,A

;Save Next 16 bits of Result
      MOVE  A0,X:RES1

;Signed x Signed Fractional Multiplication
      MOVE  A1,X:TMP    ; 16-bit arithmetic right shift
      MOVE  A2,A
      MOVE  X:TMP,A0
      MAC   X0,Y0,A     ; Upper 32 bits of result now lies in A1:A0

;Save Upper 32 bits of Result
      MOVE  A0,X:RES2
      MOVE  A1,X:RES3
```

The corresponding algorithm for integer multiplication of 32-bit values would be the same as for fractional with the addition of a final arithmetic right shift of the 64-bit result.

3.4 CONDITION CODE GENERATION

The DSP core supports many different arithmetic instructions for both word and long word operations. The flexible nature of the instruction set means that condition codes must also be generated correctly for the different combinations allowed. There are three questions to consider when condition codes are generated for an instruction:

- Is the arithmetic operation's destination an accumulator or a 16-bit register or memory location?
- Does the instruction operate on the whole accumulator or only on the upper portion?
- Is the CC bit set in the program controller's OMR register?

The CC bit in the OMR register allows condition codes to be generated without examining the contents of the extension register. This sets up a computing environment where there is effectively no extension register because its contents are ignored. Typically, the extension register is most useful in DSP operations. For the case of general-purpose computing the CC bit is often set when the program is not performing DSP tasks, although it is possible to execute any instruction with the CC bit set or cleared except for instructions that use one of the unsigned condition codes (HS, LS, HI, LO).

This section covers different aspects of condition code generation for the different instructions and configurations on the DSP core. Note that the L, E, and U bits are computed the same regardless of the size of the destination or the value of the CC bit:

- L is set if overflow occurs or limiting occurs in a parallel move.
- E is set if the extension register is in use (i.e., if bits 35-31 are not all the same).
- U is set according to the standard definition of the U bit.

3.4.1 36-bit Destinations—CC bit cleared

Most arithmetic instructions generate a result for a 36-bit accumulator. When generating condition codes for this case and the CC bit is cleared, condition codes are generated using all 36 bits of the accumulator. Examples of instructions in this category are ADC, ADD, ASL, CMP, MAC, MACR, MPY, MPYR, NEG, NORM, RND, etc.

The condition codes for 36-bit destinations are computed as follows:

- N is set if bit 35 of the corresponding accumulator is set.
- Z is set if bits 35-0 of the corresponding accumulator are all cleared.
- V is set if overflow has occurred in the 36-bit result.
- C is set if a carry (borrow) has occurred out of bit 35 of the result.

3.4.2 36-bit Destinations—CC bit set

Most arithmetic instructions generate a result for a 36-bit accumulator. When generating condition codes for this case and the CC bit is set, condition codes are generated using only the 32 bits of the accumulator located in the MSP and LSP of the accumulator. There may be values in the extension registers, but the contents of the extension register are ignored. It is effectively the same as if there is no extension register. Examples of instructions in this category are ADC, ADD, ASL, CMP, MAC, MACR, MPY, MPYR, NEG, NORM, RND, etc.

The condition codes for 32-bit destinations (CC equals 1) are computed as follows:

- N is set if bit 31 of the corresponding accumulator is set.
- Z is set if bits 31-0 of the corresponding accumulator are all cleared.
- V is set if overflow has occurred in the 32-bit result.
- C is set if a carry (borrow) has occurred out of bit 31 of the result.

3.4.3 20-bit Destinations—CC bit cleared

Two arithmetic instructions generate a result for the upper two portions of an accumulator, the MSP and the extension register, leaving the LSP of the accumulator unchanged. When generating condition codes for this case and the CC bit is cleared, condition codes are generated using the 20 bits in the upper two portions of the accumulator. The two instructions in this category are DECW and INCW.

The condition codes for DECW and INCW (CC equals 0) are computed as follows:

- N is set if bit 35 of the corresponding accumulator is set.
- Z is set if bits 35-16 of the corresponding accumulator are all cleared.
- V is set if overflow has occurred in the 20-bit result.
- C is set if a carry (borrow) has occurred out of bit 35 of the result.

3.4.4 20-bit Destinations—CC bit set

Two arithmetic instructions generate a result for the upper two portions of an accumulator, the MSP and the extension register, leaving the LSP of the accumulator unchanged. When generating condition codes for this case and the CC bit is set, the bits in the extension register and the LSP of the accumulator are not used to calculate condition codes. The two instructions in this category are DECW and INCW.

The condition codes for 16-bit destinations (CC equals 1) are computed as follows:

- N is set if bit 31 of the corresponding accumulator is set.
- Z is set if bits 31-16 of the corresponding accumulator are all cleared.
- V is set if overflow has occurred in the 16-bit result.
- C is set if a carry (borrow) has occurred out of bit 31 of the result.

3.4.5 16-bit Destinations

Some arithmetic instructions can generate a result for a 36-bit accumulator or a 16-bit destination such as a register or memory location. When generating condition codes for a 16-bit destination, the CC bit is ignored and condition codes are generated using the 16 bits of the result. Instructions in this category are ADD, CMP, SUB, DECW, INCW, MAC, MACR, MPY, MPYR, ASR, and ASL.

The condition codes for 16-bit destinations are computed as follows:

- N is set if bit 15 of the result is set.
- Z is set if bits 15-0 of the result are all cleared.
- V is set if overflow has occurred in the 16-bit result.
- C is set if a carry (borrow) has occurred out of bit 15 of the result.

Other instructions only generate results for a 16-bit destination such as the logical instructions. When generating condition codes for this case, the CC bit is ignored and condition codes are generated using the 16 bits of the result. Instructions in this category are AND, EOR, LSL, LSR, NOT, OR, ROL, and ROR. The rules for condition code generation are presented for the cases where the destination is a 16-bit register or 16 bits of a 36-bit accumulator.

The condition codes for logical instructions with 16-bit registers as destinations are computed as follows:

- N is set if bit 15 of the corresponding register is set.
- Z is set if bits 15-0 of the corresponding register are all cleared.
- V is always cleared.
- C—(Computation dependent on instruction.)

The condition codes for logical instructions with 36-bit accumulators as destinations are computed as follows:

- N is set if bit 31 of the corresponding accumulator is set.
- Z is set if bits 31-16 of the corresponding accumulator are all cleared.
- V is always cleared.
- C—(Computation dependent on instruction.)

3.4.6 Special Instruction Types

Some instructions do not follow the above rules for condition code generation, but must be considered separately. Examples of instructions in this category are the logical and bit-field instructions (ANDC, EORC, NOTC, ORC, BFCHG, BFCLR, BFSET, BFTSTL, BFTSTH, BRCLR, and BRSET), the CLR instruction, the IMPY16 instruction, the multi-bit shifting instructions (ASLL, ASRR, LSLL, LSRR, ASRAC, and LSRAC), and the DIV instruction.

The bit-field instructions only affect the C and the L bit. The CLR instruction only generates condition codes when clearing an accumulator. The condition codes are not modified when clearing any other register. Some of the condition codes are not defined after executing the IMPY16 and multi-bit shifting instructions. The DIV instruction only affects a subset of all the condition codes. See **Condition Code Computation** on page -10 for details on the condition code computation for each of these instructions.

3.4.7 TST and TSTW Instructions

There are two instructions, TST and TSTW, which are useful for checking the value in a register or memory location.

The condition codes for the TST instruction (on a 36-bit accumulator) with CC equal to 0 are computed as follows:

- L is set if limiting occurs in a parallel move.
- E is set if the extension register is in use, i.e., if bits 35-31 are not all the same.
- U is set according to the standard definition of the U bit.
- N is set if bit 35 of the corresponding accumulator is set.
- Z is set if bits 35-0 of the corresponding accumulator are all cleared.
- V is always cleared.
- C is always cleared.

The condition codes for the TST instruction (on a 36-bit accumulator) with CC equal to 0 are computed as follows:

- L is set if limiting occurs in a parallel move.
- E is set if the extension register is in use, i.e., if bits 35-31 are not all the same.
- U is set according to the standard definition of the U bit.
- N is set if bit 31 of the corresponding accumulator is set.
- Z is set if bits 31-0 of the corresponding accumulator are all cleared.
- V is always cleared.
- C is always cleared.

The condition codes for the TSTW instruction (on a 16-bit value) are computed as follows:

- N is set if the MSB of the 16-bit value is set.
- Z is set if all 16 bits of the 16-bit value are cleared.
- V is always cleared.
- C is always cleared.

3.4.8 Unsigned Arithmetic

When performing arithmetic on unsigned operands, the condition codes used to compare two values differ from those used for signed arithmetic. See **Unsigned Arithmetic** on page 3-25 for a discussion of condition code usage for unsigned arithmetic.

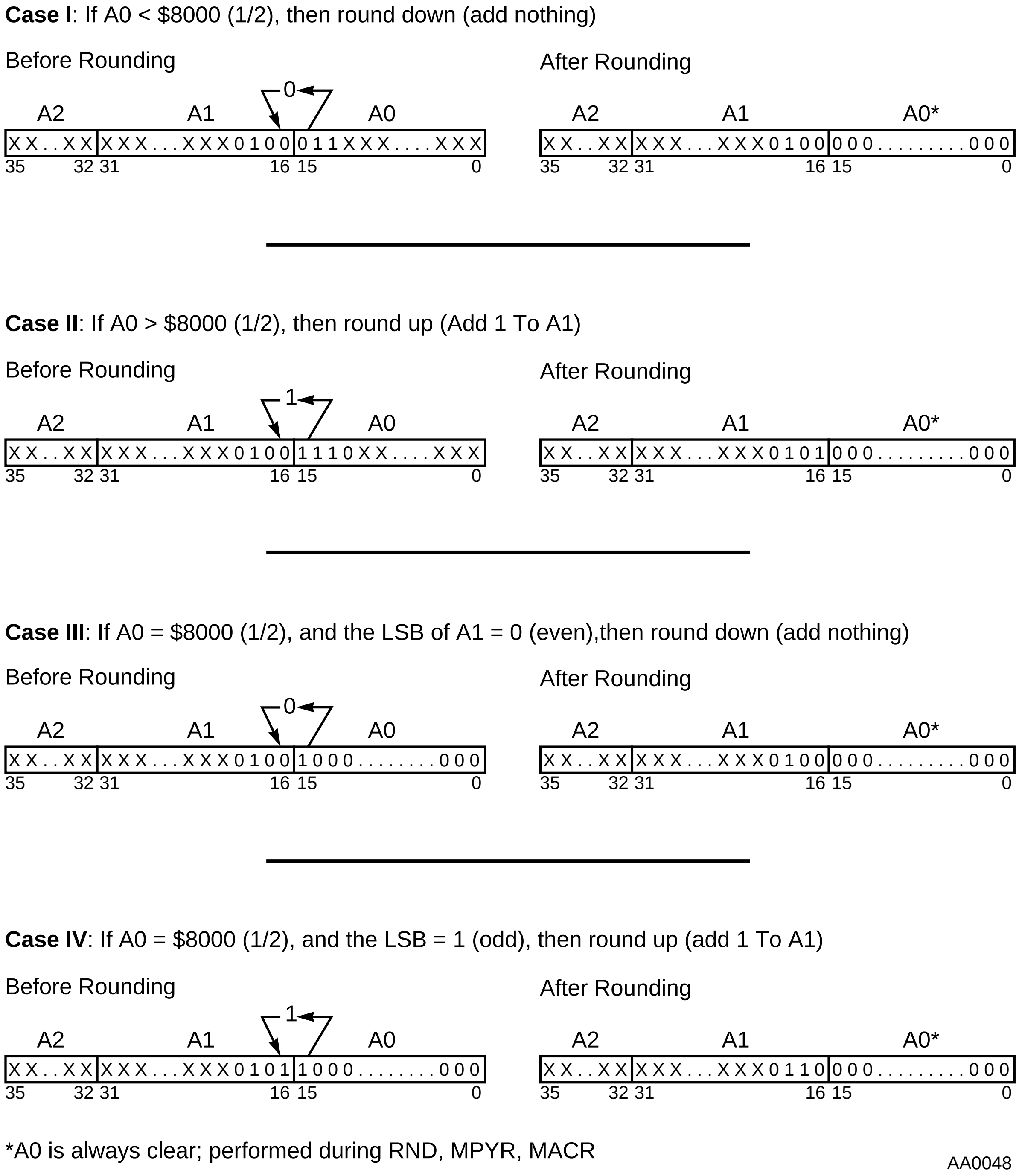
3.5 ROUNDING

The DSP56800 provides three instructions that can perform rounding—RND, MACR, and MPYR. The RND instruction simply rounds a value in the accumulator register specified by the instruction, whereas the MPYR or MACR instructions round the result calculated by the instruction in the MAC array. Each rounding instruction rounds the result to a single-precision value so the value can be stored in memory or in a 16-bit register. In addition, for instructions where the destination is one of the two accumulators, the LSP of the destination accumulator (A0 or B0) is set to $0000.

The DSP core implements two types of rounding: convergent rounding or two's complement rounding. For the DSP56800, the rounding point is between bits 16 and 15 of a 36-bit value; for the A accumulator, it is between the A1 register's LSB and the A0 register's MSB. The usual rounding method rounds up any value above one-half (i.e., LSP > $8000) and rounds down any value below one-half (i.e., LSP < $8000). The question arises as to which way the number one-half (LSP equals $8000) should be rounded. If it is always rounded one way, the results will eventually be biased in that direction. Convergent rounding solves the problem by rounding down if the number is even (bit 16 equals 0) and rounding up if the number is odd (bit 16 equals 1), whereas two's complement rounding always rounds this number up. The type of rounding is selected by the rounding bit (R) of the operating mode register (OMR) in the program controller.

3.5.1 Convergent Rounding

This is the default rounding mode. This rounding is also called “round-to-nearest even number.” For most values, this mode rounds identical to two’s-complement rounding; it only differs for the case where the least significant 16 bits is exactly $8000. For this case convergent rounding prevents any introduction of a bias by rounding down if the number is even (bit 16 equals 0) and rounding up if the rounding is odd (bit 16 equals 1). **Figure 3-13** shows the four possible cases for rounding a number in the A or B accumulator.



Case I: If A0 < $8000 (1/2), then round down (add nothing)

Case II: If A0 > $8000 (1/2), then round up (Add 1 To A1)

Case III: If A0 = $8000 (1/2), and the LSB of A1 = 0 (even),then round down (add nothing)

Case IV: If A0 = $8000 (1/2), and the LSB = 1 (odd), then round up (add 1 To A1)

*A0 is always clear; performed during RND, MPYR, MACR

AA0048

Figure 3-13 Convergent Rounding

A diagram of the convergent rounding implementation is shown in **Figure 3-14**.

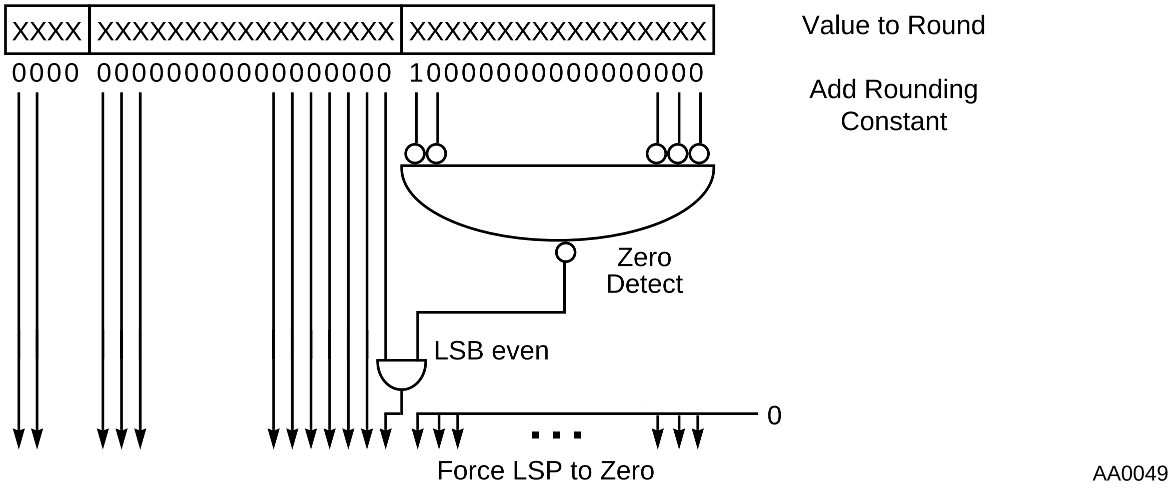
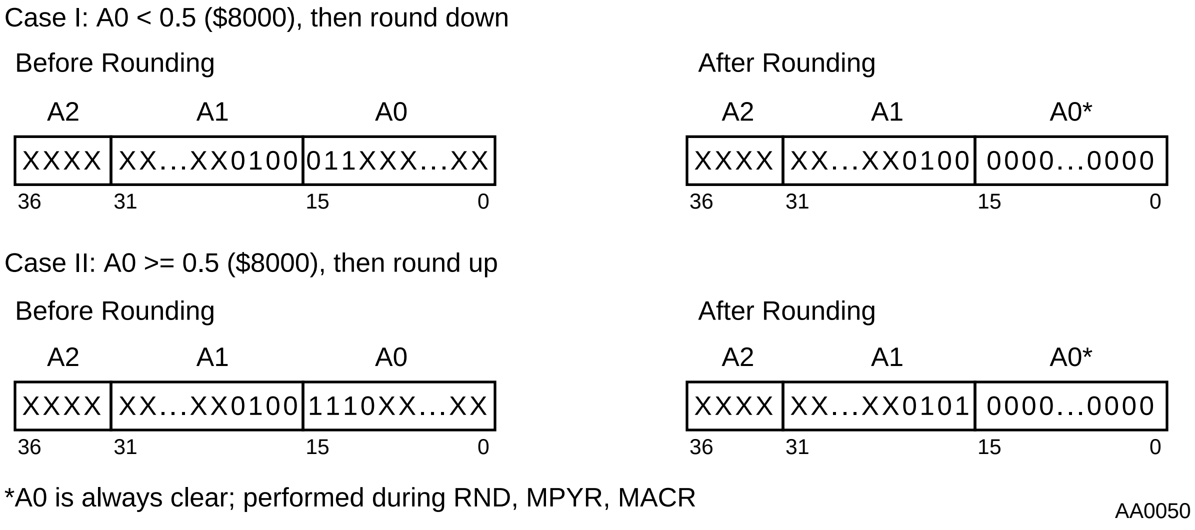


Figure 3-14 Convergent Rounding Implementation

3.5.2 Two's-complement Rounding

When this type of rounding is selected by setting the rounding bit in the OMR, 1 is added to the bit to the right of the rounding point (bit 15 of A0) before the bit truncation during a rounding operation. **Figure 3-15** shows the two possible cases.



Case I: A0 < 0.5 ($8000), then round down

	A2	A1	A0
Before Rounding	XXXX	XX...XX0100	011XXX...XX
After Rounding	XXXX	XX...XX0100	0000...0000

Case II: A0 >= 0.5 ($8000), then round up

	A2	A1	A0
Before Rounding	XXXX	XX...XX0100	1110XX...XX
After Rounding	XXXX	XX...XX0101	0000...0000

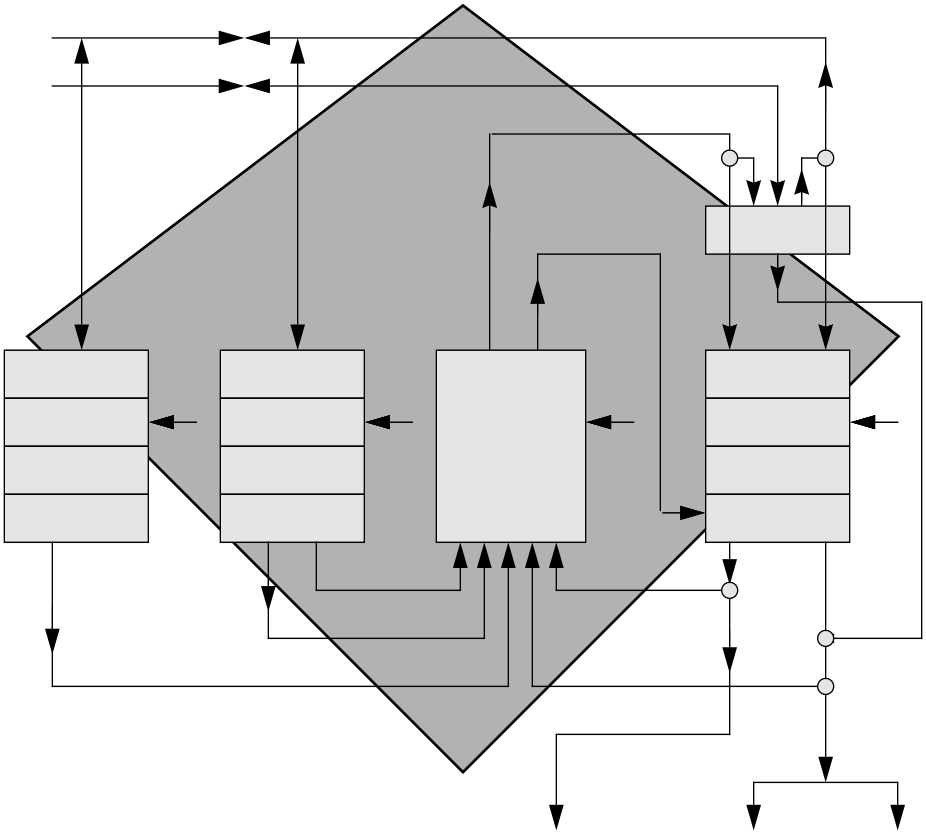
*A0 is always clear; performed during RND, MPYR, MACR

Figure 3-15 Two's-complement Rounding

Once the rounding bit has been programmed in the OMR register, there is a delay of one instruction cycle before the new rounding mode becomes active.

SECTION 4

ADDRESS GENERATION UNIT



4.1 INTRODUCTION

This section describes the architecture and the operation of the address generation unit (AGU). The address generation unit is the block where all address calculations are performed. It contains two arithmetic units—a modulo arithmetic unit for complex address calculations and an incrementer / decrementer for simple calculations. The modulo arithmetic unit can be used to calculate addresses in a modulo fashion, automatically wrapping around when necessary. A set of pointer registers, special-purpose registers, and multiple buses within the unit allow up to two address updates or a memory transfer to or from the AGU in a single cycle.

The capabilities of the address generation unit include the following operations:

- Provide one address to X data memory on the XAB1 bus
- Either pre-update or post-update an address, either before or after providing an XAB1 address
- Provide two addresses to X data memory on the XAB1 and XAB2 buses and post- update both addresses
- Provide one address to program memory for program memory data accesses and post-update the address
- Increment or decrement a counter during normalization operations
- Provide a conditional register move (Tcc instruction)

Note that in the cases where the address generation unit is generating one or two addresses to access X data memory, the program controller generates a second or third address used to concurrently fetch the next instruction.

The AGU provides many different addressing modes, which include the following:

- Indirect—no update
- Indirect—post-increment
- Indirect—post-decrement
- Indirect—post-updated by a register
- Indirect—indexed by 16-bit offset
- Indirect—indexed by a 6-bit offset
- Indirect—indexed by a register
- Immediate data
- Short immediate data
- Absolute addressing
- Absolute short addressing
- Peripheral short addressing
- Register direct
- Inherent

This section covers the architecture and programming model of the address generation unit, its addressing modes, and a discussion of the linear and modulo arithmetic capabilities of this unit. It concludes with a discussion of support for bit reversed arithmetic for fast Fourier transformation (FFT) processing and of pipeline dependencies related to the address generation unit.

4.2 ARCHITECTURE AND PROGRAMMING MODEL

The major components of the address generation unit are listed below:

- Four address registers (R0-R3)
- A stack pointer register (SP)
- An offset register (N)
- A modifier register (M01)
- A modulo arithmetic unit
- An incrementer / decrementer unit

The AGU uses integer arithmetic to perform the effective address calculations necessary to address data operands in memory. The AGU also contains the registers used to generate the addresses. It implements linear and modulo arithmetic and operates in parallel with other chip resources to minimize address-generation overhead.

Two ALUs are present within the AGU: the modulo arithmetic unit and the incrementer / decrementer unit. The two arithmetic units can generate up to two 16-bit addresses and two address updates every instruction cycle: one for XAB1 and one for XAB2 for instructions performing two parallel memory reads. The AGU can directly address 65,536 locations on XAB1 and 524,288 locations on the PAB. The AGU can directly address up to 65,536 locations on XAB2, but can only generate addresses to on-chip memory. The two ALUs work with the data memory to access up to two locations and provide two operands to the data ALU in a single cycle. The primary operand is addressed with the XAB1; and the second operand is addressed with the XAB2. The data memory, in turn, places its data on the core global data bus (CGDB) and the second external data bus (XDB2), respectively (see **Figure 4-1**). See **Introduction to Moves and Parallel Moves** on page 6-3 for more discussion on parallel memory moves.

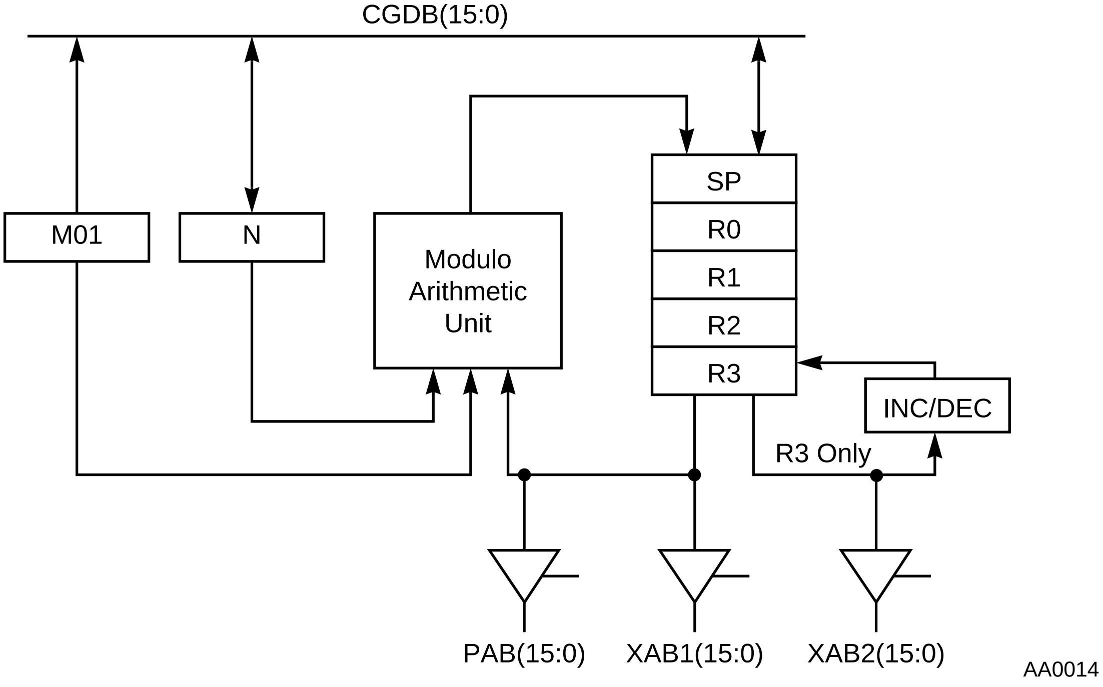


Figure 4-1 Address Generation Unit Block Diagram

All four address pointer registers and the SP are used in generating addresses in the register indirect addressing modes. The offset register can be used by all four address pointer registers and the SP, whereas the modulo register can be used by the R0 or by both the R0 and R1 pointer registers.

Whereas all the address pointer registers and the SP can be used in many addressing modes, there are some instructions that only work with a specific address pointer register. These cases are presented later in **Table 4-1** on page 4-11.

The address generation unit is connected to four major buses: CGDB, XAB1, XAB2, and PAB. The CGDB is used to read or write any of the address generation unit registers. The XAB1 and XAB2 provide a primary and secondary address respectively to the X data memory, and the PAB provides the address when accessing the program memory.

A block diagram of the address generation unit is shown in **Figure 4-1** and its corresponding programming model is shown in **Figure 4-2**. The blocks and registers are explained on the following page.

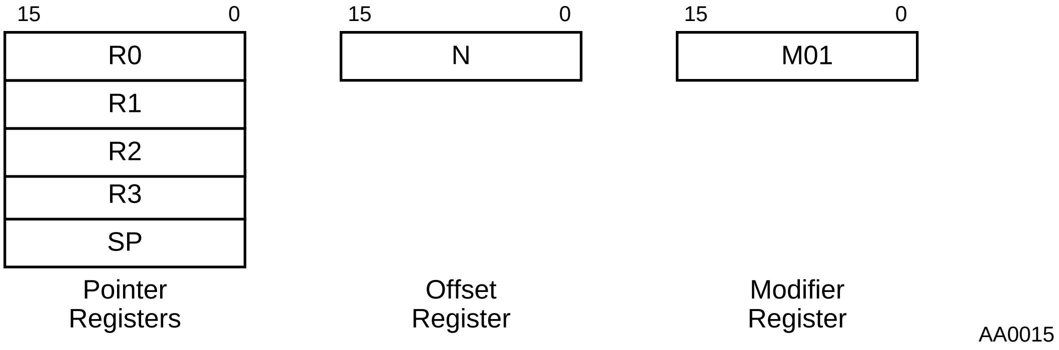


Figure 4-2 Address Generation Unit Programming Model

4.2.1 Address Registers (R0-R3)

The address register file consists of four 16-bit registers R0-R3 (Rn) that usually contain addresses used as pointers to memory. Each register may be read or written by the CGDB. High speed access to the XAB1, XAB2, and PAB buses is required to allow maximum access time for the internal and external X data memory and program memory. Each address register may be used as input for the modulo arithmetic unit for a register update calculation. Each register may be written by the output of the modulo arithmetic unit.

The R3 register may be used as input to a separate incrementer/decrementer unit for an independent register update calculation. This unit is used in the case of any instruction that performs two data memory reads in its parallel move field. For instructions where two reads are performed from the X data memory, the second read using the R3 pointer must always access on-chip memory.

Note: Due to pipelining, if an address register (Rn, SP, or M01) is changed with a MOVE or bit-field instruction, the new contents will not be available for use as a pointer until the second following instruction. If the SP is changed, no LEA or POP instructions are permitted until the second following instruction.

4.2.2 Stack Pointer Register (SP)

The stack pointer register (SP) is a single 16-bit register that is used implicitly in all PUSH instruction macros and POP instructions. The SP is used explicitly for memory references when used with the address register indirect modes. It is post-decremented on all POPs from the software stack. The SP register may be read or written by the CGDB.

Note: This register must be initialized explicitly by the programmer after coming out of reset.

Note: Due to pipelining, if an address register (Rn, SP, or M01) is changed with a MOVE or bit-field instruction, the new contents *will not* be available for use as a pointer until the second following instruction. If the SP is changed, no LEA or POP instructions are permitted until the second following instruction.

4.2.3 Offset Register (N)

The offset register (N) usually contains offset values used to update address pointers. This single register can be used to update or index with any of the address registers (R0-R3, SP). This offset register may be read or written by the CGDB. The offset register is used as input to the modulo arithmetic unit. It is often used for array indexing or indexing into a table, as discussed in **Array Indexes** on page 8-34.

Note: If the N address register is changed with a MOVE instruction, this register's contents *will* be available for use on the immediately following instruction. In this case the instruction that writes the N address register will be stretched one additional instruction cycle. This is true for the case when the N register is used by the immediately following instruction; if N is not used, then the instruction is not stretched an additional cycle. If the N address register is changed with a bit-field instruction, the new contents *will not* be available for use until the second following instruction.

4.2.4 Modifier Register (M01)

The modifier register (M01) specifies whether linear or modulo arithmetic is used when calculating a new address and may be read or written by the CGDB. This modifier register is automatically read when the R0 address register is used in an address calculation and can optionally be used also when R1 is used. This register has no effect on address calculations done with the R2, R3, or SP registers. It is used as input to the modulo arithmetic unit. This modifier register is preset during a processor reset to $FFFF (linear arithmetic).

Note: Due to pipelining, if an address register (Rn, SP, or M01) is changed with a MOVE or bit-field instruction, the new contents *will not* be available for use as a pointer until the second following instruction. If the SP is changed, no LEA or POP instructions are permitted until the following instruction.

4.2.5 Modulo Arithmetic Unit

The modulo arithmetic unit can update one address register or the SP during one instruction cycle. It is capable of performing linear and modulo arithmetic as described in **Linear and Modulo Addressing** on page 4-26. The contents of the modifier register specifies the type of arithmetic to be performed 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. Note that the modulo capability is only allowed for R0 or R1 updates; it is not allowed for R2, R3, or SP updates.

The modulo arithmetic unit first calculates the result of linear arithmetic (e.g., Rn+1, Rn-1, Rn+N) which is selected as the modulo arithmetic unit’s output for linear arithmetic. For modulo arithmetic, the modulo arithmetic unit will perform the function (Rn+N) modulo (M01+1) where N can be 1, -1, or the contents of the offset register N. If the modulo operation requires “wrap around” for modulo arithmetic, the summed output of the modulo adder will give the correct updated address register value; otherwise, if wrap around is not necessary, the linear arithmetic calculation gives the correct result.

4.2.6 Incrementer/Decrementer Unit

The incrementer / decrementer unit is used for address update calculations during dual data memory read instructions. It is used either to increment or decrement the R3 register. This adder performs only linear arithmetic; no modulo arithmetic is performed in this adder.

4.3 ADDRESSING MODES

The instruction set contains a full set of operand addressing modes, optimized for high performance signal processing as well as for efficient controller code. All address calculations are performed in the address generation unit to minimize execution time and loop overhead.

Addressing modes specify where an operand or operands for an instruction can be found (whether in a register or in memory) and provide the exact address of the operand(s). An effective address in an instruction will specify an addressing mode (i.e., where the operands can be found), and for some addressing modes the effective address will further specify an address register that points to a location in memory, how the address is calculated, and how the register is updated.

The addressing modes are grouped into three categories:

- Register direct—directly references the registers on the chip
- Address register indirect—uses an address register as a pointer to reference a location in memory
- Special—includes direct addressing, extended addressing, and immediate data

These addressing modes are described below and summarized in **Table 4-1** on page 4-11.

4.3.1 Address Register Direct Modes

The register direct addressing modes specify that the operand is in one (or more) of the nine data ALU registers, seven address registers, or four control registers.

4.3.1.1 Data or Control Register Direct

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

4.3.1.2 Address Register Direct

The operand is in one of the seven address registers (R0-R3, N, M01, or SP) specified by an effective address in the instruction. This reference is classified as a register reference.

Note: Due to pipelining, if an address register (Rn, SP, or M01) is changed with a MOVE or bit-field instruction, the new contents *will not* be available for use as a pointer until the second following instruction. If the SP is changed, no LEA or POP instructions are permitted until the second following instruction.

Note: If the N address register is changed with a MOVE instruction, this register’s contents *will* be available for use on the immediately following instruction. In this case the instruction that writes the N address register will be stretched one additional instruction cycle. This is true for the case when the N register is used by the immediately following instruction; if N is not used, then the instruction is not stretched an additional cycle. If the N address register is changed with a bit-field instruction, the new contents *will not* be available for use until the second following instruction.

4.3.2 Address Register Indirect Modes

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. The effective address in the instruction specifies the address register Rn or SP and the address calculation to be performed. These addressing modes specify that the operand(s) is (are) in memory and provide the specific address(es) of the operand(s). 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.

Address register indirect modes may require an offset and a modifier register for use in address calculations. The address register (Rn or SP) is used as the address register, the shared offset register is used to specify an optional offset from this pointer, and the modifier register is used to specify the type of arithmetic performed.

Some addressing modes are only available with certain address registers (Rn). For example, although all address registers support the “indexed by long displacement” addressing mode, only the R2 address register supports the “indexed by short displacement” addressing mode. For instructions where two reads are performed from the X data memory, the second read using the R3 pointer must always be from on-chip memory. The addressed register sets are summarized in **Table 4-1**.

Table 4-1 Address Register Indirect Addressing Modes Available

Register Set	Arithmetic Types	Addressing Modes Allowed	Notes
R0 / M01 / N	Linear / Modulo	(R0) (R0)+ (R0)- (R0)+N (R0+N) (R0+xxxx)	R0 *always* uses the M01 register to specify modulo or linear arithmetic. R0 can optionally be used as a source register for the Tcc instruction. R0 is the only register allowed as a counter for the NORM instruction.
R1 / M01 / N	Linear / Modulo	(R1) (R1)+ (R1)- (R1)+N (R1+N) (R1+xxxx)	R1 *optionally* uses the M01 register to specify modulo or linear arithmetic. R1 can optionally be used as a destination register for the Tcc instruction.
R2 / N	Linear	(R2) (R2)+ (R2)- (R2)+N (R2+N) (R2+xx) (R2+xxxx)	R2 supports a one-word indexed addressing mode. R2 is not allowed as either pointer for instructions that perform two reads from X data memory. No modulo arithmetic is allowed.
R3 / N	Linear	(R3) (R3)+ (R3)- (R3)+N (R3+N) (R3+xxxx)	R3 provides a second address for instructions with two reads from data memory. This second address can only access internal memory. It can also be used for instructions that perform one access to data memory. No modulo arithmetic is allowed.
SP / N	Linear	(SP) (SP)- (SP)+ (SP)+N (SP+N) (SP-xx) (SP+xxxx)	The SP supports a one word indexed addressing mode, that is useful for accessing local variables and passed parameters. No modulo arithmetic is allowed.

The type of arithmetic to be performed is not encoded in the instruction, but it is specified by the address modifier register (M01 for the DSP56800 core). It indicates whether linear or modulo arithmetic is performed when doing address calculations. In the case where there is not a modifier register for a particular register set (R2 or R3), linear addressing is always performed. For address calculations using R0, the modifier register is always used; for calculations using R1, the modifier register is optionally used.

Each address register indirect addressing mode is illustrated on the following pages.

4.3.2.1 No Update, (Rn), (SP)

The address of the operand is in the address register Rn or SP. The contents of the Rn register are unchanged. The M01 and N registers are ignored. This reference is classified as a memory reference (see **Figure 4-3**).

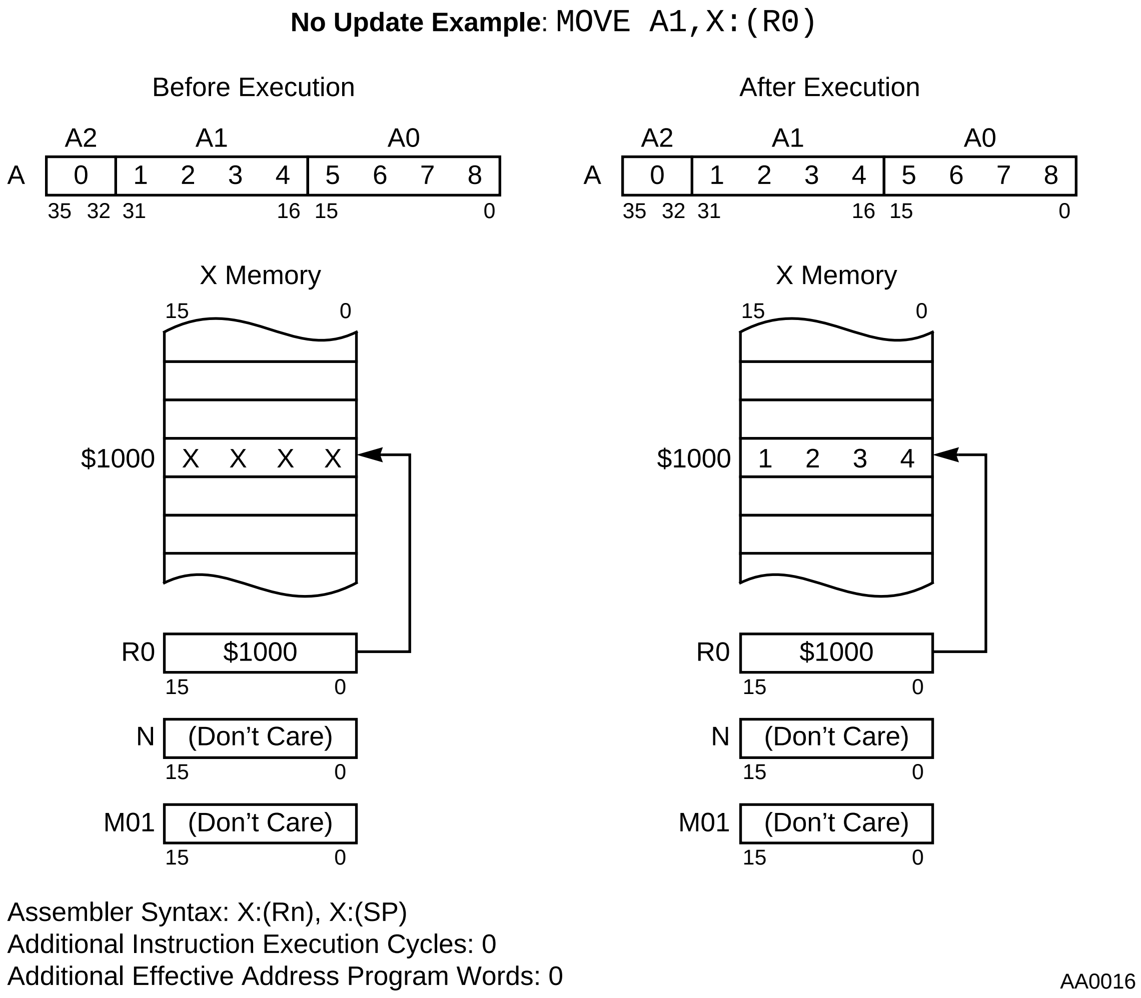
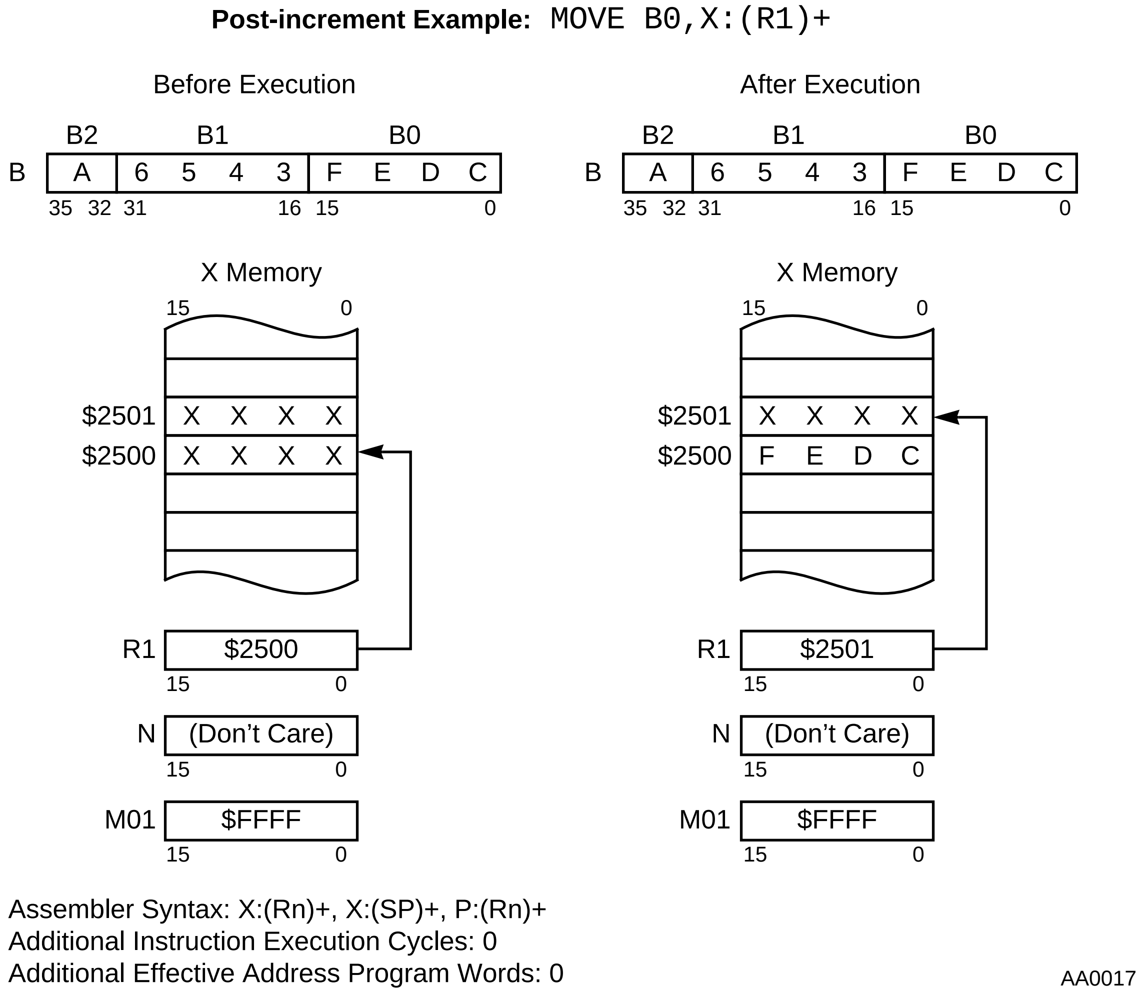

No Update Example: MOVE A1,X:(R0)
Before Execution
After Execution
A2
A1
A0
A
0
1 2 3 4
5 6 7 8
35 32 31 16 15 0
X Memory
15 0
$1000
X X X X
1 2 3 4
R0
$1000
N
(Don't Care)
M01
(Don't Care)
Assembler Syntax: X:(Rn), X:(SP)
Additional Instruction Execution Cycles: 0
Additional Effective Address Program Words: 0
AA0016

Figure 4-3 Address Register Indirect: No Update

4.3.2.2 Post-increment by 1, (Rn)+, (SP)+

The address of the operand is in the address register Rn or SP. After the operand address is used, it is incremented by one and stored in the same address register. The type of arithmetic (linear or modulo) used to increment Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. The N register is ignored. This reference is classified as a memory reference (see **Figure 4-4**).



Assembler Syntax: X:(Rn)+, X:(SP)+, P:(Rn)+
Additional Instruction Execution Cycles: 0
Additional Effective Address Program Words: 0

Figure 4-4 Address Register Indirect: Post-increment

4.3.2.3 Post-decrement by 1, (Rn)-, (SP)-

The address of the operand is in the address register Rn or SP. After the operand address is used, it is decremented by one and stored in the same address register. The type of arithmetic (linear or modulo) used to increment Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. The N register is ignored. This reference is classified as a memory reference (see **Figure 4-5**).

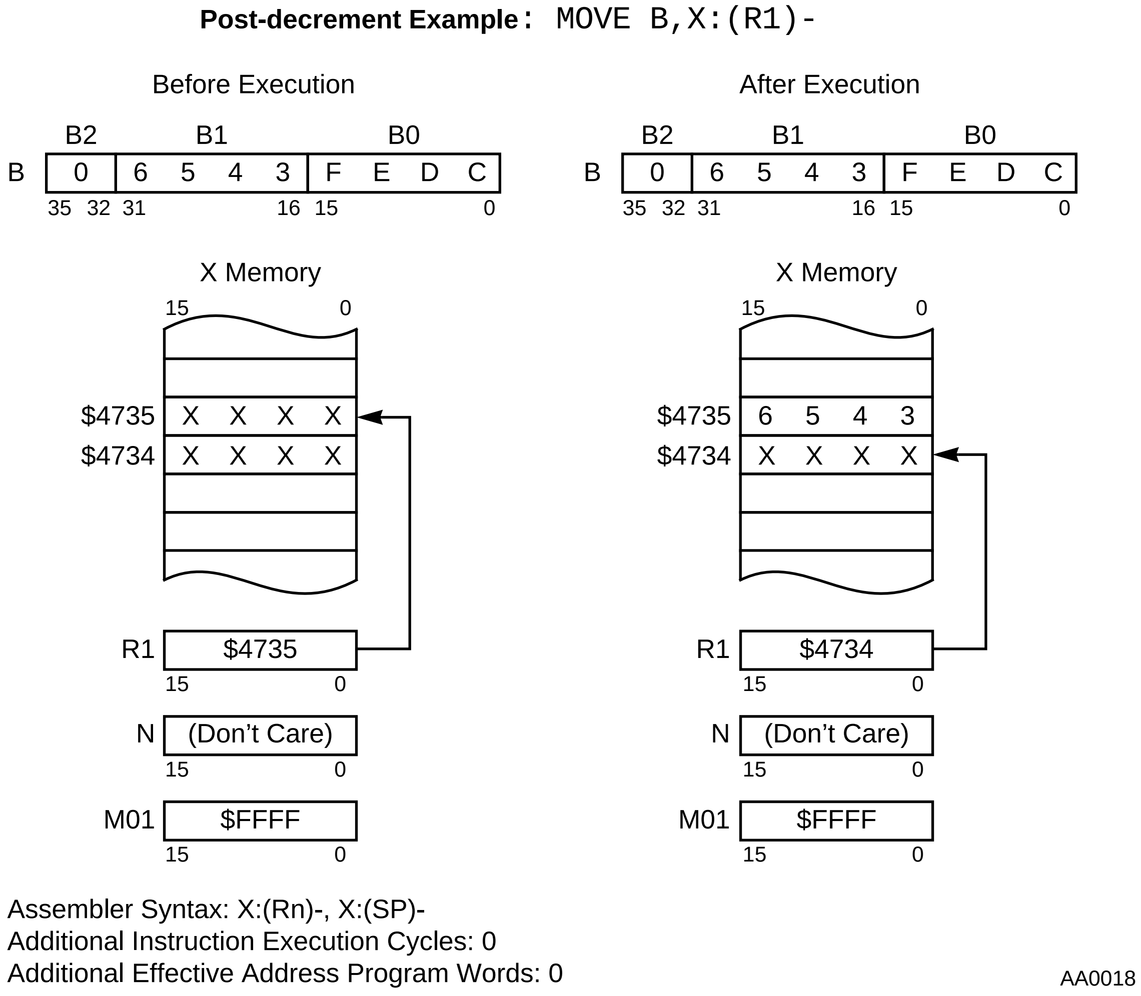
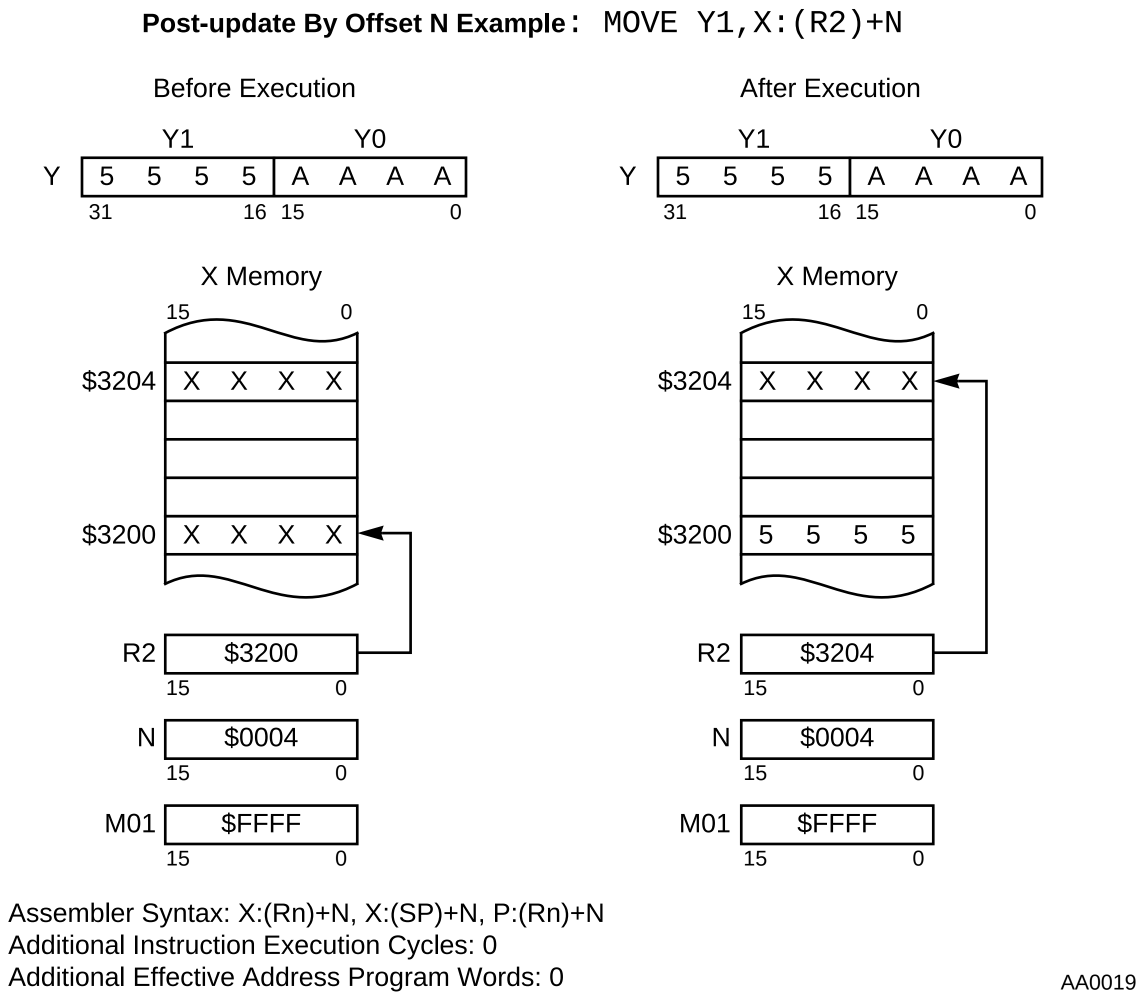

Post-decrement Example: MOVE B,X:(R1)-
Before Execution
After Execution
B2
B1
B0
B
0
6 5 4 3
F E D C
35 32 31 16 15 0
X Memory
15 0
$4735
$4734
X X X X
X X X X
6 5 4 3
R1
$4735
$4734
N
(Don’t Care)
M01
$FFFF
Assembler Syntax: X:(Rn)-, X:(SP)-
Additional Instruction Execution Cycles: 0
Additional Effective Address Program Words: 0
AA0018

Figure 4-5 Address Register Indirect: Post-decrement

4.3.2.4 Post-update by Offset N, (Rn)+N, (SP)+N

The address of the operand is in the address register Rn or SP. After the operand address is used, the contents of the N register are added to Rn and stored in the same address register. The content of N is treated as a two's-complement signed number. The contents of the N register are unchanged. The type of arithmetic (linear or modulo) used to update Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. This reference is classified as a memory reference (see **Figure 4-6**).



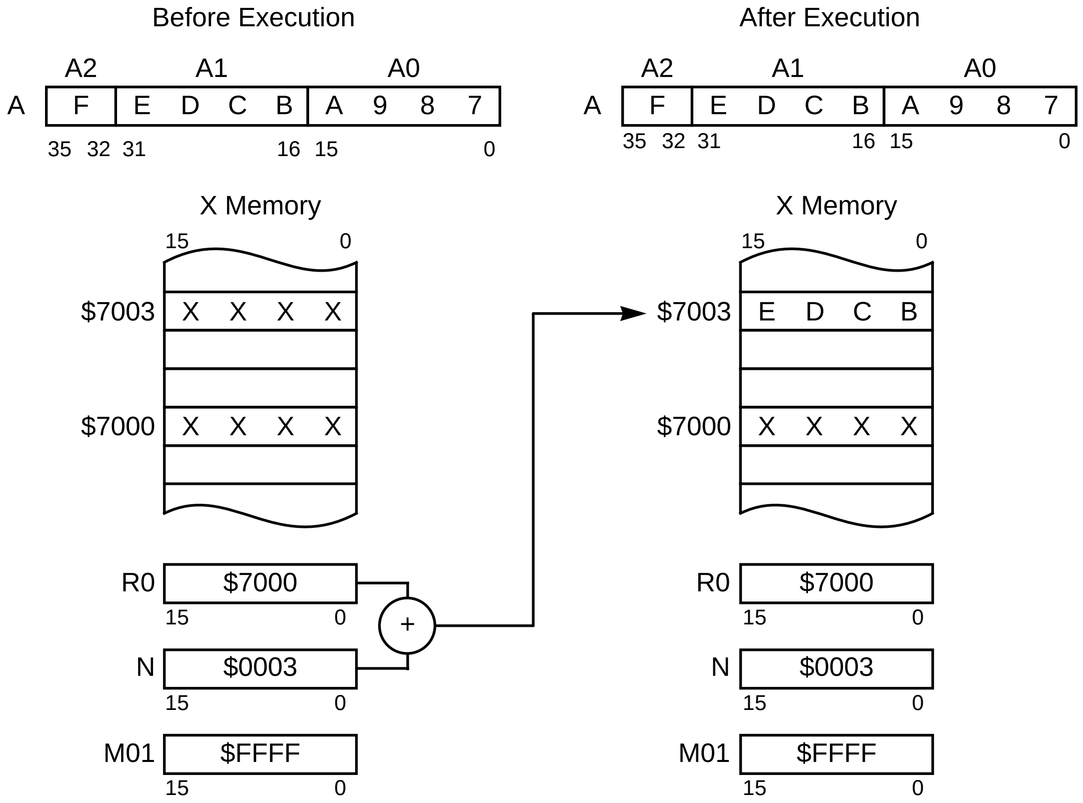
Assembler Syntax: X:(Rn)+N, X:(SP)+N, P:(Rn)+N
Additional Instruction Execution Cycles: 0
Additional Effective Address Program Words: 0

Figure 4-6 Address Register Indirect: Post-update by Offset N

4.3.2.5 Index by Offset N, (Rn+N), (SP+N)

The address of the operand is the sum of the contents of the address register Rn or SP and the contents of the address offset register N. This addition occurs before the operand can be accessed and, therefore, inserts an extra instruction cycle. The content of N is treated as a two's-complement signed number. The contents of the Rn and N registers are unchanged by this addressing mode. The type of arithmetic (linear or modulo) used to add N to Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. This reference is classified as a memory reference (see **Figure 4-7**).

Indexed By Offset N Example: `MOVE A1,X:(R0+N)`

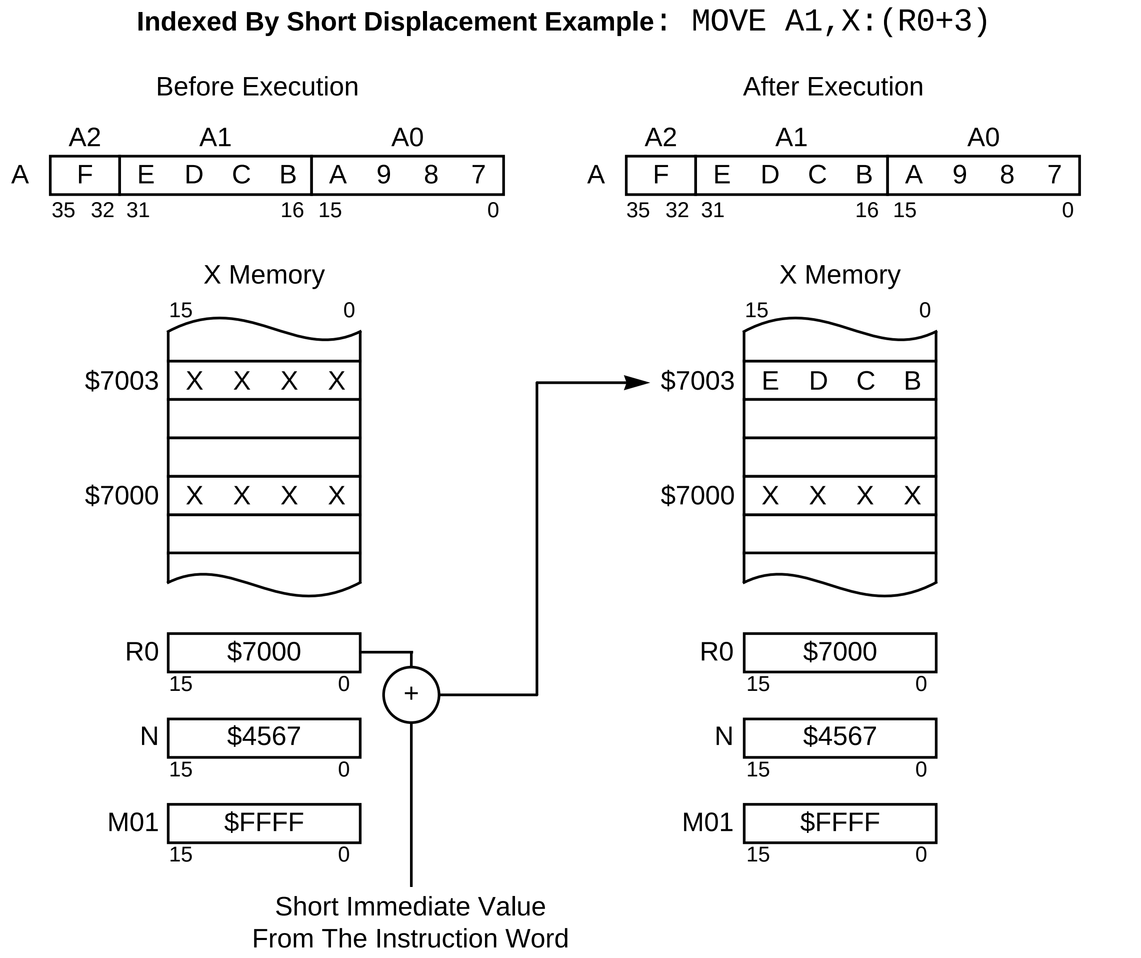


Assembler Syntax: X:(Rn+N), X:(SP+N)
Additional Instruction Execution Cycles: 1
Additional Effective Address Program Words: 0

Figure 4-7 Address Register Indirect: Indexed by Offset N

4.3.2.6 Index By Short Displacement, (SP-xx), (R2+xx)

This addressing mode contains the 6-bit short immediate index within the instruction word. This field is always one-extended to form a negative offset when the SP register is used and is always zero-extended to form a positive offset when the R2 register is used. The type of arithmetic used to add the short displacement to R2 or SP is always linear; modulo arithmetic is not allowed. This addressing mode requires an extra instruction cycle. This reference is classified as an X memory reference (see **Figure 4-8**).



Assembler Syntax: X:(Rn+xx), X:(SP-xx)
Additional Instruction Execution Cycles: 1
Additional Effective Address Program Words: 0

AA0021

Figure 4-8 Address Register Indirect: Indexed by Short Displacement

4.3.2.7 Index by Long Displacement, (Rn+xxxx), (SP+xxxx)

This addressing mode contains the 16-bit long immediate index within the instruction word. This second word is treated as a signed two’s-complement value. The type of arithmetic (linear or modulo) used to add the long displacement to Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. This addressing mode requires two extra instruction cycles. This addressing mode is available for MOVEC instructions. This reference is classified as an X memory reference (see **Figure 4-9**).

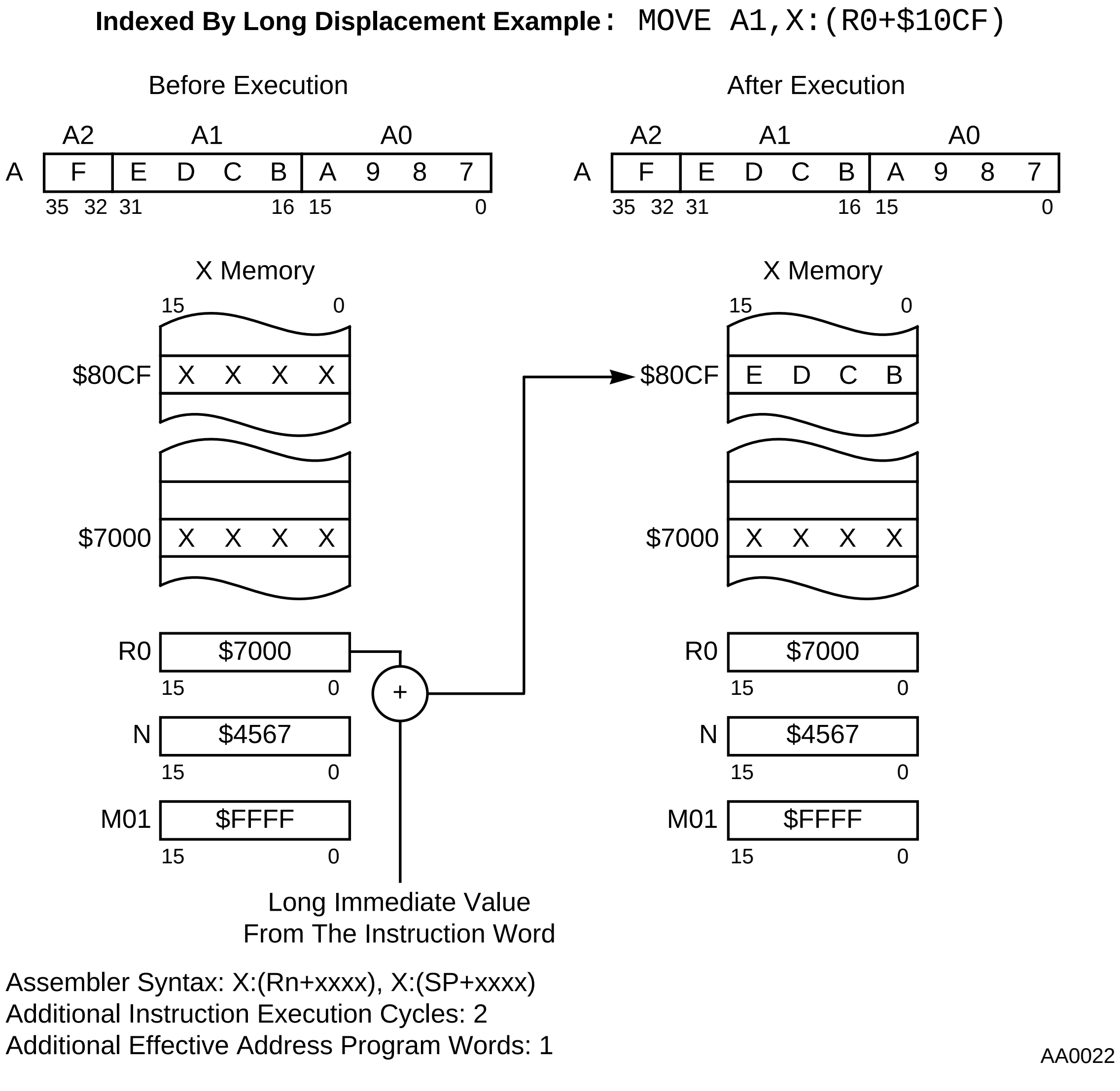

Indexed By Long Displacement Example: MOVE A1,X:(R0+$10CF)
Before Execution
After Execution
A2
A1
A0
A
F
E D C B
A 9 8 7
35 32 31 16 15 0
X Memory
15 0
$80CF
X X X X
E D C B
$7000
X X X X
R0
$7000
N
$4567
M01
$FFFF
+
Long Immediate Value
From The Instruction Word
Assembler Syntax: X:(Rn+xxxx), X:(SP+xxxx)
Additional Instruction Execution Cycles: 2
Additional Effective Address Program Words: 1
AA0022

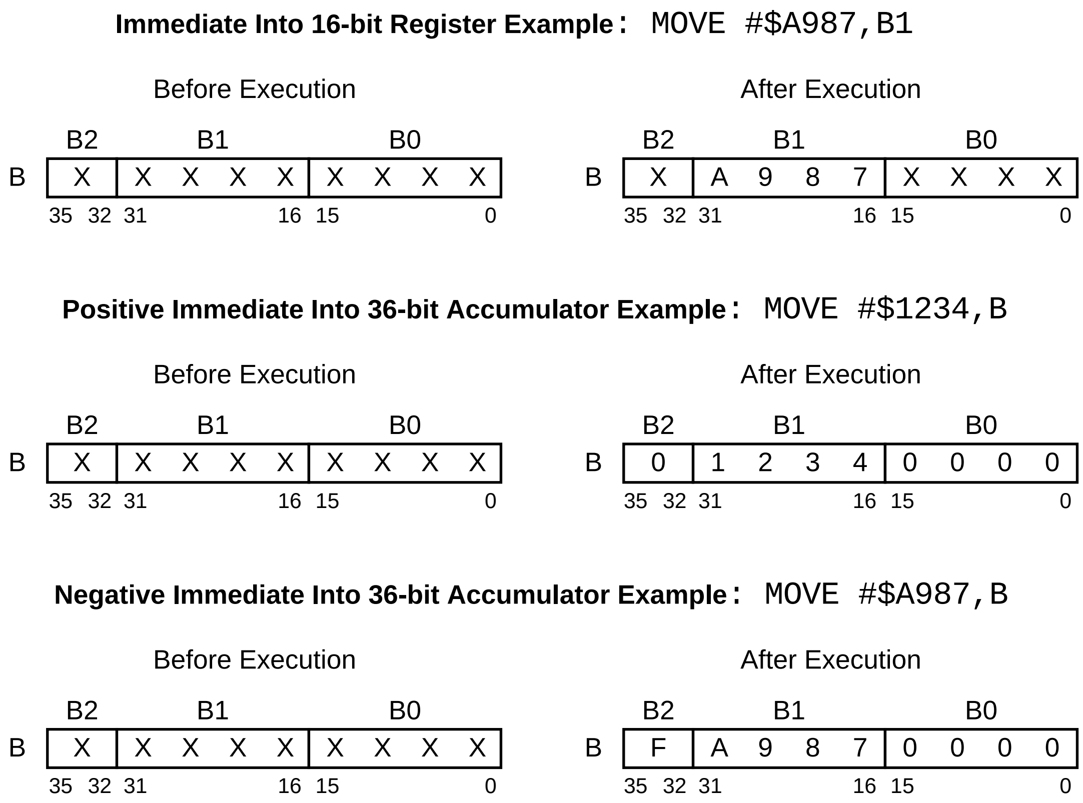
Figure 4-9 Address Register Indirect: Indexed by Long Displacement

4.3.3 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 directly in a field of the instruction, or they implicitly reference an operand. This category includes direct addressing, extended addressing, and immediate data.

4.3.3.1 Immediate Data, #xxxx

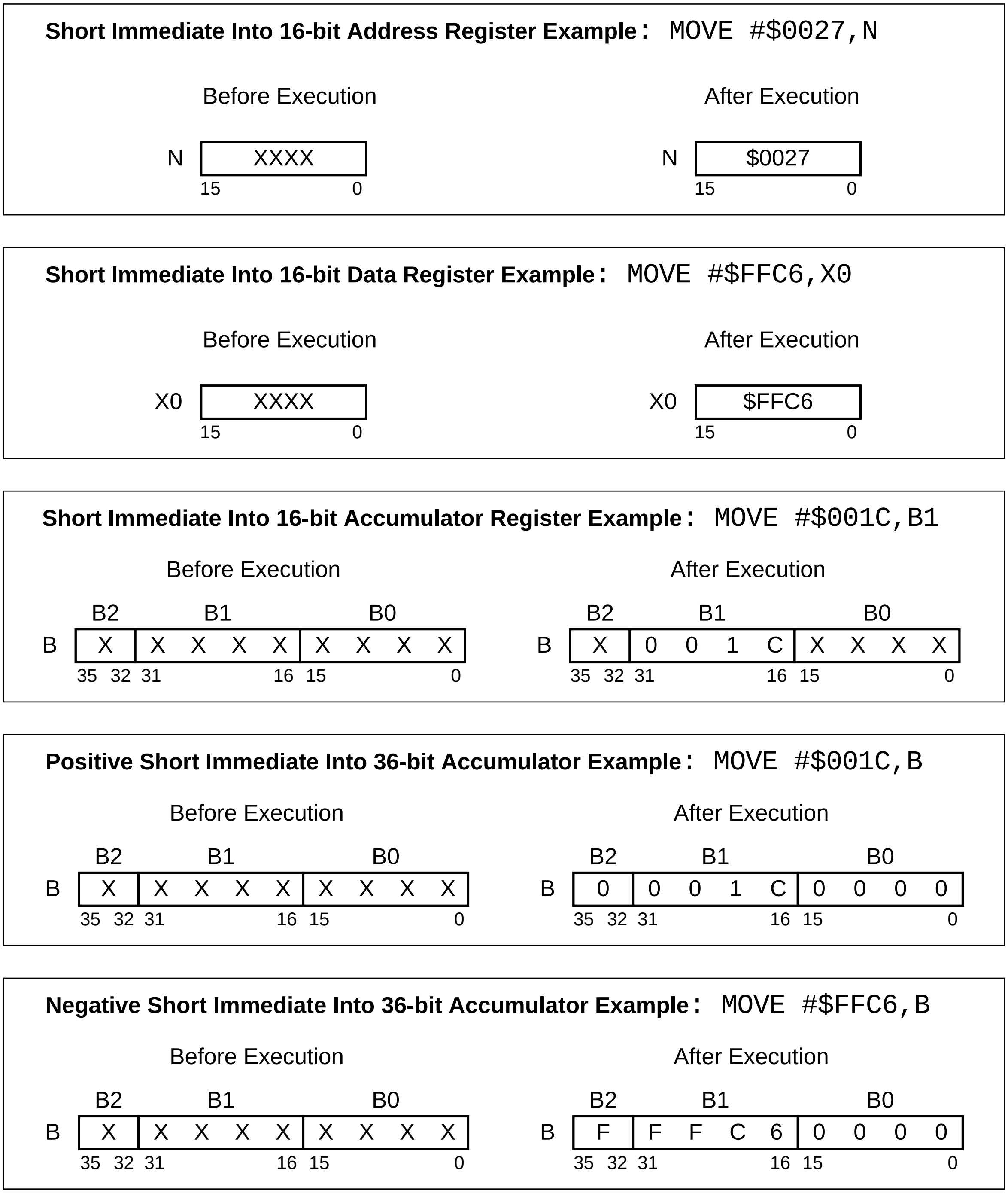
This addressing mode requires one word of instruction extension. The immediate data is a one word (16-bit) operand in the extension word of the instruction. This reference is classified as a program reference (see **Figure 4-10**).


Immediate Into 16-bit Register Example: MOVE #$A987,B1
Before Execution
After Execution
B2 B1 B0
B | X | X X X X | X X X X |
B | X | A 9 8 7 | X X X X |
35 32 31 16 15 0
Positive Immediate Into 36-bit Accumulator Example: MOVE #$1234,B
Before Execution
After Execution
B | X | X X X X | X X X X |
B | 0 | 1 2 3 4 | 0 0 0 0 |
Negative Immediate Into 36-bit Accumulator Example: MOVE #$A987,B
Before Execution
After Execution
B | X | X X X X | X X X X |
B | F | A 9 8 7 | 0 0 0 0 |

Assembler Syntax: #xxxx
Additional Instruction Execution Cycles: 1
Additional Effective Address Program Words: 1

AA0023

Figure 4-10 Special Addressing: Immediate Data



Assembler Syntax: #xx
Additional Instruction Execution Cycles: 0
Additional Effective Address Program Words: 0

AA0024

Figure 4-11 Special Addressing: Immediate Short Data

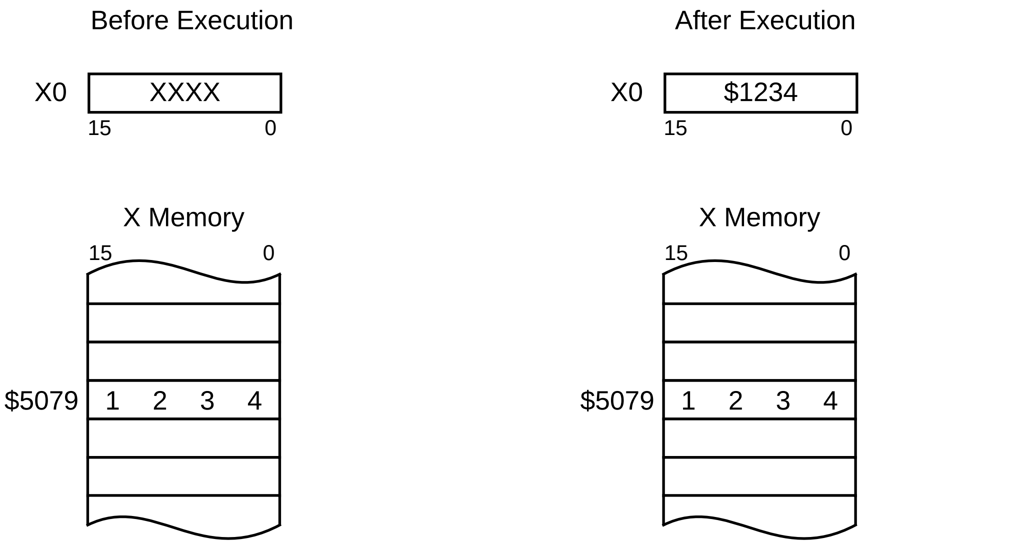
4.3.3.2 Immediate Short Data, #xx

The immediate short data operand is located in the instruction operation word. A 6-bit unsigned positive operand is used for DO and REP instructions and a 7-bit signed operand for the immediate move to an on-core register instruction. This reference is classified as a program reference (see **Figure 4-11**).

4.3.3.3 Absolute Address (Extended Addressing), xxxx

This addressing mode requires one word of instruction extension. The address of the operand is located in the extension word. No registers are used to form the address of the operand. Absolute address instructions are used with the bit manipulation and move instructions. This reference is classified as a memory reference and a program reference (see **Figure 4-12**).

Absolute Address Example: `MOVE X:$5079,X0`

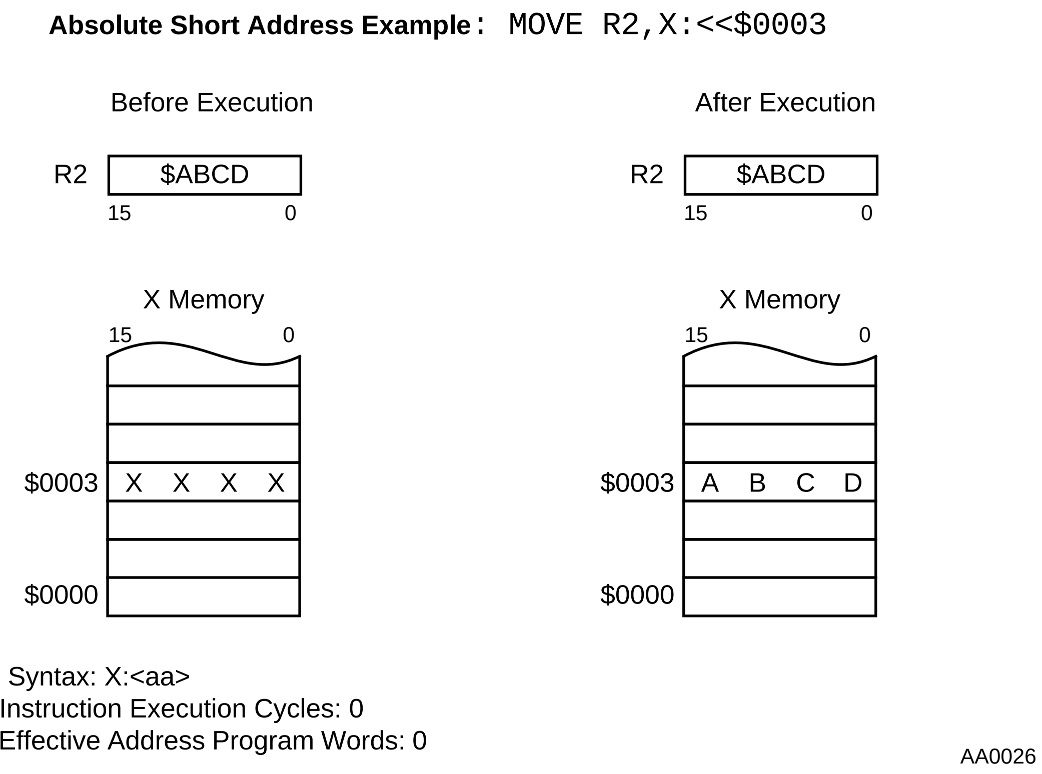


Assembler Syntax: X:xxxx
Additional Instruction Execution Cycles: 1
Additional Effective Address Program Words: 1

Figure 4-12 Special Addressing: Absolute Address

4.3.3.4 Absolute Short Address (Direct Addressing), <aa>

For the absolute short addressing mode, the address of the operand occupies six bits in the instruction operation word and is zero-extended. This allows direct access to the first 64 locations in X memory. No registers are used to form the address of the operand. Absolute short instructions are used with the bit-field manipulation and move instructions (see **Figure 4-13**).



Absolute Short Address Example: `MOVE R2,X:<<$0003`

Assembler Syntax: X:<aa>
Additional Instruction Execution Cycles: 0
Additional Effective Address Program Words: 0

Figure 4-13 Special Addressing: Absolute Short Address

4.3.3.5 I/O Short Address (Direct Addressing), <pp>

For the I/O short addressing mode the address of the operand occupies six bits in the instruction operation word and is one-extended. This allows direct access to the last sixty-four locations in X memory, which accesses the on-chip peripheral registers. No registers are used to form the address of the operand (see **Figure 4-14**).

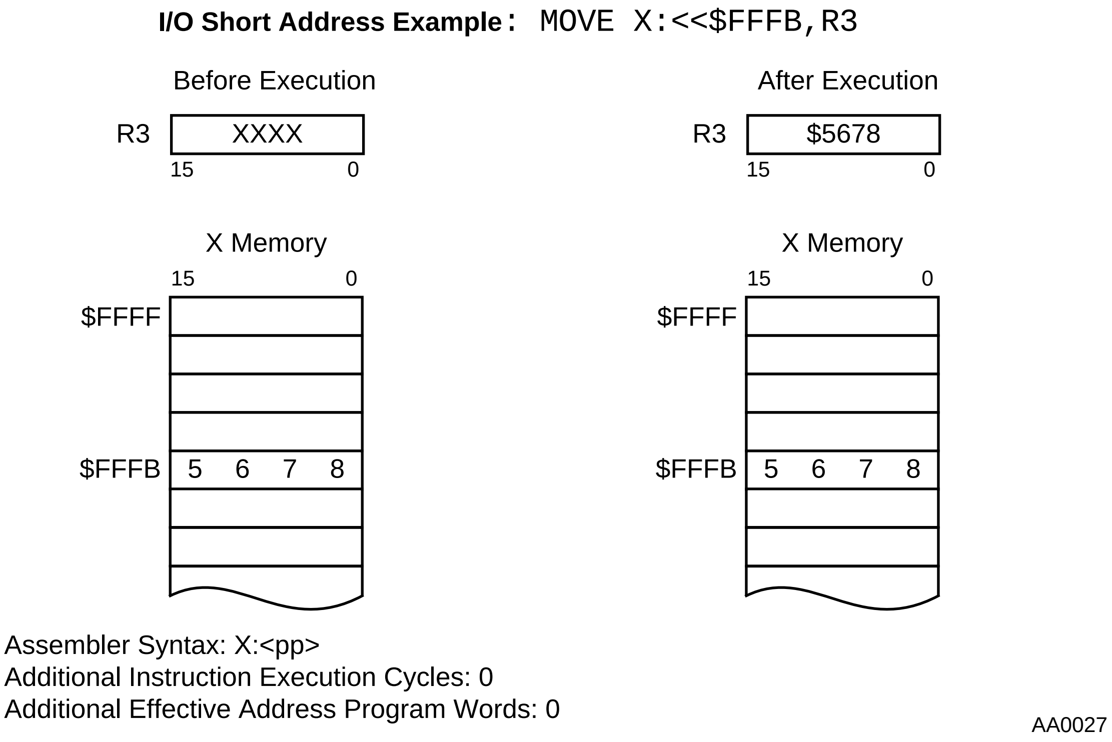


Figure 4-14 Special Addressing: I/O Short Address

4.3.3.6 Implicit Reference

Some instructions make implicit reference to the program counter (PC), software stack, hardware stack (HWS), loop address register (LA), loop counter (LC), or status register (SR). The registers implied and their use are defined by the individual instruction descriptions (see **Appendix A Instruction Set Details**).

4.3.4 Addressing Modes Summary

Table 4-2 contains a summary of the addressing modes discussed in the previous paragraphs.

Table 4-2 Addressing Mode Summary

Addressing Mode	Uses M01	Operand Reference							Assembler Syntax
		S[1]	C[2]	D[3]	A[4]	P[5]	X[6]	XX[7]	
Register Direct									
Data or control register	No		X	X					
Address register (Rn, SP)	No				X				Rn[8]
Address modifier register (M01)	No				X				M01
Address offset register (N)	No				X				N
Hardware stack (HWS)	No	X							HWS
Software stack	No						X		
Address Register Indirect									
No update	No						X		(Rn)[8]
Post-increment by 1	Yes[9]					X	X	X	(Rn)+[8]
Post-decrement by 1	Yes[9]						X		(Rn)-[8]
Post-update by offset N	Yes[9]					X	X	X	(Rn)+N[8]
Indexed by offset N	Yes[9]						X		(Rn+N)[8]
Indexed by short displacement	No						X		(R2+xx) or (SP-xx)
Indexed by long displacement	Yes[9]						X		(Rn+xxxx) or (SP+xxxx)
Special									
Immediate data	No					X			#xxxx
Immediate short data	No					X			#xx
Absolute address	No					X	X		xxxx
Absolute short address	No						X		<aa>
I/O short address	No						X		<pp>
Implicit	No	X	X			X	X		

Note:
1. Hardware stack reference
2. Program controller register reference
3. Data ALU register reference
4. Address generation unit register reference
5. Program memory reference
6. X memory reference
7. Dual X memory read
8. Any of the four address registers, R0-R3, as well as the SP are allowed here.
9. M01 modifier can be used only on the R0/N/M01 or R1/N/M01 register sets.

4.4 LINEAR AND MODULO ADDRESSING

When an arithmetic operation is performed in the address generation unit, it can be performed using either linear or modulo arithmetic. Linear arithmetic is important for general purpose address computation, and modulo arithmetic allows the creation of data structures in memory such as first-in-first-out queues (FIFOs), delay lines, circular buffers, and stacks. Data is manipulated by updating address registers (pointers) rather than moving large blocks of data. The DSP56800 core address generation unit supports both types of arithmetic for the address register indirect modes:

- (Rn)+
- (Rn)-
- (Rn)+N
- (Rn+N)
- (Rn+xxxx)

The contents of the address modifier register defines the type of address arithmetic (linear or modulo) to be performed for addressing mode calculations; and for the case of modulo arithmetic the contents of M01 also specify the modulus. Only address registers R0 and R1 have a modifier register associated with them, since linear arithmetic is always performed with the R2, R3, and SP address registers.

4.4.1 Linear Arithmetic

Linear arithmetic is the case where address arithmetic is performed using normal 16-bit two’s-complement linear arithmetic. A 16-bit offset register N or immediate data (+1, -1, or a displacement value) may be used in the address calculations. Addresses are normally considered unsigned; offsets and data are normally considered signed.

Address arithmetic using the R0 pointer uses linear arithmetic when the value in the modifier register (M01) is $FFFF. Address arithmetic using the R1 pointer uses linear arithmetic when the value in the modifier register is $FFFF or falls in the range $0000 through $3FFF. Memory accesses using the R2 and R3 pointers are always performed with linear arithmetic. **Programming the Modifier Register** on page 4-33 shows how to correctly program the M01 register.

4.4.2 Modulo Arithmetic

Modulo arithmetic is used when it is necessary to set up and step through a circular buffer in memory. Modulo arithmetic is similar to linear arithmetic, but if the result of an effective address calculation would be larger than the last address in a buffer, then an automatic wraparound is performed in the calculation. Similarly, it occurs in the case where the result of an effective address calculation calculates an address that would be smaller than the first address in a buffer, automatic wraparound is performed in the address calculation.

The address arithmetic is performed modulo M, where M is permitted to range from 2 to +16,384. 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, thus allowing a modulo size range from 2 to 16,384. The lower boundary (base address) value must have zeroes in the k LSBs, where $2^k \geq M$, 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).

The modulo register can be used for modulo arithmetic with the R0 pointer, or with both R0 and R1. In the latter case the size of the modulo buffer is the same for both pointers, but because each pointer can point to a different area of memory, the pointers can work on separate buffers. Similarly, the two pointers can work on the same buffer, if desired. The modifier register is always applied to R0, but is only applied to R1 if programmed in that manner; otherwise, the address calculations with R1 are done with linear arithmetic. All address calculations with R2 and R3 are always done with linear arithmetic. **Programming the Modifier Register** on page 4-33 shows how to correctly program the M01 register.

When executing the (Rn)+ indirect addressing mode, if the address register pointer increments past the upper boundary of the buffer (base address + M01), it will wrap around to or past the lower boundary. Similarly, when executing the (Rn)- indirect addressing mode, if the address decrements past the lower boundary, it will wrap around to or past the upper boundary (see **Figure 4-15**).

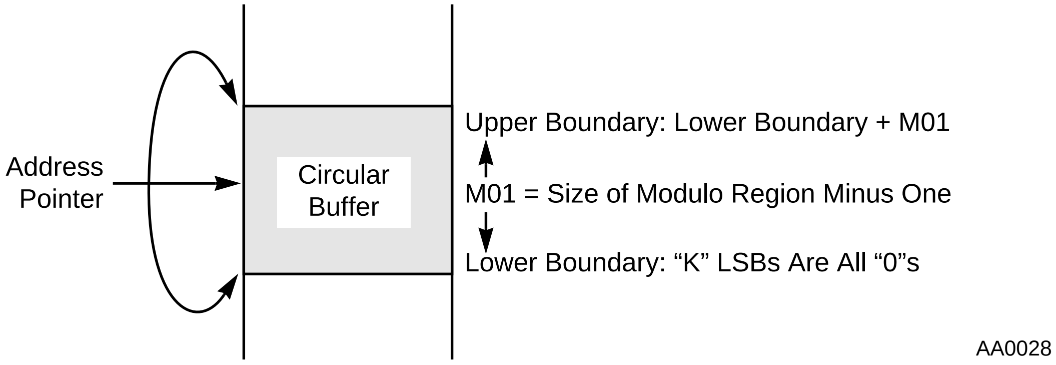

Address
Pointer
Circular
Buffer
Upper Boundary: Lower Boundary + M01
M01 = Size of Modulo Region Minus One
Lower Boundary: “K” LSBs Are All “0”s
AA0028

Figure 4-15 Circular Buffer

The steps below detail the process of setting up and using a circular buffer with 37 locations.

1. Choose the value for the M01 register:
   - The maximum allowed size of a circular buffer is 16384 locations.
   - If it is desired that modulo arithmetic is allowed only with the R0 address register:
     M01 = # locations - 1 = 37 - 1 = 36 = $0024
   - If it is desired that modulo arithmetic is allowed with both the R0 and R1 address registers:
     M01 = # locations - 1 + $8000 = 37 - 1 + 32768 = 32804 = $8024
   - If the M01 register is set to $8024, address arithmetic with both the R0 and R1 pointers is done for a circular buffer of 37 locations.
2. Find the nearest power of 2 greater than or equal to the circular buffer size:
   - $2^k \geq$ # locations
   - $2^k \geq 37$
   - $k \geq 6$
3. The lower boundary of the circular buffer must satisfy the following:
   - The “k” LSBs of the address of the lower boundary must all be 0’s.
   - Lower boundary = N x 2^k [where N and k are integers]
   - For this example:
     The 6 LSBs of the lower boundary are “0”s
     Lower boundary = (N x 2^k) = (N x 64)

- Examples of valid lower boundary addresses for this example are:
  0, 64, 128, 192, 256, …

4. The upper boundary of the circular buffer is:
   - Upper boundary = Lower boundary + # locations - 1
   - Upper boundary = (N x 2^k) + M01 [where N and k are integers]
   - For this example:
     Upper boundary = (N x 2^k) + M01 = (N x 64) + 36
5. Locating the circular buffer in memory
   - The location of the circular buffer in memory is determined by the upper (16 - k) bits of the address pointer register used in a modulo arithmetic operation. If there is an open area of memory from locations 111 to 189 ($006F to $00BD), for example, then the addresses of the lower and upper boundaries of the circular buffer will fit in this open area for N = 2:
     Lower boundary = (N x 64) = (2 x 64) = 128 = $0080
     Upper boundary = (N x 64) + 36 = (2 x 64) + 36 = 164 = $00A4
   - The exact area of memory in which a circular buffer is prepared is specified by picking a value for the address pointer register, R0 or R1, whose value is between the desired lower and upper boundaries of the circular buffer inclusively. Thus, selecting a value of 139 ($008B) for R0 would locate the circular buffer between locations 128 and 164 ($0080 to $00A4) in memory since the upper (16 - k) = 10 bits of the address indicate that the lower boundary is 128 ($0080).
   - In summary, the size and exact location of the circular buffer is defined once a value is assigned to the M01 register and the address pointer register (R0 or R1) that will be used in a modulo arithmetic calculation.
6. Selecting a value for the offset register if used in modulo operations
   - If the offset register is used in a modulo arithmetic calculation, it must be selected as follows:
     $|N| \leq$ M01 + 1 [where $|N|$ refers to the absolute value of the contents of the offset register]
   - The special case where N is a multiple of the block size, 2^k, is also discussed later in this section.
7. Performing the modulo arithmetic calculation
   - Once the appropriate registers are set up, the modulo arithmetic operation occurs when executing an instruction with any of the following addressing modes using the R0 (or R1 if enabled) register:

    - (Rn)+
    - (Rn)-
    - (Rn)+N
    - (Rn+N)
    - (Rn+xxxx)

- If the result of the arithmetic calculation would exceed the upper or lower bound, then wraparound is correctly performed.
- **Example 4-1** shows modulo arithmetic calculations for M01 = $0024, the lower boundary = $0080, and the upper boundary = $00A4.

Example 4-1 Modulo Arithmetic Calculations

```
MOVE     #$00A3,R0         ; R0 near upper boundary
NOP
LEA      (R0)+             ; R0 at upper boundary after
                           ; increment
                           ; ===> R0 = $00A4

MOVE     #$00A4,R0         ; R0 at upper boundary
NOP
LEA      (R0)+             ; R0 wraps over upper boundary
                           ; ===> R0 = $0080

MOVE     #$0080,R0         ; R0 at lower boundary
NOP
LEA      (R0)-             ; R0 wraps under lower boundary
                           ; ===> R0 = $00A4

MOVE     #$009F,R0         ; R0 near upper boundary
MOVE     #$0011,N          ; N passes upper boundary
LEA      (R0)+N            ; R0 wraps through boundary during
                           ; update
                           ; ===> R0 = $008B

MOVE     #$0083,R0         ; R0 near lower boundary
MOVE     #$FFFA,N          ; N passes past lower boundary
LEA      (R0)+N            ; R0 wraps through boundary during
                           ; update
                           ; ===> R0 = $00A1
```

Example 4-1 Modulo Arithmetic Calculations (Continued)

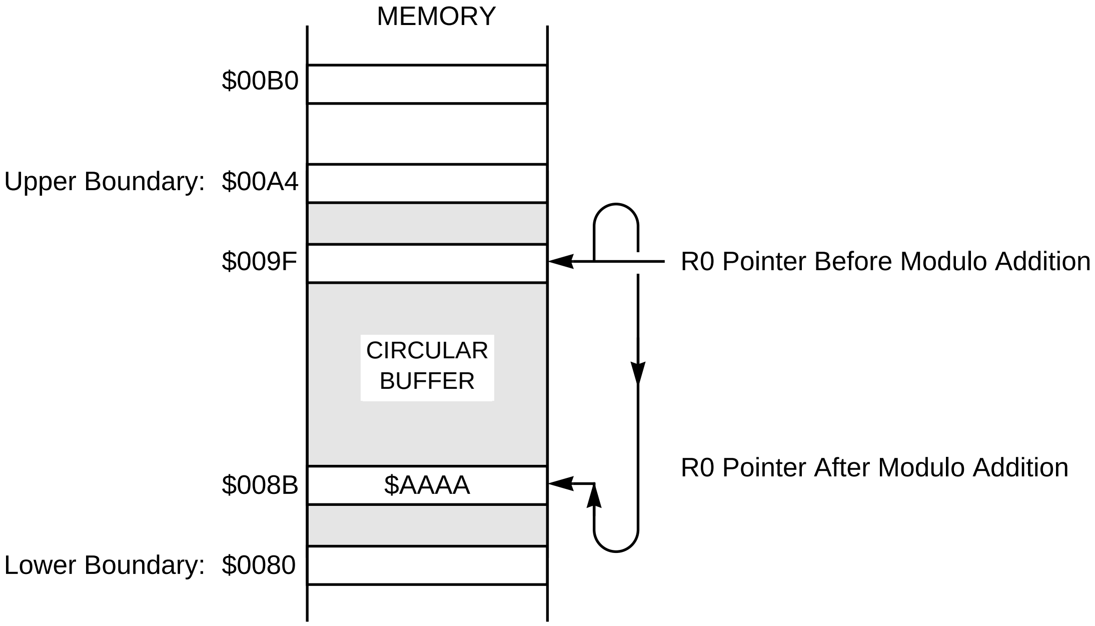
```
MOVE     #$00A4,R1          ; R1 at upper boundary
NOP
LEA      (R1)+              ; R1 does not wrap since M01
                            ; programmed for R0 only
                            ; ===> R1 = $00A5
```

The address pointer is not required to start at the lower address boundary or to end on the upper address boundary, but rather can begin anywhere within the defined modulo address range. In fact, the initial location of Rn determines the lower and upper boundaries. Neither the lower nor the upper boundary of the modulo region is stored; the upper and lower boundaries are not explicitly needed. Only the size of the modulo region is stored in M01. The boundaries are determined by the M01 register and the upper 16 minus k bits of the appropriate Rn register.

If Rn is outside the valid modulo buffer range and an operation occurs that causes Rn to be updated, the contents of Rn will be updated according to modulo arithmetic rules. For example, a MOVE B0,X:(R0)+ N instruction (where R0 = 6, M01 = 5, and N = 0) would apparently leave R0 unchanged since N0 = 0. However, since R0 is above the upper boundary, the AGU calculates R0 + N - M01 + 1 for the new contents of R0 and sets R0 = 0.

For cases where the size of a buffer is not a power of two, there will be a space of memory locations immediately after the buffer that are not accessible with modulo addressing. The space begins at the upper boundary + 1 and ends with the next usable lower boundary - 1. For the presented example, there will be a space of 27 memory locations between the upper boundary + 1 = $00A5 and next usable lower boundary address - 1= $00BF. These locations are still accessible when addressed with linear addressing, absolute addresses, or with the No Update addressing mode, but are not available for use by another modulo buffer.

If an offset N is used in the address calculations, the 16-bit absolute value |Nn| must be less than or equal to M01 + 1 for proper modulo addressing. This is because only a single modulo wraparound is detected. Note that when an address pointer is updated with an N register, it is not necessary to “land” exactly on the upper or lower boundary for the wraparound to occur correctly. If the result of the update goes beyond the upper or lower boundary, the modulo arithmetic still wraps its result correctly. This is demonstrated in **Figure 4-16** using the same boundaries as in the example with 37 locations; instead of calculating an effective address of $00B0 as it would with linear arithmetic, the effective address is $008B because the M01 register is set up for modulo addressing, and the result wraps around.



```
MOVE    #$0024,M01;     ;M01 set up for a buffer size of 37
MOVE    #$009F,R0;      ;R0 near upper boundary
MOVE    #$0011,N        ;N goes several locations past upper
                        ;boundary
MOVE    #$AAAA,X0;      ;Value to be written to Wrapped Address
MOVE    X0,X:(R0+N);    ;R0 wraps around to $008B
```

AA0029

Figure 4-16 Wraparound for N = $0011

For the normal case where | N | 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, delay lines, and sample buffers up to 16,384 words long. It is also used for decimation, interpolation, and waveform generation.

If N is greater than M, the result is data-dependent and unpredictable except for the special case where N = L*(2^k), a multiple of the block size, 2^k, where L is a positive integer. For this special case when using the (Rn)+N addressing mode, the pointer Rn will be updated using linear arithmetic to the same relative address L blocks forward in memory (see **Figure 4-17**). Note that this case requires the offset N must be a positive two's-complement integer.

This technique is useful in sequentially processing multiple tables or N-dimensional arrays. The range of values for N is -32,768 to +32,767. The modulo arithmetic unit will automatically wrap around the address pointer by the required amount.

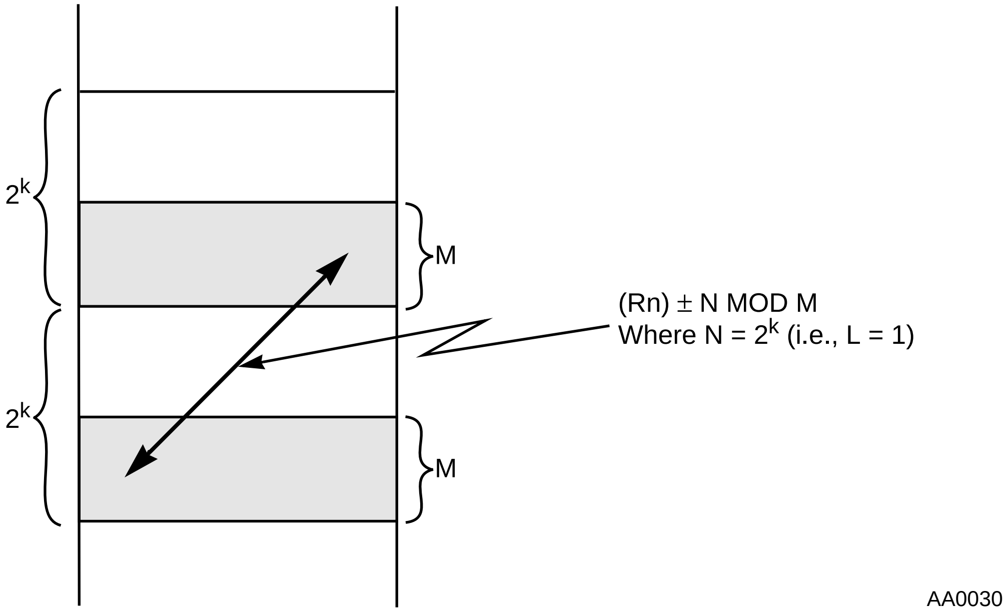


Figure 4-17 Linear Addressing with a Modulo Modifier

The special modulo case of (Rn) + N with N = L * (2^k) is useful for performing the same algorithm on multiple blocks of data in memory (e.g., implementing a bank of parallel IIR filters). The range of values for N is -16,384 to +16,383 although all values are not useful when modulo addressing as described above.

4.4.3 Programming the Modifier Register

There are two available address arithmetic types on the DSP core:

- Linear Addressing
- Modulo Addressing

Table 4-3 shows how the M01 register is correctly programmed for setting up either linear or modulo arithmetic on the R0 and R1 pointers. Linear arithmetic is always used with R2 and R3 pointer calculations and is also used for R0 and R1 calculations except for the cases stated below where the modulo register is used for both R0 and R1.

Table 4-3 Addressing Mode Modifier Summary

16-bit Modifier Register (M01) Contents	Address Calculation Arithmetic
0000 0000 0000 0000	Reserved
0000 0000 0000 0001	Modulo 2 (R0 only)
0000 0000 0000 0010	Modulo 3 (R0 only)
...	...
0011 1111 1111 1110	Modulo 16383 (R0 only)
0011 1111 1111 1111	Modulo 16384 (R0 only)
1000 0000 0000 0000	Reserved
1000 0000 0000 0001	Modulo 2 (R0 and R1)
1000 0000 0000 0010	Modulo 3 (R0 and R1)
...	...
1011 1111 1111 1110	Modulo 16383 (R0 and R1)
1011 1111 1111 1111	Modulo 16384 (R0 and R1)
1100 0000 0000 0000	Reserved
...	...
1111 1111 1111 1110	Reserved
1111 1111 1111 1111	Linear Arithmetic (R0 and R1)

The reserved sets of modifier values ($4000-$7FFF and $C000-$FFFE) should not be used.

4.4.4 Bit-Reversed Addressing

In addition to linear and modulo arithmetic, there is a third type of addressing used in DSP applications; bit-reversed arithmetic is used for 2^k-point FFT addressing. It is performed by adding together two registers but propagating the carry in the reverse direction (i.e., from the MSB to the LSB). This is equivalent to the following:

1. Bit-reversing the contents of Rn and the offset value N
2. Adding normally
3. Bit-reversing the result

If the (Rn)+N addressing mode is used with this address modifier, and N contains the value 2^{k-1} (a power of two), then post-incrementing by N is equivalent to the following:

1. Bit-reversing the k LSBs of Rn
2. Incrementing Rn by 1
3. Bit-reversing the k LSBs of Rn again

The range of values for N is 0 to +1023. This allows bit-reversed addressing for fits up to 2048 points.

As an example, consider a 1024-point FFT with real data stored in one section of data RAM and imaginary data stored in another section of data RAM. Then N would contain the value 512 and Post-incrementing 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 2π. 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 0, 1024, 2048, 3072, etc. The use of addressing modes other than post-increment by N is possible, but may not provide a useful result.

Bit reversed addressing is supported on DSP56800 chips with a small on-chip peripheral called the bit reversed address unit. This unit is a simple address generation unit that exists outside the DSP core that specifically generates bit-reversed addresses for FFT processing. See the appropriate user’s manual for information on this unit.

4.4.5 Examples

Figure 4-18 gives examples of the two addressing modifiers assuming 8-bit registers for simplification (all AGU registers are 16-bit). The addressing mode used in the example, (Rn)+N, adds the contents of the offset register to the contents of the address register after the address register is accessed.

Linear Address Modifier

M01 = 255 = 11111111 for Linear Addressing with R0	
Original Registers: N = 5,	R0=75=0100 0110
Post-increment by Offset N:	R0=80=0101 0000
Post-increment by Offset N:	R0=85=0101 0101
Post-increment by Offset N:	R0=90=0101 1010

Modulo Address Modifier

M01 = 19 = 00010011 for Modulo 20 Addressing with R0	
Original Registers: N = 5,	R0=75=0100 0110
Post-increment By Offset N:	R0=80=0101 0000
Post-increment By Offset N:	R0=65=0100 0001
Post-increment By Offset N:	R0=70=0100 0110

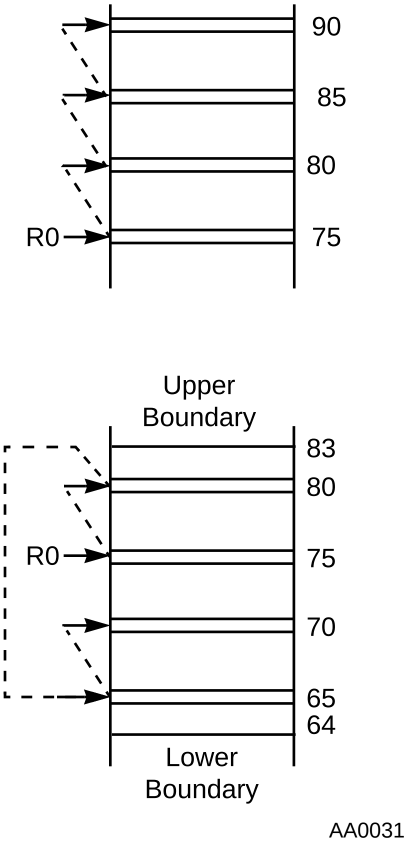


Figure 4-18 Address Modifier Examples

The results of the two examples are as follows:

- The linear address modifier addresses every fifth location, since the offset register contains $5.
- The modulo address modifier has a lower boundary at a predetermined location, and the modulo number plus the lower boundary establishes the upper boundary. This boundary creates a circular buffer so that, if the address register is pointing within the boundaries, addressing past a boundary causes a circular wraparound to the other boundary.

4.5 PIPELINE DEPENDENCIES

There are some cases within the address generation unit where the pipelined nature of the DSP core can affect the execution of a sequence of instructions. The pipeline dependencies are caused by a write to an AGU register immediately followed by an instruction that uses that same register in an address arithmetic calculation. When there is a dependency caused by a write to the N register, the DSP automatically stalls the pipeline one cycle. If a dependency is caused by a write to the R0-R3, SP, or M01 registers, however, there is no pipeline stall. This is also true if a bit-field operation is performed on the N register. Instead, the user must take care to avoid this case by rearranging the instructions or by inserting a NOP instruction to break the instruction sequence.

Several instruction sequences are presented below to examine cases where their pipeline dependency occurs, how this affects the machine, and how to correctly program to avoid these dependencies.

In **Example 4-2** there is no pipeline dependency since the N register is not used in the second instruction. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-2 No Dependency with the Offset Register

```
MOVE     #$7,N            ; Write to the N register
MOVE     X:(R2)+,X0       ; N not used in this instruction
```

In **Example 4-3** there is no pipeline dependency since the R2 register, used in the address calculation, is not written in the previous instruction. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-3 No Dependency with an Address Pointer Register

```
MOVE     #$7,R1           ; Write to R1 register
MOVE     X:(R2)+N,X0      ; R1 not used in this instruction
```

In **Example 4-4** there is no pipeline dependency since there is no address calculation performed in the second instruction. Instead, the R1 register is used as the source operand in a MOVE instruction, for which there is no pipeline dependency. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-4 No Dependency with No Address Arithmetic Calculation

```
MOVE     #$7,R1           ; Write to R1 register
MOVE     R1,X:$0004       ; No address arithmetic calculation
                          ; performed
```

Example 4-5 represents a special case. For the X:(Rn+xxxx) addressing mode, there is no pipeline dependency even if the same Rn register is written on the previous cycle.

This is true for R0-R3 as well as the SP register. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-5 No Dependency with (Rn+xxxx)

```
MOVE     #$7,R1            ; Write to R1 register
MOVE     X:(R1+$3456),X0   ; X:(Rn+xxxx) addressing mode
                           ; using R1
```

In **Example 4-6** there is a pipeline dependency since the N register is used in the second instruction. This is true for using N to update R0-R3 as well as the SP register. For the case where a dependency is caused by a write to the N register, the DSP core automatically stalls the pipeline by inserting one extra instruction cycle. Thus, this sequence is allowed. This dependency also exists for the (Rn+N) addressing mode.

Example 4-6 Dependency with a Write to the Offset Register

```
MOVE     #$7,N             ; Write to the N register
MOVE     X:(R2)+N,X0       ; N register used in address
                           ; arithmetic calculation
```

In **Example 4-7** there is a pipeline dependency since the N register is used in the second instruction. This is true for using N to update R0-R3 as well as the SP register. For the case where a dependency is caused by a bit-field operation on the N register, this sequence is not allowed and is flagged by the assembler. This sequence may be fixed by rearranging the instructions or inserting a NOP between the two instructions. This dependency only applies to the BFSET, BFCLR, or BFCHG instructions. There is no dependency for the BFTSTH, BFTSTL, BRCLR, or BRSET instructions. This dependency also exists for the (Rn+N) addressing mode.

Example 4-7 Dependency with a Bit-field Operation on the Offset Register

```
BFSET    #$7,N             ; Bit-field operation on the N
                           ; register
MOVE     X:(R2)+N,X0       ; N register used in address
                           ; arithmetic calculation
```

In **Example 4-8** there is a pipeline dependency since the address pointer register written in the first instruction is used in an address calculation in the second instruction. For the case where a dependency is caused by a write to one of these registers, this sequence is not allowed and is flagged by the assembler. This sequence may be fixed by rearranging the instructions or inserting a NOP between the two instructions.

Example 4-8 Dependency with a Write to an Address Pointer Register

```
        MOVE     #$7,R2             ; Write to the R2 register
        MOVE     X:(R2)+,X0         ; R2 register used in address
                                    ; arithmetic calculation
```

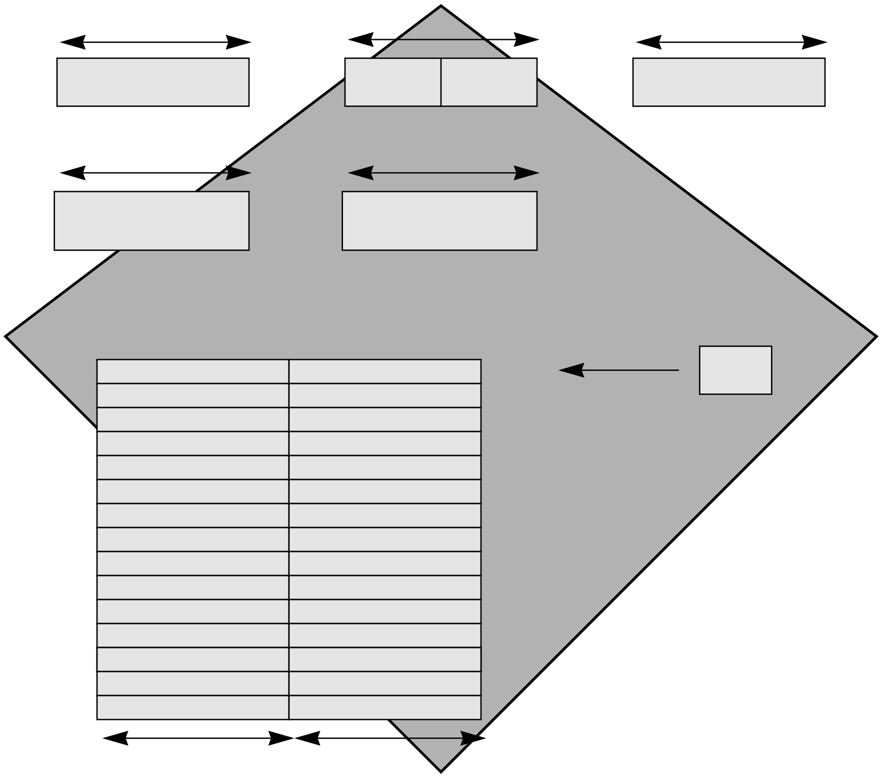
In **Example 4-9** there is a pipeline dependency since the M01 register written in the first instruction is used in an address calculation in the second instruction. For the case where a dependency is caused by a write to the M01 register, this sequence is not allowed and is flagged by the assembler. This sequence may be fixed by rearranging the instructions or inserting a NOP between the two instructions.

Example 4-9 Dependency with a Write to the Modifier Register

```
        MOVE     #$7,M01            ; Write to the M01 register
        MOVE     X:(R0)+,X0         ; M01 register used in address
                                    ; arithmetic calculation
```

SECTION 5

PROGRAM CONTROLLER



5.1 INTRODUCTION

The program controller unit is one of the three execution units in the central processing module. The program controller performs the following:

- Instruction fetching
- Instruction decoding
- Hardware DO and REP loop control
- Exception (interrupt) processing

This section covers the following:

- The architecture and programming model of the program controller
- The operation of the software stack
- A discussion of program looping
- A discussion about writing programs larger than 64K

Details of the instruction pipeline and the different processing states of the DSP chip, including reset and interrupt processing, are covered in **Section 7 Interrupts and the Processing States**.

5.2 ARCHITECTURE AND PROGRAMMING MODEL

A block diagram of the program controller is shown in **Figure 5-1** and its corresponding programming model is shown in **Figure 5-2**. The programmer views the program controller as consisting of five registers and a hardware stack (HWS). In addition to the standard program flow-control resources such as a program counter (PC) and status register (SR), the program controller features registers dedicated to supporting the hardware DO loop instruction—loop address (LA), loop counter (LC), and the hardware stack—and an operating mode register (OMR) defining the DSP operating modes.

The blocks and registers within the program controller are explained in the following sections.

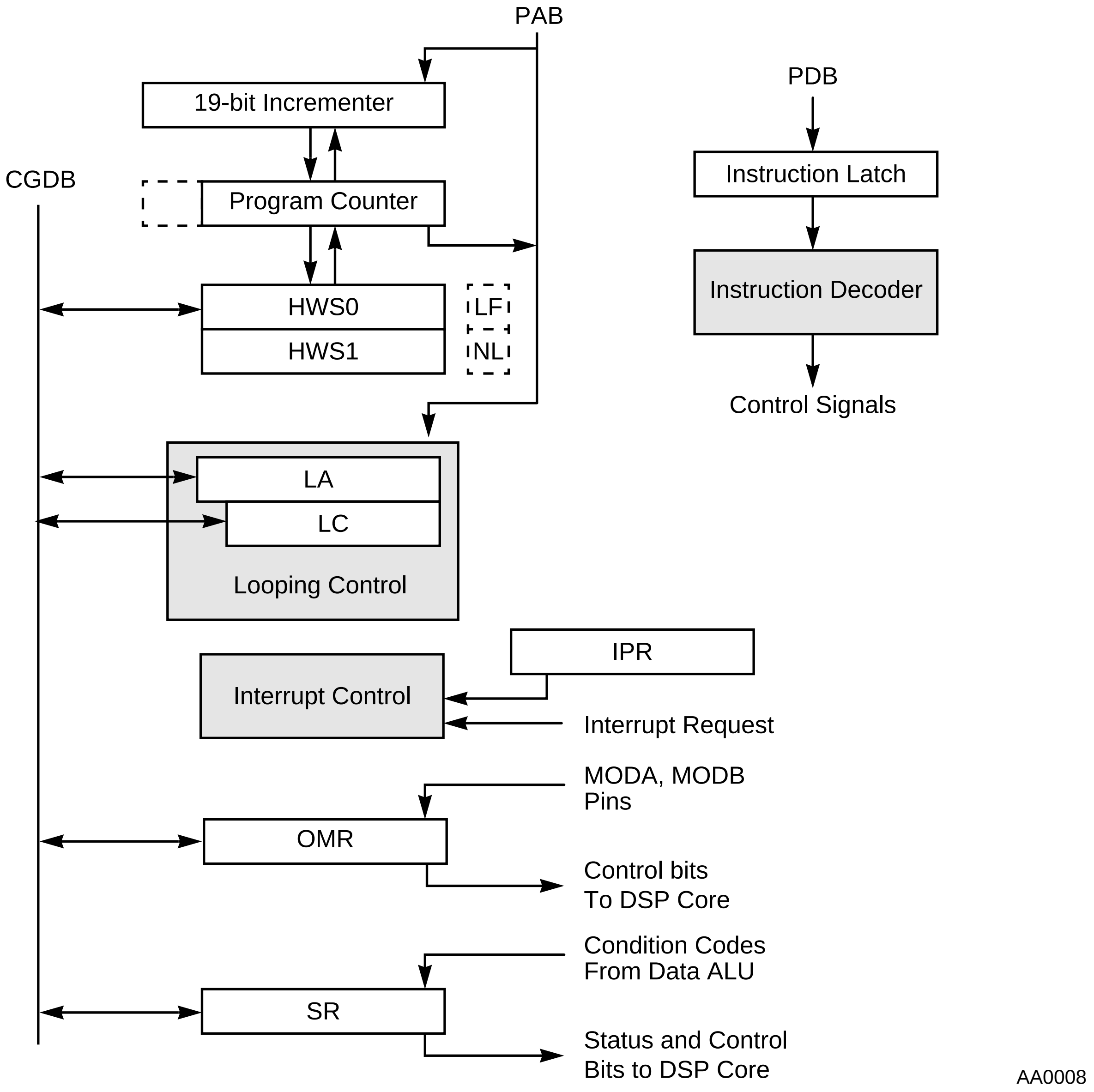


Figure 5-1 Program Controller Block Diagram

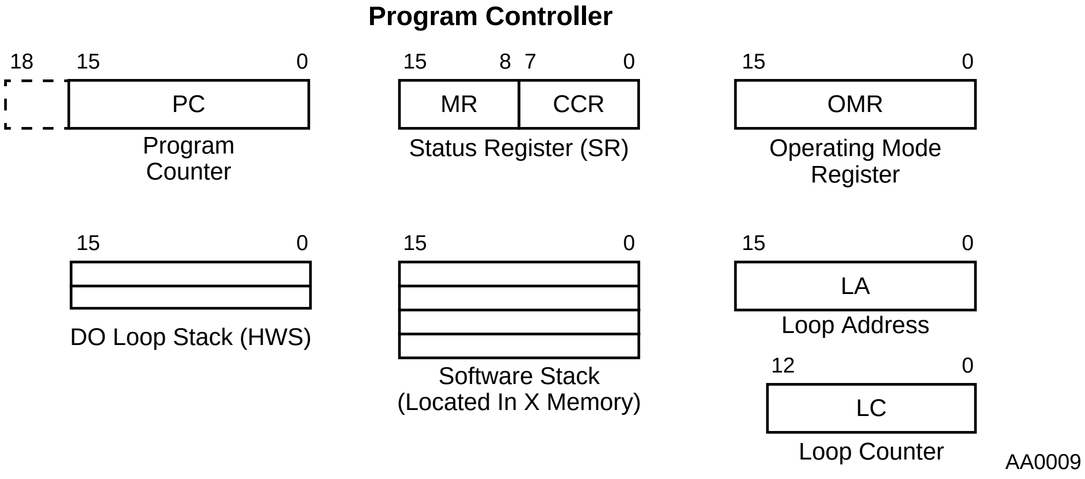


Figure 5-2 Program Controller Programming Model

5.2.1 Program Counter

The program counter (PC) is a 16-bit register that contains the address of the next location to be fetched from program memory space. Three additional bits are provided by the SR (**Program Counter Extension (P2-P0)—Bits 10-12** on page 5-13) to form a 19-bit program counter. This allows for a total linear program address space of 524,288 words. 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 hardware DO looping is initiated (on the hardware stack), when a jump to a subroutine is performed (on the software stack), and when interrupts occur (on the software stack). All jump to subroutine (JSR) and jump instructions support a 19-bit absolute address, allowing jumps anywhere within the 512K word program address space of the part. Branches work correctly across 64K word pages, but DO loops are allowed only on the first 64K page of program memory.

Located within the program controller are the following:

- Four user-accessible registers:
  - Loop address register (LA)
  - Loop count register (LC)
  - Status register (SR)
  - Operating mode register (OMR)

- A program counter (PC)
- A hardware stack (HWS)

5.2.2 Instruction Latch and Instruction Decoder

The instruction latch is a 16-bit internal register used to hold all instruction opcodes fetched from memory. The instruction decoder, in turn, uses the contents of the instruction latch to generate all control signals necessary for pipeline control—for normal instruction fetches, jumps, branches, and hardware looping.

5.2.3 Interrupt Control Unit

The interrupt control unit receives all interrupt requests, arbitrates among them, and then checks the highest priority interrupt request against the interrupt mask bits for the DSP core (I1 and I0 in the SR). If the requesting interrupt has higher priority than the current priority level of the DSP core, then exception processing begins. When exception processing begins, the interrupt control unit provides the address of the interrupt vector for interrupts generated on the DSP core, whereas the peripherals generate the vector address for interrupts generated by an on-chip peripheral.

Interrupts have a simple priority structure with levels zero or one. Level 0 is the lowest interrupt priority level (IPL) and is maskable. Level 1 is the highest level and is not maskable. Two interrupt mask bits in the SR reflect the current IPL of the DSP core and indicate the level needed for an interrupt source to interrupt the processor.

Two external interrupt sources are provided on the external interrupt request input pins—MODA/$\overline{\text{IRQA}}$ and MODB/$\overline{\text{IRQB}}$. These pins can be individually programmed as level-sensitive or negative edge-triggered. The MODA/$\overline{\text{IRQA}}$ and MODB/$\overline{\text{IRQB}}$ pins are also used to specify the operating mode of the chip coming out of reset. These pins are sampled when $\overline{\text{RESET}}$ is deasserted, and the sampled values are stored in the OMR’s operating mode (MA and MB) bits (see **Operating Mode Register** on page 5-14 for information on the operating modes). The interrupt control unit also arbitrates between interrupt requests generated by the on-chip peripherals.

Asserting the reset pin causes the DSP core to enter the reset processing state. This has higher priority and overrides any activity in the interrupt control unit and the exception processing state.

Details of interrupt arbitration and the exception processing state are discussed in **Exception Processing State** on page 7-8. The reset processing state is discussed in **Reset Processing State** on page 7-3.

5.2.4 Looping Control Unit

The looping control unit provides hardware dedicated to support loops, which are frequent constructs in DSP algorithms.

The repeat instruction (REP) loads the 13-bit LC register with a value representing the number of times the next instruction is to be repeated. The instruction to be repeated is only fetched once per loop, so power consumption is reduced, and throughput is increased when running from external program memory by decreasing the number of external fetches required.

The DO instruction loads the 13-bit LC register with a value representing the number of times the loop should be executed, loads the LA register with the address of the last instruction word in the loop (fetched only once per loop), and sets the loop flag (LF) bit in the SR. The top-of-loop address is stacked on the HWS so the loop can be repeated with no overhead. When the LF in the SR is asserted, the loop state machine will compare the PC contents to the contents of the LA to determine if the last instruction word in the loop was fetched. If the last word was fetched, the LC contents are tested for one. If LC is not equal to one, then it is decremented, and the contents of the HWS (the address of the first instruction in the loop) are read into the PC, effectively executing an automatic branch to the top of the loop. If the LC is equal to one, then the LF in the SR is restored with the contents of the OMR’s nested looping (NL) bit, the top-of-loop address is removed from the HWS, and instruction fetches continue at the incremented PC value (LA + 1).

Nested loops are supported by stacking the address of the first instruction in the loop (top-of-loop) in the HWS and copying the LF bit into the OMR’s NL bit prior to the execution of the first instruction in the loop. The user, however, must explicitly stack the LA and LC registers as described in **Nested Loops** on page 8-28.

Looping is described in more detail in **Program Looping** on page 5-19 and **Loops** on page 8-25.

5.2.5 Loop Counter

The loop counter (LC) is a special 13-bit down counter used to specify the number of times to repeat a hardware program loop (DO and REP loops). 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. If the loop counter is not one, it is decremented by one and the program loop is repeated. The loop counter may be read and written under program control. This gives software programs access to the value of the current loop iteration. This also allows for saving and restoring the LC to and from the software stack when nesting DO loops in software. (See **Figure 5-3**.)

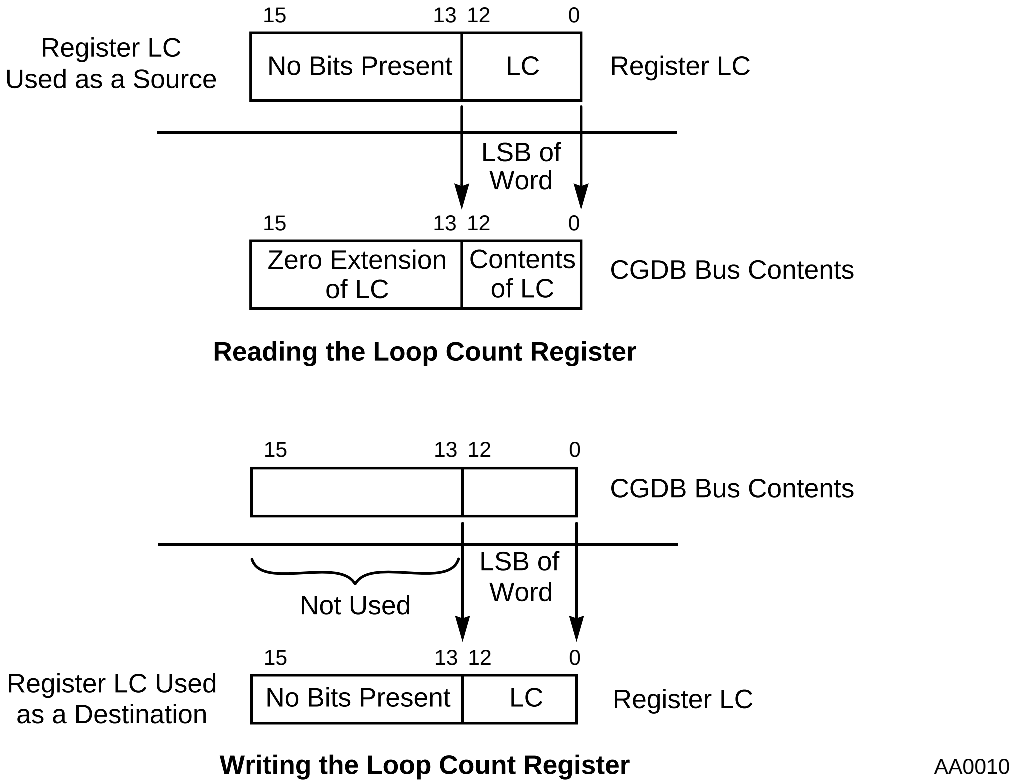


Figure 5-3 Accessing the Loop Count Register (LC)

This register is not stacked by a DO instruction and not unstacked by end-of-loop processing as is done on other Motorola DSPs. **Program Looping** on page 5-19 discusses what occurs when the loop count is zero. See **Nested Loops** on page 8-28 for a discussion of nesting loops in software.

The upper three bits of this register will read as zero during DSP read operations and should be written as zero to ensure future compatibility.

5.2.6 Loop Address

The loop address (LA) register indicates the location of the last instruction word in a hardware program loop (DO loop only). When the instruction word at the address contained in this register is fetched, the LC is checked. If it is not equal to 1, 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 LF is restored with the value in the OMR's NL bit, one location from the Hardware Stack is purged, and instruction execution continues with the instruction immediately after the loop.

The LA register is a read / write register written into by the DO instruction. The LA register can be directly accessed by the MOVE instructions as well. This also allows for saving and restoring the LA to and from the stack during nesting of loops. This register is not stacked by a DO instruction and not unstacked by end of loop processing. See **Nested Loops** on page 8-28 for a discussion of nesting loops in software.

5.2.7 Hardware Stack

The hardware stack (HWS) is a two-deep, 16-bit wide last-in-first-out (LIFO) stack. It is used for supporting hardware DO looping; the software stack is used for storing return addresses and the SR for subroutines and interrupts.

When a DO instruction is executed, the 16-bit address of the first instruction in the DO loop is pushed onto the hardware stack, the value of the LF bit is copied into the NL bit, and the LF bit is set. Each ENDDO instruction or natural end-of-loop will pop and discard the 16-bit address stored in the top location of the hardware stack, copy the NL bit into the LF bit, and clear the NL bit. One hardware stack location is used for each nested DO loop, and the REP instruction does not use the hardware stack. Thus, a two-deep hardware stack allows for a maximum of two nested DO loops and a nested REP loop within a program. Note that this includes any looping that may occur due to a DO loop in an interrupt service routine.

When the stack limit is exceeded, the oldest loop information (top-of-loop address and LF bit) is lost and a non-maskable hardware stack overflow interrupt will occur. The hardware stack overflow interrupt occurs after the stack limit has been exceeded. There is no interrupt on hardware stack underflow.

5.2.8 Status Register

The status register (SR) is a 16-bit register consisting of an 8-bit mode register (MR) and an 8-bit condition code register (CCR). The MR register is the high-order 8 bits of the SR; the CCR register is the low-order 8 bits.

The mode register is a special-purpose register that defines the operating state of the DSP core. It is conveniently located within the SR so that is it stacked correctly on an interrupt. This allows an interrupt service routine to set up the operating state of the DSP core differently.

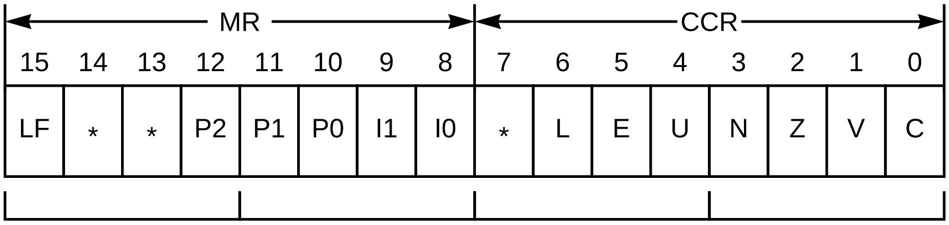
The mode register bits are affected by processor reset, exception processing, DO, ENDDO, any type of jump or branch, RTI, RTS, and SWI instructions, as well as instructions that directly reference the MR register. Also, normal program flow that crosses a 64K page boundary can affect these bits. During processor reset, the interrupt mask bits of the mode register will be set, and the LF bit and program extension bits will be cleared.

The condition code register is a special-purpose control register that defines the current status of the processor at any given time. Its bits are set as a result of status detected after certain instructions are executed. The CCR bits are affected by data ALU operations, bit-field manipulation instructions, the TSTW instruction, parallel move operations, and by instructions that directly reference the CCR register. In addition, the computation of the C, V, N, and Z condition code bits are affected by the OMR's CC bit, which specifies whether condition codes are generated using the information in the extension register. The CCR bits are not affected by data transfers over CGDB except if data limiting occurs when reading the A or B accumulators. During processor reset, all CCR bits are cleared. The standard definition of the CCR bits is given below, and more information about condition code bits is found in **Condition Code Generation** on page 3-32. Refer to **Instruction Set Details** on page A-1 computation rules.

The SR register is stacked on the software stack when a JSR is executed or when an interrupt occurs. The SR register is restored from the stack upon completion of an interrupt service routine by the return from interrupt instruction (RTI). The program extension bits in the SR are restored from the stack by the return from subroutine (RTS) instruction—all other SR bits are unaffected.

The SR format is shown in **Figure 5-4** and is described below.

SR
Status Register
Reset = $0300
Read/Write



LF—Loop Flag
P2-P0—Program Counter Extension
I1,I0—Interrupt Mask
L—Limit
E—Extension
U—Unnormalized
N—Negative
Z—Zero
V—Overflow
C—Carry

* Indicates reserved bits, read as zero and should be written with zero for future compatibility

AA0011

Figure 5-4 Status Register Format

5.2.8.1 Carry (C)—Bit 0

The carry (C) bit (SR bit 0) is set if a carry is generated out of the MSB of the result for an addition. It also is set if a borrow is generated in a subtraction. If the CC bit in the OMR register is 0, the carry or borrow is generated out of bit 35 of the result. If the CC bit in the OMR register is 1, the carry or borrow is generated out of bit 31 of the result. The carry bit is also modified by bit manipulation and shift instructions. Otherwise, this bit is cleared.

5.2.8.2 Overflow (V)—Bit 1

If the CC bit in the OMR register is 0 and if an arithmetic overflow occurs in the 36-bit result, the overflow (V) bit (SR bit 1) is set. If the CC bit in the OMR register is 1 and an arithmetic overflow occurs in the 32-bit result, the overflow bit is set. This indicates that the result is not representable in the accumulator register and the accumulator register has overflowed. Otherwise, this bit is cleared.

5.2.8.3 Zero (Z)—Bit 2

The zero (Z) bit (SR bit 2) is set if the result equals zero. Otherwise, this bit is cleared. The number of bits checked for the zero test depends on the OMR's CC bit and which instruction is executed, as documented in **Condition Code Generation** on page 3-32.

5.2.8.4 Negative (N)—Bit 3

If the CC bit in the OMR register is 0 and if bit 35 of the result is set, the negative (N) bit (SR bit 3) is set. If the CC bit in the OMR register is 1 and if bit 31 of the result is set, the negative bit is set. Otherwise, this bit is cleared.

5.2.8.5 Unnormalized (U)—Bit 4

The unnormalized (U) bit (SR bit 4) is set if the two most significant bits of the most significant product portion of the result are the same, and cleared otherwise. The U bit is computed as follows: $U = \overline{(\text{Bit 31 XOR Bit 30})}$.

The result of calculating the U bit in this fashion is that if a number is a positive normalized number, p, it satisfies the following equation: $0.5 \leq p < 1.0$. If a number is a negative normalized number, n, it satisfies the following equation: $-1.0 \leq n < -0.5$.

This bit is not affected by the OMR’s CC bit.

5.2.8.6 Extension (E)—Bit 5

The extension (E) bit (SR bit 5) is cleared if all the bits of the integer portion (bits 35-31) of the 36-bit result are all the same (the upper five bits of the value are 00000 or 11111). Otherwise, this bit is set.

If E is cleared, then the MS and LS portions of an accumulator contain all the bits with information—the extension register only contains sign extension. In this case, the accumulator extension register can be ignored. If E is set, then the extension register in the accumulator is in use.

This bit is not affected by the OMR’s CC bit.

5.2.8.7 Limit (L)—Bit 6

The limit (L) bit (SR bit 6) is set if the overflow bit is set or if the data limiters perform a limiting operation, and it is not affected otherwise. The L bit is cleared only by a processor reset or an instruction that specifically clears it. This allows the L bit to be used as a latching overflow bit. Note that L is affected by data movement operations that read the A or B accumulator registers onto the CGDB.

This bit is not affected by the OMR’s CC bit.

5.2.8.8 Interrupt Mask(I1 and I0)—Bits 8-9

The interrupt mask (I1 and I0) bits (SR bits 9 and 8) reflect the current priority level of the DSP core 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. Interrupt mask bit I0 must always be written with a 1 to ensure future compatibility and compatibility with other family members. The interrupt mask bits are set during processor reset. See **Table 5-1** for interrupt mask bit definitions.

Table 5-1 Interrupt Mask Bit Definition

I1	I0	Exceptions Permitted	Exceptions Masked
0	0	(Reserved)	(Reserved)
0	1	IPL 0, 1	None
1	0	(Reserved)	(Reserved)
1	1	IPL 1	IPL 0

5.2.8.9 Program Counter Extension (P2-P0)—Bits 10-12

The program extension (P2-P0) bits (SR bits 10-12) form the upper three bits of the program counter's value and allow for a 19-bit program address for programs larger than 65,536 locations. These bits are correctly stacked by the JSR instruction for subroutines and interrupts and are correctly unstacked by the RTS and RTI instructions for subroutines and interrupts, respectively. Jump and Jcc instructions will correctly load these bits when jumping to a new 64K page. When a program reaches the upper edge of a 64K boundary and increments into the next 64K page, the value in these three bits is correctly incremented. Likewise, these bits are correctly incremented or decremented as appropriate by any branch instruction that occurs across a page boundary. P2 corresponds to the MSB of the 19-bit program address and P0 is the least significant address bit of the three extension bits.

5.2.8.10 Reserved SR Bits—Bits 7, 13, and 14

The reserved SR bits 7, 13, and 14 are reserved for future expansion and will read as zero during DSP read operations. Bits 7, 13, and 14 should be written with 0 for future compatibility.

5.2.8.11 Loop Flag (LF)—Bit 15

The loop flag (LF) bit (SR bit 15) is set when a program loop is in progress and enables the detection of the end of a program loop. The LF bit is the only SR bit that is restored when terminating a program loop. Stacking and restoring the LF when initiating and exiting a program loop, respectively, allows the nesting of program loops (see **Nested Looping Bit (NL)—Bit 15** on page 5-17). REP looping does not affect this bit. The LF is cleared during processor reset.

Note: The LF is *not* cleared at the start of an interrupt service routine. This differs from the DSP56100 Family, where this bit is cleared upon entering an interrupt service routine. This will not cause a problem as long as the interrupt service routine code does not fetch the instruction whose address is stored in the LA register. This is typically the case because usually the interrupt service routine is located in a separate portion of program memory.

Note: This bit should never be explicitly cleared by a MOVE or bit-field instruction when the NL bit in the OMR register is set to a one.

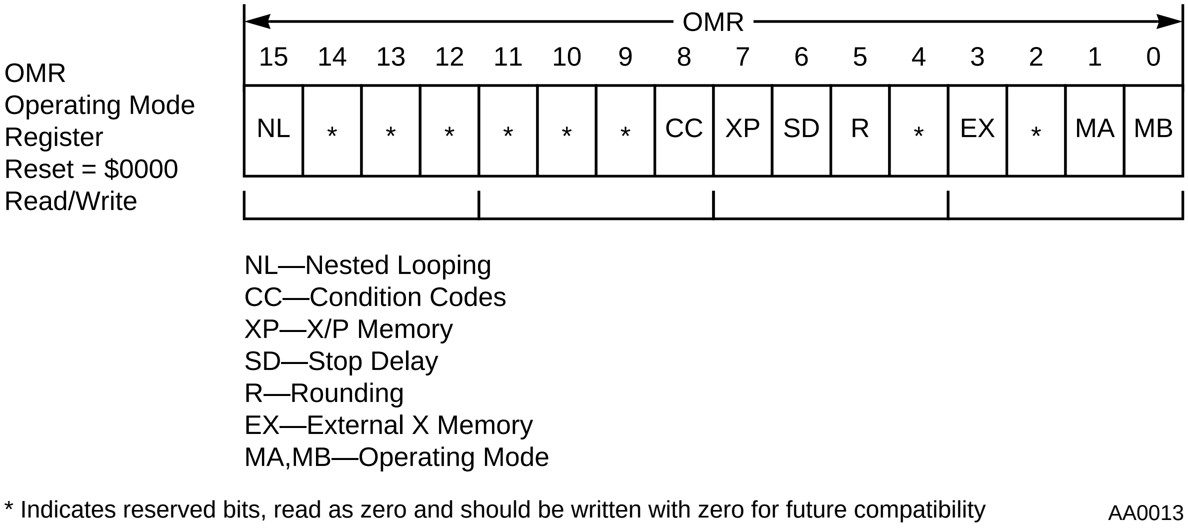
The LF bit is also affected by any accesses to the hardware stack register. Any move instruction that writes this register copies the old contents of the LF bit into the NL bit and then sets the LF bit. Any reads of this register, such as from a MOVE or TSTW instruction copy the NL bit into the LF bit and then clear the NL bit.

5.2.9 Operating Mode Register

The operating mode register (OMR) is a 16-bit register that defines the current chip operating mode of the processor. The OMR bits are affected by processor reset, operations on the HWS, and by instructions that directly reference the OMR. A DO loop will also affect the OMR, specificaly, the NL bit.

During processor reset, the chip operating mode bits will be loaded from the external mode select pins. The operating mode register format is shown in **Figure 5-5** and is described below.

Note: When a bit of the OMR is changed by an instruction, a delay of one instruction cycle is necessary before the new mode comes into effect.



OMR
Operating Mode
Register
Reset = $0000
Read/Write

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NL	*	*	*	*	*	*	CC	XP	SD	R	*	EX	*	MA	MB

NL—Nested Looping
CC—Condition Codes
XP—X/P Memory
SD—Stop Delay
R—Rounding
EX—External X Memory
MA,MB—Operating Mode

* Indicates reserved bits, read as zero and should be written with zero for future compatibility

Figure 5-5 Operating Mode Register Format

5.2.9.1 Operating Mode Bits (MB and MA)—Bits 0-1

The chip operating mode (MB and MA) bits (OMR bits 0 and 1) indicate the operating mode and memory maps of a DSP chip which has an external bus. These bits are loaded from the external mode select pins MODB and MODA respectively on processor reset. After the DSP leaves the reset state, MB and MA may be changed under program control. Possible operating modes for a program RAM part are shown in **Table 5-2**.

Table 5-2 Program ROM Operating Modes

MB	MA	Chip Operating Mode	Reset Vector	Program Memory Configuration
0	0	Bootstrap 0	BOOTROM P:$0000 (Boot from External Bus)	Int P-RAM is Write Only
0	1	Bootstrap 1	BOOTROM P:$0000 (Boot from Peripheral)	Int P-RAM is Write Only
1	0	Normal Expanded	External Pmem P:$E000	Internal Pmem Enabled
1	1	Development	External Pmem P:$0000	Internal Pmem Disabled

See the appropriate user’s manual for more detailed information on the operating modes for a specific DSP56800-based chip.

The bootstrap modes are used to initially load an on-chip program RAM upon exiting reset from external memory or through a peripheral. Operating modes 0 and 1 typically would be different for a program ROM part because no bootstrapping operation is required for a ROM part. An example of possible operating modes for a program ROM part are shown in **Table 5-3**.

Table 5-3 Program RAM Operating Modes

MB	MA	Chip Operating Mode	Reset Vector	Program Memory Configuration
0	0	Single Chip	Internal PROM P:$0000	Internal Pmem Enabled
0	1	(Reserved)	(Reserved)	(Reserved)
1	0	Normal Expanded	External Pmem P:$E000	Internal Pmem Enabled
1	1	Development	External Pmem P:$0000	Internal Pmem Disabled

5.2.9.2 External X Memory Bit (EX)—Bit 3

The external X memory (EX) bit (OMR bit 3), when set, will force all accesses to X memory on XAB1 and CGDB or PGDB to be external. The only exception to this rule is that if a MOVE or bit-field instruction is executed using the I/O short addressing mode, then the EX bit is ignored, and the access is performed to the on-chip location. The EX bit allows access to internal X memory with all addressing modes when this bit is cleared. The EX bit is necessary for providing a continuous memory map when using more than 64K of external data memory. This bit is cleared by processor reset.

The EX bit is ignored by the second read of a dual-read instruction, which uses the XAB2 and XDB2 buses and always accesses on-chip X data memory. For instructions with two parallel reads, the second read is always performed to internal on-chip memory. Refer to **Introduction to Moves and Parallel Moves** on page 6-3 for a description of the dual read instructions.

5.2.9.3 Rounding Bit (R)—Bit 5

The rounding (R) bit (OMR bit 5) selects between convergent rounding and two's-complement rounding. When set, two's-complement rounding (always round up) is used. The two rounding modes are discussed in **Rounding** on page 3-37. This bit is cleared by processor reset.

5.2.9.4 Stop Delay Bit (SD)—Bit 6

The stop delay (SD) bit (OMR bit 6) is used to select the delay that the DSP needs to exit the stop mode. When set, the processor exits quickly from stop mode. This bit is cleared by processor reset.

5.2.9.5 X/P Memory Bit (XP)—Bit 7

The X/P memory (XP) bit (OMR bit 7) is used to select from where program instructions are fetched. In most cases, this bit is cleared and instructions are fetched from Program Memory, but on devices that support execution from both memory spaces, this bit is set to 1 to fetch instructions from X data memory. Refer to the user's manual for the specific device for more information on executing programs from the X data memory. This bit is cleared by processor reset.

5.2.9.6 Condition Code Bit (CC)—Bit 8

The condition code (CC) bit (OMR bit 8) selects whether condition codes are generated using a 36-bit result from the MAC array or a 32-bit result. When set, the C, N, V, and Z condition codes are generated based on bit 31 of the data ALU result. When cleared, the C, N, V, and Z condition codes are generated based on bit 35 of the DALU result. The generation of the L, E, and U condition codes are not affected by the CC bit. This bit is cleared by processor reset.

Note: The unsigned condition tests used when branching or jumping (HI, HS, LO, LS) can only be used when the condition codes are generated with this bit set to 1. Otherwise, the chip will not generate the unsigned conditions correctly.

The effects of the CC bit on the condition codes generated by data ALU arithmetic operations is discussed in more detail in **Condition Code Generation** on page 3-32.

5.2.9.7 Nested Looping Bit (NL)—Bit 15

The nested looping (NL) bit (OMR bit 15) is used to display the status of program DO looping and the hardware stack. If this bit is set, then the program is currently in a nested DO loop (i.e., two DO loops are active). If this bit is cleared, then there may be a single or no DO loop active. This bit is necessary for saving and restoring the contents of the hardware stack, further described in **Multi-tasking and the Hardware Stack** on page 8-43. REP looping does not affect this bit.

It is important that the user never puts the processor in the illegal combination specified in **Table 5-4**. This can be avoided by ensuring that the LF bit is never cleared when the NL bit is set.

The NL bit is cleared on processor reset. Also see **Loop Flag (LF)—Bit 15** on page 5-13, which discusses the LF bit in the SR.

Table 5-4 Looping Status

NL	LF	DO Loop Status
0	0	No DO loops active
0	1	Single DO loop active
1	0	(Illegal Combination)
1	1	Two DO loops active

If both the NL and LF bits are set (i.e., two DO loops are active) and a DO instruction is executed, a hardware stack overflow interrupt occurs because there is no more space on the hardware stack to support a third DO loop.

The NL bit is also affected by any accesses to the hardware stack register. Any move instruction that writes this register copies the old contents of the LF bit into the NL bit and then sets the LF bit. Any reads of this register, such as from a MOVE or TSTW instruction copy the NL bit into the LF bit and then clear the NL bit.

5.2.9.8 Reserved OMR Bits—Bits 2, 4, and 9-14

The reserved OMR bits 2, 4, and 9-14 are reserved. They will read as zero during DSP read operations and should be written as zero to ensure future compatibility.

5.3 SOFTWARE STACK OPERATION

The software stack is a last-in-first-out (LIFO) stack of arbitrary depth implemented using memory locations in the X data memory. It is accessed through the POP instruction and the PUSH instruction macro (see **Multiple Value Pushes** on page 8-24) and will read or write the location in the X data memory pointed to by the stack pointer (SP) register. The PUSH instruction macro (two instruction cycles) pre-increments the SP register, and the POP instruction (one instruction cycle) will post-decrement the SP register.

The program counter and the SR are pushed on this stack for subroutine calls and interrupts. These registers are pulled from the stack for returns from subroutines using the RTS instruction (restores only the program extension bits in SR), and for returns from interrupt service routines that use the RTI instruction (entire SR is restored from the stack).

The software stack is also used for nesting hardware DO loops in software on the DSP56800 architecture. On the DSP56800 architecture, the user must stack and unstack the LA and LC registers explicitly if DO loops are nested. In this case, the software stack is typically used for this purpose as demonstrated in **Nested Loops** on page 8-28. The hardware stack is used, however, for stacking the address of the first instruction in the loop. Because this stack is implemented using locations in the X data memory, there is no limit to the number of interrupts or jump to subroutines or combinations of these that can be accommodated by this stack.

Note: Care must be taken to allocate enough space in the X data memory so that stack operations do not overlap other areas of data used by the program. Similarly, it may be desirable to locate the stack in on-chip memory to avoid delays due to wait states or bus arbitration.

See **Multiple Value Pushes** on page 8-24 and **Parameters and Local Variables** on page 8-36 for recommended techniques for using the software stack.

5.4 PROGRAM LOOPING

The DSP core supports looping on a single instruction (REP looping) and looping on a block of instructions (DO looping). Hardware DO looping allows fast looping on a block of instructions and is interruptible. Once the loop is set up with the DO instruction, there is no additional execution time to perform the looping tasks. REP looping repeats a one word instruction the specified number of times and can be efficiently nested within a hardware DO loop. It allows for excellent code density because blocks of in-line code of a single instruction can be replaced with a one word REP instruction followed by the instruction to be repeated. The correct programming of loops is discussed in detail in **Loops** on page 8-25.

5.4.1 Repeat (REP) Looping

The REP instruction is a one word instruction that performs single instruction repeating on one word instructions. It repeats the execution of a single instruction the amount of times specified either with a 6-bit unsigned value or with the 13 least significant bits of a DSP core register. When a repeat loop is begun, the instruction to be repeated is only fetched once from the program memory; it is not fetched each time the repeated instruction is executed. Repeat looping does not use any locations on the hardware stack. It also has no effect on the LF or NL bits in the SR and OMR respectively. Repeat looping cannot be used on an instruction that access the program memory; it is necessary to use DO looping in this case.

Note: REP loops are *not* interruptible since they are fetched only once. A DO loop with a single instruction can be used in place of a REP instruction if it is necessary to be able to interrupt while the loop is in progress.

Note: For the case of REP looping with a register value, when the register contains the value 0, then the instruction to be repeated is *not* executed (as is desired in an application), and instruction flow continues with the next sequential instruction. This is also true when an immediate value of 0 is specified.

5.4.2 DO Looping

The DO instruction is a two word instruction that performs hardware looping on a block of instructions. It executes this block of instructions the amount of times specified either with a 6-bit unsigned value or using the 13 least significant bits of a DSP core register. DO looping is interruptible and uses one location on the hardware stack for each DO loop. For cases where an immediate value larger than 63 is desired for the loop count, it is possible to use the technique presented in **Large Loops (Count Greater than 63)** on page 8-26.

The program controller register's 13-bit loop count and 16-bit loop address register are 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:

1. The LC and LA registers are loaded with values specified in the DO instruction.
2. The SR's LF bit is set, and its old value is placed in the NL bit.
3. The address of the first instruction in the program loop is pushed onto the hardware stack.

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 1. If the loop counter is not equal to 1, the loop counter is decremented and the top location in the DO Loop Stack is read (but not pulled) into the PC to return to the top of the loop. If the loop counter is equal to 1, the program loop is terminated by incrementing the PC, purging the stack (pulling the top location and discarding the contents) and continuing with the instruction immediately after the last instruction in the loop.

Note: For the case of DO looping with a register value, when the register contains the value 0, then the loop code is repeated 2^k times, where k = 13 is the number of bits in the LC register. If there is a possibility that a register value may be less than or equal to zero, then the technique outlined in **Variable Count Loops** on page 8-26 should be used. A DO loop with an immediate value of 0 is not allowed.

5.4.3 Nested Hardware DO and REP Looping

It is possible to nest up to two hardware DO loops and to nest a hardware REP loop within the two DO loops. It is recommended when nesting loops, however, that hardware DO loops should not be nested within code. Instead, a software loop should be used for an outer loop instead of a second DO loop (see **Nested Loops** on page 8-28).

The reason that nesting of hardware DO loops is supported is to provide for faster interrupt servicing. When hardware DO loops are not nested, this leaves a second hardware stack location available for immediate use by an interrupt service routine.

5.4.4 Terminating a DO Loop

A DO loop normally terminates when it has completed the last instruction of a loop for the last iteration of the loop (LC equals 1). Two techniques for early termination of the DO loops are presented in **Early Termination of a DO Loop** on page 8-32.

5.5 EXTENDING PROGRAMS BEYOND 64K WORDS

The DSP56800 architecture contains the necessary hooks to access programs larger than 65,536 memory words. It extends the 16-bit program counter with three additional SR bits to form a 19-bit program memory address capable of accessing 1 Mbyte of program memory (524,288 x 16-bit program words). Although the program is separated into 64K pages, page boundaries are transparent to the user except in a few isolated cases. The program counter will correctly increment across a page boundary, incrementing the three upper address bits. Branch and jump instructions work correctly across page boundaries, and the JSR and RTS instructions allow for correct processing of subroutines on any page, without increasing the program size.

The only restrictions occurring with programs larger than 65,536 locations is that DO loops can only exist on the first 64K page. Likewise, JSR instructions located in the interrupt vector table (**Sequence of Events in the Exception Processing State** on page 7-9) must always jump to the first 64K page; they may not go to a subroutine on any page. If interrupt service routines not located within the first 64K page, this can be solved with the software technique shown in **Example 5-1** below.

Example 5-1 Code to Locate Interrupt Service Routines to Any 64K Page

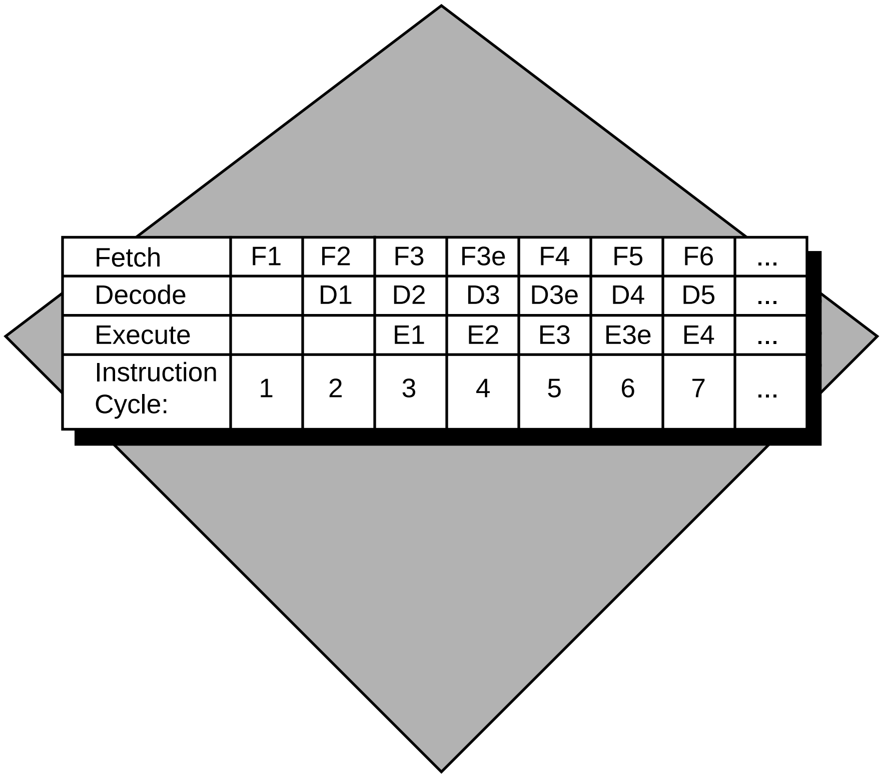
```
;
      ORG   P:$0000     ; Interrupt Vector Table
      JSR   ISR1_STEP   ; goto 1st 64K page
      JSR   ISR2_STEP   ; goto 1st 64K page


      ORG   P:$0080     ; Interrupt Jump Table in 1st 64K
ISR1_STEP
      JMP   ISR1        ; can go to any 64K page
ISR2_STEP
      JMP   ISR2        ; can go to any 64K page


      ORG   P:$20000    ; Interrupt Service Routine
ISR1
```

SECTION 6

INSTRUCTION SET INTRODUCTION



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

6.1 INTRODUCTION

As indicated by the programming model in **Figure 6-3** on page 6-8, the DSP architecture can be viewed as several functional units operating in parallel:

- Data ALU
- AGU
- Program controller
- Bit-manipulation unit

The goal of the instruction set is to keep each of these units busy each instruction cycle. This achieves maximum speed, minimum power consumption, and minimum use of program memory.

The complete range of instruction capabilities combined with the flexible addressing modes provide a very powerful assembly language for digital signal processing algorithms and general-purpose computing. (The addressing modes are presented in detail in **Addressing Modes** on page 4-8.) The instruction set has also been designed to allow for efficient coding of DSP algorithms, control code, and high-level language compilers. Execution time is enhanced by the hardware looping capabilities.

This section introduces the move instructions available on the DSP core, the concept of parallel moves, the DSP instruction formats, DSP core programming model, instruction set groups, a summary of the instruction set in tabular form, and an introduction to the instruction pipeline. The instruction summary is particularly useful because not only does it show every instruction but also the operands and addressing modes allowed for each instruction.

6.2 INTRODUCTION TO MOVES AND PARALLEL MOVES

To simplify programming, a powerful set of MOVE instructions is found on the DSP56800 core. This not only eases the task of programming the DSP, but also decreases the program code size and improves the efficiency, which in turn decreases the power consumption and MIPs required to perform a given task. Some examples of MOVE instructions are listed in **Example 6-1**.

Example 6-1 MOVE Instruction Types

```
MOVE        <any_DSPcore_register>,<any_DSPcore_register>

MOVE        <any_DSPcore_register>,<X_Data_Memory>
MOVE        <any_DSPcore_register>,<On_chip_peripheral_register>
MOVE        <X_Data_Memory>,<any_DSPcore_register>
MOVE        <On_chip_peripheral_register>,<any_DSPcore_register>

MOVE        <immediate_value>,<any_DSPcore_register>
MOVE        <immediate_value>,<X_Data_Memory>
MOVE        <immediate_value>,<On_chip_peripheral_register>
```

For any MOVE instruction accessing X data memory or an on-chip memory mapped peripheral register, seven different addressing modes are supported. Additional addressing modes are available on subset of DSP core registers that are most frequently accessed, including the registers in the data ALU, and all pointers in the address generation unit.

For all moves on the DSP56800, the syntax orders the source and destination as follows: `SRC,DST`. The source of the data to be moved and the destination are separated by a comma with no spaces either before or after the comma.

The assembler syntax also specifies which memory is being accessed (program or data memory) on any memory move. **Table 6-1** shows the syntax for specifying the correct memory space for any memory access; an example of a program memory access is shown where the address is contained in the register R2 and the address register is post-incremented after the access. The two examples for X data memory accesses show an address register indirect addressing mode in the first example and an absolute address in the second.

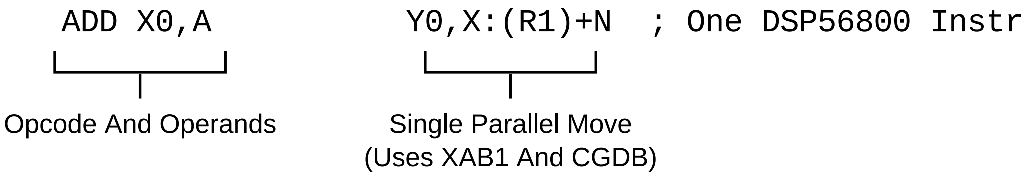
Table 6-1 Memory Space Symbols

Symbol	Examples	Description
P:	P:(R2)+	Program memory access
X:	X:(R0) X:$C000	X data memory access

The DSP56800 instruction set supports two additional types of moves—the single parallel move and the dual parallel read. Both of these are considered “parallel moves” and are extremely powerful for DSP algorithms and numeric computation.

The single parallel move allows an arithmetic operation and one memory move to be completed with one instruction, in one instruction cycle. For example, it is possible to execute an addition of two numbers while writing a value from a data ALU register to memory in the same instruction.

Figure 6-1 illustrates a single parallel move, which uses one program word and executes in one instruction cycle.



```
ADD X0,A          Y0,X:(R1)+N  ; One DSP56800 Instr
```

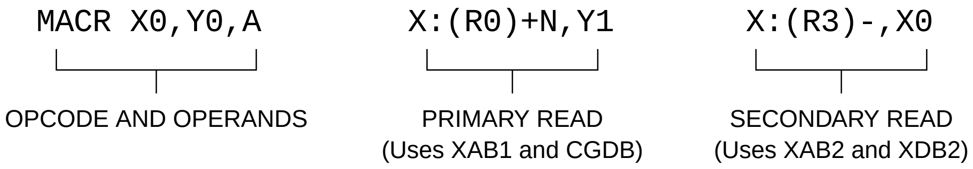
Figure 6-1 Single Parallel Move

In the single parallel move, the following occurs:

1. Register X0 is added to the register A and the result is stored in the A accumulator.
2. The contents of the Y0 register are moved into the X data memory at the location contained in the R1 register.
3. After completing the memory move, the R1 register is post-updated with the contents of the N register.

The dual parallel read allows an arithmetic operation to occur and two values to be read from X data memory with one instruction in one instruction cycle. For example, it is possible to execute in the same instruction a multiplication of two numbers with or without rounding of the result while reading two values from X data memory to two of the data ALU registers.

Figure 6-2 illustrates a double parallel move, which also uses one program word and executes in one instruction cycle.



```
MACR X0,Y0,A       X:(R0)+N,Y1       X:(R3)-,X0
```

Figure 6-2 Dual Parallel Move

In the dual parallel move, the following occurs.

1. The contents of the X0 and Y0 registers are multiplied, this result is added to the to the A accumulator, and the final result is stored in the A accumulator.
2. The contents of X data memory location pointed to with the R0 register are moved into the Y1 register.
3. The contents of X data memory location pointed to with the R3 register are moved into the X0 register.
4. After completing the memory moves, the R0 register is post-updated with the contents of the N register, and the R3 register is decremented.

Both types of parallel moves use a subset of available DSP56800 addressing modes, and the registers available for the move portion of the instruction are also a subset of the total set of DSP core registers. These subsets include the registers and addressing modes most frequently found in high- performance numeric computation and DSP algorithms. Also, the parallel moves allow a move to occur only with an arithmetic operation in the data ALU. A parallel move is not permitted, for example, with a JMP, LEA, or BFSET instruction.

6.3 INSTRUCTION FORMATS

Instructions are one, two, or three words in length. The instruction is specified by the first word of the instruction. The additional words may contain information about the instruction itself or may contain an operand for the instruction. Samples of assembly language source code for several instructions are shown in **Table 6-2**.

From the instruction formats listed in **Table 6-2**, it can be seen that the DSP offers parallel processing using the data ALU, AGU, program controller, and bit-manipulation unit. In the parallel move example the DSP can perform a designated ALU operation (data ALU), up to two data transfers specified with address register updates (AGU), 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 (i.e., operands are in data ALU registers) and allow the programmer to keep each parallel processing unit busy. Instructions that are memory-oriented (e.g., a bit-manipulation instruction), all logical instructions, or instructions that cause a control flow change (such as a jump) prevent the use of all parallel processing resources during their execution.

Table 6-2 Instruction Formats

Opcode[1]	Operands[2]	CGDB Transfer[3]	XDB2 Transfer[3]	PDB Transfer[3]	Comments
ADD	#$1234,Y1				; No parallel move
ANDC	#$7C,X:$E27				; No parallel move
ENDDO					; No parallel move
TSTW	X:(SP-9)				; No parallel move
MAC	A1,Y0,B				; No parallel move
LEA	(R2)-				; No parallel move
MOVE		R0,Y0			; No parallel move
CMP	X0,B	Y0,X:(R2)+			; Single parallel move
NEG	A	X:(R1)+N,X0			; Single parallel move
SUB	Y1,A	X:(R0)+,Y0	X:(R3)+,X0		; Dual parallel read
MPY	X1,Y0,B	X:(R1)+N,Y1	X:(R3)+,X0		; Dual parallel read
MACR	X0,Y0,A	X:(R1)+N,Y0	X:(R3)-,X0		; Dual parallel read
MOVE				X0,P:(R1)+	; Program memory move
JMP	$3FC10				; 19-bit jump address

Note: 1. Indicates data ALU, AGU, program controller, or bit-manipulation operation to be performed.
2. Specifies the operands used by the opcode.
3. Specifies optional data transfers over the respective buses.

6.4 PROGRAMMING MODEL

The registers in the DSP56800 core programming model are shown in **Figure 6-3**.

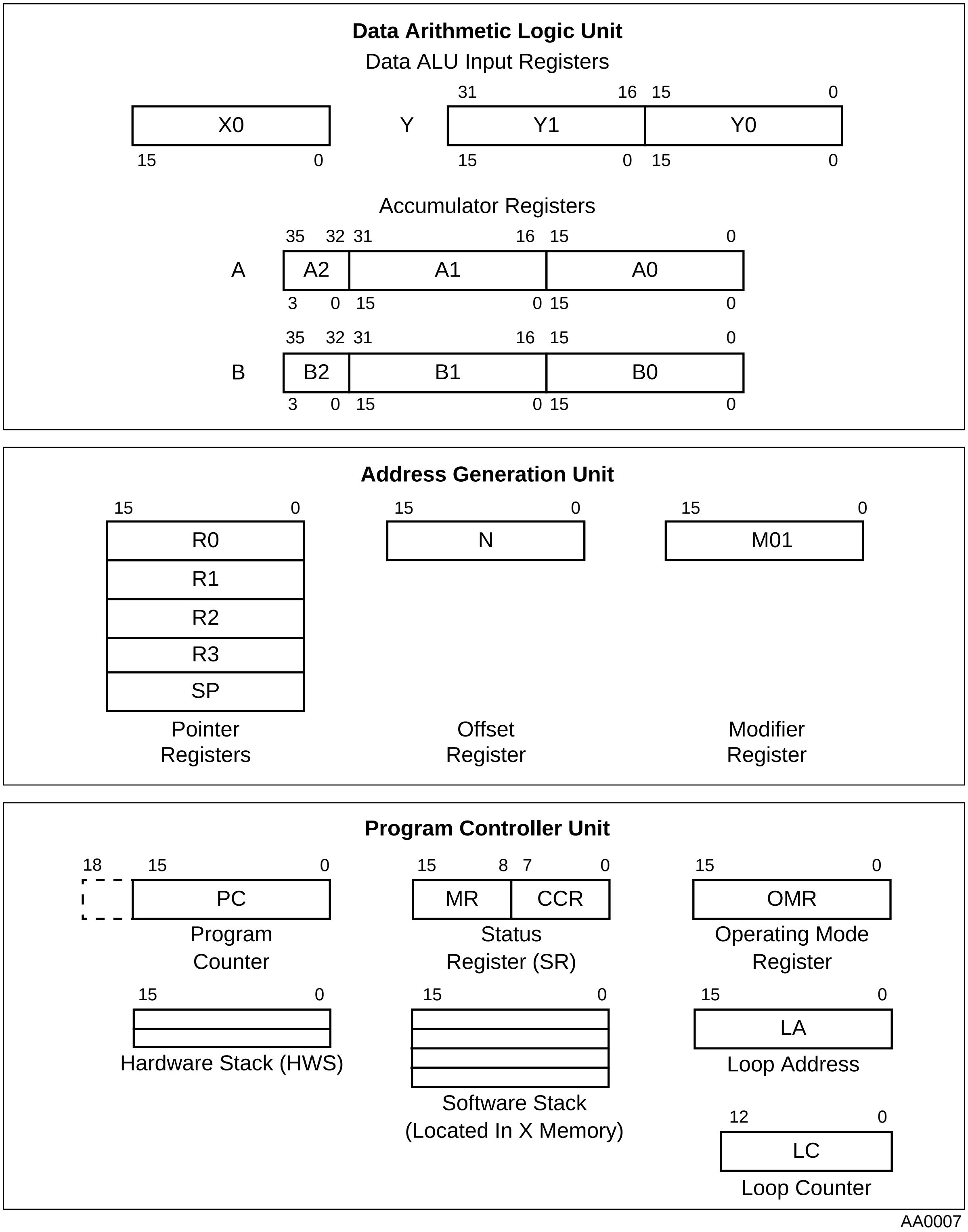


Figure 6-3 DSP56800 Core Programming Model

6.5 INSTRUCTION GROUPS

The instruction set is divided into the following groups:

- Arithmetic
- Logical
- Bit-manipulation
- Looping
- Move
- Program control

Each instruction group is described in the following sections. In addition, **Instruction Set Summary** on page 6-15 includes a useful summary for every instruction and the addressing modes and operand registers allowed for each instruction. Detailed information on each instruction is given in **Appendix A Instruction Set Details**.

6.5.1 Arithmetic Instructions

The arithmetic instructions perform all of the arithmetic operations within the data ALU. They may affect a subset or all of the condition code register bits. Arithmetic instructions are typically register-based (register direct addressing modes used for operands) so that the data ALU operation indicated by the instruction does not use the CGDB or the XDB2, although some instructions can also operate on immediate data or operands in memory.

Optional data transfers (parallel moves) may be specified with many arithmetic instructions. This allows for parallel data movement over the CGDB and over the XDB2 during a data ALU operation. This allows new data to be pre-fetched for use in following instructions and results calculated by previous instructions to be stored. Arithmetic instructions typically execute in one instruction cycle, although some of the operations may take additional cycles with different operand addressing modes. *The arithmetic instructions are the only class of instructions that allow parallel moves.*

In addition to the arithmetic shifts presented here, other types of shifts are also available in the logical instruction group (**Logical Instructions** on page 6-11). **Table 6-3** lists the arithmetic instructions.

Table 6-3 Arithmetic Instructions List

Instruction	Description
ABS	Absolute value
ADC	Add long with carry[1]
ADD	Add
ASL	Arithmetic shift left (36-bit)
ASLL	Arithmetic multi-bit shift left[1]
ASR	Arithmetic shift right (36-bit)
ASRAC	Arithmetic multi-bit shift right with accumulate[1]
ASRR	Arithmetic multi-bit shift right[1]
CLR	Clear
CMP	Compare
DEC(W)	Decrement word
DIV	Divide iteration[1]
IMPY(16)	Integer multiply[1]
INC(W)	Increment word
MAC	Signed multiply-accumulate
MACR	Signed multiply-accumulate and round
MACSU	Signed / unsigned multiply-accumulate[1]
MPY	Signed multiply
MPYR	Signed multiply and round
MPYSU	Signed / unsigned multiply[1]
NEG	Negate
NORM	Normalize[1]
RND	Round
SBC	Subtract long with carry[1]
SUB	Subtract
Tcc	Transfer conditionally[1]
TFR	Transfer data ALU register to an accumulator

Table 6-3 Arithmetic Instructions List (Continued)

Instruction	Description
TST	Test a 36-bit accumulator
TST(W)	Test a 16-bit register or memory location[1]
Note: 1. These instructions do not allow parallel data moves.	

6.5.2 Logical Instructions

The logical instructions perform all of the logical operations within the data ALU. They also affect the condition code register bits. Logical instructions are register-based as are the arithmetic instructions in **Table 6-3** and, again, some can also operate on operands in memory. Optional data transfers are not permitted with logical instructions. These instructions execute in one instruction cycle.

Table 6-4 lists the logical instructions.

Table 6-4 Logical Instructions List

Instruction	Description
AND	Logical AND
EOR	Logical exclusive OR
LSL	Logical shift left
LSR	Logical shift right
NOT	Logical complement
OR	Logical inclusive OR
ROL	Rotate left
ROR	Rotate right

6.5.3 Bit-manipulation Instructions

The bit-manipulation instructions perform one of three tasks:

- Testing a field of bits within a word
- Testing and modifying a field of a field of bits in a word

- Conditionally branching based on a test of bits within a word

Bit-field instructions can operate on any X memory location, peripheral, or DSP core register. BFTSTH and BFTSTL can test any field of the bits within a 16-bit word. BFSET, BFCLR, and BFCHG can test any field of the bits within a 16-bit word and then set, clear, or invert bits in this word, respectively. BRSET, and BRCLR can only test an 8-bit field in the upper or lower byte of the word, and then conditionally branch based on the result of the test. The carry bit of the condition code register contains the result of the bit test for each instruction. These instructions are read-modify-write type operations. The BFTSTH, BFTSTL, BFSET, BFCLR, and BFCHG instructions execute in two or three instruction cycles. The BRCLR and BRSET instructions execute in four to six instruction cycles.

Table 6-5 lists the bit-manipulation instructions.

Table 6-5 Bit-field Instruction List

Instruction	Description
ANDC	Logical AND with immediate data
BFCLR	Bit-field test and clear
BFSET	Bit-field test and set
BFCHG	Bit-field test and change
BFTSTL	Bit-field test low
BFTSTH	Bit-field test high
BRSET	Branch if selected bits are set
BRCLR	Branch if selected bits are clear
EORC	Logical exclusive OR with immediate data
NOTC	Logical complement on memory location and registers
ORC	Logical inclusive OR with immediate data

Note: Due to instruction pipelining, if an AGU register (Rn, N, SP, or M01) is directly changed with a bit-field instruction, the new contents may not be available for use until the second following instruction (see the restrictions discussed in **Pipeline Dependencies** on page 4-36).

See **Jumps and Branches** on page 8-4 for other instructions which can be synthesized.

6.5.4 Looping Instructions

The looping instructions establish looping parameters and initiate zero-overhead program looping. They allow looping on a single instruction (REP) or a block of instructions (DO). For DO looping, the address of the first instruction in the program loop is saved on the hardware stack to allow no-overhead looping. The last address of the DO loop is specified as a 16-bit absolute address. No locations in the hardware stack are required for the REP instruction. The ENDDO instruction is used only when breaking out of the loop; otherwise, it is better to use `MOVE #1,LC`. This is discussed in more detail in **Early Termination of a DO Loop** on page 8-32.

Table 6-6 lists the loop instructions.

Table 6-6 Loop Instruction List

Instruction	Description
DO	Start hardware loop
ENDDO	Disable current loop and unstack parameters
REP	Repeat next instruction

6.5.5 Move Instructions

The move instructions perform data movement over the CGDB, PGDB, XDB2, and/or PDB. Move instructions do not affect the condition code register except the limit bit if limiting is performed when reading a data ALU accumulator register. These instructions do not allow optional data transfers. In addition to the following move instructions, there are also parallel moves that can be used simultaneously with many of the arithmetic instructions. The parallel moves are shown in **Table 6-9** and **Table 6-10** and are discussed in detail in **Introduction to Moves and Parallel Moves** on page 6-3 and **Appendix A Instruction Set Details**. The LEA instruction is also included in this instruction group.

Note: There is a PUSH instruction macro, described in **Multiple Value Pushes** on page 8-24, that can be used with the POP instruction presented here.

Table 6-7 lists the move instructions.

Table 6-7 Move Instruction List

Instruction	Description
LEA	Load effective address
POP	Pop a register from the software stack
MOVE	Move data
MOVE(C)	Move control register
MOVE(I)	Move immediate
MOVE(M)	Move program memory
MOVE(P)	Move peripheral data
MOVE(S)	Move absolute short

Note: Due to instruction pipelining, if an AGU register (Rn, SP, or M01) is directly changed with a move instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in **Pipeline Dependencies** on page 4-36.

6.5.6 Program Control Instructions

The program control instructions include branches, jumps, conditional branches, conditional jumps, and other instructions that affect the program counter and software stack. Program control instructions may affect the status register bits as specified in the instruction. Also included in this instruction group are the STOP and WAIT instructions that can place the DSP chip in a low power state. See **Jumps and Branches** on page 8-4 and **Jumps and JSRs Using a Register Value** on page 8-41 for additional jump and branch instructions which can be synthesized from existing DSP56800 instructions.

Table 6-8 lists the program control instructions.

Table 6-8 Program Control Instruction List

Instruction	Description
Bcc	Branch conditionally
BRA	Branch
DEBUG	Enter debug mode
Jcc	Jump conditionally

Table 6-8 Program Control Instruction List (Continued)

Instruction	Description
JMP	Jump
JSR	Jump to subroutine
NOP	No operation
RTI	Return from interrupt
RTS	Return from subroutine
STOP	Stop processing (lowest power stand-by)
SWI	Software interrupt
WAIT	Wait for interrupt (low power stand-by)

6.6 INSTRUCTION SET SUMMARY

A summary of the DSP56800 instruction set is presented in this section in tabular form. The tables provide a quick reference to the entire instruction set because they show not only the instructions themselves, but also show the registers and addressing modes available to each instruction. From these tables, it is very easy to determine if a particular operation can be performed with a desired register or addressing mode.

6.6.1 Using the Instruction Summary Tables

The following abbreviations are used in the instruction tables:

- F—A or B accumulator
- F1—MSP of one of the accumulators (A1 or B1)
- DD—X0, Y0, or Y1 register
- Rn—R0, R1, R2, or R3 register

For instructions that contain parentheses in the instructions name, such as DEC(W) or IMPY(16), the portion within the parenthesis is optional. The assembler will accept the instruction in either form:

- DECW A ; Valid Instruction
- DEC A ; Valid Instruction
- IMPY16 Y1,X0,B ; Valid Instruction
- IMPY Y1,X0,B ; Valid Instruction

As an example, **Table 6-9** shows all registers and addressing modes allowed when performing a dual read instruction, one of the DSP56800's parallel move instructions discussed in **Introduction to Moves and Parallel Moves** on page 6-3. The instructions in **Example 6-2** are allowed:

Example 6-2 Valid Instructions

```
      MOVE                    X:(R0)+,Y0  X:(R3)+,X0
      MACR        X0,Y1,A     X:(R1)+N,Y1 X:(R3)-,X0
      ADD         Y0,B        X:(R1)+N,Y0 X:(R3)+,X0
```

The instruction in **Example 6-3** is *not* allowed:

Example 6-3 Invalid Instruction

```
      ADD         X0,Y1,A     X:(R2)-,X0  X:(R3)+N,Y0
```

The instruction is not allowed for many different reasons, any of which make it an invalid instruction. The information in **Table 6-9** shows this instruction is *not* valid for the following reasons:

- The only operands accepted for ADD or SUB is X0,F, X1,F, and Y0,F, where F is either the A or B accumulator register. Thus, `X0,Y1,A` is an invalid entry.
- The pointer R2 is not allowed for Read1.
- The post-decrement addressing mode is not available for Read1.
- The X0 register may not be a destination for Read1 because it is not listed in the Dest1 column.
- The post-update by N addressing mode is not allowed for Read2. Read2 is always identified as the memory move that uses R3 in instructions with two memory moves. For Read2, only the post-increment and post-decrement addressing modes are allowed.
- The Y0 register may not be a destination for Read2 because it is not listed in the Dest2 column.

6.6.2 Instruction Summary Tables

The following tables give a summary of the DSP56800 instruction set:

- **Table 6-9 Data ALU Instructions—Dual Parallel Read** on page 6-18
- **Table 6-10 Data ALU Instructions—Single Parallel Move** on page 6-19
- **Table 6-11 Data ALU Multiply Instructions—No Parallel Move** on page 6-20
- **Table 6-12 Other Data ALU Instructions—No Parallel Move** on page 6-21
- **Table 6-13 Data ALU Instructions using Memory—No Parallel Move** on page 6-22
- **Table 6-14 Data ALU Instructions with Immediate Data—No Parallel Move** on page 6-23
- **Table 6-15 Multi-bit Shifting Instructions—No Parallel Move** on page 6-24
- **Table 6-16 Division Instruction** on page 6-25
- **Table 6-17 Normalization Instruction** on page 6-25
- **Table 6-18 Clear Instructions** on page 6-25
- **Table 6-19 Move Instructions** on page 6-26
- **Table 6-20 Immediate Move Instructions** on page 6-27
- **Table 6-21 Move Peripheral Data Instructions** on page 6-27
- **Table 6-21 Move Peripheral Data Instructions** on page 6-27
- **Table 6-23 Test Word Instructions** on page 6-28
- **Table 6-24 Conditional Register Transfer Instructions** on page 6-28
- **Table 6-25 Bit-manipulation Instructions** on page 6-29
- **Table 6-26 Branch on Bit-field Instructions** on page 6-29
- **Table 6-27 Jump And Branch Instructions** on page 6-30
- **Table 6-28 Effective Address Update** on page 6-30
- **Table 6-29 DO and REP Instructions** on page 6-30
- **Table 6-30 Special Instructions** on page 6-31

A more detailed description of each instruction is given in **Appendix A Instruction Set Details**.

Table 6-9 Data ALU Instructions—Dual Parallel Read

Data ALU Operation		First & Second Memory Reads		Destinations For Memory Reads	
Operation	**Operands**	**Read1**	**Read2**	**Dest1**	**Dest2**
MAC MPY MACR MPYR	X0,Y1,A X0,Y0,A Y1,Y0,A X0,Y1,B X0,Y0,B Y1,Y0,B	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
				Y1	X0
				This column lists the valid destination registers for the Read1 memory access.	This column lists the valid destination registers for the Read2 memory access.
ADD SUB	X0,A Y1,A Y0,A X0,B Y1,B Y0,B				
MOVE					

Each instruction in **Table 6-9** performs two reads from the X data memory. In addition, an arithmetic operation typically occurs in parallel with these two memory accesses. If the MOVE instruction is specified then no arithmetic operation is performed in parallel with these two accesses.

Example 6-4 Data ALU Instructions—Dual Parallel Read

```
MOVE                    X:(R0)+,Y0          X:(R3)+,X0
MACR        X0,Y0,A     X:(R1)+N,Y0         X:(R3)-,X0
SUB         Y1,B        X:(R0)+,Y1          X:(R3)+,X0
```

Table 6-10 Data ALU Instructions—Single Parallel Move

Data ALU Operation		Parallel Memory Read Or Write	
Operation	**Operands**	**Mem Access**	**Src/Dest**
MAC MPY MACR MPYR	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
ADD SUB TFR CMP	X0,F Y1,F Y0,F A,B B,A		
RND TST ABS INC(W) DEC(W) CLR NEG ASL ASR	A B		

Each instruction in **Table 6-10** performs one access (read or write) to the X data memory. The memory access, referred to as a single parallel move, uses the XAB1 and CGDB. In addition, an arithmetic operation always occurs in parallel with this memory access.

Example 6-5 Data ALU Instructions—Single Parallel Move

```
MPY         A1,Y0,A     X:(R0)+N,B
MACR        Y1,X0,B     X0,X:(R1)+
ADD         B,A         X:(R2)+,Y0
SUB         Y0,B        Y1,X:(R3)+
NEG         B           A1,X:(R0)+N
INC         A           X:(R1)+,Y1
TFR         X0,B        B,X:(R2)+N
```

Table 6-11 Data ALU Multiply Instructions—No Parallel Move

Operation	Operands	Comments
MAC MPY MACR MPYR	(±)Y1,X0,F (±)Y0,X0,F (±)Y1,Y0,F (±)Y0,Y0,F (±)A1,Y0,F (±)B1,Y1,F	Allows multiplication result to be inverted if desired
	(±)Y1,X0,DD (±)Y0,X0,DD (±)Y1,Y0,DD (±)Y0,Y0,DD (±)A1,Y0,DD (±)B1,Y1,DD	Fractional multiplication, 16-bit result
IMPY(16)	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	Integer multiplication, 16-bit result
	Y1,X0,DD Y0,X0,DD Y1,Y0,DD Y0,Y0,DD A1,Y0,DD B1,Y1,DD	Multiplication result may not be inverted
MPYSU MACSU	X0,Y1,F X0,Y0,F Y0,Y1,F Y0,Y0,F Y0,A1,F Y1,B1,F	The first operand is treated as signed and the second as unsigned.
	X0,Y1,DD X0,Y0,DD Y0,Y1,DD	Fractional multiplication, 16-bit result
	Y0,Y0,DD Y0,A1,DD Y1,B1,DD	Multiplication result may not be inverted

Example 6-6 Multiply Instructions—No Parallel Move

```
      MAC         -Y1,Y0,B
      IMPY16      Y0,X0,Y1
      MPYSU       X0,Y1,A
```

Table 6-12 Other Data ALU Instructions—No Parallel Move

Operation	Operands	Comments
ADD SUB CMP AND OR EOR	X0,F Y1,F Y0,F F1,X0 F1,Y1 F1,Y0 Y0,X0 Y1,X0 X0,Y0 Y1,Y0 X0,Y1 Y0,Y1	
TFR	X0,F Y1,F Y0,F	The TFR instruction does not allow 16-bit destinations such as X0, Y0, or Y1, because it is more appropriate to use a MOVE instruction that performs the same data move. The TFR is most useful with a parallel move, allowing one register transfer and one memory move in the same instruction.
ADD SUB TFR CMP	A,B B,A	The TFR is most useful with a parallel move, allowing one register transfer and one memory move in the same instruction. It is also useful when transfering one 36-bit accumulator to the other, since all 36-bits are transferred and no limiting is performed.
INC(W), DEC(W) ASL, ASR ROL, ROR LSL, LSR NOT	A B X0 Y1 Y0	
TST, RND ABS, NEG	A B	
ADD, SUB ADC, SBC	Y,A Y,B	

Example 6-7 Other Data ALU Instructions—No Parallel Move

```
ADD         Y0,A
SUB         A,B
ASL         X0
```

Table 6-13 Data ALU Instructions using Memory—No Parallel Move

Operation	Operands	Comments
ADD	X:<aa>,F X:<aa>,DD	<aa>: first 64 locationsof X memory
	F,X:<aa> DD,X:<aa>	
	X:xxxx,F X:xxxx,DD	xxx: long 16-bit absolute address
	F,X:xxxx DD,X:xxxx	
	X:(SP-xx),F X:(SP-xx),DD	xx = [1 to 64]
	F,X:(SP-xx) DD,X:(SP-xx)	
SUB CMP	X:<aa>,F X:<aa>,DD	(See comments above)
	X:xxxx,F X:xxxx,DD	Performs <reg> - X:<ea>
	X:(SP-xx),F X:(SP-xx),DD	

INC(W)
DEC(W)
	X:<aa>	(See comments above)

	X:xxxx	
	X:(SP-xx)	

Example 6-8 Data ALU Instructions with Memory Operands

```
ADD         X:$001C,X0
ADD         B,X:$ACE0
CMP         X:(SP-37),Y0
SUB         X:$000F,A
INCW        X:$0009
DECW        X:(SP-1)
```

Table 6-14 Data ALU Instructions with Immediate Data—No Parallel Move

Operation	Operands	Comments
ADD SUB CMP	#xx,A #xx,B #xx,X0 #xx,Y1 #xx,Y0	5-bit positive integer ranging from 0 to 31
ADD SUB CMP	#xxxx,A #xxxx,B #xxxx,X0 #xxxx,Y1 #xxxx,Y0	16-bit signed integer

Example 6-9 Data ALU Instructions using Immediate Operands

```
      ADD          #21,X0
      ADD          #$1234,B
      CMP          #$CA79,Y1
      SUB          #9,A
```

Table 6-15 Multi-bit Shifting Instructions—No Parallel Move

Operation	Operands	Comments
LSRR ASRR ASLL	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F Y1,X0,DD Y0,X0,DD Y1,Y0,DD Y0,Y0,DD A1,Y0,DD B1,Y1,DD	Multi-bit Logical & Arithmetic Shifting First register is value to be shifted, second register is the shift amount (uses 4 LSBs)
LSLL	Y1,X0,DD Y0,X0,DD Y1,Y0,DD Y0,Y0,DD A1,Y0,DD B1,Y1,DD	Multi-bit Logical Left Shift First register is the value to be shifted, second register is the shift amount (uses 4 LSBs)
ASRAC LSRAC	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	Multi-bit Arithmetic Shifting with Accumulation First register is the value to be shifted, second register is the shift amount (uses 4 LSBs)

The ASRAC and LSRAC are multi-bit arithmetic and logical shifting instructions with accumulation and can be used for right shifting of a multi-precision value.

Example 6-10 Multi-bit Shifting Instructions

```
        ASLL        A1,Y0,A
        LSRR        Y1,Y0,X0
        ASRAC       Y1,X0,B
```

Table 6-16 Division Instruction

Operation	Operands
DIV	X0,A Y1,A Y0,A X0,B Y1,B Y0,B

Table 6-17 Normalization Instruction

Operation	Operands
NORM	R0,A R0,B

Table 6-18 Clear Instructions

Operation	Destination	Comments
CLR	X0, Y1, Y0, A1, B1, R0-R3, N	Identical to move #0,<reg>; does *not* set condition codes
CLR	A B	Clears 36-bit accumulator and *sets* condition codes

Table 6-19 Move Instructions

Operation	Src/dest	Dest/Src	Comments
MOVE(C)	X:xxxx X:(Rn) X:(Rn)+ X:(Rn)- X:(Rn)+N X:(Rn+N) X:(Rn+xxxx) X:(SP) X:(SP)+ X:(SP)- X:(SP)+N X:(SP+N) X:(SP+xxxx)	Any register	X:xxxx: long 16-bit absolute address xxxx: signed 16-bit offset

MOVE(C)	X:(R2+xx)
	X:(SP-xx)
		X0, Y1, Y0, A, B, A1, B1, R0-R3, N	

MOVE(C)	Any register	Any register	
MOVE(M)	P:(Rn)+ P:(Rn)+N	X0, Y1, Y0, A, B, A1, B1, R0-R3, N	Only instruction that directly addresses program memory
POP		Any register	A single stack location is popped into any register
		(No register specified)	A single stack location can also be popped with no destination specified

The MOVE(P) and MOVE(S) instructions in **Table 6-20** are a more efficient way to access the first and last 64 locations in X data memory. Immediate moves are done using MOVE(I). There is also a PUSH instruction macro, two words and two instruction cycles, discussed in **Appendix A Instruction Set Details**.

Example 6-11 Move Instructions

```
      MOVEC       X:$C00F,LA
      MOVE        A2,X:(R1+$1234)
      MOVE        X:(R0)+,B
      MOVE        B2,M01
      MOVEM       X0,P:(R2)+N
      POP         LC
```

Table 6-20 Immediate Move Instructions

Operation	Destination	Comments
MOVE(I) #xx,	X0, Y1, Y0, A, B, A1, B1, R0-R3, N	Signed 7-bit integer data (data is put in the LSP)

MOVE(I) #xxxx,	Any register
	X:xxxx
	X:(R2+xx)
	X:(SP-xx)
		Signed 16-bit immediate data to a register or memory location

MOVE(P) #xxxx,	X:<pp>	Signed 16-bit immediate data to the last 64 locations of X data memory
MOVE(S) #xxxx,	X:<aa>	Signed 16-bit immediate data to the first 64 locations of X data memory

Table 6-21 Move Peripheral Data Instructions

Operation	Src/dest	Dest/src	Comments
MOVE(P)[1]	X:<pp>	X0, Y1, Y0, A, B, A1, B1 R0-R3, N	<pp>: last 64 locations of X memory
Note: 1. The MOVE(P) instruction allows moving 16-bit immediate data as shown in**Table 6-20**.			

Table 6-22 Move Absolute Short Instructions

Operation	Src/Dest	Dest/Src	Comments
MOVE(S)	X:<aa>	X0, Y1, Y0, A, B, A1, B1 R0-R3, N	<aa>: first 64 locations of X memory
Note: 1. The MOVE(S) instruction allows moving 16-bit immediate data as shown in **Table 6-20**.			

Table 6-23 Test Word Instructions

Operation	Source	Comments
TSTW	x:<aa> x:<pp> X:(R2+xx) X:(SP-xx) X:xxxx X:(Rn) X:(Rn)+ X:(Rn)- X:(Rn)+N X:(Rn+N) X:(Rn+xxxx) X:(SP) X:(SP)+ X:(SP)- X:(SP)+N X:(SP+N) X:(SP+xxxx) Any register	<aa>: first 64 locations of X memory <pp>: last 64 locations of X memory X:xxxx: long 16-bit absolute address xxxx: signed 16-bit offset
TST(W)	(Rn)-	Test an Rn register and then post-decrement it.

Example 6-12 Test Word Instructions

```
TSTW         X:$FFF7
TSTW         (R3)-
```

Table 6-24 Conditional Register Transfer Instructions

Operation	Data ALU	AGU Unit
Tcc[1]	A,B B,A	(No transfer)
	X0,A Y1,A Y0,A X0,B Y1,B Y0,B	R0,R1
Note: 1. The Tcc instruction does not recognize the following condition codes: HI, LS, NN, and NR.		

Table 6-25 Bit-manipulation Instructions

Operation	Operand	Comments
BFSET #iiii BFCLR #iiii BFCHG #iiii BFTSTH #iiii BFTSTL #iiii ANDC #iiii ORC #iiii EORC #iiii NOTC[1]	X:(R2+xx)	xx = [0 to 63]
	X:(SP-xx)	xx = [1 to 64]
	X:<aa>	First 64 words of X memory
	X:<pp>	Last 64 words of X memory
	X:xxxx	16-bit absolute address
	Any register except HWS	
Note: 1. No immediate value is specified with the NOTC, because it simply performs a one's complement of the specified operand.		

Example 6-13 Bit-manipulation Instructions

```
      BFSET        #$80C0,X:$001A
      ANDC         #$7FFF,R2
      NOTC         X:(SP-7)
```

Table 6-26 Branch on Bit-field Instructions

Operation	Operand	Comments	Offset
BRSET #00ii BRCLR #00ii BRSET #ii00 BRCLR #ii00	X:(R2+xx)	xx = [0 to 63]	7-bit PC-relative address [-64,63]
	X:(SP-xx)	xx = [1 to 64]	
	X:<aa>	First 64 words of X memory	
	X:<pp>	Last 64 words of X memory	
	X:xxxx	16-bit absolute address	
	Any register except HWS		

Example 6-14 Branch on Bit-field Instructions

```
      BRSET        #$8100,X:$FFE7,LABEL1
      BRCLR        #$0055,X:$A097,LABEL2
```

Table 6-27 Jump And Branch Instructions

Operation	Operand	Comments
JMP Jcc	xxxxx	19-bit absolute address
JSR	xxxxx	19-bit absolute address
BRA Bcc	xx	7-bit PC relative address [-64,63]

Table 6-28 Effective Address Update

Operation	Source Register
LEA	(Rn)+ (Rn)- (Rn)+N
	(SP)+ (SP)- (SP)+N

Table 6-29 DO and REP Instructions

Operation	Operand	Comments
DO REP	#xx	6-bit unsigned short immediate data
	X0,Y1,Y0, R0,R1,R2,R3, N, LC, LA, A2,B2,A1,B1, A0,B0,A,B	Any register (uses 13 LSBs of register) except M01, SR, OMR, and HWS

Example 6-15 DO and REP Instructions

```
DO          #7,LOOP_LABEL1
DO          R2,LOOP_LABEL2
REP         #5
```

Table 6-30 Special Instructions

Operation
WAIT
STOP
RTI
RTS
NOP
ENDDO
SWI
DEBUG
ILLEGAL

6.7 THE INSTRUCTION PIPELINE

Instruction execution is pipelined to allow most instructions to execute at a rate of one instruction every clock cycle. However, certain instructions will require additional time to execute, including instructions with the following properties:

- Exceed length of one word
- Use an addressing mode that requires more than one cycle
- Access the program memory
- Cause a control flow change

In the case of a control flow change, a cycle is needed to clear the pipeline.

6.7.1 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 executed, 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 6-4 demonstrates pipelining; F1, D1 and E1 refer to the fetch, decode and execute operations, respectively, of the first instruction. Note that the third instruction contains an instruction extension word and takes two cycles to execute.

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

Figure 6-4 Pipelining

Each instruction requires a minimum of three instruction cycles (six machine cycles) to be fetched, decoded, and executed. A new instruction may be started after two machine cycles, making the throughput rate one instruction executed every instruction cycle for single cycle instructions. Two word instructions require a minimum of eight machine cycles to execute and a new instruction may start after four machine cycles.

6.7.2 Memory Access Processing

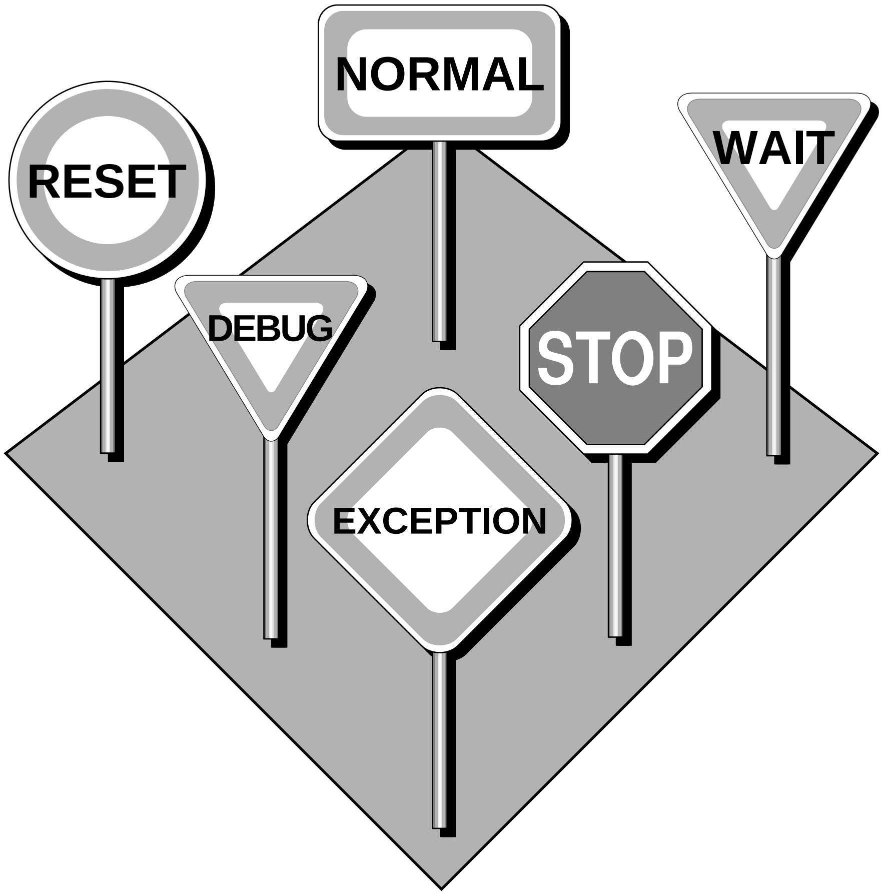
One or more of the DSP memory sources (X data memory and program memory) may be accessed during the execution of an instruction. Three address buses (XAB1, XAB2, and PAB) and three data buses (CGDB, XDB2, and PDB) are available for internal memory accesses during one instruction cycle, but only one address bus and one data bus are available for external memory accesses (when the external bus is available). If all memory sources are internal to the DSP, one or more of the two memory sources may be accessed in one instruction cycle (i.e., program memory access, or program memory access plus an X memory reference, or program memory access with two X memory references).

Note: For instructions that contain two X memory references, the second transfer using XAB2 and XDB2 may not access external memory. All accesses across these buses must access internal memory only.

See **Instruction Pipeline with Off-chip Memory Accesses** on page 7-5 for a discussion of off-chip memory accesses.

SECTION 7

INTERRUPTS AND THE PROCESSING STATES



7.1 INTRODUCTION

The DSP56800 Family processors have six processing states and are always in one of these states (see **Table 7-1**). Each processing state is described in detail below except the debug processing state, which is discussed in **OnCE Port** on page 9-7. In addition, special cases of interrupt pipelines are discussed at the end of the section. **Interrupts** on page 8-39 discusses software techniques for interrupt processing.

Table 7-1 Processing States

State	Description
Reset	The state where the DSP core is forced into a known reset state. Typically, the first program instruction is fetched upon exiting this state.
Normal	The state of the DSP core where instructions are normally executed.
Exception	The state of interrupt processing, where the DSP core transfers program control from its current location to an interrupt service routine using the interrupt vector table.
Wait	A low power state where the DSP core is shut down but the peripherals and interrupt machine remain active.
Stop	A low power state where the DSP core, the interrupt machine, and most (if not all) of the peripherals are shut down.
Debug	The state where the DSP core is halted and all registers in the On-Chip Emulation (OnCE) port of the processor are accessible for program debug.

7.2 RESET PROCESSING STATE

The processor enters the reset processing state when the external $\overline{\text{RESET}}$ pin is asserted and a hardware reset occurs. On devices with a computer operating properly (COP) timer, it is also possible to enter the reset processing state when this timer reaches zero. The DSP is typically held in reset during power-up by asserting the $\overline{\text{RESET}}$ pin, making this the first processing state entered by the DSP. The reset state performs the following:

1. Resets internal peripheral devices
2. Sets the M01 modifier register to $FFFF
3. Clears the interrupt priority register (IPR)

4. Sets the wait state fields in the bus control register (BCR) to their maximum value, thereby inserting the maximum number of wait states for all external memory accesses
5. Clears the status register's (SR) loop flag and condition code bits and sets the interrupt mask bits
6. Clears the following bits in the operating mode register: nested looping, condition codes, X/P memory, stop delay, rounding, and external X memory

The DSP remains in the reset state until the RESET pin is deasserted. When the processor deasserts the $\overline{\text{RESET}}$ pin the following occurs:

1. The chip operating mode bits of the OMR are loaded from the external mode select pins (MODA and MODB).
2. A delay of 16 instruction cycles (NOPs) occurs to sync local clock generator and state machine.
3. The chip begins program execution at program memory address defined by the state of the MODA and MODB bits in the OMR and the type of reset (hardware or COP time-out). The first instruction must be fetched and then decoded before executing. Therefore, the first instruction execution is two instruction cycles after the first instruction fetch.

After this last step, the DSP enters the normal processing state upon exiting reset. It is also possible for the DSP to enter the debug processing state upon exiting reset when system debug is underway.

7.3 NORMAL PROCESSING STATE

The normal processing state is the typical state of the processor where it executes instructions in a three-stage pipeline. This includes the execution of simple instructions such as moves or ALU operations, as well as jumps, hardware looping, bitfield instructions, instructions with parallel moves, etc. Details about the execution of the individual instructions can be found in **Appendix A Instruction Set Details**. The chip must be reset before it can enter the normal processing state.

7.3.1 Instruction Pipeline Description

The instruction execution pipeline is a three-stage pipeline, which allows most instructions to execute at a rate of one instruction per instruction cycle. For the case where there are no off-chip memory accesses, or for the case of a single off-chip access with no wait states, one instruction cycle is equivalent to two machine cycles. A machine cycle is defined as one cycle of the clock provided to the DSP core. Certain instructions, however, require more than one instruction cycle to execute; these instructions include the following:

- Instructions longer than one word
- Instructions using an addressing mode that requires more than one cycle
- Instructions that cause a control-flow change

Pipelining allows instruction executions to overlap so that the fetch-decode-execute operations of a given instruction occur concurrently with the fetch-decode-execute operations of other instructions. Specifically, while the processor is executing one instruction, it is decoding the next instruction, and fetching a third instruction from program memory. The processor fetches only one instruction word per instruction cycle; if an instruction is two words in length, it fetches the additional word with an additional cycle before it fetches the next instruction.

Table 7-2 Instruction Pipelining

Operation	**Instruction Cycle**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**•**	**•**	**•**
Fetch	F1	F2	F3	F3e	F4	F5	F6	•	•	•
Decode		D1	D2	D3	D3e	D4	D5	•	•	•
Execute			E1	E2	E3	E3e	E4	•	•	•

Table 7-2 demonstrates pipelining. “F1,” “D1,” and “E1” refer to the fetch, decode, and execute operations of the first instruction, respectively. The third instruction, which contains an instruction extension word, takes two instruction cycles to execute. Although it takes three instruction cycles (6 machine cycles) for the pipeline to fill and the first instruction to execute, an instruction usually executes on each instruction cycle thereafter (2 machine cycles).

7.3.2 Instruction Pipeline with Off-chip Memory Accesses

The three sets of internal on-chip address and data buses (XAB1 / CGDB, XAB2 / XDB2, PAB / PDB) allows for fast memory access when accessing memories on-chip. The DSP can perform memory accesses on all three bus pairs in a single instruction cycle, permitting the fetch of an instruction concurrently with up to two accesses to the X data memory. Thus, for applications where all program and data is located in on-chip memory, there is no speed penalty when performing up to three memory accesses in a single instruction.

Similarly, the external address and data bus also allow for fast program execution. For the case where only program memory is external to the chip or only X data memory is external (XAB1 / CDGB bus pair), the DSP chip will still execute programs at full speed if there are no wait states programmed on the external bus by the user. For the case where an instruction requires an external program fetch *and* an external X data memory access simultaneously, the instruction will still operate correctly. The instruction is automatically stretched an additional instruction cycle so that the two external accesses may be performed correctly, and wait states are inserted accordingly. All this occurs transparently to the user to allow for easier program development.

This information is summarized in **Table 7-3**, which shows how the chip automatically inserts instruction cycles and wait states for an instruction that is simultaneously accessing program and data memory. For dual parallel read instructions the second X memory access that uses XAB2 / XDB2 must always be done to on-chip memory. This second access may never access external off-chip memory.

Table 7-3 Additional Cycles for Off-chip Memory Accesses

Memory Space			Number of Additional Cycles	Comments
Program Fetch	X Memory 1st Access	X Memory 2nd Access		
On-chip	On-chip	On-chip	0	All accesses internal
External	On-chip	On-chip	0 + mvm	One external access
On-chip	External	On-chip	0 + mv	One external access
External	External	On-chip	1 + mv + mvm	Two external accesses

7.3.3 Instruction Pipeline Dependencies and Interlocks

The pipeline is normally transparent to the user. However, there are certain instruction-sequence combinations where the pipeline will affect the program execution. Such situations are best described by case studies. Most of these restricted sequences occur because either all addresses are formed during instruction decode or they are the result of contention for an internal resource such as the SR.

If the execution of an instruction depends on the relative location of the instruction in a sequence of instructions, there is a pipeline effect.

It is possible to see if there is a pipeline dependency. To test for a suspected pipeline effect, compare between the execution of the suspect instruction when it directly follows the previous instruction and when four NOPs are inserted between the two. If there is a difference, it is caused by a pipeline effect. The assembler flags instruction sequences with potential pipeline effects so that the user can determine if the operation will execute as expected.

Example 7-1 Pipeline Dependencies in Similar Code Sequences

No Pipeline Effect

```
      ORC #$0001,SR      ; Changes carry bit at the end of execution
                         ; time slot
      JCS LABEL          ; Reads condition codes in SR in its execution
                         ; time slot
```

The JCS instruction will test the carry bit modified by the ORC without any pipeline effect in the code segment above.

Pipeline Effect

```
      ORC #$0008,OMR     ; Sets EX bit at execution time slot
      MOVE X:$17,A       ; Reads internal memory instead of external
                         ; memory
```

A pipeline effect occurs because the address of the MOVE is formed at its decode time before the ORC changes the EX bit (which changes the memory map) in the ORI’s execution time slot. The following code produces the expected results of reading the external ROM:

```
      ORC #$0008,OMR     ; Sets EX bit at execution time slot
      NOP                ; Delays the MOVE so it will read the updated
                         ; memory map
      MOVE X:$17,A       ; Reads external memory
```

Example 7-2 Common Pipeline Dependency Code Sequence

```
MOVE X0,R2        ; Move a value into register R2
MOVE X:(R2),A     ; Uses the OLD contents of R2 to address
                  ; memory.
```

In this case before the first MOVE instruction has written R2 during its execution cycle, the second MOVE has accessed the old R2, using the old contents of R2. This is because the address for indirect moves is formed during the decode cycle. This overlapping instruction execution in the pipeline causes the pipeline effect.

One instruction cycle should be allowed after an address register has been written by a MOVE instruction before the new contents are available for use as an address register by another MOVE instruction. The proper instruction sequence is listed below:

```
MOVE X0,R2        ; Move a number into register R2
NOP               ; Execute any instruction or instruction
                  ; sequence not using the R2 register written
                  ; in the previous instruction
MOVE X:(R2),A     ; Use the new contents of R2
```

Pipeline Dependencies on page 4-36 contains more details on interlocks caused during address generation.

7.4 EXCEPTION PROCESSING STATE

The exception processing state is the state where the DSP core recognizes and processes interrupts that can be generated by conditions inside the DSP or from external sources. Upon the occurrence of an event, interrupt processing transfers control from the currently executing program to an interrupt service routine with the ability to later return to the current program upon completion of the interrupt service routine. In digital signal processing, some of the main uses of interrupts are to transfer data between DSP memory and a peripheral device or to begin execution of a DSP algorithm upon reception of a new sample. An interrupt can also be used to exit the DSP’s low power wait processing state.

An interrupt will cause the processor to enter the exception processing state. Upon entering this state, the current instruction in decode executes normally. The next fetch address is supplied by the interrupt controller and points into the interrupt vector table (**Table 7-4** on page 7-11). During this fetch the PC is not updated. The instruction located at these two addresses in the interrupt vector table must always be to a two-word unconditional jump to subroutine instruction (JSR). Note that the interrupt controller only fetches the second word of the JSR instruction. This results in the program changing flow to an interrupt routine, and a context switch is performed.

There are many sources for interrupts on the DSP56800 family of chips, and some of these sources can generate more than one interrupt. Interrupt requests can be generated from conditions within the DSP core, from the DSP peripherals, or from external pins. The DSP core features a prioritized interrupt vector scheme with up to 64 vectors to provide faster interrupt servicing. The interrupt priority structure is discussed in **Interrupt Priority Structure** on page 7-12.

7.4.1 Sequence of Events in the Exception Processing State

The following steps occur in exception processing:

1. A request for an interrupt is generated either on a pin, from the DSP core, from a peripheral on the DSP chip, or from an instruction executed by the DSP core. Any hardware interrupt request from a pin is first synchronized with the DSP clock.
2. The request for an interrupt by a particular source is latched in an interrupt pending flag if it is an edge or non-maskable interrupt (all other interrupts are not latched and must remain asserted in order to be serviced). For peripherals that can generate more than one interrupt request and have more than one interrupt vector, the interrupt arbiter only sees one request from the peripheral active at a time.
3. All pending interrupt requests are arbitrated to select which interrupt will be processed. The arbiter automatically ignores any interrupts with an interrupt priority level (IPL) lower than the interrupt mask level specified in the SR. If there are any remaining requests, the arbiter selects the remaining interrupt with the highest IPL, and the chip enters the exception processing state (see **Figure 7-1** on page 7-10).
4. The interrupt controller then freezes the program counter (PC) and fetches the JSR instruction located at the two interrupt vector addresses associated with the selected interrupt. It is required that the instruction located at the interrupt vector address must be a two-word JSR instruction. Note that only the second word of the JSR instruction is fetched; the first word of the JSR is provided by the interrupt controller.
5. The interrupt controller places this JSR instruction into the instruction stream and then releases the PC, which is used for the next instruction fetch. Arbitration among the remaining interrupt requests is allowed to resume. The next interrupt arbitration then begins.
6. The execution of the JSR instruction stacks the PC and the SR as it transfers control to the first instruction in the interrupt service routine. These two

stacked registers contain the 19-bit return address that will later be used to return to the interrupted code. In addition, the IPL is raised to level 1 to disallow any level 0 interrupts. Note that the OnCE trap, stack error, illegal instruction, and SWI can still generate interrupts because these are level 1 interrupts and are non-maskable.

The exception processing state is completed when the processor executes the JSR instruction located in the interrupt vector table and the chip enters the normal processing state. As it enters the normal processing state, it begins executing the first instruction in the interrupt service routine. Each interrupt service routine should return to the main program by executing an RTI instruction.

Interrupt routines for level 0 interrupts are interruptible by higher priority interrupts. **Figure 7-1** shows an example of processing an interrupt.

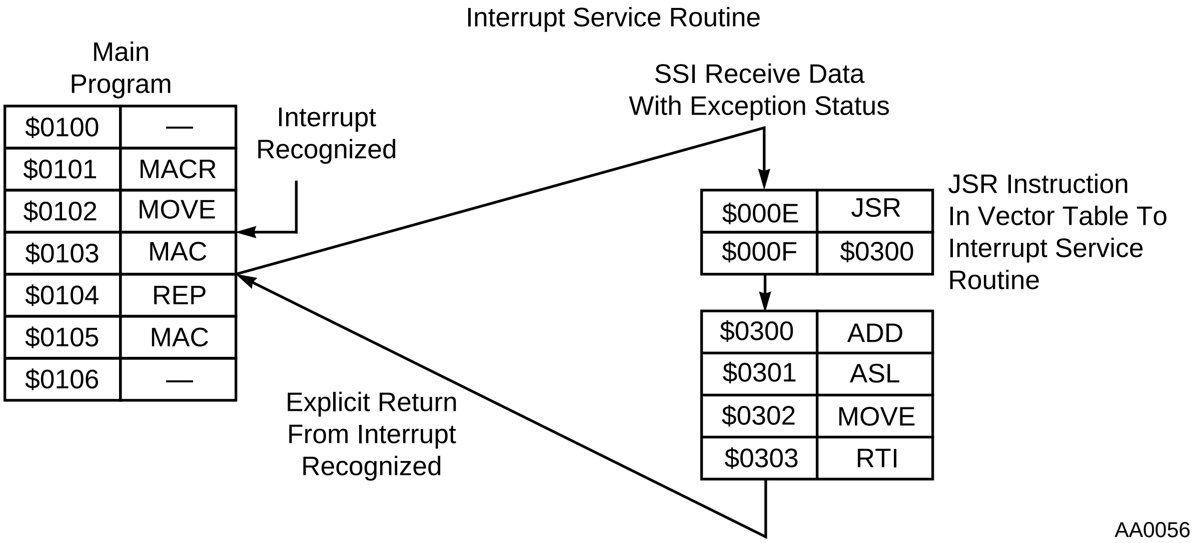

Interrupt Service Routine
Main Program
$0100 —
$0101 MACR
$0102 MOVE
$0103 MAC
$0104 REP
$0105 MAC
$0106 —
Interrupt Recognized
SSI Receive Data With Exception Status
$000E JSR
$000F $0300
JSR Instruction In Vector Table To Interrupt Service Routine
$0300 ADD
$0301 ASL
$0302 MOVE
$0303 RTI
Explicit Return From Interrupt Recognized
AA0056

Figure 7-1 Interrupt Processing

Steps 1 through 3 listed on page 7-9 require two additional instruction cycles, effectively making the interrupt pipeline five levels deep.

7.4.2 Reset and Interrupt Vector Table

The interrupt vector table specifies the addresses that the processor accesses once it recognizes an interrupt and begins exception processing. Since peripherals can also generate interrupts, the interrupt vector map for a given chip is specified by all sources on the DSP core as well as all peripherals that can generate an interrupt.

Table 7-4 lists the reset and interrupt vectors available on DSP56800-based DSP chips. The interrupt vectors used by the on-chip peripherals of a DSP chip will be listed in the user's manual for that chip.

Table 7-4 DSP56800 Core Reset and Interrupt Vector Table

Interrupt Starting Address	Interrupt Priority Level	Interrupt Source
$0000	-	Hardware Reset
$0002	-	COP Watchdog Reset
$0004	-	(Reserved)
$0006	1	Illegal Instruction Trap
$0008	1	SWI
$000A	1	Hardware Stack Overflow
$000C	1	OnCE Trap
$000E	1	(Reserved)
$0010	0	IRQA
$0012	0	IRQB
$0014	0	(Vector Available for On-chip Peripherals)
$0016	0	(Vector Available for On-chip Peripherals)
$0018	0	(Vector Available for On-chip Peripherals)
$001A	0	(Vector Available for On-chip Peripherals)
$001C	0	(Vector Available for On-chip Peripherals)
$001E	0	(Vector Available for On-chip Peripherals)
$0020	0	(Vector Available for On-chip Peripherals)
$0022	0	(Vector Available for On-chip Peripherals)
$0024	0	(Vector Available for On-chip Peripherals)
$0026	0	(Vector Available for On-chip Peripherals)
$0028	0	(Vector Available for On-chip Peripherals)
$002A	0	(Vector Available for On-chip Peripherals)
$002C	0	(Vector Available for On-chip Peripherals)
$002E	0	(Vector Available for On-chip Peripherals)
$0030	0	(Vector Available for On-chip Peripherals)
$0032	0	(Vector Available for On-chip Peripherals)
$0034	0	(Vector Available for On-chip Peripherals)

Table 7-4 DSP56800 Core Reset and Interrupt Vector Table (Continued)

Interrupt Starting Address	Interrupt Priority Level	Interrupt Source
$0036	0	(Vector Available for On-chip Peripherals)
$0038	0	(Vector Available for On-chip Peripherals)
$003A	0	(Vector Available for On-chip Peripherals)
$003C	0	(Vector Available for On-chip Peripherals)
$003E	0	(Vector Available for On-chip Peripherals)
$0040	0	(Vector Available for On-chip Peripherals)
$0042	0	(Vector Available for On-chip Peripherals)
...		
$007C	0	(Vector Available for On-chip Peripherals)
$007E	0	(Vector Available for On-chip Peripherals)

It is required that a two-word JSR instruction is present in any interrupt vector location that may be fetched during exception processing. If an interrupt vector location is unused, then the JSR instruction is not required.

The hardware reset and COP reset are special cases because they are reset vectors, not interrupt vectors. There is no IPL specified for these two because these conditions reset the chip and reset takes precedence over any interrupt. Typically a two-word JMP instruction is used in the reset vectors. The hardware reset vector will either be at address $0000 or $E000 and the COP reset vector will either be at $0002 or $E002 depending on the operating mode of the chip. The operating mode is defined by the MODA and MODB pins immediately out of reset, as discussed in **Operating Mode Bits (MB and MA)—Bits 0-1** on page 5-15.

7.4.3 Interrupt Priority Structure

Interrupts are organized in a simple priority structure. Each interrupt source has an associated IPL: Level 0 or Level 1. Level 0, the lowest level, is maskable, and Level 1 is non-maskable. **Table 7-5** summarizes the priority levels and their associated interrupt sources.

Table 7-5 Interrupt Priority Level Summary

IPL	Description	Interrupt Sources
0	Maskable	On-chip peripherals, $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$
1	Non-maskable	Illegal instruction, OnCE trap, HWS overflow, SWI

The interrupt mask bits (I1, I0) in the SR reflect the current priority level and indicate the IPL needed for an interrupt source to interrupt the processor (see **Table 7-6**). Interrupts are inhibited for all priority levels below the current processor priority level. Level 1 interrupts, however, are not maskable and, therefore, can always interrupt the processor.

Table 7-6 Interrupt Mask Bit Definition in the Status Register

I1	I0	Exceptions Permitted	Exceptions Masked
0	0	(Reserved)	(Reserved)
0	1	IPL 0, 1	None
1	0	(Reserved)	(Reserved)
1	1	IPL 1	IPL 0

7.4.4 Interrupt Priority Register (IPR)

The interrupt unit on the DSP56800 core supports seven interrupt channels for use by on-chip peripherals in addition to the $\overline{\text{IRQ}}$ interrupts and interrupts generated by the DSP core. Each maskable interrupt source can individually be enabled or disabled in the IPR on the chip.

The interrupt priority register (IPR) is a read / write memory mapped register located at X:$FFFB. It specifies the IPL for each of the interrupting devices including $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$ and each peripheral device. (For specific peripheral information, see the appropriate user's manual for the desired chip.) The IPL for each on-chip peripheral device (channels 0 - 6) and for each external interrupt source ($\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$) can be enabled or disabled under software control. In addition, it specifies the trigger mode of the external interrupt sources and is used to enable or disable the individual external interrupts. The IPR is cleared on $\overline{\text{RESET}}$. All unused bits (bits 8, 7, 6, 3, 0) are read as 0 and must be written with a “0” to ensure future compatibility and compatibility with other family members. **Figure 7-2** shows the IPR and **Figure 7-3** shows how it is programmed for different interrupts on the DSP56800 chip.

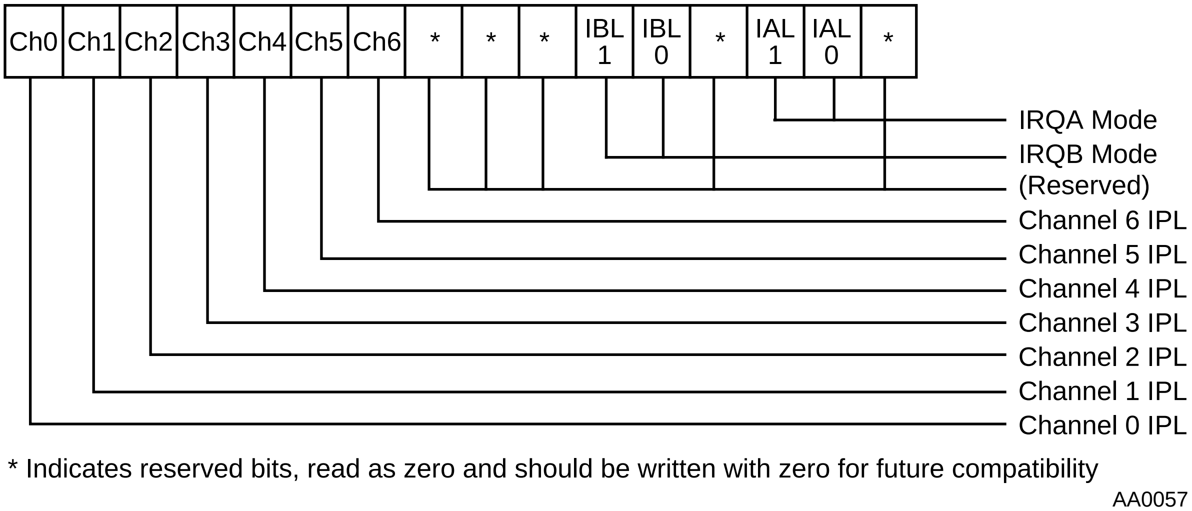


Figure 7-2 Interrupt Priority Register, X$FFFB

Chx	Enabled?	IPL
0	No	—
1	Yes	0

IBL0 IAL0	Enabled?	IPL
0	No	—
1	Yes	0

IBL1 IAL1	Trigger Mode
0	Level-sensitive
1	Edge-sensitive

AA0058

Figure 7-3 On-chip Peripheral and $\overline{\text{IRQ}}$ Interrupt Programming

7.4.5 Interrupt Sources

An interrupt request is a request to break out of currently executing code to enter an interrupt service routine. Interrupt requests in the DSP are generated from one of three sources: external hardware, internal hardware, and internal software. The internal hardware interrupt sources include all of the on-chip peripheral devices.

Each interrupt source has at least one associated interrupt vector, and some sources may have several interrupt vectors. The interrupt vector addresses for each interrupt source are listed in the interrupt vector table (**Table 7-4**). These addresses are usually located in either the first 64 or 128 locations of program memory. For further information on a device's on-chip peripheral interrupt sources, see the device's individual user's manual.

When an interrupt request is recognized and accepted by the DSP core, a two-word JSR instruction is fetched from the interrupt vector table. Because the program flow is directed to a different starting address within the table for each different interrupt, the interrupt structure can be described as "vectored." A vectored interrupt structure has low execution overhead. If it is known beforehand that certain interrupts will not be used or enabled, those locations within the table can instead be used for program or data storage.

7.4.5.1 External Hardware Interrupt Sources

The external hardware interrupt sources are listed below:

- $\overline{\text{RESET}}$ pin
- $\overline{\text{IRQA}}$ pin—priority level 0
- $\overline{\text{IRQB}}$ pin—priority level 0

An assertion of the $\overline{\text{RESET}}$ is not truly an interrupt, but rather it forces the chip into the reset processing state. Likewise, for any DSP chip that contains a COP timer, a time-out on this timer can also place the chip into the reset processing state. The reset processing state is at the highest priority and takes precedence over any interrupt, including an interrupt in progress.

Assertions on the $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ pins generate $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ interrupts, which are priority level 0 interrupts and are individually maskable. The $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ interrupt pins are internally synchronized with the processor's internal clock and can be programmed as level-sensitive or edge-sensitive.

Edge-sensitive interrupts are latched as pending when a falling edge is detected on an $\overline{\text{IRQ}}$ pin. The $\overline{\text{IRQ}}$ pin's interrupt pending bit remains set until its associated interrupt is recognized and serviced by the DSP core. Edge-sensitive interrupts are automatically cleared when the interrupt is recognized and serviced by the DSP core. In an edge-sensitive interrupt the interrupt pending bit is automatically cleared when the second vector location is fetched.

Level-sensitive interrupts, on the other hand, are never latched but go directly into the interrupt controller. A level-sensitive interrupt is examined and processed when the $\overline{\mathrm{IRQ}}$ pin is low and the interrupt arbiter allows this interrupt to be recognized. Since there is no interrupt pending bit associated with level-sensitive interrupts, the interrupt cannot not be cleared automatically when serviced; instead, it must be explicitly cleared by other means to prevent multiple interrupts.

Note: On all level-sensitive interrupts, the interrupt must be externally released before interrupts are internally re-enabled. Otherwise, the processor will be interrupted repeatedly until the release of the level-sensitive interrupt.

When either the $\overline{\mathrm{IRQA}}$ or $\overline{\mathrm{IRQB}}$ pin is disabled in the IPR, any interrupt request on its associated pin is ignored, regardless of whether the input was defined as level-sensitive or edge-sensitive. If the interrupt input is defined as edge-sensitive, its interrupt pending bit will remain in the reset state for as long as the interrupt pin is disabled. If the interrupt is defined as level-sensitive, its edge-detection latch will stay in the reset state. If the level-sensitive interrupt is disabled while it is pending, it will be cancelled. However, if the interrupt has been fetched, it normally will not be cancelled.

The level-sensitive interrupt capability is useful for the case where there is more than one external interrupt source, yet only one IRQ pin is available. In this case the interrupts are wire ORed onto a single $\overline{\mathrm{IRQ}}$ pin with a resistor pull-up, and any one of these can assert an interrupt. It is important that the interrupt service routine poll each device, and, after finding the source of the interrupt, it must clear the conditions causing the interrupt request.

7.4.5.2 DSP Core Hardware Interrupt Sources

Other interrupt sources include the following:

- Stack error interrupt—priority level 1
- OnCE trap—priority level 1
- All on-chip peripherals (such as timers and serial ports)—priority level 0

An overflow of the hardware stack (HWS) causes a stack overflow interrupt that is vectored to P:$000A (see **Hardware Stack** on page 5-9). Encountering the stack overflow condition means that too many DO loop addresses have been stacked and that the oldest top-of-loop address has been lost. The stack error is non-recoverable. The stack error condition refers to hardware stack overflow and does not affect the software stack pointed to by the stack pointer (SP) register in any manner.

The OnCE trap interrupt is an interrupt that can be setup in the OnCE debug port accessible through the JTAG pins. This gives the debug port the capability to generate an interrupt on a trigger condition such as the matching of an address in the OnCE port (see **OnCE Port** on page 9-7 for more information).

In addition to these sources there are seven general-purpose interrupt channels, Ch0 through Ch6, available for use by on-chip peripherals such as timers and serial ports. Each channel can independently generate an interrupt request, each can be individually masked, and each channel can have one or more dedicated locations in the interrupt vector table. Typically, one channel is assigned to each on-chip peripheral, but, in cases where there are more than seven peripherals that can generate interrupts, it is possible to put more than one peripheral on a single interrupt channel.

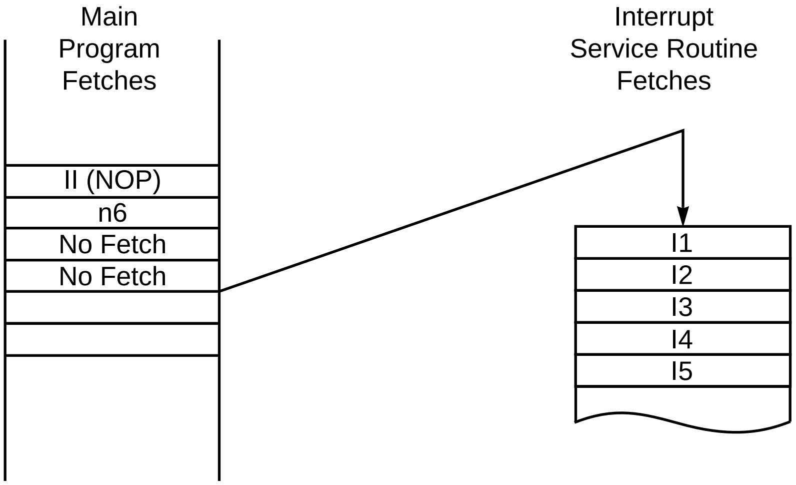
7.4.5.3 DSP Core Software Interrupt Sources

The two software interrupt sources are listed below:

- Software interrupt (SWI)—priority level 1
- Illegal instruction interrupt (Ill)— priority level 1

A SWI is a non-maskable interrupt that is serviced immediately following the SWI instruction execution (i.e., no other instructions are executed between the SWI instruction and the JSR instruction found in the interrupt vector table). The difference between an SWI and a JSR instruction is that the SWI sets the interrupt mask to prevent level 0-maskable interrupts from being serviced. The SWI’s ability to mask out lower level interrupts makes it very useful for setting breakpoints in monitor programs or for making a system call in a simple operating system. The JSR instruction does not affect the interrupt mask.

The illegal instruction interrupt is also a non-maskable interrupt (priority level 1). It is serviced immediately following the execution or attempted execution of an illegal instruction (an undefined operation code). Illegal exceptions are fatal errors. The JSR located in the illegal instruction interrupt vector will stack the address of the instruction immediately after the illegal instruction.



(a) Instruction Fetches from Memory

Illegal Instruction Interrupt Recognized As Pending

Illegal Instruction Interrupt Recognized As Pending

Interrupt Control Cycle 1								i						
Interrupt Control Cycle 2									i					
Fetch		n1	n2	n3	n4	II	n6	—	—	ii1	ii2	ii3	ii4	ii5
Decode			n1	n2	n3	n4	II	—	—	—	ii1	ii2	ii3	ii4
Execute				n1	n2	n3	n4	NOP	—	—	—	ii1	ii2	ii3
Instruction Cycle Count	1	2	3	4	5	6	7	8	9	10	11	12	13	14

i = Interrupt
ii = Interrupt Instruction Word
II = Illegal Instruction
n = Normal Instruction Word

(b) Program Controller Pipeline

AA0059

Figure 7-4 Illegal Instruction Interrupt Servicing

This interrupt can be used as a diagnostic tool to allow the programmer to examine the stack and locate the illegal instruction, or the application program can be restarted with the hope that the failure was a soft error. The ILLEGAL instruction, found in **Appendix A Instruction Set Details**, is useful for testing the illegal interrupt service routine to verify that it can recover correctly from an illegal instruction. Note that the illegal instruction trap does not fire for all invalid opcodes.

7.4.6 Interrupt Arbitration

Interrupt arbitration and control, which occurs concurrently with the fetch-decode-execute cycle, takes two instruction cycles. External interrupts are internally synchronized with the processor clock before their interrupt-pending flags are set. Each external and internal interrupt has its own flag. After each instruction is executed the DSP arbitrates all interrupts. During arbitration, each pending interrupt's IPL is compared with the interrupt mask in the SR, and the interrupt is either allowed or disallowed. The remaining pending interrupts are prioritized according to the IPLs shown in **Table 7-7**, and the interrupt source with the highest priority is selected. The interrupt vector corresponding to that source is then placed on the program address bus so that the program controller can fetch the interrupt instruction.

Table 7-7 Fixed Priority Structure Within an IPL

Priority	Exception	Enabled By
Level 1 (Non-maskable)		
Highest	Hardware $\overline{\text{RESET}}$	—
	Watchdog timer reset	—
	Illegal instruction	—
	HWS overflow	—
	OnCE trap	—
Lower	SWI	—
Level 0 (Maskable)		
Higher	IRQA (External interrupt)	IPR bit 1
	$\overline{\text{IRQB}}$ (External interrupt)	IPR bit 4
	Channel 6 peripheral interrupt	IPR bit 9
	Channel 5 peripheral interrupt	IPR bit 10
	Channel 4 peripheral interrupt	IPR bit 11
	Channel 3 peripheral interrupt	IPR bit 12
	Channel 2 peripheral interrupt	IPR bit 13
	Channel 1 peripheral interrupt	IPR bit 14
Lowest	Channel 0 peripheral interrupt	IPR bit 15

Interrupts from a given source are not buffered. The processor will not arbitrate a new interrupt from the same source until after it fetches the second word of the interrupt vector of the current interrupt.

An internal interrupt acknowledge signal clears the appropriate interrupt pending flag for DSP core interrupts. Some peripheral interrupts may also be cleared by the internal interrupt acknowledge signal, as defined in their specifications. Peripheral interrupt requests that need a read / write action to some register do not receive the internal interrupt acknowledge signal, and their interrupt request will remain pending until their registers are read / written. Further, if the interrupt comes from an IRQ pin and is programmed as level-triggered, the interrupt request will not be cleared. The acknowledge signal will be generated after the interrupt vectors have been generated, not before.

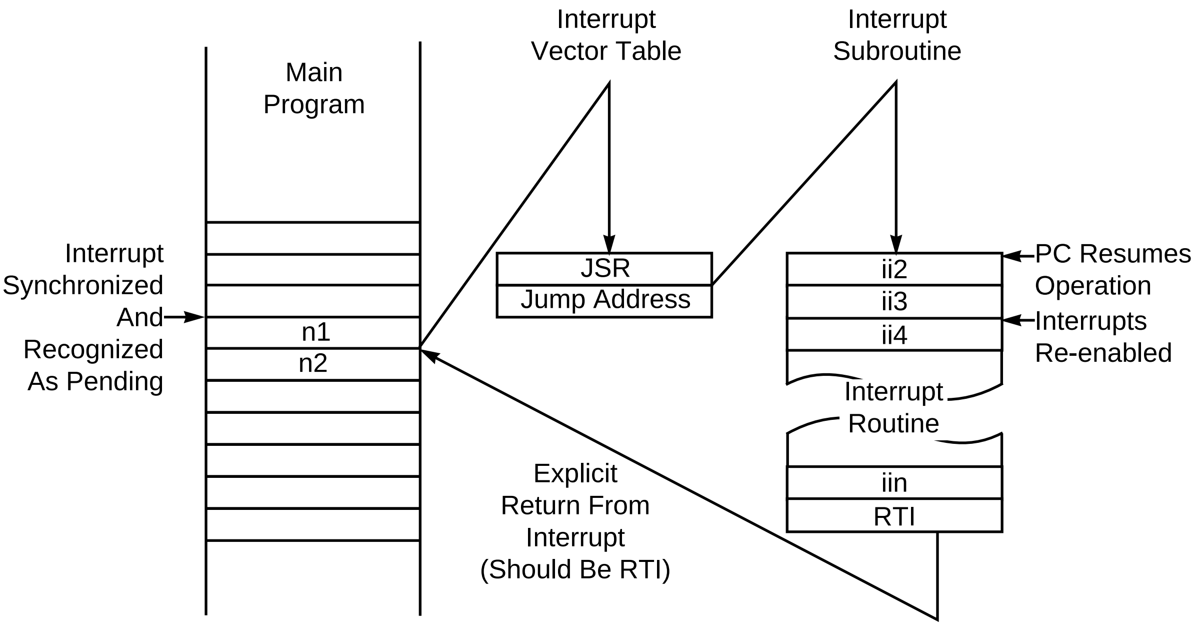
If more than one interrupt is pending when an instruction is executed, the processor will service the interrupt with the highest priority level first. When multiple interrupt requests with the same IPL are pending, a second fixed-priority structure within that IPL determines which interrupt the processor will service. For two interrupts programmed at the same priority level (non-maskable or level 0), **Table 7-7** shows the exception priorities within the same priority level. This table only applies when two interrupts arrive simultaneously or where two interrupts are simultaneously pending.

Whenever a level 0 interrupt has been recognized and exception processing begins, the DSP56800 interrupt controller changes the interrupt mask bits in the program controller's SR to allow only level 1 interrupts to be recognized. This prevents another level 0 interrupt from interrupting the interrupt service routine in progress. If an application requires that a level 0 interrupt can interrupt the current interrupt service routine, it is necessary to use one of the techniques in **Setting Interrupt Priorities in Software** on page 8-39.

7.4.7 The Interrupt Pipeline

The interrupt controller generates an interrupt instruction fetch address, which points to the second instruction word of a two-word JSR instruction located in the interrupt vector table. This address is used instead of the PC for the next instruction fetch. While the interrupt instructions are being fetched, the PC is loaded with the address of the interrupt service routine contained within the JSR instruction. After the interrupt vector has been fetched, the PC is used for any subsequent instruction fetches and the interrupt is guaranteed to be executed.

Upon executing the JSR instruction fetched from the interrupt vector table, the processor enters the appropriate interrupt service routine and exits the exception processing state. The instructions of the interrupt service routine are executed in the normal processing state and the routine is terminated with an RTI instruction. The RTI instruction restores the PC to the program originally interrupted, and the SR to its contents before the interrupt occurred. Then program execution resumes. **Figure 7-5** show the interrupt service routine. The interrupt service routine must be told to return to the main program by executing an RTI instruction.



(a) Instruction Fetches from Memory

Interrupt Synchronized And Recognized As Pending

Interrupts Re-enabled

Interrupt Control Cycle 1	i																	
Interrupt Control Cycle 2		i																
Fetch	n1	n2	—	Adr	—	ii2	ii3	ii4	ii5	iin	RTI	—	—	—	—	n2	—	—
Decode		n1	JSR	JSR	JSR	JSR	ii2	ii3	ii4	ii5	iin	RTI	RTI	RTI	RTI	RTI	n2	—
Execute			n1	JSR	JSR	JSR	JSR	ii2	ii3	ii4	ii5	iin	RTI	RTI	RTI	RTI	RTI	n2
Instruction Cycle Count	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

i = Interrupt
ii = Interrupt Instruction Word
n = Normal Instruction Word

(b) Program Controller Pipeline

AA0069

Figure 7-5 Interrupt Service Routine

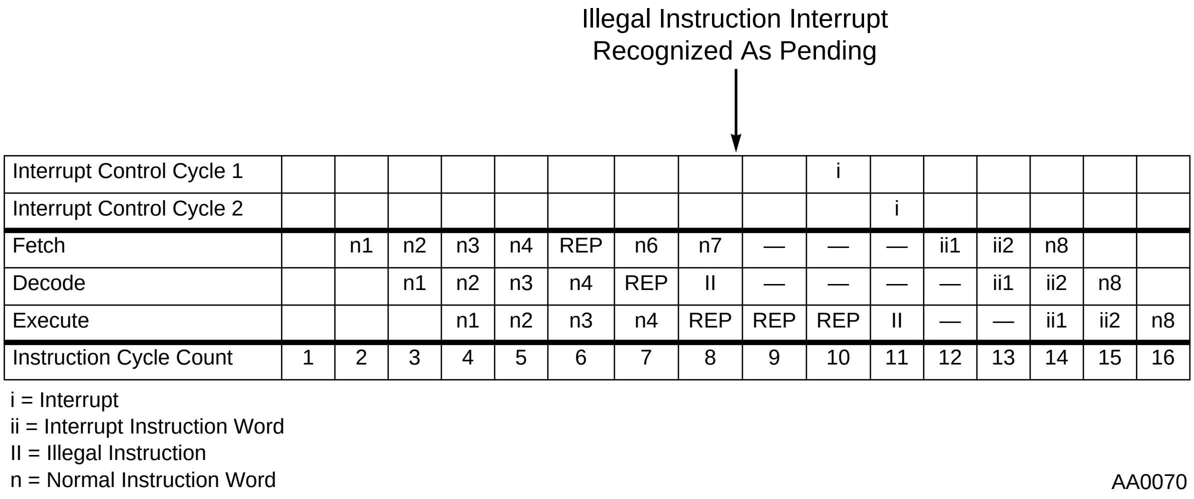
The execution of an interrupt service routine always conforms to the following rules:

1. A JSR to the starting address of the interrupt service routine is located at the first of two interrupt vector addresses.
2. The interrupt mask bits of the SR are updated to mask level 0 interrupts.
3. The first instruction word of the next interrupt service (of higher IPL) will reach the decoder only after the decoding of at least four instructions following the decoding of the first instruction of the previous interrupt.
4. The interrupt service routine can be interrupted (i.e., nested interrupts are supported).
5. The interrupt routine, which can be any length, should be terminated by an RTI, which restores the PC and SR from the stack.

Figure 7-5 demonstrates the interrupt pipeline. The point at which interrupts are re-enabled and subsequent interrupts are allowed is shown to illustrate the non-interruptible nature of the early instructions in the long interrupt service routine.

Reset is a special exception, which will normally contain only a JMP instruction at the exception start address.

The is only one case in which the stacked address will not point to the illegal instruction. If the illegal instruction follows an REP instruction (see **Figure 7-6**), the processor will effectively execute the illegal instruction as a repeated NOP and the interrupt vector will then be inserted in the pipeline. The next instruction will be fetched, decoded, and executed normally.



Illegal Instruction Interrupt
Recognized As Pending

Interrupt Control Cycle 1										i						
Interrupt Control Cycle 2											i					
Fetch		n1	n2	n3	n4	REP	n6	n7	—	—	—	ii1	ii2	n8		
Decode			n1	n2	n3	n4	REP	II	—	—	—	—	ii1	ii2	n8	
Execute				n1	n2	n3	n4	REP	REP	REP	II	—	—	ii1	ii2	n8
Instruction Cycle Count	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

i = Interrupt
ii = Interrupt Instruction Word
II = Illegal Instruction
n = Normal Instruction Word

AA0070

Figure 7-6 Repeated Illegal Instruction

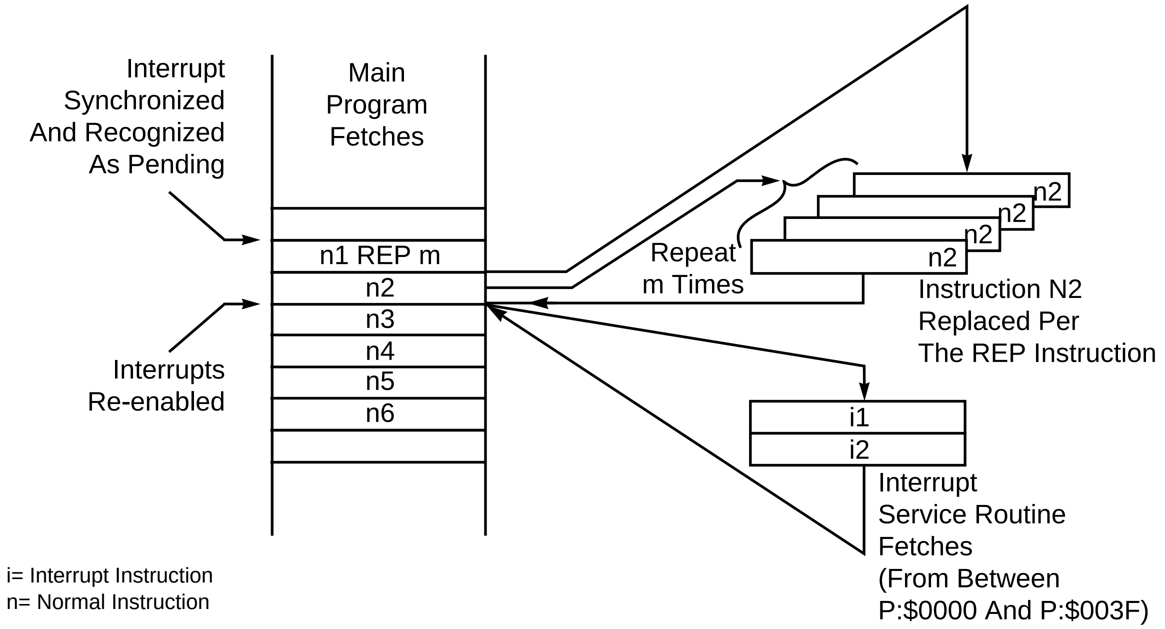
In DO loops, if the illegal instruction is in the loop address (LA) location and the instruction preceding it (i.e., at LA-1) is being interrupted, the loop counter (LC) will be decremented as if the loop had reached the LA instruction. When the interrupt service ends and the instruction flow returns to the loop, the instruction after the illegal instruction will be fetched (since it is the next sequential instruction in the flow).

7.4.8 Interrupt Latency

Interrupt latency represents the time between an interrupt request first appears and when the first instruction in an interrupt service routine is actually executed. The interrupt can only take place on instruction boundaries, and so the length of execution on an instruction affects interrupt latency.

There are some special cases to consider. The following instructions are not interruptible: SWI, STOP, and WAIT. Likewise, the REP instruction and the instruction it repeats are not interruptible.

A REP instruction and the instruction that follows it are treated as a single two-word instruction, regardless of how many times it repeats the second instruction of the pair. Instruction fetches are suspended and will be reactivated only after the LC is decremented to one (see **Figure 7-7**). During the execution of n2 in **Figure 7-7**, no interrupts will be serviced. When LC finally decrements to one, the fetches are re-initiated, and pending interrupts can be serviced.


Interrupt Synchronized And Recognized As Pending
Main Program Fetches
n1 REP m
n2
n3
n4
n5
n6
Interrupts Re-enabled
Repeat m Times
n2
n2
n2
n2
Instruction N2 Replaced Per The REP Instruction
i1
i2
Interrupt Service Routine Fetches (From Between P:$0000 And P:$003F)
i= Interrupt Instruction
n= Normal Instruction

(a) Instruction Fetches from Memory

Interrupt Synchronized And Recognized As Pending

Interrupts Re-enabled

Interrupt Control Cycle 1	i						i					
Interrupt Control Cycle 2		i%						i				
Fetch	REP	n2			n3				ii1	ii2	n5	n6
Decode		REP	REP	REP	n2	n2	n2	n2	JSR	JSR	JSR	JSR
Execute			REP	REP	REP	n2	n2	n2	n2	JSR	JSR	JSR
Instruction Cycle Count	1	2	3	4	5	6	7	8	9	10	11	12

i = Interrupt
ii = Interrupt Instruction Word
n = Normal Instruction Word
i% = Interrupt Rejected

(b) Program Controller Pipeline

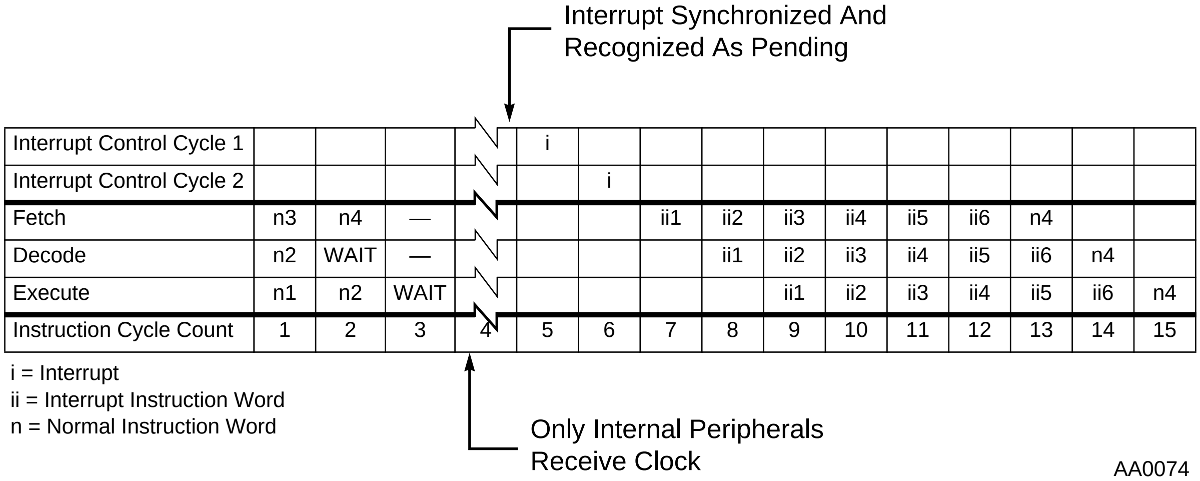
AA0071

Figure 7-7 Interrupting a REP Instruction

7.5 WAIT PROCESSING STATE

The wait instruction brings the processor into the wait processing state, which is one of two low power-consumption states. Asserting any valid interrupt request higher than the current processing level releases the DSP from the wait state. In the wait state the internal clock is disabled from all internal circuitry except the internal peripherals. All internal processing is halted until an unmasked interrupt occurs, or the DSP is reset.

Figure 7-8 shows a wait instruction being fetched, decoded, and executed. It is fetched as n3 in this example and, during decode, is recognized as a wait instruction. The following instruction (n4) is aborted, and the internal clock is disabled from all internal circuitry except the internal peripherals. The processor stays in this state until an interrupt or reset is recognized. The response time is variable due to the timing of the interrupt with respect to the internal clock.



Interrupt Synchronized And Recognized As Pending

Interrupt Control Cycle 1					i										
Interrupt Control Cycle 2						i									
Fetch	n3	n4	—				ii1	ii2	ii3	ii4	ii5	ii6	n4		
Decode	n2	WAIT	—					ii1	ii2	ii3	ii4	ii5	ii6	n4	
Execute	n1	n2	WAIT						ii1	ii2	ii3	ii4	ii5	ii6	n4
Instruction Cycle Count	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i = Interrupt
ii = Interrupt Instruction Word
n = Normal Instruction Word

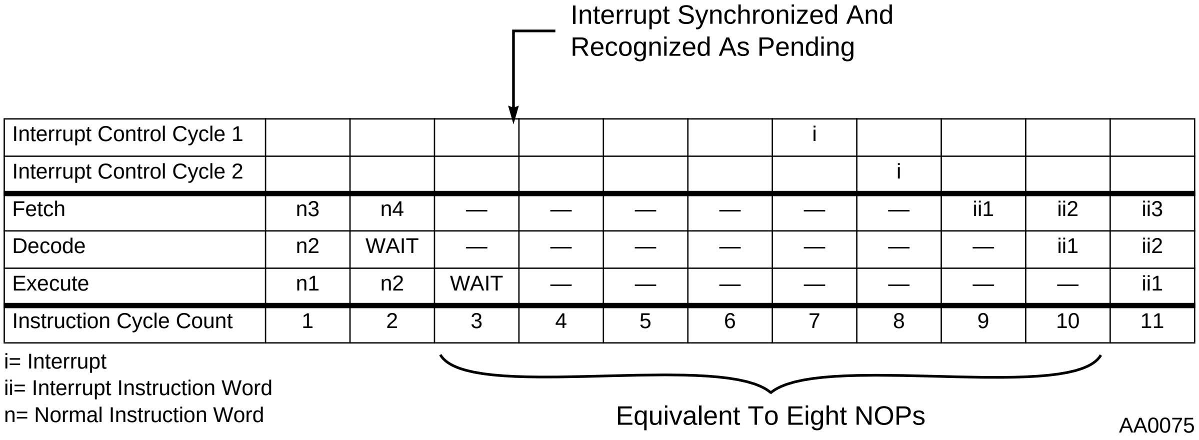
Only Internal Peripherals Receive Clock

AA0074

Figure 7-8 Wait Instruction Timing

Figure 7-8 shows the result of an interrupt bringing the processor out of the wait state. The two appropriate interrupt vectors are fetched and put in the instruction pipe. The next instruction fetched is n4, which had been aborted earlier. Instruction execution proceeds normally from this point.

Figure 7-9 shows an example of the wait instruction being executed at the same time that an interrupt is pending. Instruction n4 is aborted as before. The wait instruction causes a five-instruction-cycle delay from the time it is decoded, after which the interrupt is processed normally. The internal clocks are not turned off, and the net effect is that of executing eight NOP instructions between the execution of n2 and ii1.



Interrupt Synchronized And Recognized As Pending

Interrupt Control Cycle 1							i				
Interrupt Control Cycle 2								i			
Fetch	n3	n4	—	—	—	—	—	—	ii1	ii2	ii3
Decode	n2	WAIT	—	—	—	—	—	—	—	ii1	ii2
Execute	n1	n2	WAIT	—	—	—	—	—	—	—	ii1
Instruction Cycle Count	1	2	3	4	5	6	7	8	9	10	11

i= Interrupt
ii= Interrupt Instruction Word
n= Normal Instruction Word

Equivalent To Eight NOPs

AA0075

Figure 7-9 Simultaneous Wait Instruction and Interrupt

7.6 STOP PROCESSING STATE

The STOP instruction brings the processor into the stop processing state, which is the lowest power consumption state. In the stop state the clock oscillator is gated off, whereas in the wait state the clock oscillator remains active. The chip clears all peripheral interrupts and external interrupts ($\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, and $\overline{\text{NMI}}$) when it enters the stop state. Stack errors that were pending remain pending. The priority levels of the peripherals remain as they were before the STOP instruction was executed. The on-chip peripherals are held in their respective individual reset states while in the stop state.

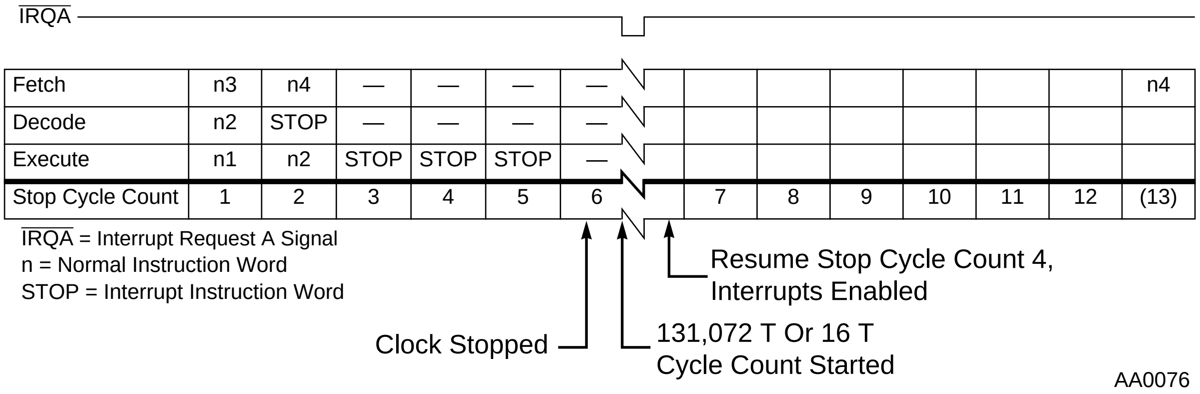
The stop processing state halts all activity in the processor until one of the following actions occurs:

- A low level is applied to the $\overline{\text{IRQA}}$ pin
- A low level is applied to the $\overline{\text{RESET}}$ pin
- An on-chip timer reaches zero

Either of these actions will activate the oscillator, and, after a clock stabilization delay, clocks to the processor and peripherals will be re-enabled. The clock stabilization delay period is determined by the stop delay (SD) bit in the OMR.

The stop sequence is composed of eight instruction cycles called stop cycles. They are differentiated from normal instruction cycles because the fourth cycle is stretched for an indeterminate period of time while the four-phase clock is turned off.

The STOP instruction is fetched in stop cycle 1 of **Figure 7-10**, decoded in stop cycle 2 (which is where it is first recognized as a stop command), and executed in stop cycle 3. The next instruction (n4) is fetched during stop cycle 2 but is not decoded in stop cycle 3 because, by that time, the STOP instruction prevents the decode. The processor stops the clock and enters the stop mode. The processor will stay in the stop mode until it is restarted.

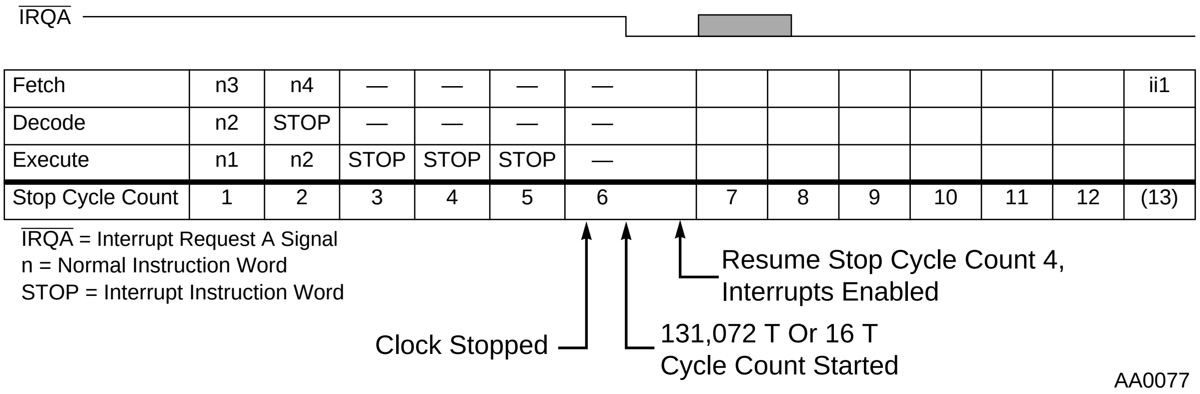


Fetch	n3	n4	—	—	—	—							n4
Decode	n2	STOP	—	—	—	—							
Execute	n1	n2	STOP	STOP	STOP	—							
Stop Cycle Count	1	2	3	4	5	6		7	8	9	10	11	12	(13)

$\overline{\text{IRQA}}$ = Interrupt Request A Signal
n = Normal Instruction Word
STOP = Interrupt Instruction Word

Figure 7-10 STOP Instruction Sequence

Figure 7-11 shows the system being restarted by asserting the IRQA signal. If the exit from stop state was caused by a low level on the $\overline{\text{IRQA}}$ pin, then the processor will service the highest priority pending interrupt. If no interrupt is pending, then the processor resumes at the instruction following the STOP instruction that brought the processor into the stop state.



Fetch	n3	n4	—	—	—	—							ii1
Decode	n2	STOP	—	—	—	—							
Execute	n1	n2	STOP	STOP	STOP	—							
Stop Cycle Count	1	2	3	4	5	6		7	8	9	10	11	12	(13)

$\overline{\text{IRQA}}$ = Interrupt Request A Signal
n = Normal Instruction Word
STOP = Interrupt Instruction Word

Figure 7-11 STOP Instruction Sequence

An $\overline{\text{IRQA}}$ deasserted before the end of the stop cycle count will not be recognized as pending. If $\overline{\text{IRQA}}$ is asserted when the stop cycle count completes, then an $\overline{\text{IRQA}}$

interrupt will be recognized as pending and will be arbitrated with any other interrupts.

Specifically, when $\overline{\text{IRQA}}$ is asserted, the internal clock generator is started and begins a delay determined by the SD bit of the OMR. When the chip uses the internal clock oscillator, the SD bit should be set to zero, to allow a longer delay time of 128K T cycles (131,072 T cycles) so that the clock oscillator may stabilize. When the chip uses a stable external clock, the SD bit may be set to one to allow a shorter (16 T cycle) delay time and a faster start up of the chip.

For example, assume that the SD equals 0 so that the 128K T counter is used. During the 128K T count the processor ignores interrupts until the last few counts and, at that time, begins to synchronize them. At the end of the 128K T cycle delay period, the chip restarts instruction processing, completes stop cycle 4 (interrupt arbitration occurs at this time), and executes stop cycles 5, 6, 7, and 8. (It takes 17T from the end of the 128K T delay to the first instruction fetch.) If the $\overline{\text{IRQA}}$ signal is released (pulled high) after a minimum of 4T but less than 128K T cycles, no $\overline{\text{IRQA}}$ interrupt will occur, and the instruction fetched after stop cycle 8 will be the next sequential instruction (n4 in **Figure 7-10**). An $\overline{\text{IRQA}}$ interrupt will be serviced as shown in **Figure 7-11** if the following conditions are true:

1. The $\overline{\text{IRQA}}$ signal had previously been initialized as level-sensitive.
2. $\overline{\text{IRQA}}$ is held low from the end of the 128K T cycle delay counter to the end of stop cycle count 8.
3. No interrupt with a higher interrupt level is pending.

If $\overline{\text{IRQA}}$ is not asserted during the last part of the STOP instruction sequence (6, 7, and 8) and if no interrupts are pending, the processor will re-fetch the next sequential instruction (n4). Since the $\overline{\text{IRQA}}$ signal is asserted, the processor will recognize the interrupt and fetch and execute the JSR instruction located at P:$0010 and P:$0011 (the $\overline{\text{IRQA}}$ interrupt vector locations).

To ensure servicing $\overline{\text{IRQA}}$ immediately after leaving the stop state, the following steps must be taken before the execution of the STOP instruction:

1. Define $\overline{\text{IRQA}}$ as level-sensitive; an edge-triggered interrupt will not be serviced.
2. Define $\overline{\text{IRQA}}$ priority as higher than the other sources and higher than the program priority.
3. Ensure that no stack error is pending.
4. Execute the STOP instruction and enter the stop state.

5. Recover from the stop state by asserting the $\overline{\text{IRQA}}$ pin and holding it asserted for the entire clock recovery time. If it is low, the IRQA vector will be fetched.
6. The exact elapsed time for clock recovery is unpredictable. The external device that asserts $\overline{\text{IRQA}}$ must wait for some positive feedback, such as specific memory access or a change in some predetermined I/O pin, before deasserting $\overline{\text{IRQA}}$.

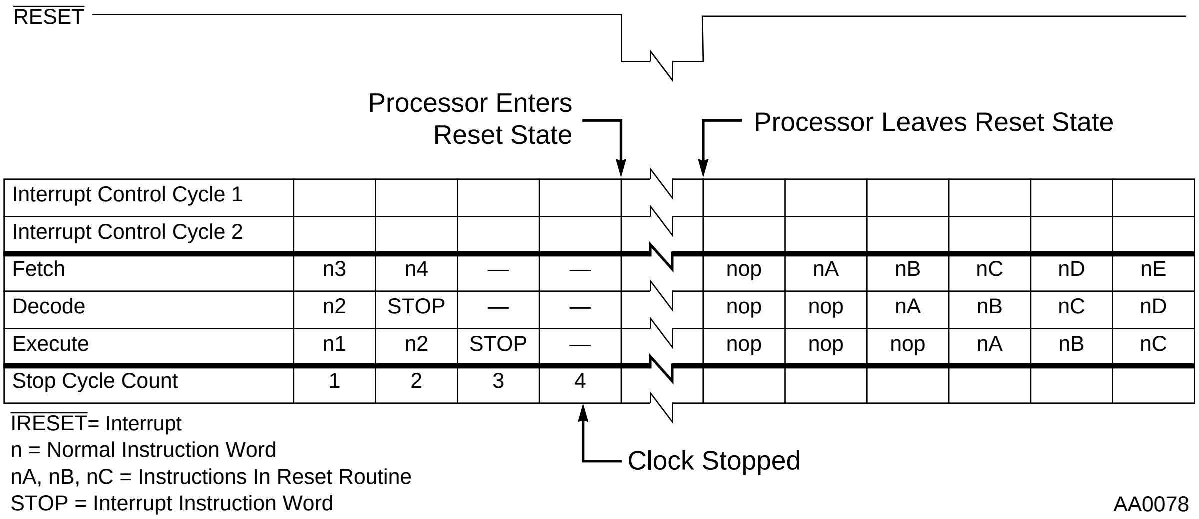
The STOP sequence totals 131,104 T cycles (if the SD equals 0) or 48 T cycles (if the SD equals 1) in addition to the period with no clocks from the stop fetch to the $\overline{\text{IRQA}}$ vector fetch (or next instruction). However, there is an additional delay if the internal oscillator is used. An indeterminate period of time is needed for the oscillator to begin oscillating and then stabilize its amplitude. The processor will still count 131,072 T cycles (or 16 T cycles), but the period of the first oscillator cycles will be irregular; thus, an additional period of 19,000 T cycles should be allowed for oscillator irregularity (the specification recommends a total minimum period of 150,000 T cycles for oscillator stabilization). If an external oscillator is used that is already stabilized, no additional time is needed.

The PLL may be disabled or not when the chip enters the stop state. If it is disabled and will not be re-enabled when the chip leaves the stop state, the number of T cycles will be much greater because the PLL must regain lock.

If the STOP instruction is executed when the $\overline{\text{IRQA}}$ signal is asserted, the clock generator will not be stopped, but the four-phase clock will be disabled for the duration of the 128K T cycle (or 16 T cycle) delay count. In this case the STOP instruction looks like a 131,072 T + 35 T cycle (or 51 T cycle) NOP, since the STOP instruction itself is eight instruction cycles long (32 T) and synchronization of $\overline{\text{IRQA}}$ is 3T, which equals 35T.

A stack error interrupt pending before entering the stop state is not cleared and will remain pending. During the clock stabilization delay in stop mode, any edge-triggered IRQ interrupts are cleared and ignored.

If $\overline{\text{RESET}}$ is used to restart the processor (see **Figure 7-12**), the 128K T cycle delay counter would not be used, all pending interrupts would be discarded, and the processor would immediately enter the reset processing state as described in **Reset Processing State** on page 7-3. For example, the stabilization time recommended in the *DSP56L811 Technical Data Sheet* for the clock ($\overline{\text{RESET}}$ should be asserted for this time) is only 50 T for a stabilized external clock but is the same 150,000 T for the internal oscillator. These stabilization times are recommended and are not imposed by internal timers or time delays. The DSP fetches instructions immediately after exiting reset. If the user wishes to use the 128K T (or 16 T) delay counter, it can be started by asserting $\overline{\text{IRQA}}$ for a short time (about two clock cycles).


$\overline{\text{RESET}}$
Processor Enters Reset State
Processor Leaves Reset State
Clock Stopped
AA0078

Interrupt Control Cycle 1											
Interrupt Control Cycle 2											
Fetch	n3	n4	—	—		nop	nA	nB	nC	nD	nE
Decode	n2	STOP	—	—		nop	nop	nA	nB	nC	nD
Execute	n1	n2	STOP	—		nop	nop	nop	nA	nB	nC
Stop Cycle Count	1	2	3	4							

$\overline{\text{IRESET}}$= Interrupt
n = Normal Instruction Word
nA, nB, nC = Instructions In Reset Routine
STOP = Interrupt Instruction Word

Figure 7-12 STOP Instruction Sequence Recovering with $\overline{\text{RESET}}$

7.7 DEBUG PROCESSING STATE

The debug processing state is a state where the DSP core is halted and under the control of the OnCE debug port. Serial data is shifted in and out of this port, and it is possible to execute single instructions from this processing state. The debug processing state is covered in more detail in **Section 10 Development Tools**.

SECTION 8

SOFTWARE TECHNIQUES

```
; JRSET Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTH      #xxxx,X:<ea>      ; 16-bit mask allowed
      JCS         label             ; 19-bit jump address allowed

; JRCLR Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTL      #xxxx,X:<ea>      ; 16-bit mask allowed
      JCS         label             ; 19-bit jump address allowed

; BR1SET Operation
; Emulated in 5 Icyc (4 Icyc if false), 3 Instruction Words
      BFTSTL      #xxxx,X:<ea>      ; 16-bit mask allowed
      BCC         label             ; 7-bit signed PC rel offset
                                    ; allowed
```

8.1 INTRODUCTION

The features of the DSP56800 architecture can be further enhanced using different software techniques to get the most out of the DSP56800 architecture's resources. For example, more powerful instructions can be often emulated with small sequences of DSP56800 instructions. This chapter discusses how more performance can be obtained from the DSP56800 architecture using software techniques. The following topics are covered:

- Synthesizing useful new instructions
- Techniques for shifting 16- and 32-bit values
- Incrementing and decrementing
- Division techniques
- Pushing variables onto the software stack
- Different looping and nested looping techniques
- Different techniques for array indexing
- Parameter passing and local variables
- Freeing up registers for time critical loops
- Interrupt programming
- Jumps and JSRs using a register value
- Freeing one hardware stack (HWS) location
- Multi-tasking and the HWS

8.2 USEFUL INSTRUCTION OPERATIONS

The flexible instruction set of the DSP56800 architecture allows new instructions to be synthesized from existing DSP56800 instructions. This section presents some of these useful operations that are not directly supported by the DSP56800 instruction set, but can be efficiently synthesized. **Table 8-1** lists operations that can be synthesized using DSP56800 instructions.

Table 8-1 Operations Synthesized using DSP56800 Instructions

Operation	Description
JRSET, JRCLR	Jumps if all selected bits in bit-field set or clear
BR1SET, BR1CLR	Branches if at least one selected bit in bit-field set or clear
JR1SET, JR1CLR	Jumps if at least one selected bit in bit-field set or clear
JVS, JVC, BVS, BVC	Jumps or branches if the overflow bit is set or clear
JPL, JMI, JES, JEC, JLMS, JLMC, BPL, BMI, BES, BEC, BLMS, BLMC	Jumps or branches on other condition codes
NEGW	Negates of upper two registers of an accumulator
NEG	Negates another data ALU register, an AGU register, or a memory location
XCHG	Exchanges any two registers
MAX	Returns the maximum of two registers
MIN	Returns the minimum of two registers
Accumulator sign-extend	Sign-extends the accumulator into the A2 or B2 portion
Accumulator unsigned load	Zeros the accumulator LSP and extension register

8.2.1 Jumps and Branches

Several operations for jumping and branching can be emulated, depending on selected bits in a bit-field, overflows, or other condition codes.

8.2.1.1 JRSET and JRCLR Operations

The JRSET and JRCLR operations are very similar to the BRSET and BRCLR instructions. They still test a bit-field and go to another address if all masked bits are either set or cleared. The BRSET and BRCLR instructions only allow branches of 64 locations away from the current instruction and can only test an 8-bit bit-field; however, JRSET and JRCLR operations allow jumps to anywhere in the 512K word program address space and can specify a 16-bit mask. The code below shows that these two operations allow the same addressing modes as the BFTSTH and BFTSTL instructions.

```
; JRSET Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTH      #xxxx,X:<ea>      ; 16-bit mask allowed
      JCS         label             ; 19-bit jump address allowed
```

```
; JRCLR Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTL      #xxxx,X:<ea>      ; 16-bit mask allowed
      JCS         label             ; 19-bit jump address allowed
```

8.2.1.2 BR1SET and BR1CLR Operations

The BR1SET and BR1CLR operations are very similar to the BRSET and BRCLR instructions. They still test a bit-field and branch to another address based on the result of some test. The difference is that for BRSET and BRCLR the condition is true if all selected bits in the bit-field are 1s or 0s, respectively, whereas for BR1SET and BR1CLR the condition is true if at least one of the selected bits in the bit-field is a 1 or 0, respectively. BR1SET and BR1CLR operations can also specify a 16-bit mask, compared to an 8-bit mask for BRSET and BRCLR. The code below shows that these two operations allow the same addressing modes as the BFTSTH and BFTSTL instructions.

```
; BR1SET Operation
; Emulated in 5 Icyc (4 Icyc if false), 3 Instruction Words
      BFTSTL      #xxxx,X:<ea>      ; 16-bit mask allowed
      BCC         label             ; 7-bit signed PC relative offset
                                    ; allowed

; BR1CLR Operation
; Emulated in 5 Icyc (4 Icyc if false), 3 Instruction Words
      BFTSTH      #xxxx,X:<ea>      ; 16-bit mask allowed
      BCC         label             ; 7-bit signed PC relative offset
                                    ; allowed
```

8.2.1.3 JR1SET and JR1CLR Operations

The JR1SET and JR1CLR operations are very similar to the JRSET and JRCLR instructions. They still test a bit-field and jump to another address based on the result of some test. The difference is that for JRSET and JRCLR the condition is true if all selected bits in the bit-field are 1s or 0s, respectively, whereas for JR1SET and JR1CLR the condition is true if at least one of the selected bits in the bit-field is a 1 or 0, respectively. JR1SET and JR1CLR operations allow jumps to anywhere in the 512K word program address space and can specify a 16-bit mask. The code below shows that these two operations allow the same addressing modes as the BFTSTH and BFTSTL instructions.

```
; JR1SET Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTL      #xxxx,X:<ea>      ; 16-bit mask allowed
      JCC         label             ; 19-bit jump address allowed
```

```
; JR1CLR Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTH      #xxxx,X:<ea>      ; 16-bit mask allowed
      JCC         label             ; 19-bit jump address allowed
```

8.2.1.4 JVS, JVC, BVS, and BVC Operations

Although there is no instruction for jumping or branching on overflow, this operation can be emulated as shown in the code below. Note that the carry bit will be destroyed by this operation since it receives the result of the BFTSTH instruction. The code below shows JVS and BVC.

```
; JVS Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTH      #$0002,SR         ; Test V bit in SR
      JCS         label             ; 19-bit jump address allowed

; BVC Operation
; Emulated in 5 Icyc (4 Icyc if false), 3 Instruction Words
      BFTSTH      #$0002,SR         ; Test V bit in SR
      BCC         label             ; 7-bit signed PC relative offset
                                    ; allowed
```

8.2.1.5 Other Jumps and Branches on Condition Codes

Jumping and branching using some of the other condition codes (PL, MI, EC, ES, LC, LS) can be accomplished in the same manner as was done for overflow (**JVS, JVC, BVS, and BVC Operations** on page 8-6). Remember that this technique destroys the value in the carry bit. The code below shows JPL and BES.

```
; JPL Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTH      #$0008,SR         ; Test the N bit in SR
      JCC         label             ; 19-bit jump address allowed

; BES Operation
; Emulated in 5 Icyc (4 Icyc if false), 3 Instruction Words
      BFTSTH      #$0020,SR         ; Test E bit in SR
      BCS         label             ; 7-bit signed PC relative offset
                                    ; allowed
```

Similar code can be written for JMI, JEC, JES, JLMC, JLMS, BPL, BMI, BEC, BLMC, and BLMS. The JLMS and JLMC are used for “jump if limit set” and “jump if limit clear,” respectively; this is done to avoid any confusion with the JLS (“jump if lower or same”) instruction.

8.2.2 Negates

The NEGW operation can be used to negate the upper two registers of the accumulator. The NEG operation can be used to negate the X0, Y0, or Y1 data ALU registers, negate an AGU register, or negate a memory location.

8.2.2.1 NEGW Operation

The NEGW operation can be emulated as shown in the code below:

```
; 20-bit NEGW Operation
; Operates on EXT:MSP, Clears LSP, 3 Icyc
      MOVE        #0,A0       ; Clear LSP
      NEG         A           ; Now negates upper 20 bits of accumulator
                              ; since A0 = 0
```

This correctly negates the upper 20 bits of the accumulator, but also destroys the A0 register.

The NEG instruction can be used directly, executing in 1 instruction cycle, in cases where it is already known that the least significant portion (LSP) of an accumulator is $0000. This is true immediately after a value is moved to the A or B accumulator from memory or a register as shown in the code below:

```
; Example of 1 Icyc NEGW Operation
; Works because A0 is already equal to $0000
      MOVE        X:(R0),A    ; Move a 16-bit value to an accumulator,
                              ; Clearing A0 register
      NEG         A           ; Now negates upper 20 bits of accumulator
                              ; since A0 = 0
```

The technique shown in the code below can be used for cases in which 16-bit data is being processed and it can be guaranteed that the LSP or extension register of the accumulator contains no required information:

```
; 16-bit NEGW Operation
; Operates on MSP, Forces EXT to sign extension, LSP to $0, 2 Icyc
      MOVE        A1,A        ; Force A2 to sign extension,
                              ; force A0 cleared
      NEG         A           ; Now negates upper 20 bits of accumulator
                              ; since A0 = 0
```

The following technique may be used for the case where the CC bit in the SR is set to a 1, the LSP may not be $0000, and the user is not interested in the values in the accumulator extension registers:

```
; 16-bit NEGW Operation
; CC bit must be set, Operates on MSP, Doesn't affect A0, 2 Icyc
      NOT         A           ; Ones complement of A1, A2 unchanged
      INCW        A           ; Increment to get two's-complement,
                              ; A2 may be incorrect
```

8.2.2.2 Negating the X0, Y0, or Y1 Data ALU registers

Although the NEG instruction is supported on accumulators only, NEG can be emulated as shown in the code below to perform a negation of the data ALUs X0, Y0, or Y1 registers:

```
; NEG Operation
; Emulated at 2 Icyc
      NOT         Y0
      INCW        Y0
```

8.2.2.3 Negating an AGU register

It is possible to negate one of the AGU registers (Rn) without destroying any other register as shown in the code below:

```
; NEG Operation
; Emulated at 3 Icyc
      NOTC        R0
      LEA         (R0)+
```

8.2.2.4 Negating a Memory Location

It is possible to negate a memory location as shown in the code below:

```
; NEG Operation
; Emulated at 5 Icyc
      NOTC        X:$19
      INCW        X:$19
```

When an accumulator is available, it may be faster to do this operation simply by moving the value to an accumulator, performing the operation there, and moving the result back to memory.

8.2.3 Register Exchanges

The XCHG operation can be emulated as shown in the code below:

```
; XCHG Operation
; Emulated at 4 Icyc
      PUSH        X0
      MOVE        A,X0
      POP         A
```

If a register is available, the exchange of any two registers can be emulated as shown in the code below:

```
; XCHG Operation
; Emulated at 3 Icyc
      MOVE        X0,N
      MOVE        A,X0
      MOVE        N,A
```

A faster exchange of any two registers can be emulating using one address register and N equals 0 as shown in the code below:

```
; XCHG Operation
; N register is 0, Emulated at 2 Icyc
      MOVE        A,X:(R0)
      TFR X0,A    X:(R0)+N,X0
```

8.2.4 Minimum and Maximum Values

The MAX operation returns the maximum of two values; the MIN operation return the minimum.

8.2.4.1 MAX Operation

The MAX operation can be emulated as shown in the code below:

```
; MAX Operation
      MAX         X0,A

;     ------ becomes ------
; MAX operation
; Emulated at 4 Icyc
      CMP         X0,A
      TGT         X0,A        ; (can also use TGE if desired)
```

8.2.4.2 MIN Operation

The MIN operation can be emulated as shown in the code below:

```
; MIN Operation
      MIN         Y0,A

;     ------ becomes ------

; MIN Operation
; Emulated at 4 Icyc
      CMP         Y0,A
      TLT         Y0,A        ; (can also use TLE if desired)
```

8.2.5 Accumulator Sign-extend

There are two versions of this operation. In the first, the accumulator only contains 16 bits of useful information in A1 or B1, and it is necessary to sign-extend into A2 or B2. In the second version, both A1 and A0 or B1 and B0 contain useful information. The code below shows both versions:

```
; Sign Extension Operation of 16-bit Accumulator Data
; Emulated in 1 Icyc, 1 Instruction Words
      MOVE        A1,A        ; Sign-extend into A2, clear A0 register

; Sign Extension Operation of 32-bit Accumulator Data
; Emulated in 4 Icyc, 4 Instruction Words
      PUSH        A0          ; Save A0 register
      MOVE        A1,A        ; Sign-extend into A2, clear A0 register
      POP         A0          ; Restore A0 register to correct contents
```

8.2.6 Unsigned Load of an Accumulator

The unsigned load of an accumulator, which zeros the LSP and extension register, can be exactly emulated as shown in the code below:

```
; DSP56100 Family Unsigned Load
; Emulated at 2 Icyc
      MOVE        x:(R0),A
      ZERO        A

;     ------ becomes ------

; DSP56800 Family Unsigned Load
; Emulated at 2 Icyc
      CLR         A
      MOVE        x:(R0),A1
```

This operation is important for processing unsigned numbers when the CC bit in the operating mode register (OMR) register is a 0, so that the condition codes are set using information at bit 35. This operation is useful for performing unsigned additions and subtractions on 36-bit values.

8.3 16- AND 32-BIT SHIFTS

This technique presents many different methods for performing shift operations on the DSP56800 architecture. Different techniques offer different advantages. Some techniques require several registers, where others can be performed only on the register to be shifted. It is even possible to shift the value in one register, but place the result in a different register. Techniques are also presented for shifting 36-bit values by large immediate values.

8.3.1 Small Immediate 16- or 32-bit Shifts

If it is only necessary to shift a register or accumulator by a small amount, one of the two techniques shown in the code below may be adequate. These techniques may also be appropriate if there are not any registers available for use in the shifting operation, since more than one register is required with the multi-bit shifting instructions. For cases where the amount of bit positions to shift is larger than three for 16-bit registers or five for a 32-bit value, then it may be appropriate to use another technique.

```
; First Technique - Shift an Accumulator by 3 bits - Use Inline Code
      ASL         A
      ASL         A
      ASL         A

; Second Technique - Shift an Accumulator by 6 bits - Use REP Loop
      REP         #6
      ASL         A
```

For places in a program that are executed infrequently, the second technique of using a REP (or DO) loop results in the smallest code size.

8.3.2 General 16-bit Shifts

For fast 16-bit shifting, the ASLL, ASRR, LSLL, and LSRR allow for single cycle shifting of a 16-bit value where the shift count is specified by a register. If it is desired to shift by an immediate value, the immediate value must first be loaded into a register as shown in the code below:

```
; Shifting a 16-bit Value by an Immediate Value
; Executes in 2 Icyc, 2 Instruction Words
      MOVE        #7,X0       ; Load shift count into the X0 register
      ASLL        Y0,X0,Y0    ; Arithmetically shift the contents of Y0
                              ; 7 bits to the left
```

Note that these instructions clear the LSP of an accumulator. It is possible to perform a right shift where the bits shifted into the LSP of the accumulator are not lost; instead of using the ASRR or LSRR instructions, a CLR instruction is first used to clear the accumulator, and then an ASRAC or LSRAC instruction is performed. This technique allows a 16-bit value to be right shifted into a 32-bit field as shown in the code below:

```
; Shifting a 16-bit Value into a 32-bit field
; Executes in 2 Icyc, 2 Instruction Words
      CLR         A           ; Clear accumulator
      ASRAC       Y0,X0,A     ; Arithmetically shift into a 32-bit field
```

8.3.3 General 32-bit Arithmetic Right Shifts

It is possible to perform right shifting of up to 15 bits on 32-bit values using the techniques presented in this section.

The example below shows how to arithmetically shift the 32-bit contents of the Y1:Y0 registers, storing the results into the A accumulator. Note that this technique uses many of the data ALU registers: Y1 and Y0 to hold the value to be shifted, X0 to hold the amount to be shifted, and the A accumulator to store the result. The code shown below allows shifts of 0 to 15 bits and executes in five instruction cycles:

```
; Arithmetically Shift Y1:Y0 Register Combination by 8 bits
; Emulated in 5 Icyc, 5 Instruction Words
      MOVE        #8,X0
      LSRR        Y0,X0,A           ; Logically shift lower word
      MOVE        A1,A0             ; 16-bit arithmetic right shift
      MOVE        A2,A1
      ASRAC       Y1,X0,A           ; Arithmetically shift upper word and
                                    ; Combine with lower word
```

If it is necessary to shift by more than 15 bits, then the code below should be preceded by a shift of 16 bits, as documented later in this section.

Similar code below shows how to arithmetically shift the 32-bit value in the A accumulator. Again, this technique takes several registers: Y1 to hold the most significant word (MSW) to be shifted and Y0 to hold the amount to be shifted. This, perhaps, is only useful when the amount to be shifted is a variable amount or when the amount to be shifted is 8 or more and the Y1 and Y0 registers are available. Note that the extension register (A2) is not shifted in this case.

```
; Arithmetically Shift A1:A0 Accumulator by 11 bits
; Emulated in 7 Icyc, 7 Instruction Words
      MOVE        #11,Y0
      MOVE        A1,Y1             ; Save copy of A1 register (upper word
                                    ; to be shifted)
      MOVE        A0,A1
      LSRR        A1,Y0,A           ; Logically shift lower word
      MOVE        A1,A0             ; 16 bit arithmetic right shift
      MOVE        A2,A1
      ASRAC       Y1,Y0,A           ; Arithmetically shift upper word and
                                    ; Combine with lower word
```

8.3.4 General 32-bit Logical Right Shifts

Right shifting logically is identical to right shifting arithmetically except for the final shift instruction. For arithmetic shifts of 32-bit values the final instruction is an

ASRAC instruction, and for logical shifts of 32-bit values the final instruction is a LSRAC instruction. This is shown in the code below:

```
; Logically Shift Y1:Y0 Register Combination by 8 bits
; Emulated in 5 Icyc, 5 Instruction Words
      MOVE        #8,X0
      LSRR        Y0,X0,A           ; logically shift lower word
      MOVE        A1,A0             ; 16-bit arithmetic right shift
      MOVE        A2,A1
      LSRAC       Y1,X0,A           ; logically shift upper word and
                                    ; combine with lower word
```

8.3.5 Arithmetic Shifts by a Fixed Amount

Arithmetic shifts (left or right) by a fixed amount can be emulated with the ASRxx operations.

8.3.5.1 Right Shifts (ASR12 - ASR20)

For arithmetic right shifts there is a faster way to shift an accumulator for large shift counts. The code below shows how to perform arithmetic right shifts of 12 through 20 bits on an accumulator. This emulation works without destroying any registers on the chip. It is possible to use this technique greater than 20 bit shifts if desired, but not possible to use this technique for shifts of 11 bits or less without losing information.

```
; ASR12 Operation
; Emulated in 8 Icyc, 8 Instruction Words
      ASL         A
      ASL         A
      ASL         A
      ASL         A
      PUSH        A1          ; (PUSH is a two word, two Icyc macro)
      MOVE        A2,A
      POP         A0

; ASR13 Operation
; Emulated in 7 Icyc, 7 Instruction Words
      ASL         A
      ASL         A
      ASL         A
      PUSH        A1          ; (PUSH is a two word, two Icyc macro)
      MOVE        A2,A
      POP         A0

; ASR14 Operation
; Emulated in 6 Icyc, 6 Instruction Words
      ASL         A
      ASL         A
      PUSH        A1          ; (PUSH is a two word, two Icyc macro)
      MOVE        A2,A
      POP         A0

; ASR15 Operation
; Emulated in 5 Icyc, 5 Instruction Words
      ASL         A
      PUSH        A1          ; (PUSH is a two word, two Icyc macro)
      MOVE        A2,A
      POP         A0

; ASR16 Operation
; Emulated in 2 Icyc, 2 Instruction Words
      MOVE        A1,A0       ; (Assumes EXT contains sign extension)
      MOVE        A2,A1

; ASR17 Operation
; Emulated in 3 Icyc, 3 Instruction Words
      ASR         A
      MOVE        A1,A0       ; (Assumes EXT contains sign extension)
      MOVE        A2,A1

; ASR18 Operation
; Emulated in 4 Icyc, 4 Instruction Words
      ASR         A
      ASR         A
      MOVE        A1,A0       ; (Assumes EXT contains sign extension)
      MOVE        A2,A1

; ASR19 Operation
; Emulated in 5 Icyc, 5 Instruction Words
      ASR         A
      ASR         A
      ASR         A
      MOVE        A1,A0       ; (Assumes EXT contains sign extension)
      MOVE        A2,A1

; ASR20 Operation
; Emulated in 6 Icyc, 6 Instruction Words
      ASR         A
      ASR         A
      ASR         A
      ASR         A
      MOVE        A1,A0        ; (Assumes EXT contains sign extension)
      MOVE        A2,A1
```

8.3.5.2 Left Shifts (ASL16 - ASL19)

For arithmetic left shifts there is a faster way to shift an accumulator for large shift counts. The code below shows how to perform arithmetic left shifts of 16 through 19 bits on an accumulator. This emulation works without destroying any registers on the chip. It is possible to use this technique greater than 19 bit shifts if desired but not possible for shifts of 15 bits or less without losing information.

```
; ASL16 Operation
; Emulated in 4 Icyc, 4 Instruction Words
      PUSH        A1           ; (PUSH is a two word, two Icyc macro)
      MOVE        A0,A
      POP         A2

; ASL17 Operation
; Emulated in 5 Icyc, 5 Instruction Words
      ASL         A
      PUSH        A1           ; (PUSH is a two word, two Icyc macro)
      MOVE        A0,A
      POP         A2

; ASL18 Operation
; Emulated in 6 Icyc, 6 Instruction Words
      ASL         A
      ASL         A
      PUSH        A1           ; (PUSH is a two word, two Icyc macro)
      MOVE        A0,A
      POP         A2

; ASL19 Operation
; Emulated in 7 Icyc, 7 Instruction Words
      ASL         A
      ASL         A
      ASL         A
      PUSH        A1           ; (PUSH is a two word, two Icyc macro)
      MOVE        A0,A
      POP         A2
```

8.4 INCREMENTS AND DECREMENTS

Increments and decrements on the DSP56800 core can be done on almost any piece of data. This section summarizes the different increments and decrements available to both registers and memory locations. It is important to note the LEA instruction, which is used to increment or decrement AGU pointer registers. The TSTW instruction is also used for decrementing AGU pointer registers; this instruction is similar to LEA but also sets the condition codes, making it useful for program looping and other tasks. The LEA and TSTW instructions do not cause a pipeline dependency in the AGU (**Pipeline Dependencies** on page 4-36). The TSTW instruction is not available for incrementing an AGU pointer or for decrementing the SP register.

```
; Different ways to Increment on the DSP56800 core
      INCW        A           ; on a Data ALU Accumulator
      INCW        X0          ; on a Data ALU Input Register
      LEA         (Rn)+       ; on an AGU pointer register (R0-R3 or SP)
      INCW        X:$0        ; on anywhere within the first 64 locations
                              ; of X data memory
      INCW        X:$C200     ; on anywhere within the first 64K locations
                              ; of X data memory
      INCW        X:(SP-37)   ; on a value located on the stack

; Different ways to Decrement on the DSP56800 core
      DECW        A           ; on a Data ALU Accumulator
      DECW        X0          ; on a Data ALU Input Register
      LEA         (Rn)-       ; on an AGU pointer register (R0-R3 or SP)
      TSTW        (Rn)-       ; on an AGU pointer register (R0-R3)
      DECW        X:$0        ; on anywhere within the first 64 locations
                              ; of X data memory
      DECW        X:$C200     ; on anywhere within the first 64K locations
                              ; of X data memory
      DECW        X:(SP-37)   ; on a value located on the stack
```

The many different techniques available helps to prevent registers from being destroyed. Otherwise, as found on other architectures, it is necessary to first move data to an accumulator to perform an increment.

8.5 DIVISION

It is possible to perform fractional or integer division on the DSP56800 core. There are several questions to consider when implementing a division on the DSP core:

- Are both operands always guaranteed to be positive?
- Are operands fractional or integer?
- Is the quotient only needed, or is the remainder needed as well?
- Will the calculated quotient fit in 16 bits in integer division?
- Are the operands signed or unsigned?
- How many bits of precision are in the dividend?
- What about overflow in fractional and integer division?
- Will there be “integer division” effects?

Note: In a division equation, the “dividend” is the numerator, the “divisor” is the denominator, and the “quotient” is the result.

It is then possible to select the appropriate division algorithm once all these questions have been answered. The fractional algorithms support a 32-bit signed dividend, and the integer algorithms support a 31-bit signed dividend. All algorithms support a 16-bit divisor.

Note that the most general division algorithms are the fractional and integer algorithms for 4 quadrant division that generate both a quotient and a remainder. These take the most number of instruction cycles to complete and use the most registers.

For extended precision division, where the number of quotient bits required is more than 16, the DIV instruction and routines presented in this section are no longer applicable. For further information on division algorithms, refer to pages 524-530 of *Theory and Application of Digital Signal Processing* by Rabiner and Gold (Prentice-Hall, 1975), pages 190-199 of *Computer Architecture and Organization* by John Hayes (McGraw-Hill, 1978), or other references as required.

8.5.1 Positive Dividend and Divisor with Remainder

The algorithms in the code below are the fastest and take the least amount of program memory. In order to use these algorithms, it must be guaranteed that both the dividend and divisor are both positive, signed two’s-complement numbers. One algorithm is presented for division of fractional numbers and a second is presented for the division of integer numbers. Both algorithms generate the correct positive quotient and positive remainder.

```
; Division of Fractional, Positive Data (B1:B0 / X0)
      BFCLR       #$0001,SR   ; Clear carry bit: required for 1st DIV instr
      REP         16
      DIV         X0,B        ; Form positive quotient in B0
      ADD         X0,B        ; Restore remainder in B1
                              ; (At this point, the positive quotient is in
                              ; B0 and the positive remainder is in B1)

; Division of Integer, Positive Data (B1:B0 / X0)
      ASL         B           ; Shift of dividend required for integer
                              ; division
      BFCLR       #$0001,SR   ; Clear carry bit: required for 1st DIV instr
      REP         16
      DIV         X0,B        ; Form positive quotient in B0
      MOVE        B0,Y1       ; Save quotient in Y1
                              ; (At this point, the positive quotient is in
                              ; B0 but the remainder is not yet correct)
      ADD         X0,B        ; Restore remainder in B1
      ASR         B           ; Required for correct integer remainder
                              ; (At this point, the correct positive
                              ; remainder is in B1)
```

8.5.2 Signed Dividend and Divisor with No Remainder

The algorithms in the code below are fast ways to divide two signed numbers, where both the dividend or the divisor are signed two’s-complement numbers. These algorithms are faster because they generate the quotient only; they do not generate a correct remainder. The algorithms are referred to as 4 quadrant division because these algorithms allow any combination of positive or negative operands for the dividend and divisor. One algorithm is presented for division of fractional numbers and a second is presented for the division of integer numbers.

```
; 4 Quadrant Division of Fractional, Signed Data (B1:B0 / X0)
; Generates signed quotient only, no remainder
; Setup
      MOVE        B,Y1        ; Save Sign Bit of dividend (B1) in MSB of Y1
      ABS         B           ; Force dividend positive
      EOR         X0,Y1       ; Save sign bit of quotient in N bit of SR
      BFCLR       #$0001,SR   ; Clear carry bit: required for 1st DIV instr
; Division
      REP         16
      DIV         X0,B        ; Form positive quotient in B0
```

```
; Correct quotient
      BGE         DONE        ; If correct result is positive, then done
      NEG         B           ; Else negate to get correct negative result
DONE
                              ; (At this point, the correctly signed
                              ; quotient is in B0 but the remainder is not
                              ; correct)

; 4 Quadrant Division of Integer, Signed Data (B1:B0 / X0)
; Generates signed quotient only, no remainder
; Setup
      ASL         B           ; Shift of dividend required for integer
                              ; division
      MOVE        B,Y1        ; Save Sign Bit of dividend (B1) in MSB of Y1
      ABS         B           ; Force dividend positive
      EOR         X0,Y1       ; Save sign bit of quotient in N bit of SR
      BFCLR       #$0001,SR   ; Clear carry bit: required for 1st DIV instr
; Division
      REP         16
      DIV         X0,B        ; Form positive quotient in B0
; Correct quotient
      BGE         DONE        ; If correct result is positive, then done
      NEG         B           ; Else negate to get correct negative result
DONE
                              ; (At this point, the correctly signed
                              ; quotient is in B0 but the remainder is not
                              ; correct)
```

8.5.3 Signed Dividend and Divisor with Remainder

The algorithms in the code below are another way to divide two signed numbers, where both the dividend or the divisor are signed two's-complement numbers (positive or negative). These algorithms are the most general because they generate both a correct quotient and a correct remainder. The algorithms are referred to as 4 quadrant division because these algorithms allow any combination of positive or negative operands for the dividend and divisor. One algorithm is presented for division of fractional numbers and a second is presented for the division of integer numbers.

```
; 4 Quadrant Division of Fractional, Signed Data (B1:B0 / X0)
; Generates Signed quotient and remainder
; Setup
      MOVE        B1,A        ; Save sign bit of dividend (B1) in MSB of A1
      MOVE        B1,N        ; Save sign bit of dividend (B1) in MSB of N
      ABS         B           ; Force dividend positive
      EOR         X0,Y1       ; Save sign bit of quotient in N bit of SR
      BFCLR       #$0001,SR   ; Clear carry bit: required for 1st DIV instr
; Division
      REP         16
      DIV         X0,B
; Correct quotient
      TFR         B,A
      BGE         QDONE       ; If correct result is positive, then done
      NEG         B           ; Else negate to get correct negative result
QDONE
      MOVE        A0,Y1       ; Y1 <- True quotient
      MOVE        X0,A        ; A <- Signed divisor
      ABS         A           ; A <- Absolute value of divisor
      ADD         B,A         ; A1 <- Restored remainder
      BRCLR       #$8000,N,DONE
      MOVE        #0,A0
      NEG         A
DONE
                              ; (At this point, the correctly signed
                              ; quotient is in Y1 and the correct remainder
                              ; in A1)

; 4 Quadrant Division of Integer, Signed Data (B1:B0 / X0)
; Generates Signed quotient and remainder
; Setup
      ASL         B           ; Shift of dividend required for integer
                              ; division
      MOVE        B1,A        ; Save sign bit of dividend (B1) in MSB of A1
      MOVE        B1,N        ; Save sign bit of dividend (B1) in MSB of N
      ABS         B           ; Force dividend positive
      EOR         X0,Y1       ; Save sign bit of quotient in N bit of SR
      BFCLR       #$0001,SR   ; Clear carry bit: required for 1st DIV instr
;Division
      REP         16
      DIV         X0,B
; Correct quotient
      TFR         B,A
      BGE         QDONE ; If correct result is positive, then done
      NEG         B     ; Else negate to get correct negative result
QDONE
```

```
      MOVE        A0,Y1 ; Y1 <- True quotient
      MOVE        X0,A  ; A <- Signed divisor
      ABS         A     ; A <- Absolute Value of divisor
      ADD         B,A   ; A1 <- Restored remainder
      BRCLR       #$8000,N,DONE
      MOVE        #0,A0
      NEG         A
      ASR         B     ; Shift required for correct integer remainder
DONE
                        ; (At this point, signed quotient in Y1, correct
                        ; remainder in A1)
```

8.5.4 Algorithm Examples

Examples of values calculated with the division algorithms in this section are presented in the examples below:

Example 8-1 Simple Fractional Division

A simple example of fractional division is the following case:

0.125 / 0.5 = 0.25

For this case a positive fractional algorithm can be selected. Converting the fractional numbers into hex gives the following division:

$10000000 / $4000

This gives the following results:

quotient = $2000 = 0.25
remainder = 0

Example 8-2 Signed Fractional Division

Another example of fractional division is the following case:

-0.2628712165169417858123779297 / 0.39035034179687500
= -0.6734008789062500

For this case a 4 quadrant fractional algorithm can be selected. Converting the fractional numbers into hex gives the following division:

$de5a3c69 / $31f7

This gives the following results:

quotient = $a9ce = -0.6734008789062500

Example 8-3 Simple Integer Division

A simple example of integer division is the following case:

64 / 9 = 7 (remainder = 1)

For this case a positive integer algorithm can be selected. Converting the integer numbers into hex gives the following division:

$00000040 / $0009

This gives the following results:

quotient = $0007 = 7
remainder = 1

Example 8-4 Signed Integer Division

Another example of integer division is the following case:

-492789125 / -15896 = 31000

For this case a 4 quadrant integer algorithm can be selected. Converting the integer numbers into hex gives the following division:

$e2a0a27b / $c1e8

This gives the following results:

quotient = $7918 = 31000

The results can be easily checked by multiplying the quotient with the divisor and adding the remainder to this product. This final answer should be the same as the original dividend.

8.5.5 Overflow Cases

Both integer and fractional division are subject to division overflow. Overflow is the case where the correct value of the quotient will not fit into the destination available to store it.

For division of fractional numbers the result must be a 16-bit signed fractional value greater than or equal to -1.0 and less than 1.0 - $2^{-[N-1]}$; in other words, it must satisfy the following:

$$-1.0 \le \text{quotient} < +1.0 - 2^{-[N-1]}$$

For the case where the magnitude of the dividend is larger than the magnitude of the divisor, this inequality will not be true because any result generated will be larger in magnitude than 1.0. Thus, division overflow occurs with fractional numbers for the case where the absolute value of the divisor is less than or equal to the absolute value of the dividend; in other words, it must satisfy the following:

$$|\,\text{divisor}\,| \leq |\,\text{dividend}\,|$$

If this condition can be true when dividing fractional numbers, it is first necessary to scale the dividend to prevent this from occurring.

For division of integer numbers, the result must be a 16-bit signed integer value greater than or equal to $-2^{-[N-1]}$ and less than or equal to $[2^{[N-1]} -1]$, where N is equal to 16; in other words:

$$-2^{-[N-1]} \leq \text{quotient} \leq [2^{[N-1]} -1], \text{ where } N = 16$$

When dividing integer numbers, it must be guaranteed that the final result can fit into a signed 16-bit integer value. Otherwise, it is first necessary to scale the numerator to prevent this from occurring.

8.6 MULTIPLE VALUE PUSHES

The DSP56800 core currently supports a one word, one instruction cycle POP instruction for removing information from the stack. The PUSH operation, however, is a two word, two instruction cycle macro, which expands to the code shown below. (This instruction macro works quite well when pushing a single variable.)

```
; Expansion of the PUSH Instruction Macro
; Emulated in 2 Icyc, 2 Instruction Words
      LEA         (SP)+              ; Increment the SP (1 Icyc, 1 Word)
      MOVE        <register>,X:(SP) ; Place value onto the stack
                                     ; (1 Icyc, 1 Word)
```

However, there is a better technique when it is necessary to push more than one value onto the software stack. Instead of using consecutive PUSH instruction macros, it is more efficient and saves more instruction words by expanding out the PUSH operation as shown below.

```
; Faster technique for pushing multiple values onto the stack
; Finishes in 5 Icyc, 5 Instruction Words
      LEA         (SP)+              ; Increment SP
      MOVE        X0,X:(SP)+
      MOVE        Y0,X:(SP)+
      MOVE        R0,X:(SP)+
      MOVE        R1,X:(SP)          ; No post-increment SP on last MOVE
```

In this case five instruction cycles and five words are used to push four values onto the software stack. If the PUSH instruction macro had been used instead, it would have performed the same function in eight instruction cycles with eight words.

Another use of the PUSH instruction is for temporary storage. Sometimes a temporary variable is required, such as when swapping two registers. There are two techniques for doing this: one using an unused register and the second technique using a location on the stack. The second technique uses the PUSH instruction macro and works whenever there is no other registers available. The two techniques are shown in the code below:

```
; Swapping two registers (X0, R0) using an Available Register (N)
; 3 Icyc, 3 Instruction Words
      MOVE        X0,N              ; X0 -> TEMP
      MOVE        R0,X0             ; R0 -> X0
      MOVE        N,R0              ; TEMP -> R0

; Swapping two registers (X0, R0) using a Stack Location
; 4 Icyc, 4 Instruction Words
      PUSH        X0                ; X0 -> TEMP
      MOVE        R0,X0             ; R0 -> X0
      POP         R0                ; TEMP -> R0
```

The operation is faster using an unused register if one is available. Often, the N register is a good choice for temporary storage as in the example above.

8.7 LOOPS

The DSP56800 core contains a powerful and flexible hardware DO loop mechanism. It allows for loop counts up to 8,192, it allows a large number of instructions (maximum of 64K) to reside within the body of the loop, and hardware DO loops can be interrupted. In addition loops execute correctly from both on-chip and off-chip program memory, and it is possible to single step through the instructions in the loop using the OnCE port for emulation.

The DSP56800 core also contains a useful hardware REP loop mechanism, which is very useful for very simple, fast looping on a single instruction. It is very useful for simple nesting when the inner loop only contains a single instruction. For a REP loop the instruction to be repeated is only fetched once from program memory, reducing activity on the buses. This is very useful when executing code from off-chip program memory. However, REP loops are not interruptible.

8.7.1 Large Loops (Count Greater than 63)

Currently, the DO instruction allows an immediate value up to the value 63 to be specified for the loop count. In cases where it is necessary to specify an immediate value larger than 63, this is done using one of the registers on the DSP56800 core to specify the loop count. Since registers are a precious resource, it is desirable not to use any important registers that may contain valid data. The code below shows a technique for specifying loop counts greater than 63 without destroying any register values.

```
      MOVE        #2048,LC          ; Specify a loop count greater than 63
                                    ; using the LC register
      DO          LC,LABEL          ; (LC reg used to avoid destroying
                                    ; another register)
;     (instructions)
LABEL
```

Since the LC register is already a dedicated register used for looping and is always loaded by the DO instruction, no information is lost when this register is used to specify a larger loop count. Note that this technique will also work with the LC register for nested loops, as long as the loading of the LC register with immediate data occurs AFTER the LC register is pushed for nested loops.

Note: This technique should NOT be used for the REP instruction because it will destroy the value of the LC register if done by a REP instruction nested within a hardware DO loop.

8.7.2 Variable Count Loops

There are cases where it is useful to loop for a variable number of times instead of a constant number of times. For these cases the loop count is then specified using a register. Since the LC register is 13 bits, this allows a variable number of loop iterations from 2^k times (where k is the number of bits in the LC register) 13 to 1 iteration. It is important to consider what takes place if this variable is zero or negative. For the case of DO looping on a register value, when the register contains the value 0, the loop will execute 2^k times, where k is the number of bits in the LC register (13). This is also true when an immediate value of 0 is specified. For the case where the number of iterations is negative, the number will simply be interpreted as an unsigned positive number and the loop will be entered. If there is a possibility that a register value may be less than or equal to zero, then it is necessary to insert extra code outside of the loop to detect this and branch over the loop. This is demonstrated in the code below.

```
; Hardware Looping When the Loop count can be Negative or Zero
      TSTW        X0               ; Skip over loop if loop count <= 0
      BLE         LABEL
      DO          X0,LABEL
      ASL         A
LABEL
```

For the case of REP looping on a register value, when the register contains the value 0, the instruction to be repeated is simply skipped as desired; no extra code is required. This is also true when an immediate value of 0 is specified. For the case where the number of iterations can be negative, this responds in the same manner as the DO loop, and can be solved using the technique presented above for DO looping.

8.7.3 Software Loops

The DSP56800 provides the capability for implementing loops in either hardware or software. For non-nested loops in critical code sections, the hardware looping mechanism is always the fastest. However, there is a limitation, when using the hardware looping mechanism. The DSP56800 allows a maximum of two nested hardware DO loops. Any looping beyond this generates a HWS overflow interrupt.

Software looping techniques are also efficiently implemented on the DSP core. Software looping simply uses a register or memory location and decrements this value until it reaches zero. A branch instruction conditionally branches to the top of the loop.

There are three different techniques are shown below for implementing a loop in software: one using a data ALU register, one using an AGU register, and one using a memory location to hold the loop count. Each of these is shown in the code below:

```
; Software Looping
; Data ALU Register Used for Loop Count
      MOVE        #3,X0       ; Load loop count to execute the loop 3 times
LABEL                         ; Enters loop at least once
;     (instructions)
      DECW        X0
      BGT         LABEL       ; Back to top-of-Loop if positive and not 0
```

```
; Software Looping
; AGU Register Used for Loop Count
      MOVE        #3-1,R0     ; Load loop count to execute the loop 3 times
LABEL                         ; Enters loop at least once
;     (instructions)
      TSTW        (R0)-
      BGT         LABEL       ; Back to top-of-Loop if positive and not 0

; Software Looping
; Memory Location (one of first 64 XRAM locations) Used for Loop Count
      MOVE        #3,X:$7     ; Load loop count to execute the loop 3 times
LABEL                         ; Enters loop at least once
;     (instructions)
      DECW        X:$7
      BGT         LABEL       ; Back to top-of-Loop if positive and not 0
```

8.7.4 Nested Loops

This section gives recommendations for and detailed discussion of nested loops.

8.7.4.1 Recommendations

For nested looping it is recommended that the innermost loop be a hardware DO loop when appropriate and all outer loops are implemented as software loops. Even though it is possible to nest hardware DO loops, it is better to implement all outer loops using software looping techniques for two reasons:

1. The DSP56800 allows only two nested hardware DO loops.
2. The execution time of an outer hardware loop compares with the execution time of a software loop.

Likewise, there is little difference in code size between a software loop and an outer loop implemented using the hardware DO mechanism.

The hardware nesting capability of DO loops should instead be used for efficient interrupt servicing. It is recommended that the main program and all subroutines use no nested hardware DO loops. It is also recommended that software looping be used whenever there is a JSR instruction within a loop and the called subroutine requires the hardware DO loop mechanism. If these two rules are followed, then it can be guaranteed that no more than one hardware DO loop is active at a time. If this is the case, then the second HWS location is always available to ISRs for faster interrupt processing. This significantly reduces the amount of code required to free up and restore the hardware looping resources such as the HWS when entering and exiting an ISR, since it is already known upon entering the ISR that a HWS location is available.

If this technique is used, the ISR should not themselves be interruptible, or if they can be interrupted, then any ISR that can interrupt an ISR already in progress must save off one HWS location (see **Freeing One Hardware Stack Location** on page 8-42).

The code below shows the recommended nesting technique:

```
; Nesting Loops Recommended Technique

      MOVE        #3,X:$0003  ; Set up loop count for Outer Loop
                              ; (Software Loop)
OUTER
;     (instructions)
      DO          X0,INNER    ; DO loop is Inner Loop (Hardware Loop)
      ASL         A
      MOVE        A,X:(R0)+
INNER
;     (instructions)
      DECW        X:$0003     ; Decrement Outer Loop Count
      BGT         OUTER       ; Branch to top of Outer Loop if not done
```

It would also be possible to use a data ALU or AGU register if more speed is needed.

An exception to the above recommendation for nesting loops is for the unique case where the innermost loop executes a single word instruction. In this case it is possible to use a REP loop for the innermost loop and a hardware DO loop for the outermost loop. This is demonstrated in the code below:

```
; Nesting Loops Recommended Technique for Special Case of REP loop nested
; within a Hardware DO Loop
      INCW        A
      DO          X0,LABEL    ; DO loop is outer loop (interruptible)
      MOVE        B,Y1
;     (instructions)
      REP         #4          ; REP loop is inner loop (non-interruptible)
      ASL         A           ; (Must be a one word instruction)
;     (instructions)
      MOVE        A,X:(R0)+
LABEL
```

The REP loop may not be interrupted; however, so this technique may not be useful for large loop counts on the innermost loop if there are tight requirements for interrupt latency in an application. If this is the case, then the first example with a software outer loop and an inner DO loop may be appropriate.

8.7.4.2 Nested Hardware DO and REP Loops

Nesting of hardware DO loops is permitted on the DSP56800 architecture. However, it is not recommended that this technique be used for nesting loops within a program, but rather that the hardware nesting of DO loops be used to provide more efficient interrupt processing as described in **Recommendations** on page 8-28.

Since the HWS is two locations deep, it is possible to nest one DO loop within another DO loop. Furthermore, since the REP instruction does not use the HWS, it is possible to place a REP instruction within these two nested DO loops. The code below shows the maximum nesting of hardware loops allowed on the DSP56800 processor:

```
; Hardware Nested Looping Example of the Maximum Depth Allowed
;
      DO          #3,OLABEL         ; Beginning of outer loop
      PUSH        LC
      PUSH        LA
      DO          X0,ILABEL         ; Beginning of inner loop
;     (instructions)
      REP         Y0                ; Skips ASL if y0 = 0
      ASL         A
;     (instructions)
ILABEL                              ; End of inner loop
      POP         LA
      POP         LC
      NOP                           ; 3 instructions required after POP
      NOP                           ; 3 instructions required after POP
      NOP                           ; 3 instructions required after POP
OLABEL                              ; End of outer loop
```

The HWS’s current depth can be determined by the NL and LF bits, as shown in **Table 5-3 Program RAM Operating Modes** on page 5-16. From these bits it is possible to determine whether there are no loops currently in progress, a single loop, or two nested loops. Refer to **Reserved OMR Bits—Bits 2, 4, and 9-14** on page 5-18 for the values of these bits in these different conditions.

For nested DO loops it is required that there are at least three instructions after the pop of the LA and LC registers and before the label of any outer loop. This shows up in the above example as three NOPs but can be any other instructions.

Further hardware nesting is possible by saving the contents of the HWS and later restoring the stack on completion, as described in **Multi-tasking and the Hardware Stack** on page 8-43.

8.7.4.3 Comparison of Outer Looping Techniques

A comparison of the execution overhead and extra code size of software and hardware outer loops shows that for nesting of loops, it is just as efficient to nest in software (see **Table 8-1**). If a data ALU register or AGU register is available for use as the loop count, each loop executes one cycle faster than nesting loops in hardware.If there are no on-chip registers available for the loop counter, then the third technique can be used that uses one of the first 64 locations of X data memory. This technique executes one cycle slower per loop than nesting loops in hardware. Each of the software techniques also uses fewer instruction words.

Table 8-1 Outer Loop Performance Comparison

Loop Technique	# Icyc to Setup Loop	Additional #Icyc Executed Each Loop	Total # of Instr Words
Hardware nested DO loops	3	5	7
Software using data ALU register	1	4	3
Software using AGU register	1	4	3
Software using memory location	2	6	4

It is recommended that the nesting of hardware DO loops not be used for implementing nested loops. Instead, it is recommended that all outer loops in a nested looping scheme be implemented using software looping techniques. Likewise, it is recommended that software looping techniques be used when a loop contains a JSR and the called routine contains many instructions or contains a hardware DO loop.

8.7.5 Hardware DO Looping in Interrupt Service Routines

Upon entering an ISR, it is possible that one or two hardware DO loops are currently in progress. This means that the hardware looping resources (the LA and LC registers and the HWS) are currently in use and may need to be freed up if hardware looping is required within the ISR.

If the recommendations presented in **Nested Loops** on page 8-28 are followed, then it may be possible to guarantee that a maximum of one DO loop is active. In this case the HWS is guaranteed to have at least one open location, and the LF and NL bits will correctly indicate the looping status. In this case an ISR simply pushes the LA and LC registers upon entering the routine and pops them upon exit. This is very efficient as demonstrated in the code below:

```
; Example of an ISR that uses the Hardware DO Looping Mechanism
; Assumes that at least one HWS location is free
; Overhead is 5 instruction cycles, 5 instruction words
ISR
      LEA         (SP)+             ; Save Hardware Looping Resources
      MOVE        LC,X:(SP)+
      MOVE        LA,X:(SP)
;     (instructions)
      DO          #7,LABEL          ; Example of a DO loop within an ISR
      INC         A
LABEL
;     (instructions)
      POP         LA          ; Restore Hardware Looping Resources
      POP         LC
      RTI
```

Note that this five cycle, five word overhead is not required if the hardware DO loop is not required by the interrupt service routine. Note that this overhead is also not required if only the hardware REP loop is used by the ISR.

If this technique is used, it is important that any ISR that uses hardware DO looping cannot be interrupted by a maskable interrupt and that any non-maskable ISRs save one location of the HWS if they require hardware looping.

For ISRs where it is possible that there are two DO loops currently in progress upon entering the routine, it is necessary to free up one HWS location as well. This is accomplished using the technique described in **Freeing One Hardware Stack Location** on page 8-42.

8.7.6 Early Termination of a DO Loop

There are two techniques that can be used to terminate a DO loop early. In the first technique the loop is terminated such that it continues executing the remainder of the instructions in the loop but will not return back up to the top of the loop. In this case it is best to use the following instruction instead of ENDDO:

```
      MOVE #1,LC.
```

This way, the HWS will purge its value at the correct time, as if there is nesting of hardware DO loops; the LC and LA registers will be popped correctly in software.

There is also the case where it is desirable to conditionally break out of the loop immediately without executing anymore instructions in the loop. In this case it is recommended to use the technique shown in the code below:

```
      PUSH        LC          ; Save outer loop registers if nested loop
      PUSH        LA
      DO          #N,LABEL
      (instructions in loop)
      Bcc         EXITLP      ; 2 Icyc for each iteration
                              ; 3 Icyc if loop terminates when true
;     (instructions)
LABEL
      BRA         OVER        ; 3 additional Icyc for BRA when exiting loop
                              ; if normal exit
EXITLP ENDDO                  ; 1 additional Icyc for ENDDO when exiting
                              ; loop if exit via Bcc
OVER
      POP         LA          ; Restore outer loop registers if nested loop
      POP         LC
;
;     ------ or with another technique ------
;
      PUSH        LC          ; Save outer loop registers if nested loop
      PUSH        LA
      DO          #N,LABEL
;     (instructions)
      Bcc         OVER        ; 3 Icyc for each iteration
      ENDDO                   ; 6 Icyc if loop terminates when true
      BRA         LABEL
OVER
      (instructions)
LABEL
      POP         LA          ; Restore outer loop registers if nested loop
      POP         LC
```

8.8 ARRAY INDEXES

The flexible set of addressing modes on the DSP56800 architecture allow for several different ways to index into arrays. Array indexing usually involves a base address and an offset from this base. The base address is the address of the first location in the array, and the offset indicates the location of the data in the array. For example, the first value in the array typically has an offset of 0, whereas the fourth element has an offset of 3. The n^{th} element is always accessed with an offset of n - 1.

There are two types of arrays typically implemented: global arrays (whose base address is fixed and known at assembly time) and local arrays (whose base address may vary as the program is running). Global arrays that are small in size can benefit from the single word instruction that directly accesses the first 128 locations of the X data memory, as well as the indexed with short displacement addressing mode.

8.8.1 Global or Fixed Array with a Constant

This type of array indexing is performed with the X:#xxxx or X:<aa> addressing mode, where the assembler adds the base address to the constant offset into the array. For arrays that are small in size they can be indexed using the X:<aa> addressing mode, saving one program word and one instruction cycle. It is also possible to use the X:(Rn+xxxx) or X:(R2+xx) addressing modes, if the base address of the array is stored in a Rn register.

8.8.2 Global or Fixed Array with a Variable

This type of array indexing is performed with the X:(Rn+xxxx), X:(R2+xx), or X:(Rn+N) addressing mode.

In first two addressing modes—X:(Rn+xxxx) or X:(R2+xx)—the constant value specifies the base address of the array, and Rn or R2 specifies the offset into the array. These first two are similar to the method used by microcontrollers and are useful when only one or two accesses are performed with a particular base address because it is not necessary to load a register with the base address. The X:(R2+xx) addressing mode executes in one less instruction cycle and uses one less instruction word than the X:(Rn+xxxx) addressing mode. It is useful for arrays whose base address is located in the first few locations in X data memory.

In the last addressing mode—X:(Rn+N)—Rn is the base address of the array, and N specifies the offset. This addressing mode is best for the case where many accesses are to be performed into an array. In this case the base address is only loaded once into the Rn register and then many accesses can be performed using the X:(Rn+N) addressing mode. This addressing mode uses a single program word and executes in two instruction cycles.

8.8.3 Local Array with a Constant

This type of array indexing is done with the X:(Rn+xxxx) or X:(R2+xx) addressing mode, where Rn holds the base address of the array, and the constant value specifies the constant offset into the array. It can also be done with the X:(SP+#xxxx) or X:(SP-#xx) addressing mode, but this is not as straightforward. In this case SP holds the address of the end of the stack frame, and the base address of the array is located using a constant offset value from the stack pointer. The constant used to index into this local array is added to the offset of the base address from the stack pointer to access the desired location of an array stored within the stack frame. Stack frames are discussed in **Parameters and Local Variables** on page 8-36.

8.8.4 Local Array with a Variable

This type of array indexing is done with the X:(Rn+N) or X:(SP+N) addressing modes. It is similar to the technique described in **Local Array with a Constant** on page 8-35, but, instead of using a constant index, the register N holds the variable offset into the array.For the case of X:(SP+N), the N register contains the sum of the index into the array and the offset of the array’s base address from the stack pointer.

8.8.5 Array with an Incrementing Pointer

Often it is desired to sequentially access the elements in an array. This type of array indexing is most often done with the X:(Rn)+ addressing mode, where Rn is initialized to the first element of the array of interest and sequentially advances to each next element in the array by the automatic post-incrementing address mode. In special cases it is also possible to use X:(Rn+N), where N holds the base address and Rn is the incrementing array index that is advanced using an LEA (Rn)+ instruction. The latter is useful where it is also necessary to have access to the variable that holds the index into the array, which is held in the Rn register.

8.9 PARAMETERS AND LOCAL VARIABLES

The DSP56800 software stack supports structured programming techniques, such as parameter passing to subroutines and local variables. These techniques can be used for both assembly language programming as well as high level language compilers.

Parameters can be passed to a subroutine by placing these variables on the software stack immediately before performing a JSR to the subroutine. Placing these variables on the stack is referred to as building a “stack frame.” These passed parameters are then accessed in the called subroutines using the stack addressing modes available on the DSP56800. This is demonstrated in the example below (destroys the x0 register):

```
; Example of Subroutine Call With Passed Parameters
      MOVE        X:$35,X0    ; Pointer variable to be passed to subroutine
      LEA         (SP)+       ; Push variables onto stack
      MOVE        X0,X:(SP)+
      MOVE        X:$21,X0    ; 1st data variable to be passed to
subroutine
      MOVE        X0,X:(SP)+  ; Push onto stack
      MOVE        X:$47,X0    ; 2nd data variable to be passed to
                              ; subroutine
      MOVE        X0,X:(SP)   ; Push onto stack
      JSR         ROUTINE1
      POP                     ; Remove the three passed parameters from
                              ; Stack when done
      POP
      POP


ROUTINE1
      MOVE        #5,N        ; Allocate room for local variables
      LEA         (SP)+N
;     (instructions)
      MOVE        X:(SP-9),r0 ; Get pointer variable
      MOVE        X:(SP-7),B  ; Get 2nd data variable
      MOVE        X:(R0),X0   ; Get data pointed to by pointer variable
      ADD         X0,B
      MOVE        B,X:(SP-8)  ; Store sum in 1st data variable
;     (instructions)
      MOVE        #-5,N
      LEA         (SP)+N
      RTS
```

In a similar manner it is also possible to allocate space and to access variables that are locally used by a subroutine, referred to as local variables. This is done by reserving stack locations above the location that stores the return address stacked by the JSR instruction. These locations are then accessed using the DSP56800's stack addressing modes. For the case of local variables, the value of the stack pointer is updated to accommodate the local variables. For example, if five local variables are to be allocated, then the stack pointer is increased by the value of five to allocate space on the stack for these local variables. When a large numbers of variables are allocated on the stack, it is often more efficient to use the (SP)+N addressing mode.

It is possible to support passed parameters and local variables for a subroutine at the same time. In this case the program first pushes all passed parameters onto the stack (see **Figure 8-1**) using the technique outlined in **Multiple Value Pushes** on page 8-24. Then the JSR instruction is executed, which pushes the return address and the SR onto the stack. Upon entering the subroutine, the first thing the subroutine does is to allocate space for local variables by updating the SP, ensuring that any writes to the SP register are always with even values. Then, both passed parameters and local variables can be accessed with the stack addressing modes.

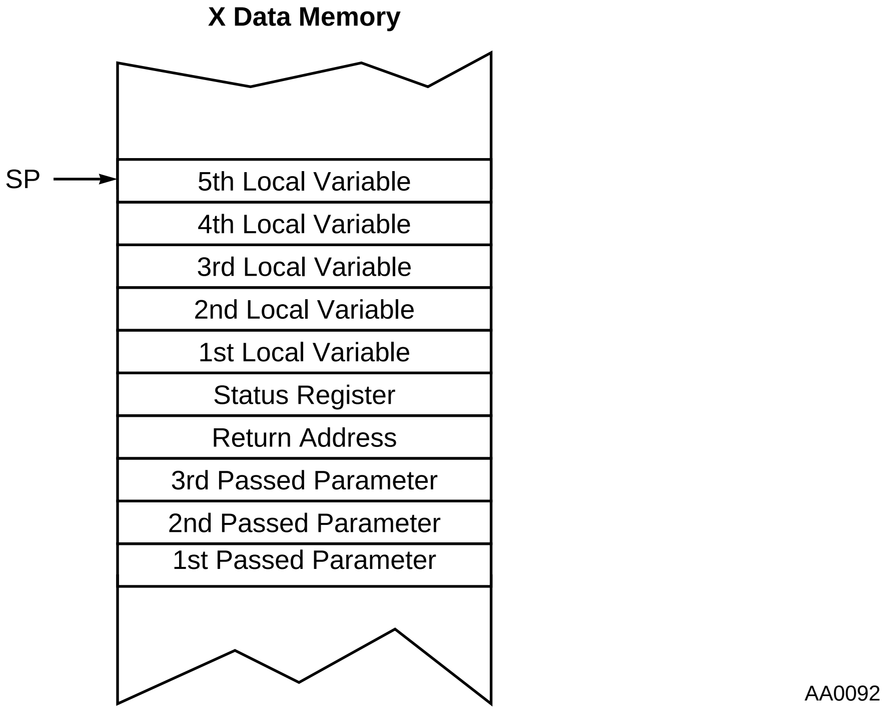


Figure 8-1 Example of a DSP56800 Stack Frame

8.10 TIME-CRITICAL DO LOOPS

Often, the most time a program spends will be in time-critical loops. For these loops it is important to have an adequate number of registers for efficient execution of the loop. However, sometimes the registers already contain data that is not necessary for the critical loop but must not be lost. In this case the DSP56800 architecture provides a convenient mechanism for freeing up these registers using the software stack. The programmer pushes any registers containing values not required in the tight loop, freeing up these registers for use. After completion of the loop, these registers are popped. An example is shown in the code below.

```
      MOVE        #$1234,R3         ; Contents of this register not
                                    ; required in tight loop
      MOVE        #$5aa,A           ; Contents of this register not
                                    ; required in tight loop

      PUSH        R3                ; Prepare for tight loop: X0, Y0 are
                                    ; unused and available, and R0 already
                                    ; points to that required for loop
      PUSH        A0
      PUSH        A1
      PUSH        A2

; Enter Section with Tight Loop - R3 and A can now be used by tight loop
      MOVE        $C000,R3
      CLR         A
      MOVE                    X:(R0)+,Y0  X:(R3)+,X0
      REP         #32
      MAC         X0,Y0,A     X:(R0)+,Y0  X:(R3)+,X0
      MOVE        A,X:(R2)+         ; store result

      POP         A2                ; tight loop completed, restore
                                    ; borrowed registers
      POP         A1
      POP         A0
      POP         R3
```

In the example above there are four PUSH instruction macros in a row. For more efficient and compact code, the technique outlined in **Multiple Value Pushes** on page 8-24 can be used. In certain cases it may also be possible to store critical information within the first 64 locations of X data memory, on the top of the stack, or in an unused register such as N when an extra location is required within a tight loop itself.

8.11 INTERRUPTS

The interrupt mechanism on the DSP56800 is simple, yet flexible. There are two levels of interrupts: maskable and non-maskable. All maskable interrupts on the chip can be masked at one spot in the SR. Likewise, individual peripherals can be individually masked within one register, the interrupt priority register (IPR), or at the peripheral itself. It is beneficial to have a single register in which all maskable interrupts can be individually masked. This capability gives the user the capability of setting up interrupt priorities within software.

When programming interrupts, it is necessary to correctly set up the following:

1. Initialize and program the peripheral, enabling interrupts within the peripheral.
2. Program the IPR to enable interrupts on that particular interrupt channel.
3. Enable interrupts in the SR.

8.11.1 Setting Interrupt Priorities in Software

This section demonstrates several different styles of coding possible for ISRs on the DSP56800 core. In counting the number of overhead instruction cycles below, it is important to remember that the JSR instruction executes in four instruction cycles when entering an interrupt, and that the RTI instruction now takes five instruction cycles to complete.

8.11.1.1 High Priority or a Small Number of Instructions

For ISRs that are short it is recommended that level 0 interrupts remain disabled during these interrupt routines. Since they are short, it is not nearly so important to interrupt them, because they are guaranteed to complete execution quickly. This is also recommended for ISRs with a very high priority, which should not be interrupted by some other source.

```
; Interrupt Service Routine
; DSP56800 core (Interrupts Remain Masked, 9 Overhead Cycles)
      JSR         ISR   ; located in interrupt vector table
ISR                     ; Long ISR
;     (interrupt code)
      RTI
```

8.11.1.2 Many Instructions, Equal Priorities

For ISRs that require a significant number of instruction cycles to complete, it is possible to reduce the interrupt servicing overhead if all interrupts can be considered to have the same priority. This is shown in the following generic ISR.

```
; Interrupt Service Routine for Long Interrupt
; DSP56800 core (Interrupts Remain Masked, 11 Overhead Cycles)
      JSR         ISR   ; located in interrupt vector table
ISR                     ; Long ISR
      BFCLR       #$0200,SR; re-enable interrupts with new mask
;     (interrupt code)
      RTI
```

8.11.1.3 Many Instructions, Programmable Priorities

For ISRs that require a significant number of instruction cycles to complete, it is possible for the user to still program interrupt priorities in software. This is shown in the following generic ISR.

```
; Generic Long ISR on the DSP56100 architecture (5 Overhead Cycles)
      JSR         ISR         ; Instr located in Interrupt Vector Table
;     (instructions)
ISR   ;(interrupt code)       ; Interrupt Service Routine
      RTI

; Generic ISR - DSP56800 core (20 Overhead Cycles)
      JSR         ISR         ; Instr located in Interrupt Vector Table
;     (instructions)
ISR                           ; ISR
      LEA         (SP)+
      MOVE        N,X:(SP)+   ; Save “N” register for usage by ISR
      MOVE        X:IPR,N     ; Save interrupted task’s IPR
      MOVE        N,X:(SP)
      MOVE        #xxxx,X:IPR ; Load new mask - defines which can interrupt
                              ; this ISR
      BFCLR       #$0200,SR   ; Re-enable interrupts with new mask
;     (interrupt code)
      POP         N           ; Restore interrupted task’s IPR
      MOVE        N,X:IPR
      POP         N           ; Restore saved register used by ISR
      RTI
```

8.11.2 Hardware Looping in Interrupt Routines

Since an interrupt can occur at any location in a program, it is possible that the HWS used by hardware DO loops may already be full. If an ISR needs to use the DO looping mechanism, it may be necessary to free up one location in the HWS. This can be done using the technique outline in **Freeing One Hardware Stack Location** on page 8-42. Alternatively, if it can be guaranteed that the main program will never use more than one DO loop at a time (i.e., no nested loops) it may then be possible for an ISR to simply use hardware DO loops without using this technique to free up a stack location.

8.11.3 Interrupts in Programs Larger than 64K

For programs larger than 65,536 locations, it is important to note that the JSR instruction in the interrupt vector table must always go to the first 64K program memory page. For cases where the ISR resides on a different memory page, it is necessary to use the technique outlined in **Extending Programs Beyond 64K Words** on page 5-21.

8.11.4 Identifying System Calls by a Number

In operating systems, system calls are often made by using an SWI instruction when a user’s task needs assistance from the operating system. Often, it is useful to have several different types of system calls, each identified with a number. The following code shows how system calls can have an associated number when an SWI instruction is executed.

```
MOVE        #xx,N ; Put number associated with system call in N reg
PUSH        N     ; Push this value on the stack so accessible by O/S
SWI               ; Generate interrupt to return to O/S
```

8.12 JUMPS AND JSRS USING A REGISTER VALUE

Sometimes it is necessary to perform a jump or jump to subroutine using the value stored in an on-chip register instead of using an absolute address. The RTS instruction is used to perform this task because it takes the value on the software stack and loads it into the program counter, effectively performing a jump.

The register used for the jump can be any register on the DSP core. The code shown below only applies to changes of program flow with a program memory page of 65,536 locations; these techniques cannot cross a 64K program memory page.

```
; JMP <register> Operation
; 8 Icyc
      LEA         (SP)+
                              ;Note: Can use any core register
      MOVE        <register>,X:(SP)+
      MOVE        SR,X:(SP)
      RTS

; Jcc <register> Operation
; 10 Icyc (3 Icyc if condition false)
      Bcc~        OVER        ; (cc~ is the condition exactly opposite the
                              ; desired cc)
      LEA         (SP)+
      MOVE        <register>,X:(SP)+
      MOVE        SR,X:(SP)
      RTS
OVER

; JSR <register> Operation - destroys one register, N
; 11 Icyc
      MOVE        #NEXT,N
      LEA         (SP)+
      MOVE        N,X:(SP)+   ; Push return address onto stack
      MOVE        SR,X:(SP)   ; Push SR onto stack
      MOVE        <register>,X:(SP)+
                              ; Push address of subroutine onto stack
      MOVE        SR,X:(SP)   ; Push SR onto stack
      RTS                     ; Go to address in top 2 values on stack
NEXT
```

8.13 FREEING ONE HARDWARE STACK LOCATION

There are certain cases where a section of code should use DO looping, but it is clear whether the HWS is full or not. An example is an ISR, which may be called when two nested DO loops are in progress. In these cases it may be desirable to free a single location on the HWS for use by a section of code such as an ISR. The code below shows how to free one location for an ISR:

```
; Interrupt Service Routine - Frees Up One HWS Location
; 14 extra Icyc, 12 extra words
;
ISR
      LEA         (SP)+         ; Push four registers onto the stack
      MOVE        LA,X:(SP)+    ; Save LA register in case already in loop
      MOVE        SR,X:(SP)+    ; Save LF bit in SR register...
      MOVE        LC,X:(SP)+    ; Save LC register...
      MOVE        HWS,X:(SP)    ; Save HWS register...
;     (instructions)
      DO          #3,LABEL
      INCW        A
LABEL
;     (instructions)
      POP         LA            ; Conditionally restore HWS
      BRCLR       #$8000,X:(SP-1),_OVER
      MOVE        LA,HWS
_OVER
      POP         LC            ; Restore LC register from stack
      POP                       ; Toss SR register from stack
      POP         LA            ; Restore LA register from stack
      RTI
```

For ISRs that are maskable, it is better to follow the recommendations outlined in **Nested Loops** on page 8-28 to reduce the overhead needed for freeing up one HWS location. This greatly simplifies the setup code required when entering and exiting the ISR.

8.14 MULTI-TASKING AND THE HARDWARE STACK

For multi-tasking, it is important to be able to save and later restore the hardware DO loop stack (HWS). This section shows code that will perform the save and restore operations. When reading the HWS two locations of the stack are read, as well as the current state of the HWS, contained in the NL and LF bits of the OMR and SR, respectively. Each read of the HWS register pops the HWS one value, and each write of the HWS register pushes the HWS one value.

8.14.1 Saving the Hardware Stack

An example of reading the entire contents of the HWS to X memory is shown in the code below:

```
; Save HWS
; 4 Icyc, 4 words
      MOVE        SR,X:(R2)+  ; Read HWS pointer’s LSB (LF) and
                              ; save to memory
      MOVE        HWS,X:(R2)+ ; Read 1st stack location and
                              ; save in X memory
      MOVE        SR,X:(R2)+  ; Read HWS pointer’s MSB (NL) and
                              ; save to memory
      MOVE        HWS,X:(R2)+ ; Read 2nd stack location and
                              ; save in X memory
```

8.14.2 Restoring the Hardware Stack

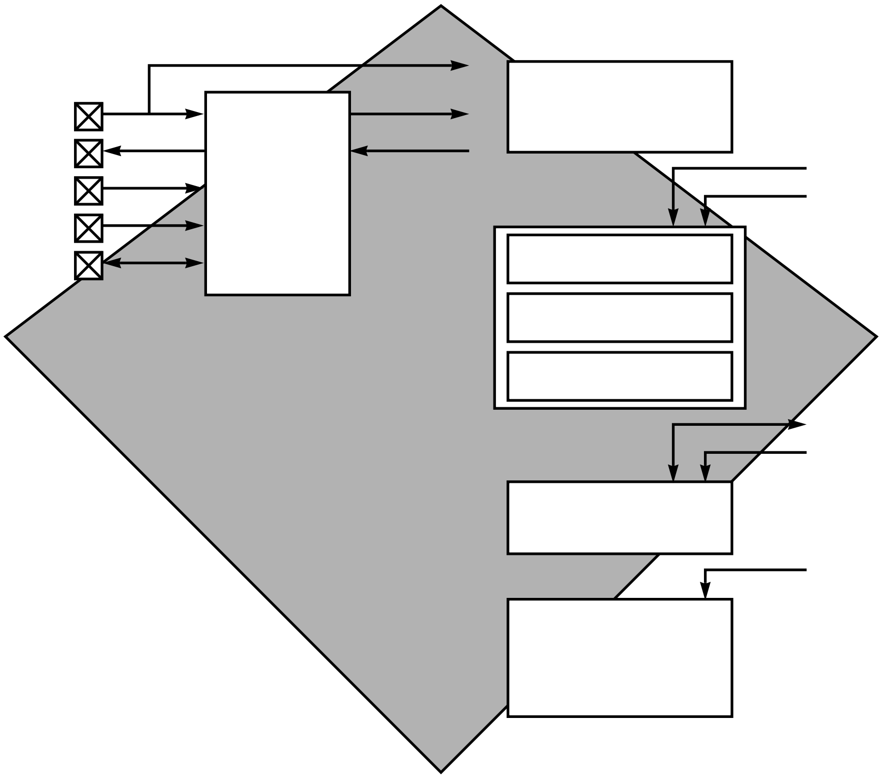
When restoring the HWS, it is first necessary that the HWS be empty. If this is unclear, performing two reads from the HWS will ensure that the stack is empty. Once this is true, then the HWS can be restored. An example of restoring the contents of the HWS from X data memory is shown below:

```
; Restore HWS, 10 words, 14 Icyc worst case
; Assumes R2 points to “stored” HWS
; Destroys R2 register

      MOVE        HWS,LA      ; 1st read of HWS ensures NL bit is cleared
      MOVE        HWS,LA      ; 2nd read of HWS ensures LF bit is cleared
      BRCLR       #$8000,X:(R2),OVER
                              ; If LF bit set, then push a value onto HWS
      LEA         (R2)+
      MOVE        X:(R2)+,HWS ; Puts one value onto stack and sets LF bit
      BRCLR       #$8000,X:(R2),OVER
                              ; If NL bit set, then push a value onto HWS
      LEA         (R2)+
      MOVE        X:(R2)+,HWS
OVER
```

SECTION 9

JTAG /ON-CHIP EMULATION (OnCE)



9.1 INTRODUCTION

The DSP56800 family provides board and chip-level testing capability through two on-chip modules that are both accessed through the JTAG / OnCE port. These modules are:

- On-Chip Emulation (OnCE) port
- Test access port and 16-state controller (commonly called the JTAG port)

The presence of the JTAG / OnCE interface allows the user to insert the DSP chip into a target system while retaining debug control. This capability is especially important for devices without an external bus, because it eliminates the need for a costly cable to bring out the footprint of the chip, as required by a traditional emulator system.

The OnCE port is a Motorola-designed module used in DSP chips to debug application software used with the chip. The port is a separate on-chip block that allows non-intrusive interaction with the DSP and is accessible through the pins of the JTAG interface. The OnCE port makes it possible to examine registers, memory or on-chip peripherals contents in a special debug environment. This avoids sacrificing any user accessible on-chip resources to perform debugging. The capabilities of the OnCE Port include:

- Interrupt / break into debug mode on a program memory address
- Interrupt / break into debug mode on a data memory address (read, write, or access)
- Interrupt / break into debug mode on an on-chip peripheral register access
- Enter debug mode using a DSP microprocessor instruction
- Display / modify the contents of any DSP core register
- Display / modify the contents of peripheral memory-mapped registers
- Display / modify any desired sections of program or data memory
- Full speed stepping on one (single stepping) or more instructions (up to 256)
- Trace one or more instructions
- Save or restore the current state of the DSP chip’s pipeline
- Display the contents of the real-time instruction trace buffer
- Return to user mode from debug mode

See **OnCE Port** on page 9-7 for a detailed description of this port.

The JTAG port conforms to the IEEE 1149.1a-1993 IEEE Standard Test Access Port and Boundary Scan Architecture specification defined by the Joint Test Action Group (JTAG). Five dedicated pins interface to a test access port (TAP) that contains a 16-state controller. The TAP uses a boundary scan technique to test the interconnections between integrated circuits after they are assembled onto a printed circuit board. Boundary scan allows a tester to observe and control signal levels at each component pin through a shift register placed next to each pin. This is important for testing continuity and determining if pins are stuck at a one or zero level.

The JTAG port has the following capabilities:

- Perform boundary scan operations to test circuit-board electrical continuity
- Bypass the DSP for a given circuit-board test by replacing the boundary scan register with a single bit register
- Sample the DSP system pins during operation, and transparently shift out the result in the boundary scan register. Pre-load values to output pins prior to invoking the EXTEST instruction
- Disable the output drive to pins during circuit-board testing
- Provide a means of accessing the OnCE controller and circuits to control a target system
- Query identification information (manufacturer, part number, and version) from a DSP IC
- Force test data onto the outputs of a DSP IC(s) while replacing its boundary scan register in the serial data path with a single bit register
- Enable a weak pull-up current device on all input signals of a DSP IC(s); this helps to ensures deterministic test results in the presence of a continuity fault during interconnect testing

See **TAP Controller** on page 9-59 for a detailed description of this port.

9.2 COMBINED JTAG/OnCE INTERFACE OVERVIEW

The JTAG and OnCE blocks are tightly coupled. **Figure 9-1** shows the block diagram of the JTAG / OnCE interface with its two distinct parts: the JTAG port and the OnCE port. The JTAG port is the master; it must enable the OnCE port before the port can be accessed.

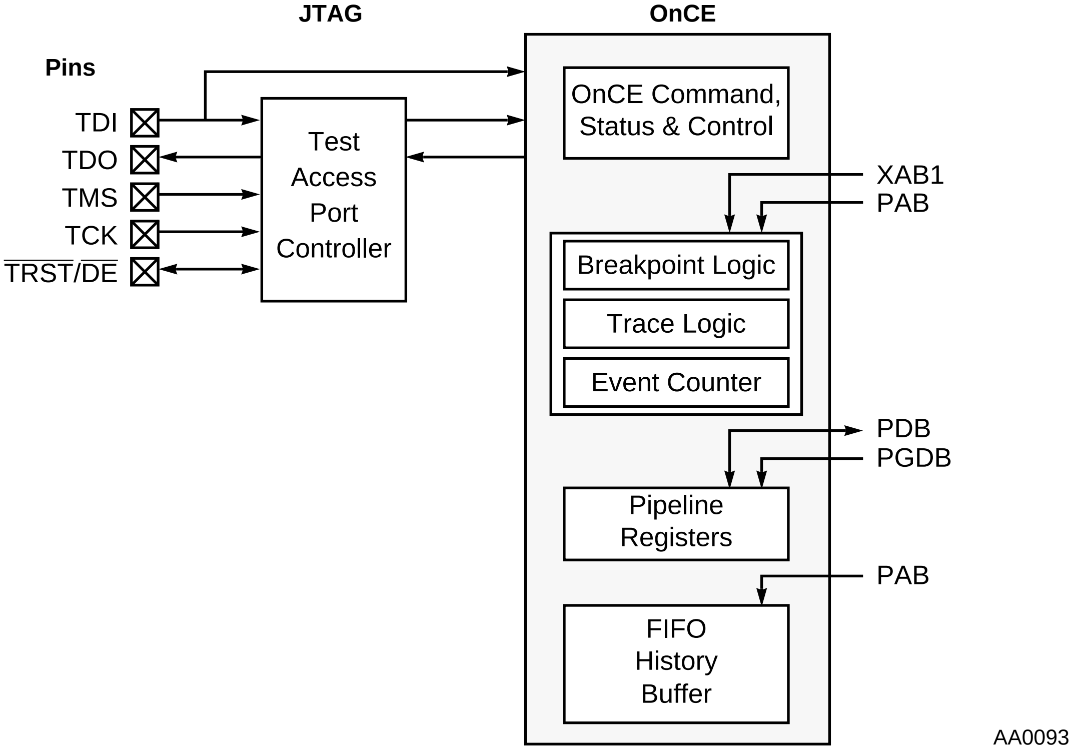


Figure 9-1 JTAG/OnCE Interface Block Diagram

There are three different programming models to consider when using the JTAG / OnCE interface:

- OnCE programming model—accessible through the JTAG port
- OnCE programming model—accessible from the DSP core
- JTAG programming model—accessible through the JTAG port

The programming models are discussed in more detail in **OnCE Port** on page 9-7 and **JTAG Port Architecture** on page 9-51.

9.2.1 JTAG and OnCE Port Pinout

As described in the IEEE 1149.1a-1993 specification, the JTAG port requires a minimum of 4 pins to support TDI, TDO, TCK, and TMS signals. The DSP56800 family also uses the optional $\overline{\text{TRST}}$ input signal and multiplexes it so that the same pin can support the debug event ($\overline{\text{DE}}$) output signal used by the OnCE interface. The pin functions are described on the following page in **Table 9-1**.

Table 9-1 JTAG Pin Descriptions

Pin Name	Pin Description
TDI	Test Data Input—This input pin provides a serial input data stream to the JTAG and the OnCE port. It is sampled on the rising edge of TCK and has an on-chip pull-up resistor.
TDO	Test Data Output—This tri-state-able output pin provides a serial output data stream from the JTAG and the OnCE port. It is driven in the Shift-IR and Shift-DR controller states of the JTAG state machine, and changes on the falling edge of TCK.
TCK	Test Clock Input—This input pin proves a gated clock to synchronize the test logic and shift serial data to and from the JTAG / OnCE port. The maximum frequency for TCK is 1 / 8 the maximum frequency specified for the DSP56800 core (i.e., 5 MHz for TCK if the maximum CLK input is 40 MHz)
TMS	Test Mode Select Input—This input pin is used to sequence the JTAG TAP controller's state machine. It is sampled on the rising edge of TCK and has an on-chip pull-up resistor.
$\overline{\text{TRST}}$/$\overline{\text{DE}}$	Test Reset / Debug Event—This bidirectional pin, when configured as an input, provides a reset signal to the JTAG TAP controller. When configured as an output, it signals debug events detected on a trigger condition. The operational mode of the pin is configured by bit 14 of the Once Control Register (OCR). (See **OnCE Control Register (OCR)** on page 9-14.)

9.2.2 Using the Bidirectional $\overline{\text{TRST}}$/$\overline{\text{DE}}$ Pin

The $\overline{\text{TRST}}$ / $\overline{\text{DE}}$ pin provides useful debug acknowledge features similar to those in earlier OnCE implementations. Additionally, the pin can be used for debugging multiple DSP configurations. **Figure 9-2** shows the internal configuration of the $\overline{\text{TRST}}$ / $\overline{\text{DE}}$ pin. The open drain output function is used to indicate a debug event has occurred. For example, a trace or breakpoint occurrence will cause the $\overline{\text{DE}}$ output to go low for a period of 8 phi clocks. The circuitry in this pin will be further described in **Event Modifier (EM1 and EM0)—Bits 6-5** on page 9-17.

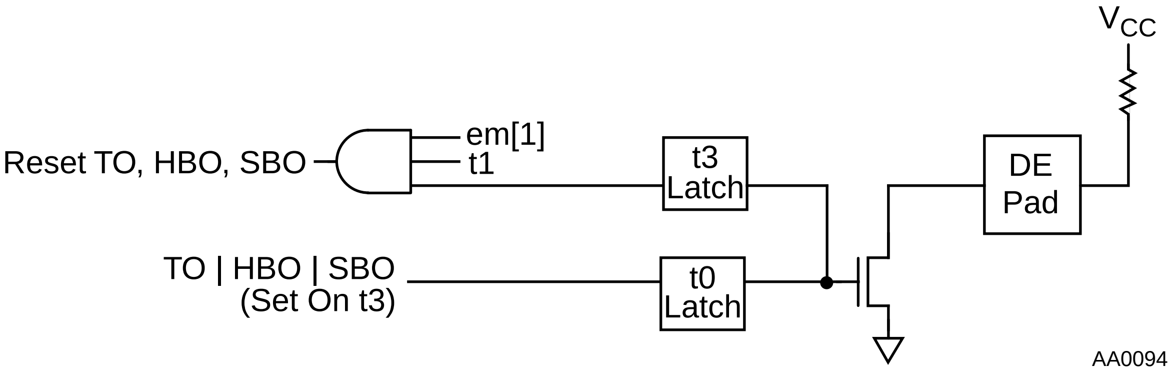


Figure 9-2 Debug Event Pin

- $\overline{\text{TRST}}$—When enabled, this input resets of the JTAG TAP controller state machine.
- $\overline{\text{DE}}$—When enabled, this tri-statable output provides an open drain signal that indicates an event has occurred in the OnCE debug logic. This event can be an entry to debug mode, a stop of the FIFO, a trace count decrement to zero, or a vectored interrupt taken due to one of the above.

9.3 OnCE PORT

While the JTAG port described in **JTAG Port Architecture** on page 9-51 provides board test capability, the OnCE port provides emulation and debug capability to the user. The OnCE port permits full-speed, non-intrusive emulation on a user's target system or on a Motorola Application Development Module (ADM) board. This section describes the environment in which the OnCE port is used for debugging a real-time embedded application.

A typical debug environment consists of a target system where the DSP resides in the user-defined hardware. The DSP's JTAG port interfaces to a command convertor board over a six-wire link consisting of the four JTAG serial lines, a ground, and reset wire. The reset wire is optional and is only used to reset the DSP and its associated circuitry. See **Hardware Development Environment** on page 10-8 for more information on the DSP56800 hardware development tools.

9.3.1 OnCE Port Architecture

The OnCE port is composed of four different sub-blocks, each of which performs a different task:

- Command, status, and control
- Breakpoint and trace
- Pipeline registers
- FIFO history buffer

A block diagram of the OnCE port is shown in **Figure 9-3**. **Figure 9-4** shows the OnCE registers accessible through the JTAG port. **Figure 9-5** shows the OnCE registers accessible through the DSP core and the corresponding OnCE interrupt vector.

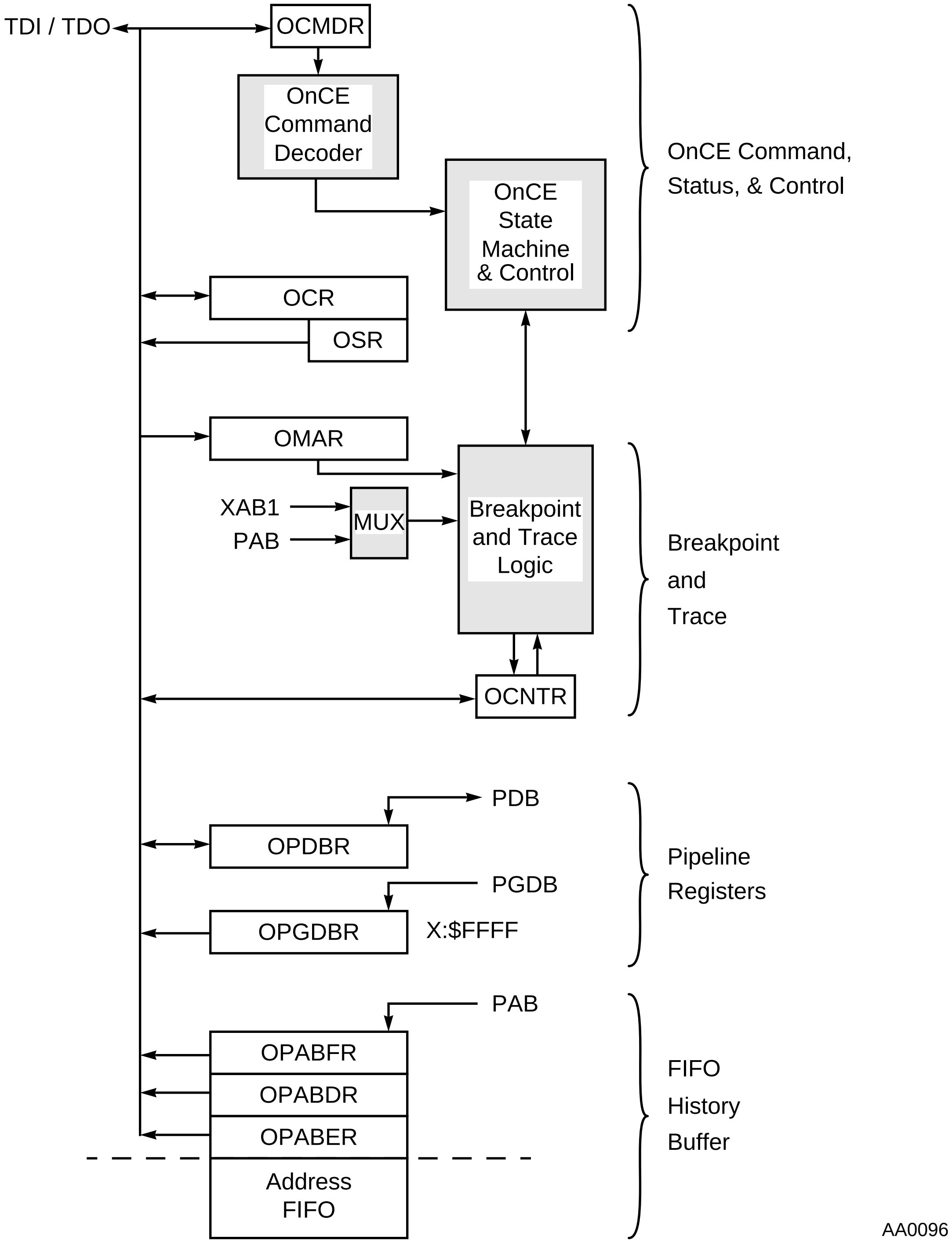
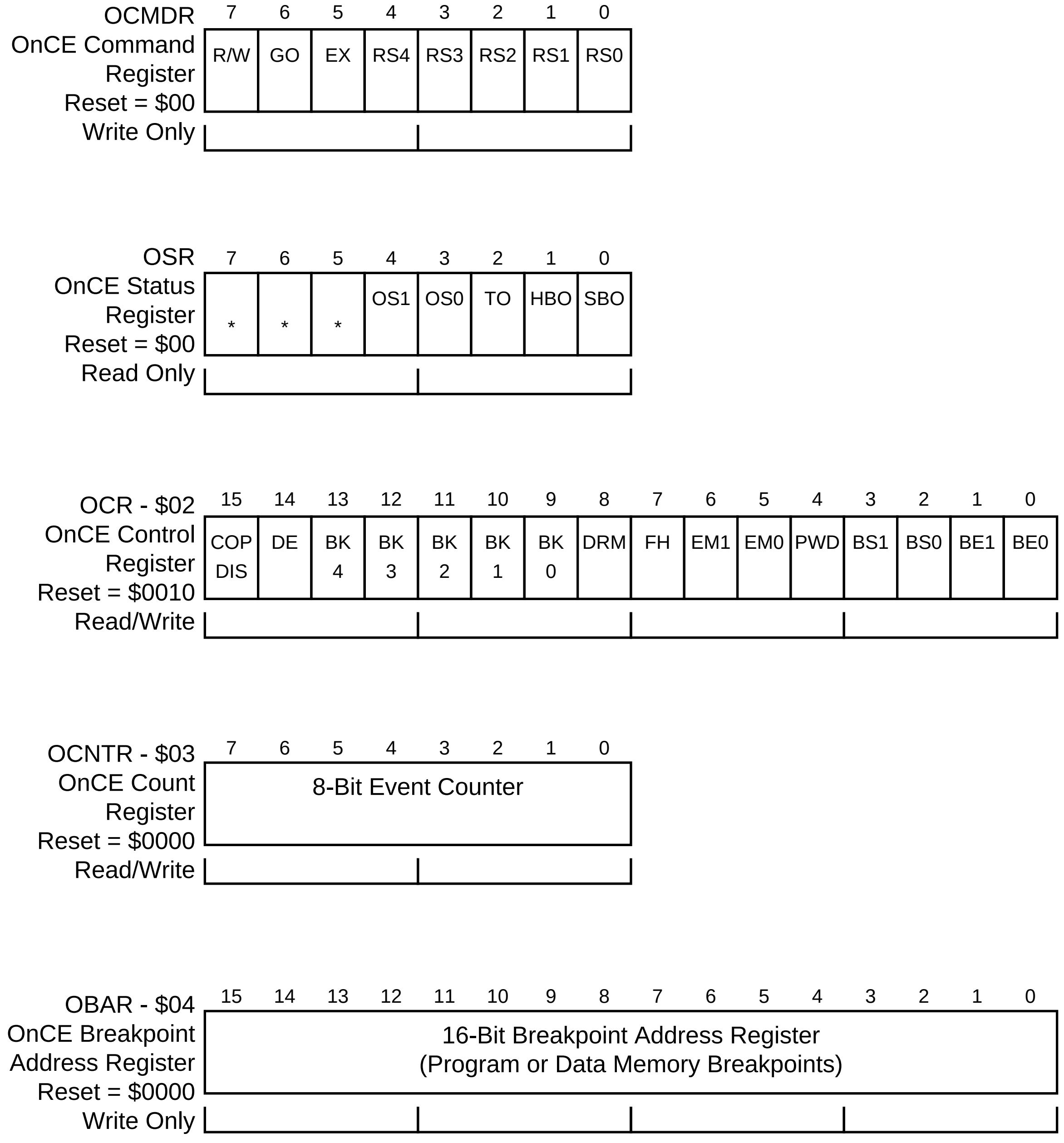


Figure 9-3 OnCE Block Diagram



OCMDR
OnCE Command Register
Reset = $00
Write Only

7	6	5	4	3	2	1	0
R/W	GO	EX	RS4	RS3	RS2	RS1	RS0

OSR
OnCE Status Register
Reset = $00
Read Only

7	6	5	4	3	2	1	0
*	*	*	OS1	OS0	TO	HBO	SBO

OCR - $02
OnCE Control Register
Reset = $0010
Read/Write

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COP DIS	DE	BK 4	BK 3	BK 2	BK 1	BK 0	DRM	FH	EM1	EM0	PWD	BS1	BS0	BE1	BE0

OCNTR - $03
OnCE Count Register
Reset = $0000
Read/Write

7	6	5	4	3	2	1	0
8-Bit Event Counter							

OBAR - $04
OnCE Breakpoint Address Register
Reset = $0000
Write Only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16-Bit Breakpoint Address Register (Program or Data Memory Breakpoints)															

* Indicates reserved bits, written as zero for future compatibility

AA0104a

Figure 9-4 OnCE Port Programming Model—Accessed Through JTAG

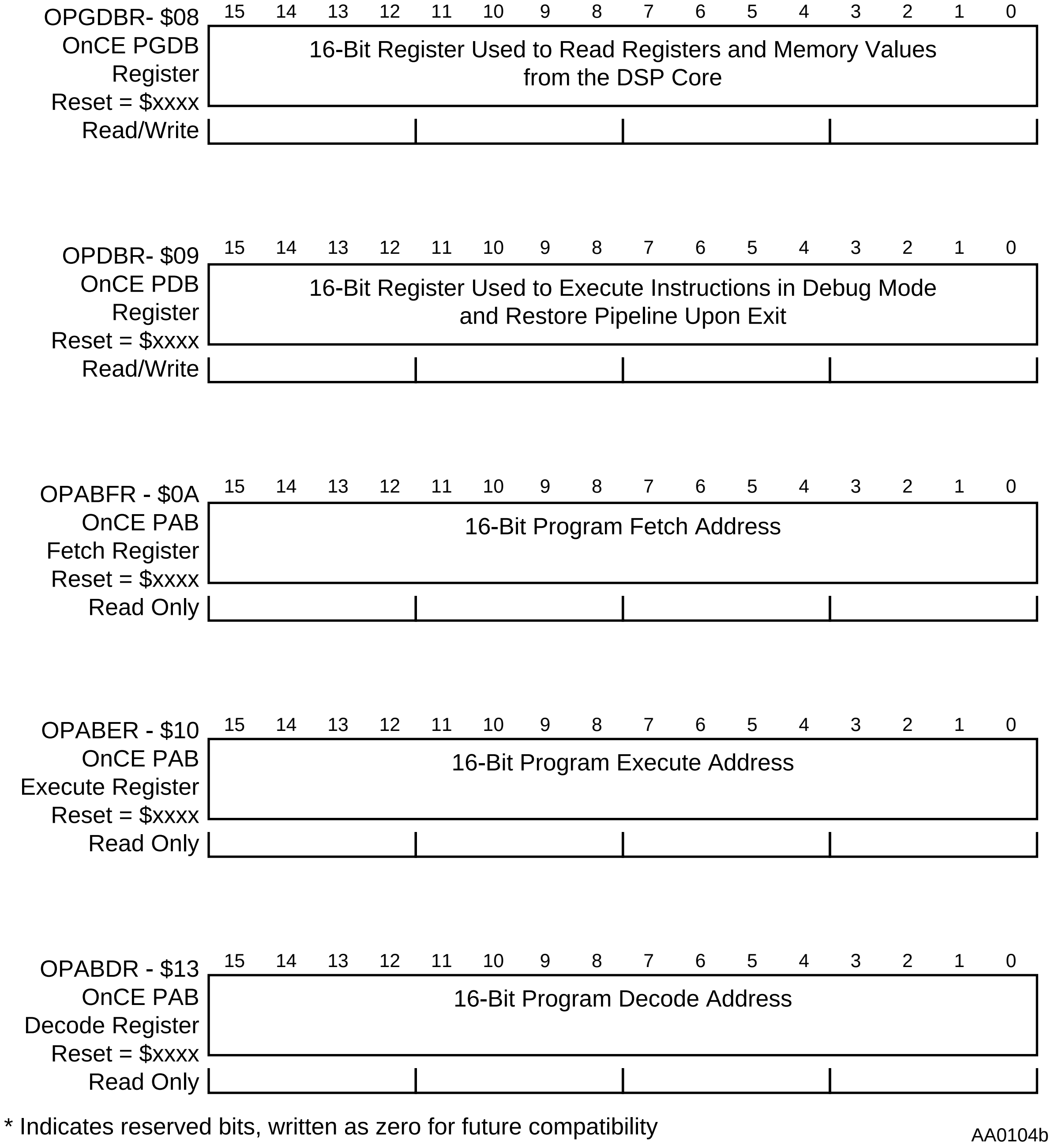


Figure 9-4 OnCE Port Programming Model—Accessed Through JTAG (Continued)

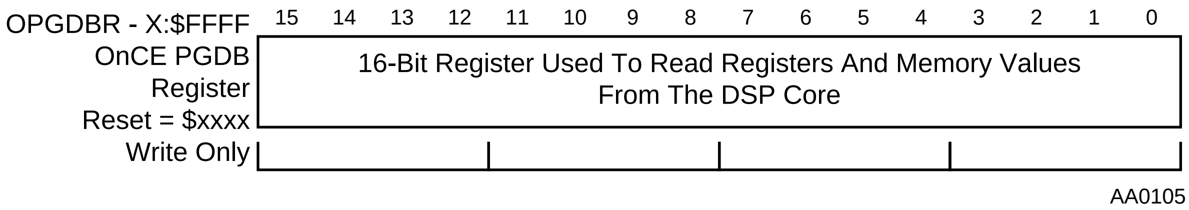


Figure 9-5 OnCE Port Programming Model—Accessed from the Core

The OnCE port has an associated interrupt vector 000C. The OnCE exception trap is available to the user so that when a debug event (breakpoint or trace occurrence) is detected, a level 1 non-maskable interrupt can be generated, and the program can initiate the appropriate handler routine.

9.3.1.1 OnCE Command, Status, And Control Section

The OnCE controller and serial interface contains the following blocks:

- OnCE input shift register (OISR) and bit counter
- OnCE command register (OCMDR)
- OnCE decoder (ODEC)
- OnCE control register (OCR)
- OnCE status register (OSR)
- OnCE state machine and control

9.3.1.1.1 OnCE Input Shift Register (OISR)

The OnCE input shift register (OISR) is a 16- or 8-bit shift register that receives the serial data from the TDI line. The data is clocked into the register on the falling edge of the clock applied to the TCK pin until after the 8th bit is received. Data is latched into the OCMDR as input for the OnCE decoder. Data is always shifted into the OISR MSB first.

9.3.1.1.2 OnCE Command Register (OCMDR)

The OnCE Port has its own instruction register and instruction decoder. After a command is latched into the OnCE command register (OCMDR), the command decoder implements the instruction through the OnCE state machine and control block. There are two type of commands: read commands that cause the chip to deliver required data, and write commands that transfer data into the chip and write it in one of the on-chip resources. The commands are 8 bits long and have the format shown in **Figure 9-6**. The lowest 5 bits (RS4-RS0) identify the source for the operation, defined in **Table 9-2**. Bit 5 (EX), bit 6 (GO), and bit 7 ($R/\overline{W}$) define the exit command bit (**Table 9-3**), the execute bit (**Table 9-4**), and the read / write bit (**Table 9-5**), respectively.

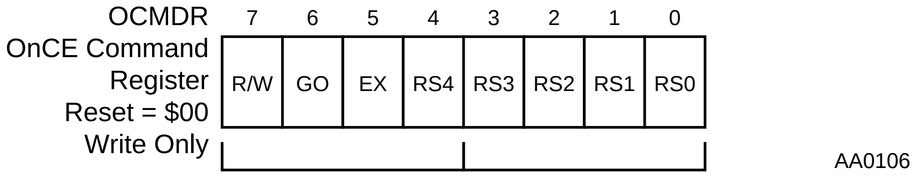


Figure 9-6 OnCE Command Format

Table 9-2 Register Select Encoding

RS4-RS0	Register/Action Selected
00000	No Register Selected
00001	Breakpoint and trace counter (OCNTR)
00010	Debug control register (OCR)
00011	(Reserved for an EPAB memory address register)
00100	OnCE breakpoint address register (OBAR)
00101	(Reserved for a second breakpoint address, OBAR2)
00110	(Reserved for a third breakpoint address, OBAR3)
00111	(Reserved for a fourth breakpoint address, OBAR4)
01000	PGDB bus transfer register (OPGDBR)
01001	Program data bus register (OPDBR)
01010	Program address register—fetch cycle (OPABFR)
01011	(Reserved)
01100	Clear OCNTR
01101	(Reserved)
01110	(Reserved)
01111	Debug request
10000	Program address register—execute cycle (OPABER)
10001	Program address FIFO (OPFIFO)
10010	(Reserved)
10011	Program address register—decode cycle (OPABDR)
101xx	(Reserved)
11xxx	(Reserved)

Table 9-3 EX Bit Definition

EX	Action
0	Remain in debug mode
1	Leave debug mode
Note: 1. Bit 5 in the OnCE command word is the exit command. To leave the OnCE mode and re-enter the normal operating mode, both the EX and GO bits must be asserted in the OnCE input command register.	

Table 9-4 GO Bit Definition

GO	Action
0	Inactive—no action taken
1	Execute DSP instruction

Table 9-5 $R/\overline{W}$ Bit Definition

$R/\overline{W}$	Action
0	Write to the register specified by the RS4-RS0 bits
1	Read from the register specified by the RS4-RS0 bits

9.3.1.1.3 OnCE Decoder (ODEC)

The OnCE decoder (ODEC) decodes all OnCE instructions received in the OCMDR. It receives as input the 8-bit command from the OISR, two signals from JTAG port (one indicating that 8 bits have been received and the other that 16 bits have been received), and one signal indicating that the DSP has halted. The ODEC generates all the strobes required for reading and writing the selected OnCE registers.

9.3.1.1.4 OnCE Control Register (OCR)

The 16-bit OnCE control register (OCR) selects the events that will put the chip in debug mode. **Figure 9-7** illustrates the OnCE control register organization.

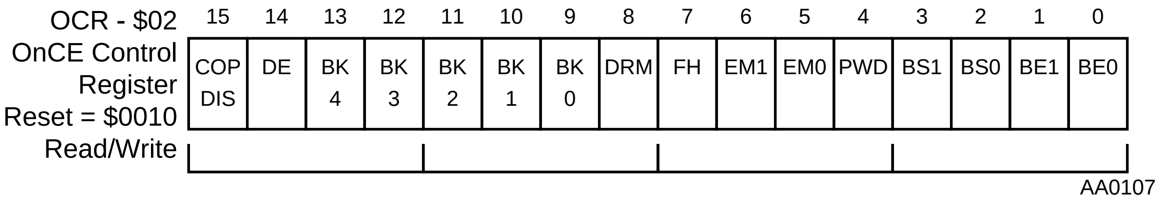


Figure 9-7 OnCE Control Register

The OCR includes the following control bits and bit fields:

- Breakpoint enable field (BE1 and BE0)
- Breakpoint selection field (BS1 and BS0)
- Power down mode (PWD)
- Event modifier field (EM1 and EM0)
- FIFO halt (FH)
- Debug request mask (DRM)
- Breakpoint configuration field (BK4-BK0)
- $\overline{\text{DE}}$ pin enable (DE)
- COP timer disable (COPDIS)

9.3.1.1.4.1 Breakpoint Enable (BE1 and BE0)—Bits 1-0

The breakpoint enable (BE1 and BE0) control bits (OCR bits 1-0) enable or disable the breakpoint logic and select the type of memory operations (read, write, or access) upon which the breakpoint logic operates. These bits are cleared on hardware reset. **Table 9-6** describes the bit functions.

Table 9-6 BE1-BE0 Bit Definition

BE1	BE0	Selection
0	0	Breakpoint disabled
0	1	Breakpoint enabled on memory write
1	0	Breakpoint enabled on memory read
1	1	Breakpoint enabled on memory access

Note: Breakpoints should remain disabled until the OMAR register is loaded. See **OnCE Breakpoint and Trace Section** on page 9-26 for a more complete description of tracing and breakpoints, and **Entering Debug Mode from User Mode** on page 9-38 for a description of how to correctly set up a breakpoint. Breakpoints may be disabled or enabled for one memory space. The trace mode of operation is selected through OSCR.

9.3.1.1.4.2 Breakpoint Selection (BS1 and BS0)—Bits 3-2

The breakpoint selection (BS1 and BS0) control bits (OCR bits 3-2) select if the breakpoints will be recognized on program memory fetch, program memory access, X memory access or second X memory read. These bits are cleared on hardware reset (see **Table 9-7**).

Table 9-7 BS1-BS0 Bit Definition

BS1-BS0	Action on Occurrence of an Event
00	Breakpoint on program memory fetch (fetch of the first word of instructions that are actually executed, not of those that are killed, not of those that are the second word of two-word instructions, and not of jumps that are not taken)
01	Breakpoint on any program memory access (any MOVEM instructions, fetches of instructions that are executed and of instructions that are killed, fetches of second word of two-word instructions, and fetches of jumps that are not taken)
10	Breakpoint on the first X memory access—XAB1 / CGDB access
11	(Reserved)

Note: It is not possible to set a breakpoint on the XAB2 bus, used in the second access of a dual read instruction.

The BS1-BS0 bits work in conjunction with the BE1-BE0 bits to determine how the address breakpoint hardware is setup. The decoding scheme for BS1-BS0 and BE1-BE0 is shown in **Table 9-8**.

Table 9-8 Breakpoint Programming with the BS1-0, BE1-0 Bits

Function	BS1-BS0	BE1-BE0
Disable all breakpoints	XX	00
(Reserved)	00	01
Program Instruction Fetch	00	10
(Reserved)	00	11
Any program write or fetch	01	01
Any program read or fetch	01	10
Any program access or fetch	01	11
XAB1 write	10	01
XAB1 read	10	10
XAB1 access	10	11
(Reserved)	11	01
(Reserved)	11	10
(Reserved)	11	11

Note: When all breakpoints are disabled with the BE1-BE0 bits set to 00, the full-speed instruction tracing capability is not affected. See **OnCE Breakpoint and**

Trace Section on page 9-26 for a more complete description of tracing and breakpoints and **Entering Debug Mode from User Mode** on page 9-38 for a description of how to correctly set up a breakpoint.

9.3.1.1.4.3 Power Down Mode (PWD)—Bit 4

The user may set/reset the power down mode (PWD) bit (OCR bit 4) by writing to the OCR at any time. On core reset, the bit automatically sets to one (power down mode) if the JTAG TAP controller is not decoding an ENABLE_ONCE command. If the ENABLE_ONCE command is being decoded, the bit can only be set/cleared through a OnCE write command to the OCR.

When OnCE is powered down (PWD equals 1), much of the OnCE is shut down, though the following two things can still occur:

- JTAG DEBUG_REQUEST instruction still halts the core.
- The OnCE state machine is still accessible so that the user can write to the OCR.

Note: DEBUG instructions executed by the DSP56800 core are ignored if PWD is set.

To monitor the activity of the DSP core when exiting reset, the JTAG TAP controller must be decoding ENABLE_ONCE, and the PWD bit must be cleared.

9.3.1.1.4.4 Event Modifier (EM1 and EM0)—Bits 6-5

The event modifier (EM1 and EM0) bits (OCR bits 6-5) allow different actions to take place when an event (breakpoint or trace occurrence) happens. If EM[1:0] equals 00, the core halts when the current instruction finishes executing and the DSP goes into the debug processing state. If EM[1:0] equals 01, the core does not halt when an event occurs but FIFO, OPABFR and OPABDR capture is halted and the respective event occurrence flag gets set in the OSR. If EM[1:0] equals 10 and an event occurs, the core does not halt but a level 1 interrupt will occur and the program will vector to location P:$000C in program memory. P:$000C is the location for the OnCE trap in the interrupt vector table (**Figure 7-2 Interrupt Priority Register, X$FFFB** on page 7-14). If EM[1:0] equals 11, the core does not halt when an event occurs, and the event is automatically re-enabled. Again, the respective event occurrence flag will also be set in OSR. In all cases, the DE pin, when present, will be pulsed low for 8 phi clocks when an event occurs. **Table 9-9** summarizes the different EM encodings.

Table 9-9 Event Modifier Selection

EM1-EM0	Function	Action on Occurrence of an Event
00	Enter debug	Halt core and enter the debug processing state (FIFO capture is automatically halted)

Table 9-9 Event Modifier Selection (Continued)

EM1-EM0	Function	Action on Occurrence of an Event
01	FIFO halt	Halt capture by OPABFR, OPABDR, FIFO, Re-arm event, and user program continues running
10	Vector enable	User program interrupted by OnCE TRAP, Program execution goes to P:$000C, Continue FIFO capturing, re-arm event, and user program continues running
11	Re-arm	Re-arm event, continue FIFO capturing, and user program continues running

When EM[1:0] equals 00, the event occurrence flags are automatically cleared when the core leaves debug mode. When EM[1:0] equals 01, the event occurrence flags are cleared by any write to the OCR. When EM[1:0] equals 10 or 11, the respective flag is cleared automatically, thereby re-arming the event; but the breakpoint and trace counters will not be reloaded (i.e., trace / breakpoint counters will be 0 so that next occurrence of either will cause an event). This automatic re-arming is desirable if the 10 encoding is being used for a PROM patch or the 11 encoding is used for profiling code.

The event modifier bits add some powerful debug techniques to the OnCE module. Users can profile code more easily with the 01 encoding or perform special tasks when events occur with the 10 encoding. The most attractive feature of the 10 encoding is the ability to patch the PROM. If a section of code in PROM is incorrect, the user can set a breakpoint at the starting address of the bad code and vector off to a PRAM location where the patch resides. The 11 encoding is useful for toggling the $\overline{\text{DE}}$ pin output. The user can count events on the $\overline{\text{DE}}$ output and determine how much time is being spent in a certain subroutine or other useful things. $\overline{\text{DE}}$ is held low for two cycles to avoid transmission line problems at the board level at high internal clock speeds. This restricts event recognition to no more than 1 every 3 internal clock cycles, limiting its usefulness during tracing. **Figure 9-8** shows a timing diagram of the $\overline{\text{DE}}$ pin during an automatic re-arm after a breakpoint occurrence.

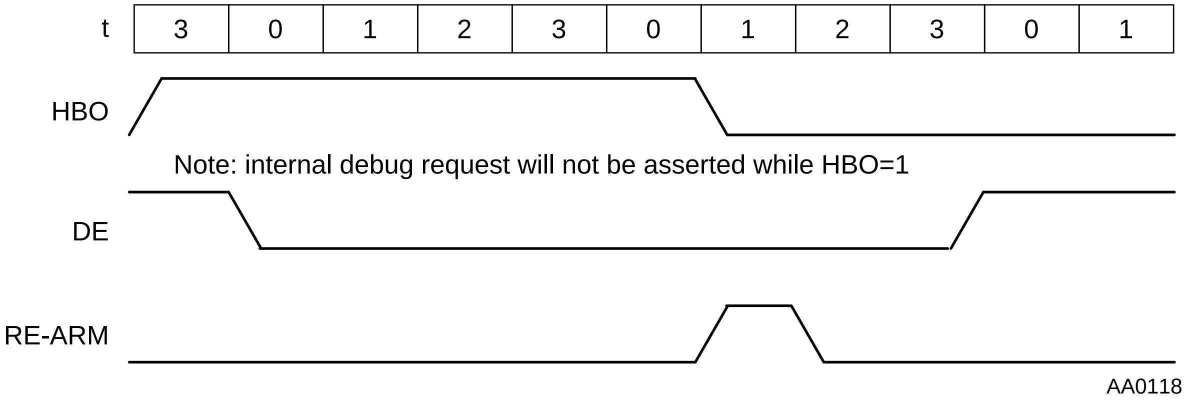

t
3
0
1
2
3
0
1
2
3
0
1
HBO
Note: internal debug request will not be asserted while HBO=1
DE
RE-ARM
AA0118

Figure 9-8 Re-arm Timing Diagram

9.3.1.1.4.5 FIFO Halt (FH)—Bit 7

The FIFO halt (FH) bit (OCR bit 7) allows the user to halt data capture in the FIFO, OPABFR, and OPABDR to read the register contents when FH equals 1. To re-enable capture, FH must be set to 0. FH will be set to 0 on reset to allow for normal FIFO capture.

9.3.1.1.4.6 Debug Request Mask (DRM)—Bit 8

Some members of the DSP56800 Family may implement an external debug request input. In such chips, the debug request mask (DRM) bit (OCR bit 8) is used to mask this external debug request. When this bit is set to 0, a pulse on the debug request input pin will cause the DSP to enter the debug mode of operation. When this bit is set to 1, the DSP will not enter debug mode when a pulse enters the debug request input pin.

9.3.1.1.4.7 Breakpoint Configuration (BK4-BK0)—Bits 13-9

The breakpoint configuration (BK4-BK0) bits (OCR bits 13-9) are used to configure the operation of the OnCE module when it enters the debug operating state. **Table 9-10** describes the functions performed for each configuration combination.

Table 9-10 Breakpoint Configuration Bits Encoding

BK4-BK0	Final Trigger Combination to Enter Debug Mode or Interrupt
00000	Breakpoint 1 must occur to enter debug mode
00001	Either Breakpoint 1 or breakpoint 2 can occur to enter debug mode
00010	Both Breakpoint 1 and breakpoint 2 must occur to enter debug mode
00011	Illegal—may cause unpredictable results

Table 9-10 Breakpoint Configuration Bits Encoding (Continued)

BK4-BK0	Final Trigger Combination to Enter Debug Mode or Interrupt
00100	Breakpoint 1 must occur to enter debug mode; Breakpoint 2 occurrence generates a OnCE interrupt
00101	Illegal—may cause unpredictable results
00110	Illegal—may cause unpredictable results
00111	Illegal—may cause unpredictable results
01000	Illegal—may cause unpredictable results
01001	Illegal—may cause unpredictable results
01010	Illegal—may cause unpredictable results
01011	If Breakpoint 2 occurs, breakpoint 1 must occur the number of times specified by the value in ONCTR before entering debug mode
01100	Illegal—may cause unpredictable results
01101	Illegal—may cause unpredictable results
01110	Illegal—may cause unpredictable results
01111	Breakpoint 2 must occur the number of times specified by the value in ONCTR; after this, a breakpoint 1 occurrence invokes debug mode
10000	Breakpoint 1 occurrence invokes trace mode (ONCTR contents specify number of steps)
10001	Occurrence of breakpoint 1 or breakpoint 2 invokes trace mode (ONCTR contents specify number of steps)
10010	Breakpoints 1 AND 2 must occur to invoke Trace Mode (ONCTR contents specify number of steps)
10011	Illegal—may cause unpredictable results
10100	Breakpoint 1 occurrence invokes trace mode (ONCTR contents specify number of steps); breakpoint 2 occurrence generates a OnCE interrupt
10101	Illegal—may cause unpredictable results
10110	Illegal—may cause unpredictable results
10111	Trace mode using the value set in ONCTR.
11000	Illegal—may cause unpredictable results
11001	Illegal—may cause unpredictable results
11010	Illegal—may cause unpredictable results
11011	Breakpoint 2 must occur first, followed by breakpoint 1 to enter trace mode (ONCTR contents specify number of steps)
11100	Illegal—may cause unpredictable results
11101	Illegal—may cause unpredictable results

Table 9-10 Breakpoint Configuration Bits Encoding (Continued)

BK4-BK0	Final Trigger Combination to Enter Debug Mode or Interrupt
11110	Illegal—may cause unpredictable results
11111	Illegal—may cause unpredictable results

Note: 1. If a hardware address breakpoint is enabled, full-speed stepping *will* ignore the address breakpoints. This is not an issue for single stepping but becomes an issue when a step count greater than 1 is used.
2. The chip must have two accumulators to use all of the specified functionality described in this table. Refer to the appropriate user's manual to determine which configurations are available. For example, chips with a single accumulator can only use three encoding values: 00000 (single breakpoint), 10000 (single breakpoint followed by trace mode), and 10111 (simple trace mode).

9.3.1.1.4.8 $\overline{\mathrm{DE}}$ Pin Output Enable (DE)—Bit 14

The $\overline{\mathrm{DE}}$ pin enable (DE) bit (OCR bit 14) is used to setup the $\overline{\mathrm{TRST}}$ / $\overline{\mathrm{DE}}$ pin as an output. When DE is set to 0, the pin is setup as the $\overline{\mathrm{TRST}}$ input. When DE is set to 1, the $\overline{\mathrm{DE}}$ output is enabled, and the pin no longer provides the $\overline{\mathrm{TRST}}$ capability.

9.3.1.1.4.9 COP Timer Disable (COPDIS)—Bit 15

The COP timer disable (COPDIS) bit (OCR bit 15) is used to prevent the COP timer from resetting the DSP chip when it times out. When COPDIS is set to 0, the COP timer is enabled. Setting the COPDIS bit to 1 disables the COP timer.

9.3.1.1.5 OnCE Status Register (OSR)

The OnCE status register (OSR) is shown in **Figure 9-9**. The OSR contains SBO, HBO, and TO, along with OS1 and OS0. The 8-bit OnCE status register (OSR), which is readable in the STATCOM state of the OnCE state machine (see **OnCE State Machine and Control Block** on page 9-23). By observing the values of all 5 status bits, the user can determine if the core has halted and what caused it to halt, or if and why the core has not halted in response to a debug request. The OSR is automatically read while the OnCE module is in the STATCOM state. The same serial clocks used to shift in a new command also shift the status information out of the OSR, allowing efficient status polling. New status information is captured in the OSR only on transitions to the STATCOM state, except for transition 3 which occurs during the STATCOM state (see **Figure 9-10** on page 9-24 and **Table 9-12** on page 9-25).

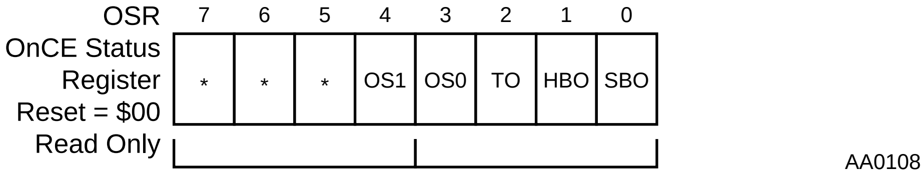


Figure 9-9 OnCE Status Register (OSR)

9.3.1.1.5.1 Software Breakpoint Occurrence (SBO)—Bit 0

The read-only software breakpoint occurrence (SBO) status bit (OSR bit 0) is set when the debug mode is entered using a DEBUG instruction. It is used by the external command controller to determine how the debug mode was entered. This bit is cleared when leaving the debug mode and is also cleared on hardware reset.

9.3.1.1.5.2 Hardware Breakpoint Occurrence (HBO)—Bit 1

The read-only hardware breakpoint occurrence (HBO) status bit (OSR bit 1) is set when a OnCE hardware breakpoint occurs. It is used by the external command controller to determine how the debug mode was entered. This bit is cleared when leaving the debug mode and it is also cleared on hardware reset.

9.3.1.1.5.3 Trace Occurrence (TO)—Bit 2

The read-only trace occurrence (TO) status bit (OSR bit 2) 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 reset and when leaving the debug mode.

9.3.1.1.5.4 OnCE Status (OS1 and OS0)—Bits 4-3

The OnCE status (OS1 and OS0) bits (OSR bits 4-3) hold the same status information in the same encoding as the existing DSI/OS0 and DSCK/OS1 pins with one difference: the existing, reserved encoding 11 would instead indicate the DSP core has been halted. This additional encoding will provide complete acknowledge information to the user when OSR is polled. **Table 9-11** summarizes the core status bit description.

Table 9-11 DSP Core Status Bit Description

OS1-OS0	Instruction	Description
00	Normal	DSP core executing instructions
01	STOP/WAIT	DSP core in stop or wait mode
10	Busy	DSP doing external or peripheral access
11	Debug	DSP core halted and in debug mode

Note: The OS bits are also captured by the JTAG instruction register.

9.3.1.1.5.5 Reserved OSR Register Bits—Bits 7-5

The reserved OSR register bits (OSR bits 7-5) are reserved for future expansion and are read as zero during DSP read operations. The user should never write a 1 to these bits.

9.3.1.1.6 OnCE State Machine and Control Block

The OnCE state machine has the following states:

- IDLE—Do nothing
- STATCOM—Poll status and get command
- DEC1—Decode 1
- DEC2—Decode 2
- RWREG—Read / write a OnCE register
- WPDBR—Write OPDBR

Figure 9-10 shows the possible paths through the OnCE state machine. **Table 9-12** gives a description of each state transition (as shown in **Figure 9-10**).

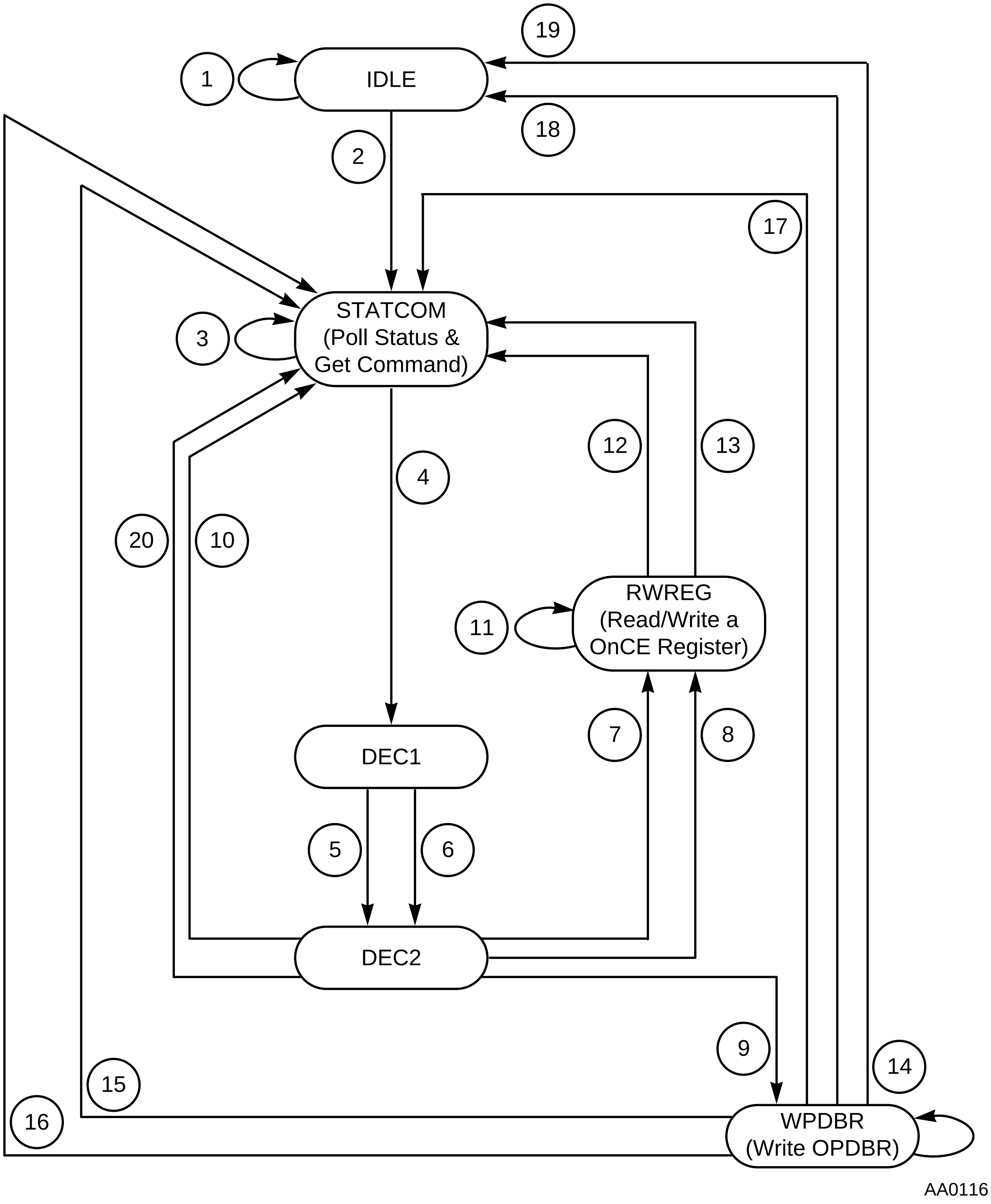


Figure 9-10 OnCE State Machine

Table 9-12 OnCE State Machine Transitions

Transition	Description
1	Chip is in normal mode.
2	Chip enters debug mode. Go to STATCOM state.
3	Get command. Wait for external controller to finish sending an 8-bit command.
4	External controller has finished sending the command. Start decoding OnCE command.
5	OnCE will not access any registers. The core is to repeat executing the previous instruction.
6	OnCE is to access a dedicated register. This is the default state when the OnCE command selects a reserved option.
7	Read or write any OnCE dedicated register besides the PDB register, OBDBR. The core will not be asked to execute any instruction.
8	Read OPDBR. The core will not be asked to execute any instruction.
9	Write to the OPDBR. The core will eventually be asked to execute an instruction.
10	OnCE is to clear either the OnCE breakpoint counter (OMBC) or the OnCE trace counter (OTC).
11	Read or write the OnCE dedicated register. Wait for 16 input or output bits to be shifted.
12	Finished writing a OnCE register. Send an acknowledge pulse.
13	Finished writing a OnCE register. Do not send an acknowledge pulse.
14	Write to the OPDBR. Wait for the 16-bit core command or operand to be shifted.
15	Finished writing to the OPDBR. Execute a one-word core instruction, but do not exit from debug mode.
16	Finished writing to the OPDBR. Execute a two-word core instruction, but do not exit from debug mode.
17	Finshed writing to the OPDBR. Transfer its contents to the PDB. This is the first word of a two-word instruction. Get the second word.
18	Finished writing to the OPDBR. Execute a one-word core instruction while exiting the OnCE debug mode.
19	Finished writing to the OPDBR. Execute a two-word core instruction while exiting the OnCE debug mode.
20	The core has finished executing the current instruction. Get the next OnCE command.

9.3.1.2 OnCE Breakpoint and Trace Section

Two capabilities useful for real-time debugging of embedded control applications are address breakpoints and full-speed instruction stepping. Traditionally, processors had set a breakpoint in program memory by replacing the instruction at the breakpoint address with an illegal instruction that causes a breakpoint exception. This technique is limiting in that breakpoints can only be set in RAM at the beginning of an opcode and not on an operand. In addition, breakpoints can never be set on data memory locations. The DSP56800 core provides on-chip address comparison hardware for setting breakpoints on program or data memory accesses. This allows breakpoints to be set on program ROM as well as program RAM locations. Breakpoints can be programmed for reads, writes, program fetches, or memory accesses using the OCR's BS and BE bits (**Breakpoint Selection (BS1 and BS0)—Bits 3-2** on page 9-15).

The breakpoint logic can be enabled for the following:

- Program instruction fetches
- Program memory accesses via the MOVE(M) instruction (read, write, or access)
- X data memory accesses (read, write, or access)
- On-chip peripheral register accesses (read, write, or access)

Breakpoints are possible during on-chip peripheral register accesses because these are implemented as memory-mapped registers in the X data space. The breakpoint unit contains an address latch, a register that stores the breakpoint address, an address comparator and a counter. **Figure 9-11** illustrates a block diagram of the OnCE breakpoint unit.

In addition, the DSP56800 OnCE port provides a full-speed stepping capability, the capability to execute up to 256 instructions at full speed and then re-enter the debug processing state. This allows the user to single step through a program in the simplest case, then the counter is set for one occurrence or to execute many instructions at full speed before returning to the debug processing state.

The breakpoint and trace logic has been designed to work together so that it is possible to set up more sophisticated trigger conditions that combine up to two breakpoints and trace logic. The specific combinations are configured through the breakpoint configuration (BK4-BK0) bits in the OCR. See **Breakpoint Configuration (BK4-BK0)—Bits 13-9** on page 9-19 for more information about the available trigger combinations. The individual events and conditions that lead to triggering can also be modified by event modifier (EM1-EM0) bits in the OCR. See **Event Modifier (EM1 and EM0)—Bits 6-5** on page 9-17 for additional information about using these bits.

While debugging, it is sometimes the case that there are not enough breakpoints available. In that case, the core-based DEBUG instruction can be useful in helping to catch difficult bugs. The DEBUG instruction is a one word instruction that places the DSP core in the debug processing state when it is executed. The JTAG instruction DEBUG_REQUEST (0111) can also be used to force the debug mode. (See **Table 9-14 JTAG Instruction Register Encodings** on page 9-56 for a list of JTAG instructions.)

9.3.1.2.1 OnCE Breakpoint Logic Operation

The address comparator register is useful in halting a program at a specific point to examine or change registers or memory. Using the address comparator to set breakpoints enables the user to set breakpoints in RAM or ROM while in any operating mode.

The address comparator will cause a logic true signal when the comparison of its value is equal to the address on the bus. The breakpoint counter is then decremented if greater than zero. If the breakpoint counter is equal to zero, it is not decremented, and a breakpoint occurs.

Conditional jump addresses produced by the instruction pipeline that are equal to the program address being monitored are only valid if the conditional jump instruction occurs; otherwise the conditional jump address 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 that formed the data memory address, and the breakpoint counter has decremented to zero. The breakpoint registers are controlled by the OnCE control register (OCR).

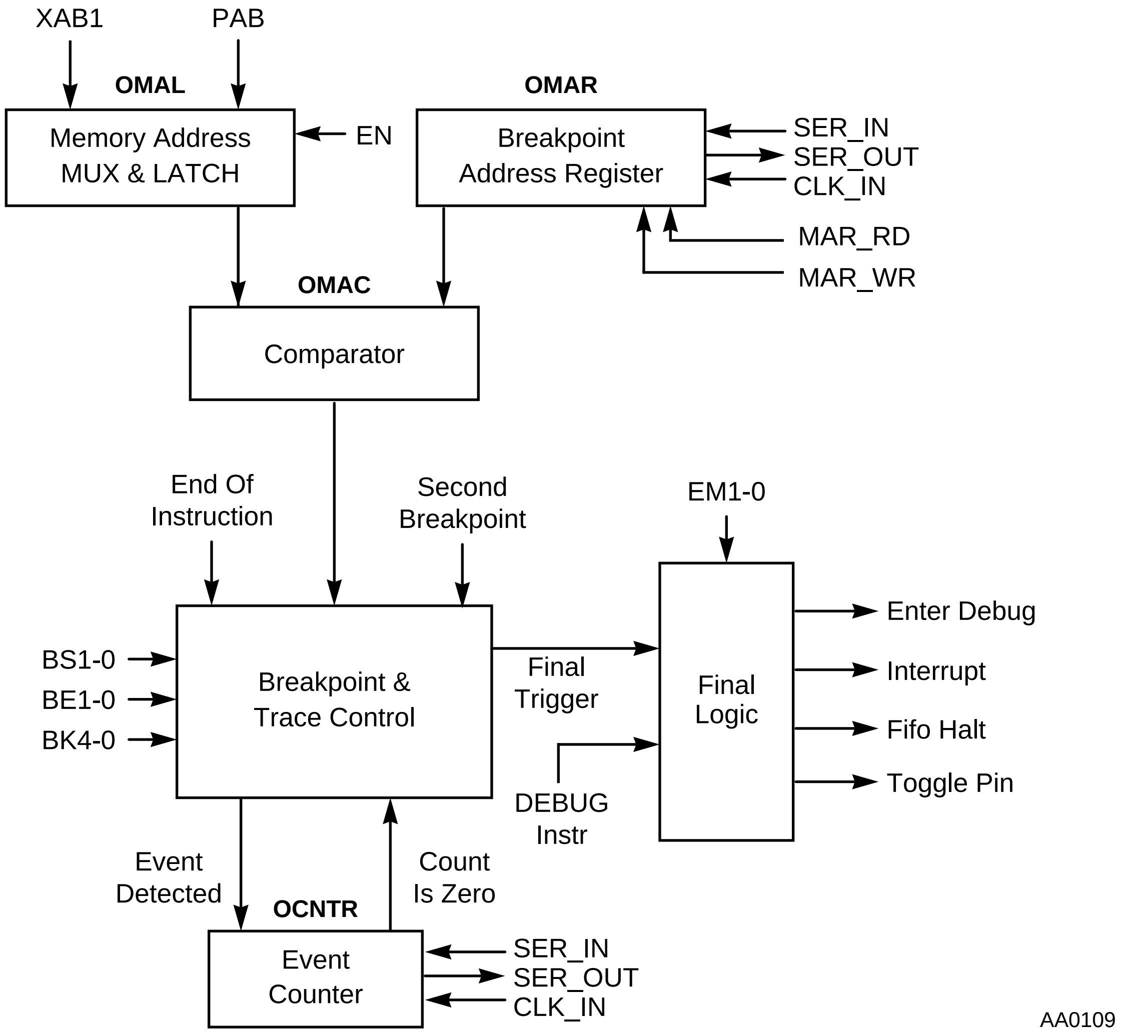


Figure 9-11 Breakpoint and Trace Counter Unit

When in the special trace mode, the DSP will not cause an interrupt exception but instead will enter the debug operation mode and wait for further instructions from the debug serial port. Single or multiple instructions can be traced.

9.3.1.2.2 OnCE Breakpoint Counter (OCNTR)

The OnCE breakpoint counter (OCNTR) is an 8-bit counter that 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 debuging and provides a means of checking hot spots in program segments as well as peripheral or data memory accesses.

When used as a breakpoint counter, the ONCTR becomes a powerful tool to debug real-time interrupt sequences such as servicing an A/D or D/A convertor or stopping after a specific number of transfers from a peripheral have occurred. The ONCTR is cleared by hardware reset.

When used as a trace mode counter, the ONCTR allows more than one instruction to 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 in real-time with no interference from the debug mode. This feature helps the software developer debug sections of code that do not have a normal flow or are getting hung up in infinite loops. Using the trace counter also enables the user to debug areas of code that are time critical.

In either case, 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, and the breakpoint mode (either breakpoint, trace, or a combination of breakpoint and trace events) is selected in the debug control register (OCR) BK4 to BK0 bits. The DSP 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 or breakpoint encountered (depending on whether it is configured as a breakpoint counter or instruction counter). Interrupts are serviceable. If selected as an instruction counter, all instructions executed, including fast interrupt services, decrement the trace counter. Upon decrementing to zero the DSP will re-enter the debug mode (or proceed to the next step in its configuration until it re-enters debug mode), the trace occurrence bit or breakpoint occurrence bit in the debug status register (OSR) are set, and the debug event output will be toggled to indicate that the DSP OnCE port is requesting service.

Note: The trace count should be loaded with one less than (i.e., N-1) the number of instructions that the user wants to execute (e.g., to single step one instruction, the trace counter is loaded with a zero).

9.3.1.2.3 OnCE Memory Address Latch (OMAL)

The OnCE memory address latch (OMAL) is a 16-bit register that latches the PAB or XAB1 on every cycle. This latching is disabled if the OnCE port is powered down with the OCR’s PWD bit.

Note: The OMAL register does not latch the XAB2 bus. As a result, it is not possible to set an address breakpoint on any access done on the XAB2/XDB2 bus pair used for the second read in any dual-read instruction.

9.3.1.2.4 OnCE Breakpoint Address Register (OBAR)

The OnCE breakpoint address register (OBAR) is a 16-bit OnCE register that stores the memory breakpoint address. OBAR is available for read / write operations only through the JTAG / OnCE serial interface. Before enabling breakpoints, OMAR must be loaded by the command controller.

9.3.1.2.5 OnCE Memory Address Comparator (OMAC)

The OnCE memory address comparator (OMAC) is a 16-bit comparator that compares the current memory address (stored by OMAL) with memory address register (OBAR). If OMAC is equal to OMAL then the comparator delivers a signal indicating that the breakpoint address has been reached.

9.3.1.3 OnCE Pipeline

The previous chip pipeline state must be reconstructed to resume normal chip activity when returning from the debug mode. **Figure 9-12** illustrates a block diagram of pipeline information registers. Only the OPDBR and the OPGDBR are used to reconstruct the pipeline as it was before debug. The FIFO history buffer, OPABFR, and OPABDR only provide status information. When loading a one-word instruction into the OPDBR and issuing a GO command, the hardware internally transfers the OPDBR to the OPGDBR and then executes the instruction. When loading a two-word instruction, the first word is loaded into the OPDBR. As the second word is loaded to the OPDBR, the first word is automatically transferred to the OPGDBR and then execution takes place.

9.3.1.3.1 OnCE PDB Register (OPDBR)

The OnCE PDB register (OPDBR) is a read / write, 16-bit latch that stores the value of the program data bus generated by the last program memory access of the DSP before the debug mode is entered. OPDBR is available for read / write operations only through the JTAG / OnCE 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 write access to the OPGDBR, this task is accomplished by writing the OPDBR first and then the data from OPDBR is latched in OPGDBR.

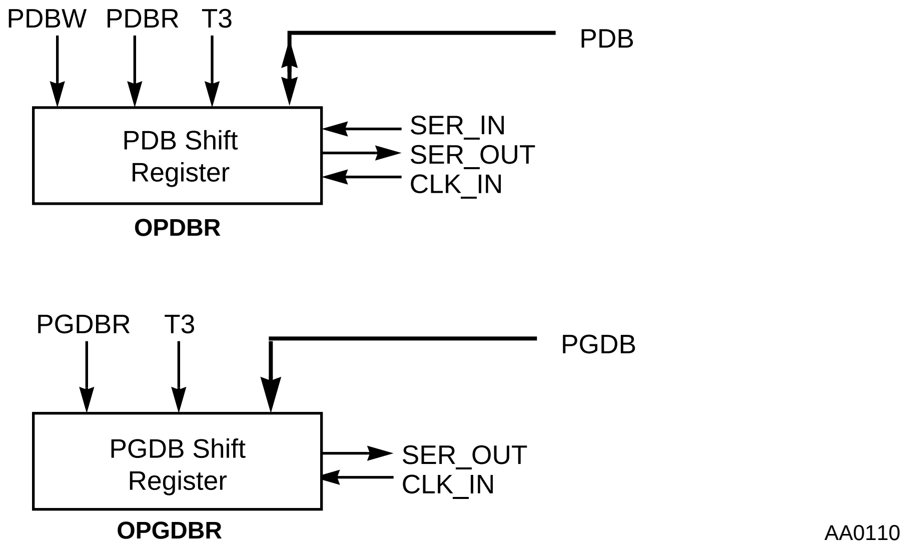


Figure 9-12 Pipeline Information Registers

9.3.1.3.2 OnCE PGDB Register (OPGDBR)

The OnCE PGDB register (OGDBR) is a read only 16-bit latch that stores the value of the global data bus. OGDBR is available for read operations only through the serial interface. OGDBR is required as a means of passing information between the chip and the command controller. OGDBR will be mapped on the X internal IO space at address $FFFF. Whenever the command controller needs information such as a register or memory value, it will force the chip to execute an instruction that brings that information to the OGDBR. Then, the contents of the OGDBR will be delivered serially to the command controller by the command READ GDB REGISTER.

9.3.1.4 OnCE FIFO History Buffer

To ease the debugging activity and keep track of the program flow, a FIFO, read-only buffer is provided. It 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 9-13** illustrates a block diagram of the program address bus FIFO.

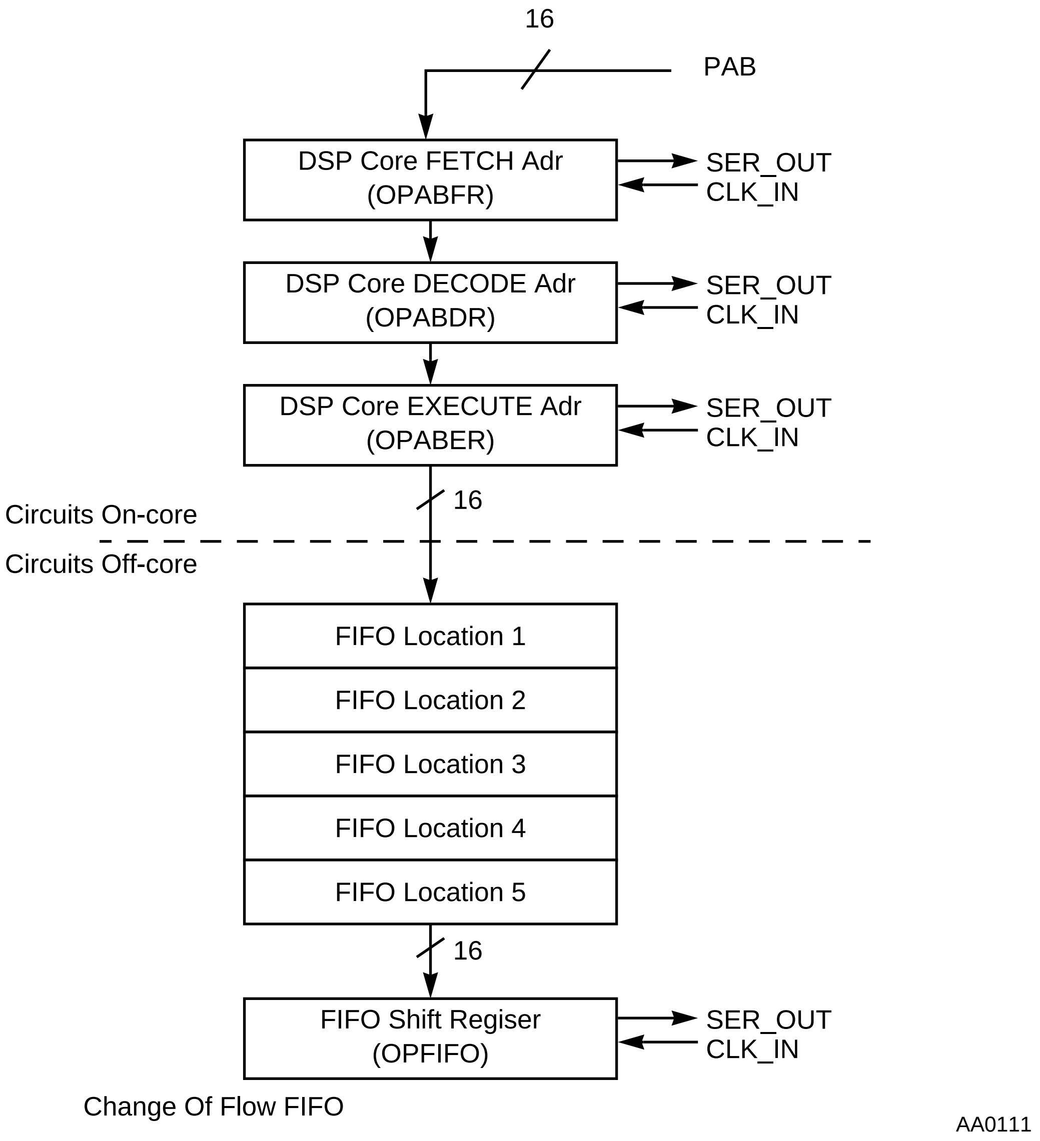


Figure 9-13 Program Address Bus FIFO

9.3.1.4.1 OnCE PAB Fetch Register (OPABFR)

The OnCE PAB register for fetching (OPABFR) is a read-only 16-bit latch that stores the address of the last instruction that was fetched before the debug mode was entered. OPABFR is available for read operations only through the serial interface. This register is not affected by the operations performed during the debug mode.

9.3.1.4.2 OnCE PAB Decode Register (OPABDR)

The 16-bit OnCE PAB register for decoding (OPABDR) 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 is available for read operations only through the serial interface. This register is not affected by the operations performed during the debug mode.

9.3.1.4.3 OnCE PAB Execute Register (OPABER)

The 16-bit OnCE PAB register for executing (OPABER) stores the address of the instruction currently executing. OPABER is available for read operations only through the serial interface. This register is not affected by the operations performed during the debug mode.

9.3.1.4.4 OnCE PAB FIFO

The FIFO is implemented as a true FIFO buffer with five 16-bit locations. All locations are accessed through the same address, and each read access to the FIFO is followed by a shift of the values in the FIFO. The registers are serially available for read to the command controller through their common FIFO address. The FIFO is not affected by the operations performed during the debug mode except for the shifting performed after reading a FIFO value.

Note: Not all implementations use the external five-level FIFO locations. Refer to the appropriate user’s manual for chip-specific information.

The OnCE FIFO will no longer capture the address of every executed instruction. Instead, only the addresses of the following instructions will be placed into the FIFO:

- BRA
- JMP
- JSR
- Bcc (with conditional true)
- Jcc (with condition true)

The first location of the FIFO will however always hold the address of the last executed instruction, regardless of whether it caused change of program flow or not. This change gives the user much more information on program flow than the current implementation, since sequential flow may be assumed between instructions that change flow.

Reads of the FIFO register will read the oldest value in the FIFO, and then shift all values one location towards the shift register. Each read destroys one location in the FIFO after it is read, so that when the entire FIFO has been read, the contents are no longer available within the FIFO.

The change of flow nature of the FIFO begins *after* the OPABER (i.e., changes of flow only affect the contents of the FIFO), and not the fetch, decode, or execute registers (OPABFR, OPABDR, or OPABER).

9.3.2 The Debug Processing State

Introduction on page 7-3 describes six processing states for the DSP chip:

- Reset
- Normal
- Exception
- Wait
- Stop
- Debug

The sixth processing state debug supports the on-chip emulation features of the chip. In this mode, the DSP core is halted and is set to accept OnCE commands through the JTAG port. Once the OnCE port is set up correctly, the DSP leaves the debug state and returns control to the user program. The DSP reenters the debug processing state when the previously-set trigger condition occurs.

Capabilities available in the debug state include the following:

- Read and write the OnCE registers
- Read the instruction FIFO
- Reset the OnCE event counter
- Execute a single DSP instruction and return to this state
- Execute a single DSP instruction and exit this state

9.3.2.1 OnCE Normal and Debug Modes

The OnCE module has two operational modes: normal and debug. The OnCE module is in the normal mode except when the DSP enters the debug processing state. The OnCE is in the debug mode whenever the DSP enters the debug processing state. The major difference between the two states is register access. The following OnCE module registers can be accessed in normal or debug mode:

- OCR

- OSR
- OISR
- OCNTR
- OBAR

The following OnCE module registers can only be accessed when the module is in debug mode:

- OPGDBR
- OPDBR
- OPABFR
- OPABDR
- OPABER
- OPFIFO

9.3.2.2 Entering the Debug Mode

There are seven ways to enter the debug mode. They are:

1. External request during hardware reset (with external debug request pin)
2. External request during normal activity (with external debug request pin)
3. External request during stop (with external debug request pin)
4. External request during wait (with external debug request pin)
5. Software request during normal activity
6. Trigger events (breakpoint/trace modes)
7. JTAG DEBUG_REQUEST (0111) instruction execution

Each is explained below.

Note: The presence of an external debug request pin is chip-specific. Refer to the appropriate user's manual to determine if this feature is included for a specific chip implementation. If present, this pin, when asserted, causes the DSP 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 JTAG/OnCE serial input line.

The status bits provide information about the chip status when the debug processing state cannot be entered in response to an external request to enter the debug processing state.

Table 9-13 shows the status of the chip as a function of the two output pins OS0:OS1.

Table 9-13 Function of OS1:OS0

OS1	OS0	Status
0	0	Normal state
0	1	Stop or wait mode
1	0	DSP busy state (external accesses with wait state)
1	1	Debug mode

9.3.2.2.1 External Request During Hardware Reset

Asserting the debug request pin 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 debug request line.

Note: In this case the chip does not perform any fetch or memory access before entering the debug mode.

9.3.2.2.2 External Request During Normal Activity

Asserting the debug request line during the normal chip activity causes the chip to finish execution of the current instruction and then enter the debug mode. After receiving the acknowledge the command controller must deassert the debug request line.

Note: In this case the chip completes execution of the current instruction and stops after the newly-fetched instruction enters the instruction latch. This process is the same for any newly-fetched instruction, including instructions fetched during interrupt processing or instructions that will be killed by the interrupt processing.

9.3.2.2.3 External Request During Stop

Asserting the debug request line 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. The chip will wake up from the stop state normally (finish executing the STOP instruction) and halt after the next instruction enters the instruction latch. After receiving the acknowledge, the command controller must deassert the debug request line.

Note: In this case the chip completes the execution of the STOP instruction and halts after the next instruction enters the instruction latch.

9.3.2.2.4 External Request During Wait

Asserting the debug request line when the chip is in the wait state (i.e., has executed a WAIT instruction) causes the chip to exit wait state and enter the debug mode. The chip will wake up from the wait state normally (finish executing the WAIT instruction) and halt after the next instruction enters the instruction latch. After receiving the acknowledge, the command controller must deassert the debug request input.

Note: In this case the chip completes execution of the WAIT instruction and halts after the next instruction enters the instruction latch.

9.3.2.2.5 Software Request During Normal Activity

Upon executing the DEBUG instruction, the chip will enter debug processing state after the instruction following the DEBUG instruction has entered the instruction latch.

9.3.2.2.6 Trigger Events (Breakpoint/Trace Modes)

The DSP56800 Family allows the user to configure specific trigger events. These events can include breakpoints, trace modes, or combinations of breakpoints and trace mode operations. The chip enters the debug mode *after* completing execution of the instruction that caused the OCNTR to decrement when either of the following occurs:

- Operating in the trace mode when the OCNTR has reached zero
- When operating in normal mode with breakpoint enabled, the breakpoint value is detected, and, if multiple breakpoint iterations are specified, the OCNTR has reached zero.

In the case of breakpointing on program memory addresses, the breakpoint will be acknowledged immediately after the execution of the instruction accessed at the specified address. However, when breakpointing on data memory addresses the breakpoint will be acknowledged after the completion of the instruction following the instruction that caused the access at the specified address.

Note: In trace mode, only those instructions that are actually executed may cause the OCNTR to decrement (i.e., a killed instruction—instruction discarded during the interrupt process—will not decrement the OCNTR and will not cause the chip to enter the debug mode).

9.3.2.2.7 JTAG DEBUG_REQUEST Instruction

The user can also invoke the debug mode through a special JTAG instruction: DEBUG_REQUEST (0111). To enter the debug mode from the JTAG port, you must issue the JTAG DEBUG_REQUEST instruction, then enter the ENABLE_ONCE instruction to begin programming the OnCE module. See **Serial Protocol Description** on page 9-38 for more information about using the serial interface.

Note: See **JTAG Port Architecture** on page 9-51 for general information about using the JTAG TAP controller and its instructions.

9.3.2.3 Exiting Debug Mode

To exit debug mode, the EX bit (bit 5) in the OnCE command register (OCMDR) is used . The TAP controller is put in the Shift-DR state and shifted in an 8-bit binary value with bit 5 set to one. This can be done as part of the last command entered in a set of debugging instructions, or as a separate command with all other bits set to zero. See **Returning from Debug Mode to Normal Mode** on page 9-49 for additional information.

9.3.3 Serial Protocol Description

In order to permit an efficient means of communication between the command controller and the DSP chip, the following protocol has been adopted. Before starting any debugging activity, the command controller has to wait for an acknowledge from the chip that informs the command controller that it has entered the debug mode.

Note: In case of a breakpoint, trace, or software DEBUG / DEBUGcc instruction, the acknowledge itself initiates the debug session. The command controller communicates with the chip by sending 8-bit commands that may be accompanied by 16-bit data. After sending a command, the command processor starts waiting 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.

9.3.3.1 Entering Debug Mode from User Mode

When shifting in the ENABLE_ONCE command, the OS bits will be shifted out so the user can determine if the core has acknowledged the request and halted. To enter the debug mode from the JTAG port, the user must issue the JTAG DEBUG_REQUEST instruction and then enter the ENABLE_ONCE instruction to begin programming the OnCE module.

9.3.3.2 Entering Debug Mode from DSP Reset

The JTAG state machine will be reset via a POR signal. JTAG will therefore be accessible soon after power-up. The OnCE state machine will be placed at the IDLE state at reset assertion by a pulse. The pulse would be valid for only a few cycles just after reset assertion, allowing for access to the OnCE state machine when the DSP is still in reset. The user can then execute the JTAG instruction DEBUG_REQUEST followed by ENABLE_ONCE. After de-asserting DSP reset, the chip will be in debug mode.

Note: Providing OnCE access in DSP reset allows the user to set breakpoints while in reset. This aids in debugging when the DSP has problems leaving reset.

9.3.3.3 Polling for OnCE Status

It is necessary to poll for OnCE status in the following three situations:

- After loading the JTAG instruction DEBUG_REQUEST
- After issuing a DSP instruction while in debug mode
- When breakpoints are enabled in user mode.

In these situations the user must have some way of knowing if the core has halted. Since the TDO pin (which replaces the existing DSO pin) can no longer provide the acknowledge pulse, it has been proposed that the new OSR contain the OS[1:0] bits (as well as the JTAG-IR on Capture-IR) that provide this sort of status information.

Note: If the core is executing a STOP / WAIT instruction, the OSR will not be readable (loss of internal clocks) and status must be read via the JTAG IR. Alternately, the user can use the DE output to indicate that a debug event has occurred and that the OnCE module needs servicing.

The user can poll the OSR in debug or user mode provided that the OnCE state machine is in the STATCOM state and the JTAG state machine is in Shift-DR. The OSR will capture new status only when traversing to the STATCOM state from a state other than STATCOM itself (see **OnCE State Machine and Control Block** on page 9-23).

9.3.3.4 Setting Breakpoints in User Mode

Setting breakpoints in user mode is done the same way as in debug mode, except that it is recommended that the user disable breakpoints before writing to the OMAC, OBAR, or the BK4-BK0 bits in OCR. This avoids generating any indeterminate results that could arise if a breakpoint is occurring while the above registers are being modified.

9.3.3.5 Reading Pipeline Information in User Mode

The user can access pipeline information while the program is still executing. One way of doing this is for the user to follow the steps below:

1. Make sure breakpoints and trace are disabled.
2. Write 01 to the EM bits in OCR.
3. Set the OCNTR to 1.
4. Set BK4-BK0 in the OCR to 10111 to enable trace mode.
5. Poll the OSR until TO equals 1.
6. Read the FIFO.

9.3.3.6 Displaying a Specified Register

The multiple DSP configuration shown in **Figure 9-14** will be used for **Example 9-1**. The user must assume before the procedure begins that all three DSPs are running and that their JTAG ports are executing the BYPASS instruction. The user wishes to display the value of a register ("reg") of DSP#2

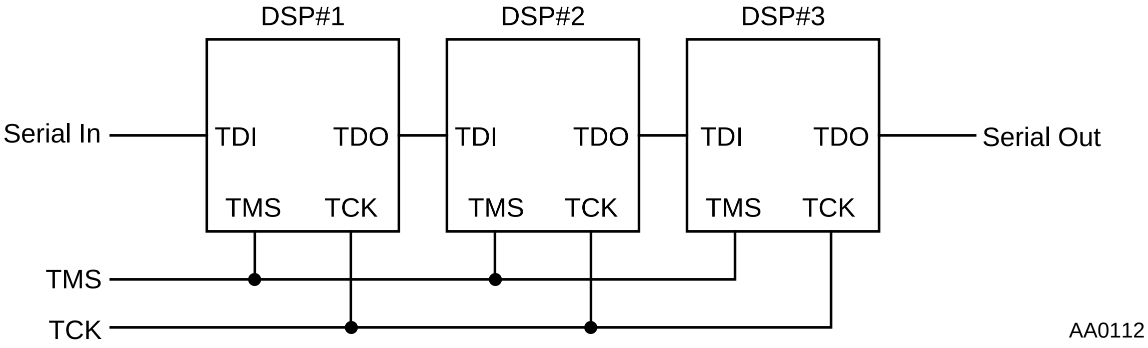


Figure 9-14 Multiple DSP Configuration for Example Procedures

Note: "Shift-IR," "Shift-DR," "Update-IR," and "Update-DR" in the example below refer to operating states in the JTAG module. The TAP controller in the JTAG module uses the current operating state of the TAP controller combined with the level of the TMS input to transition between the operating states. A detailed description of the TAP controller state machine is presented in **TAP Controller** on page 9-59.

Note: BYPASS, DEBUG_REQUEST, and ENABLE_ONCE are JTAG instructions. (See **Table 9-14 JTAG Instruction Register Encodings** on page 9-56 for a list of JTAG instructions.)

Example 9-1 Display a Specified Register—Serial

1. Enter Shift-IR, shift in BYPASS to DSP#1 and DSP#3, and DEBUG_REQUEST to DSP#2 (12 clocks in Shift-IR). Pass through Update-IR. OnCE state machine of DSP#2 is in STATCOM state.
2. Enter Shift-IR, ENABLE_ONCE into DSP#2 and BYPASS into others. Pass through Update-IR. OnCE state machine is in STATCOM state.
3. Enter Shift-DR. Begin polling to see if DSP#2 has entered debug mode (or poll via JTAG IR if STOP / WAIT may be executing). When status information indicates the core has halted and is in debug mode, enter Shift-DR, shift in “WRITE OPDBR with GO and no EX” (OnCE command = 01001001). Pass through Update-DR. OnCE state machine is in WPDBR state.
4. Enter Shift-DR. Send the 16-bit opcode: “MOVE reg, x:OGDB” (17 clocks in Shift-DR). Pass through Update-DR to actually write the OPDBR and begin execution of the MOVE instruction. OnCE state machine is in STATCOM state to allow for polling.
5. Enter Shift-DR and begin polling for OS[1:0] = 11 again. While polling, shift in “READ OPGDBR” (OnCE command = 11001000). When OS[1:0] = 11, pass through Update-DR. In this case, polling really would not be necessary unless internal wait states are inserted. And if they are, user should have reduce JTAG clock speed enough such that polling still is not necessary. OnCE state machine is in RWREG state in preparation for reading of OPGDBR.
6. Enter Shift-DR. Clock 17 times to display the value in the OGDBR (i.e., register contents). Pass through Update-DR. OnCE state machine is in STATCOM state in preparation for further OnCE commands to be shifted in.

9.3.3.7 Displaying X Memory Area Starting at Address xxxx

For **Example 9-2**, the user should assume the multiple DSP configuration shown in **Figure 9-14 Multiple DSP Configuration for Example Procedures** on page 9-40. Before the procedure begins, all three DSPs are in user mode and their JTAG ports are executing the BYPASS instruction. The user wishes to display memory contents in DSP#2.

Note: “Shift-IR,” “Shift-DR,” “Update-IR,” and “Update-DR” in the example below refer to operating states in the JTAG module. The TAP controller in the JTAG module uses the current operating state of the TAP controller combined with the level of the TMS input to transition between the operating states. A detailed description of the TAP controller state machine is presented in **TAP Controller** on page 9-59.

Note: BYPASS, DEBUG_REQUEST, and ENABLE_ONCE are JTAG instructions. (See **Table 9-14 JTAG Instruction Register Encodings** on page 9-56 for a list of JTAG instructions.)

Example 9-2 Display X Memory Area from Address xxxx—Serial

1. Perform steps 1-6 of the “Displaying a Specified Register” procedure where the register in this case is R0. R0 will be used in displaying the memory area, so its original value should be saved and later restored just before leaving the debug mode. OnCE state machine is in STATCOM state.
2. Enter Shift-DR, shift “WRITE OPDBR with no GO and no EX” (OnCE command = 00001001) into DSP#2 (9 clocks in Shift-DR). Pass through Update-DR. OnCE state machine is in WPDBR state.
3. Enter Shift-DR. Send the 16-bit opcode of the two-word DSP instruction: “MOVE #$xxxx, R0” (17 clocks in Shift-DR). Pass through Update-DR to actually write the OPDBR. OnCE state machine is in STATCOM state.
4. Enter Shift-DR, shift “WRITE OPDBR with GO and no EX” (OnCE command = 01001001) into DSP#2 (9 clocks in Shift-DR). Pass through Update-DR. OnCE state machine is in WPDBR state.
5. Enter Shift-DR. Send the 16-bit data of the two-word DSP instruction: “MOVE #$xxxx, R0” (the xxxx field). Pass through Update-DR to actually write the OPDBR and begin execution of the MOVE instruction. OnCE state machine is in STATCOM state to allow for polling.
6. Enter Shift-DR and begin polling for OS[1:0] = 11. While polling, shift in “WRITE OPDBR with no GO and no EX” (OnCE command = 00001001). When OS[1:0] = 11, indicating that the MOVE instruction has completed and the core has halted once again, R0 has been loaded with the base address of the memory area to be displayed. Pass through Update-DR. OnCE state machine is in WPDBR state.

Example 9-2 Display X Memory Area from Address xxxx—Serial (Continued)

7. Enter Shift-DR. Send the 16-bit opcode: “MOVE X:(R0)+, x:OGDB” (17 clocks in Shift-DR). Pass through Update-DR to actually write the OPDBR and begin execution of the MOVE instruction. OnCE state machine is in STATCOM state to allow for polling.
8. Enter Shift-DR and begin polling for OS[1:0] = 11. While polling, shift in “READ OPDBR” (OnCE command = 11001001). When OS[1:0] = 11, the MOVE instruction has completed, the core has halted once again, the contents of the first memory location have been placed in OGDBR and R0 has been incremented such that it points to the next memory location. Pass through Update-DR. OnCE state machine is in RWREG state.
9. Enter Shift-DR. Clock 17 times to display the value in the OGDBR (i.e. the contents of memory location xxxx). Pass through Update-DR. OnCE state machine is in STATCOM state.
10. Enter Shift-DR. Shift in “NO SELECTION with GO and no EX” (OnCE command = 11000000). Pass through Update-DR to execute “MOVE X:(R0)+, x:OGDB” again. Go to step 8 to continue reading memory contents. When finished, restore original value of R0.

9.3.3.8 Returning from Debug Mode to Normal Mode

For **Example 9-3**, the user should assume the multiple DSP configuration shown in **Figure 9-14 Multiple DSP Configuration for Example Procedures** on page 9-40. The user wishes to bring DSP#2 back into user mode from debug mode. DSP#1 and DSP#3 are in user mode and their JTAG ports are executing the BYPASS instruction.

Note: “Shift-IR,” “Shift-DR,” “Update-IR,” and “Update-DR” in the example below refer to operating states in the JTAG module. The TAP controller in the JTAG module uses the current operating state of the TAP controller combined with the level of the TMS input to transition between the operating states. A detailed description of the TAP controller state machine is presented in **9.4.1.4 TAP Controller** on page 9-59.

Note: BYPASS, DEBUG_REQUEST, and ENABLE_ONCE are JTAG instructions. (See **Table 9-14 JTAG Instruction Register Encodings** on page 9-56 for a list of JTAG instructions.)

Example 9-3 Return from Debug Mode to Normal Mode—Serial

1. Enter Shift-DR, shift “WRITE OPDBR with no GO and no EX” (OnCE command = 00001001) into DSP#2 (9 clocks in Shift-DR). Pass through Update-DR. PAB is driven with value in OPABDR which is the value latched from the PAB just before entering debug mode. OnCE state machine is in WPDBR state.
2. Enter Shift-DR. Send the saved contents of the OPDBR (17 clocks in Shift-DR). Pass through Update-DR to actually load the OPDBR. OPDBR will drive the PDB and the instruction latch loads the value on the PDB. OnCE state machine is in the STATCOM state.
3. Enter Shift-DR, shift “WRITE OPDBR with GO and EX” (OnCE command = 01101001) into DSP#2 (9 clocks in Shift-DR). Pass through Update-DR. OnCE state machine is in WPDBR state.
4. Enter Shift-DR. Send the saved contents of the OPDBR (17 clocks in Shift-DR). Pass through Update-DR to actually load the OPDBR. The OnCE state machine goes into the IDLE state then back to STATCOM as long as ENABLE_ONCE is executing in JTAG machine. DSP#2’s core pipeline is restored to its state prior to debug, and the DSP is back in user mode.

9.3.3.9 Recovering from STOP/WAIT Execution

If a STOP or WAIT instruction is executed while the user is accessing OnCE in user mode problems will occur since little or no internal clocks are running anymore. This possibility should be avoided; the user can recognize the occurrence by capturing the OS bits in the JTAG IR in Capture-IR. The user can then choose to send a DEBUG_REQUEST to bring the core out of STOP / WAIT.

9.3.4 Using The OnCE

The following notations are used:

- ACK—Wait for acknowledge on $\overline{\text{DS}}$ line
- CLK—Issue 16 clocks to read out data from selected register

Commands require eight clocks.

9.3.4.1 Begin Debug Activity

Debug activity begins on an instruction boundary after the debug request pin is asserted, a DEBUGcc opcode is executed, a trigger event occurs, or a JTAG

DEBUG_REQUEST is executed. If the instruction executing when the debug request pin is asserted is a REP instruction or the instruction following a REP instruction, then the debug activity begins after the instruction following the REP instruction finishes its repetitions. The first ACK indicates that the OnCE controller is ready to receive commands and data. Most of the debug activities will have the beginning specified in **Example 9-4**.

Example 9-4 Begin Debug Activity

ACK

1. Save pipeline information:
    1. Send command “Read OPDBR” (OnCE command = 11001001)
    2. ACK
    3. CLK
    4. Send command “READ OPGDR” (OnCE command = 11001000)
    5. ACK
    6. CLK
2. Read PAB FIFO and fetch / decode info (this step is optional):
    1. Send command “READ OPABFR” (OnCE command = 11001010)
    2. ACK
    3. CLK
    4. Send command “READ OPABDR” (OnCE command = 11010011)
    5. ACK
    6. CLK

Example 9-4 Begin Debug Activity (Continued)

g. Send command “READ OPFIFO” (OnCE command = 11010001)

h. ACK

i. CLK

j. Send command “READ OPFIFO” (OnCE command = 11010001)

k. ACK

l. CLK

m. Send command “READ OPFIFO” (OnCE command = 11010001)

n. ACK

o. CLK

p. Send command “READ OPFIFO” (OnCE command = 11010001)

q. ACK

r. CLK

s. Send command “READ OPFIFO” (OnCE command = 11010001)

t. ACK

u. CLK

9.3.4.2 Displaying a Specified Register

Example 9-5 shows the procedure for displaying a specified register.

Example 9-5 Display a Specified Register—OnCE

1. Send command “WRITE OPDBR with GO and no EX” (OnCE command = 01001001); ODEC selects OPDBR as destination for serial data.
2. ACK

Example 9-5 Display a Specified Register—OnCE (Continued)

3. Send the 16-bit opcode: “MOVE reg, x:OGDB
After all 16-bits have been received, the PDB drives the OPDBR. ODEC generates PRNEW and releases the chip from the halt state, and the contents of the register specified in the instruction is loaded in the OPGDBR. The PRCYC1 signal (an internal signal) that marks the end of the instruction brings the chip again in the halt state, and an acknowledge is issued to the command controller.
4. ACK
5. Send command READ OPGDBR (OnCE command = 11001000)
ODEC selects PGDB as the source for serial data, and an acknowledge is issued to the command controller.
6. ACK
7. CLK

9.3.4.3 Displaying X Memory Area Starting at Address xxxx

Example 9-6 shows the procedure for displaying X memory area; this procedure uses Rn to minimize serial traffic.

Example 9-6 Display X Memory Area from Address xxxx—OnCE

1. Send command “WRITE OPDBR with GO and no EX” (OnCE command = 01001001); ODEC selects OPDBR as destination for serial data.
2. ACK
3. Send the 16-bit opcode: “MOVE R0,x:OGDB”
After all 16-bits have been received, the PDB drives the OPDBR. ODEC generates PRNEW and releases the chip from the halt state, and the contents of R0 are loaded in the OPGDBR. The PRCYC1 signal that marks the end of the instruction brings the chip again to the halt state, and an acknowledge is issued to the command controller.
4. ACK
5. Send command READ OPGDBR (OnCE command = 11001000)
ODEC selects PGDB as the source for serial data, and an acknowledge is issued to the command controller.
6. ACK

Example 9-6 Display X Memory Area from Address xxxx—OnCE (Continued)

7. CLK
   The command controller generates 16 clocks that shift out the contents of the OPGDBR. The value of R0 is thus saved and will be restored before exiting the debug mode.
8. Send command “WRITE OPDBR with no GO and no EX” (OnCE command = 00001001); ODEC selects OPDBR as destination for serial data.
9. ACK
10. Send the 16-bits of opcode: “MOVE #$xxxx,R0”
    After all 16-bits have been received, the PDB drives the OPDBR. ODEC generates PRNEW, so the PILR is loaded with the opcode. An acknowledge is issued to the command controller.
11. ACK
12. Send command “WRITE OPDBR with GO and no EX” (OnCE command = 01001001); ODEC selects OPDBR as destination for serial data.
13. ACK
14. Send the 16-bits of the 2nd word of: “MOVE #$xxxx,R0” (the xxxx field) where xxxx is the address to be read.
    After all 16-bits have been received, the PDB drives the OPDBR. ODEC releases the chip from the halt state, and the instruction starts execution. The PRCYC1 signal that marks the end of the instruction brings the chip again to the halt state, and an acknowledge is issued to the command controller.
15. ACK
16. Send command “WRITE OPDBR with GO and no EX” (OnCE command = 01001001); ODEC selects OPDBR as destination for serial data.
17. ACK
18. Send the 16-bit opcode: “MOVE X:(R0)+,x:OGDB”
    After all 16-bits have been received, the PDB drives the OPDBR. ODEC generates PRNEW and releases the chip form the halt state, and the contents of X:(R0) are loaded in the OPGDBR. The PRCYC1 signal that marks the end of the instruction brings the chip again in the halt state, and an acknowledge is issued to the command controller.
19. ACK

Example 9-6 Display X Memory Area from Address xxxx—OnCE (Continued)

20. Send command “READ OPGDBR” (OnCE command = 11001000)
    ODEC selects PGDB as source for serial data and an acknowledge is issued to the command controller.
21. ACK
22. CLK
23. Send command “NO SELECTION with GO and no EX” (OnCE command 11000000); ODEC releases the chip from the halt state, and the instruction is executed once again (in a REPEAT-like fashion). The PRCYC1 signal that marks the end of the instruction brings the chip again to the halt state, and an acknowledge is issued to the command controller.
24. ACK
25. Send command “READ OPGDBR” (OnCE command = 11001000)
    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 must be restored.

9.3.4.4 Returning from Debug Mode to Normal Mode

There are two cases for returning from the debug mode. In **Example 9-7**, control will be returned to the program that was running before debug was initiated, and in **Example 9-8**, the registers will be changed to jump to a different program. There is *no acknowledgment* on the DSO pin when the chip leaves the OnCE mode following a GO, EX. This is a special case of the “write a register” option.

Example 9-7 Returning from Debug Mode to Normal Mode—OnCE

1. Send command “WRITE OPDBR with no GO and no EX” (OnCE command = 00001001). ODEC selects the OPDBR as destination for serial data. Also ODEC selects the on-chip PAB register as source for the PAB bus. After the PAB was driven, an acknowledge is issued to the command controller.
2. ACK

Example 9-7 Returning from Debug Mode to Normal Mode—OnCE (Continued)

3. Send the 16-bits of the saved PGDB) value.
   After all 16-bits have been received, the PDB drives the OPDBR.
   ODEC generates PRNEW, so the entire chip loads the opcode.
   An acknowledge is issued to the command controller.
4. ACK
5. Send command “WRITE OPDBR with GO and EX” (OnCE command = 01101001). ODEC selects OPDBR as destination for serial data.
6. ACK
7. Send the 16-bits of the saved OPDBR value.
   After all 16-bits have been received, the PDB drives the OPDBR.
   ODEC releases the chip form the halt state, and the debug mode bit in OSR is cleared. The chip continues to execute instructions until a debug mode condition occurs.

Example 9-8 Jump to a New Program—Go from Address $xxxx

1. Send command “WRITE OPDBR with no GO and no EX” (OnCE command = 00001001). ODEC selects OPDBR as destination for serial data.
2. ACK
3. Send 16 bits of the opcode of a two word jump instruction instead of the saved OPGDBR (instruction latch) value.
   After all the 16-bits have been received, the PDB drives the OPDBR. ODEC causes the DSP to load the opcode. An acknowledge is issued to the command controller.
4. ACK
5. Send command “WRITE OPDBR with GO and EX” (OnCE command = 01101001). ODEC selects OPDBR as destination for serial data.
6. ACK

Example 9-8 Jump to a New Program—Go from Address $xxxx (Continued)

7. Send 16 bits of the target absolute address ($xxxx). The chip will resume fetching from the target address (with no pipeline problems). 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.
   In other words, after 16 bits have been received, the PDB drives the OPDBR. 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.

9.4 JTAG PORT ARCHITECTURE

The DSP56800 core provides a dedicated user-accessible test access port (TAP) that is fully compatible with the *IEEE 1149.1 Standard Test Access Port and Boundary Scan Architecture*. Problems associated with testing high-density circuit boards have led to development of this proposed standard under the sponsorship of the Test Technology Committee of IEEE and the Joint Test Action Group (JTAG). The DSP56800 core implementation supports circuit-board test strategies based on this standard. This section includes aspects of the JTAG implementation that are specific to the DSP. It is intended to be used with the supporting IEEE 1149.1a-1993 document. The discussion includes those items required by the standard to be defined and, in certain cases, provides additional information specific to the DSP implementation. For internal details and applications of the standard, refer to the IEEE 1149.1a-1993 document.

The TAP controller is a simple state machine, discussed in **TAP Controller** on page 9-59, used to sequence the JTAG port through its valid operations:

- Serially shift in or out a JTAG command
- Update (and decode) the JTAG instruction register
- Serially input or output a data value
- Update a JTAG (or OnCE) register.

Note: The JTAG port oversees the shifting of data into and out of the OnCE port through the TDI and TDO pins, respectively. The shifting, in this case, is guided by the same tap controller used when shifting JTAG information.

The test logic includes a TAP consisting of five dedicated signal pins, a 16-state controller, and three test data registers. A boundary scan register links all device signal pins into a single shift register. The test logic, implemented utilizing static logic design, is independent of the device system logic. The DSP56800 core implementation provides the following capabilities:

1. Perform boundary scan operations to test circuit-board electrical continuity (EXTEST).
2. Sample the DSP56800 core based device system pins during operation and transparently shift out the result in the boundary scan register. Pre-load values to output pins prior to invoking the EXTEST instruction (SAMPLE/PRELOAD).
3. Query identification information (manufacturer, part number and version) from an DSP56800 core-based device (IDCODE).
4. Add a weak pull-up device on all input signals to cause all open inputs to report a logic 1 and force a predictable internal state while performing external boundary scan operations (EXTEST_PULLUP).
5. Disable the output drive to pins during circuit-board testing (HIGHZ).
6. Force test data onto the outputs of an DSP56800 core-based device while replacing its boundary scan register in the serial data path with a single bit register (CLAMP).
7. Provide a means of accessing the OnCE controller and circuits to control a target system (ENABLE_ONCE).
8. Provide a means of entering the debug mode of operation (DEBUG_REQUEST).
9. Bypass the DSP56800 core for a given circuit-board test by effectively reducing the boundary scan register to a single cell (BYPASS).

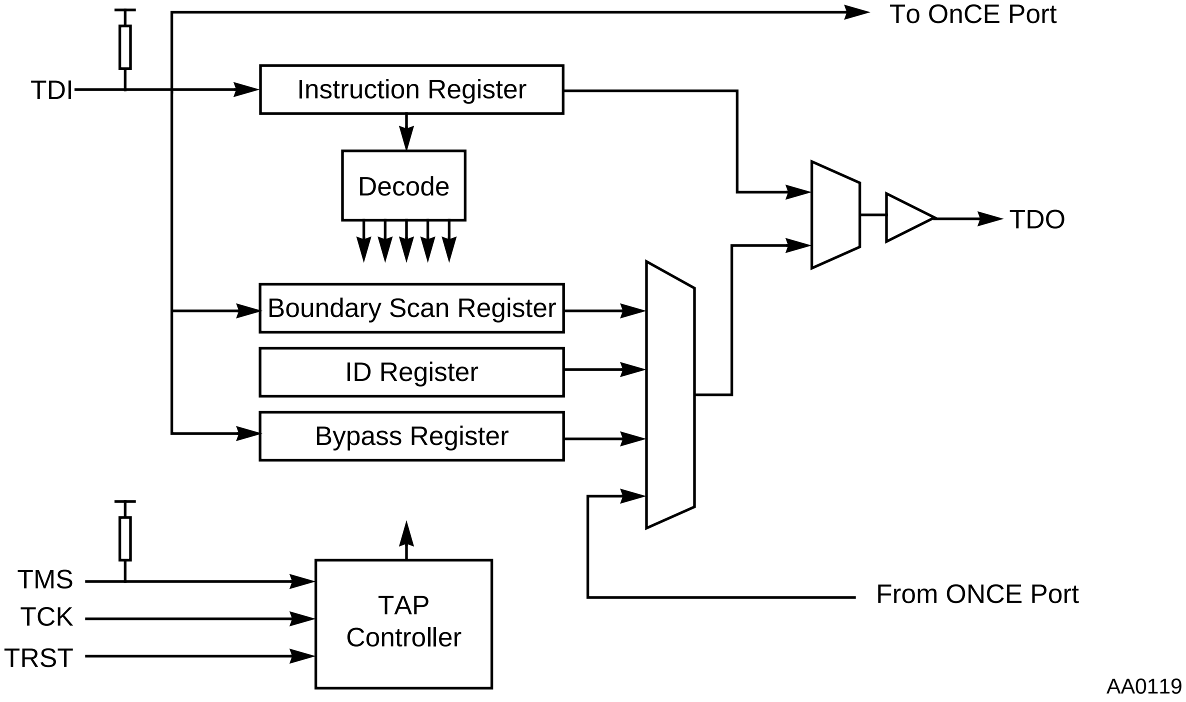
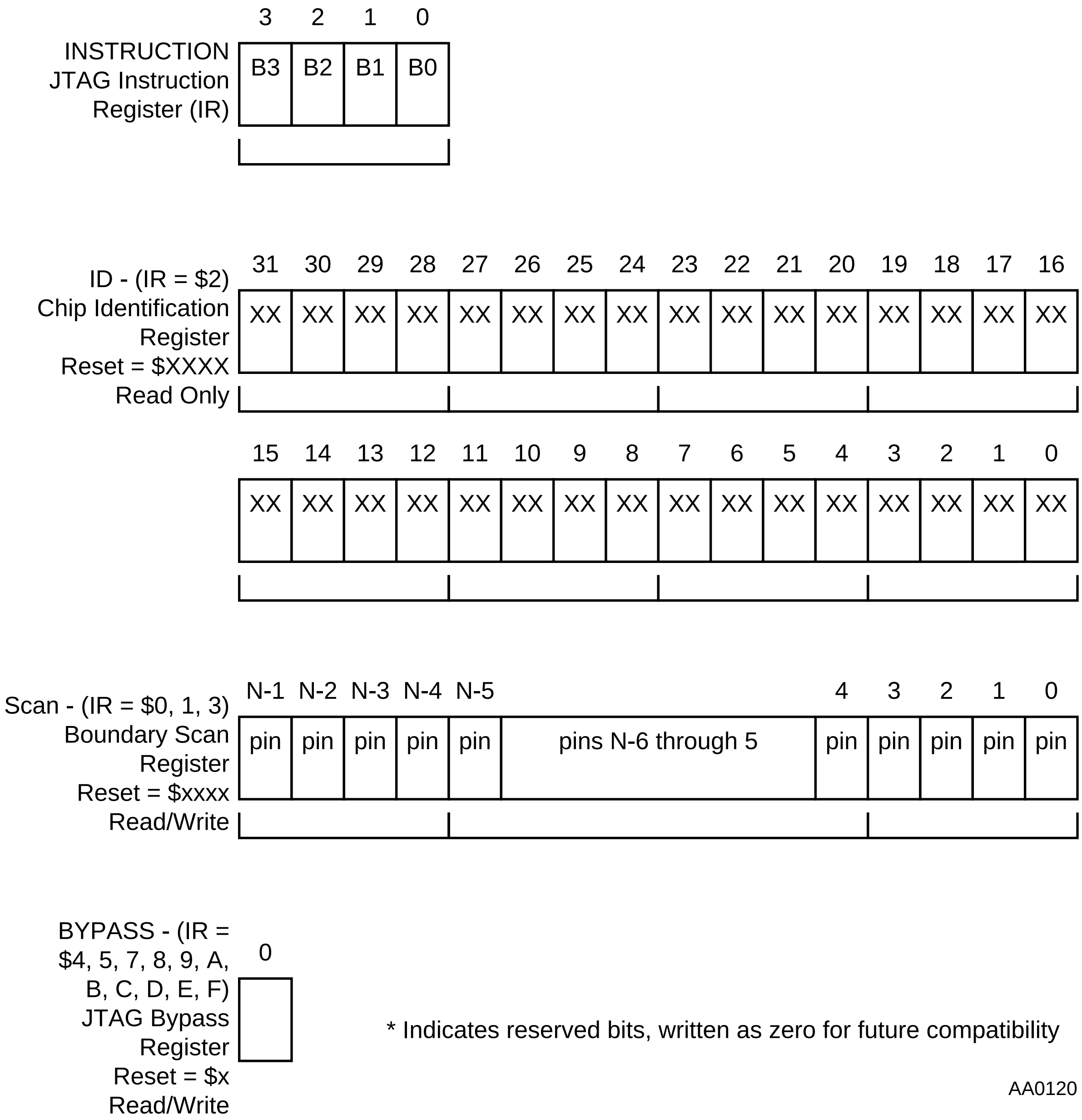


Figure 9-15 JTAG Block Diagram

A block diagram of the JTAG port is shown in **Figure 9-15**, and its corresponding programming model is shown in **Figure 9-16**. There are 3 read / write registers in the JTAG port: the instruction register, boundary scan register, and the bypass register. There is also one read-only register: ID Register.

By virtue of the TAP controller, the JTAG instruction register is always accessible through the JTAG port, whereas the other JTAG registers must be individually selected by the JTAG instruction register. The blocks and registers within the JTAG port are explained below.



* Indicates reserved bits, written as zero for future compatibility

AA0120

Figure 9-16 JTAG Port Programming Model

9.4.1 JTAG Chip Identification Register (CID)

The chip identification register (CID) is a 32-bit register used to provide a unique JTAG ID for each DSP chip. It is provided as a public instruction to allow the manufacturer, part number, and version of a component to be determined through the TAP. **Figure 9-17** shows the CID register configuration. One application of the device identification register is to distinguish the manufacturer(s) of components on a board when multiple sourcing is used. As more components emerge that conform to IEEE 1149.1a-1993, it is desirable to allow for a system diagnostic controller unit to blindly interrogate a board design in order to determine the type of each component in each location. This information is also available for factory process monitoring and for failure mode analysis of assembled boards.

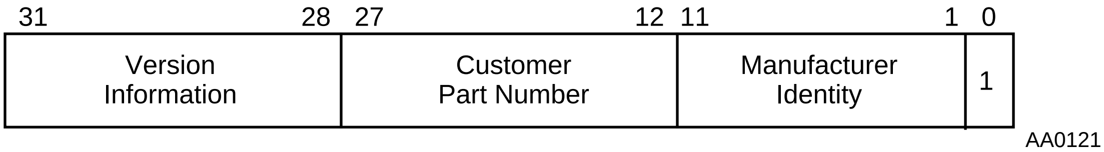


Figure 9-17 Identification Register Configuration

Motorola’s Manufacturer Identity is 00000001110. The Customer Part Number consists of two parts: Motorola Design Center Number (bits 27:22) and Design Center Assigned Sequence Number (bits 21:12). DSP’s Design Center Number is 000101. Version information and Design Center Assigned Sequence Number values vary depending on the current revision and implementation of a specific chip.

9.4.1.1 JTAG Boundary Scan Register

The boundary scan register is a register used to examine or control the “scan-able” pins on a DSP chip. The number of bits in this register varies from chip to chip, depending on the number of scan-able pins found on a particular device. One register bit is present for each scan-able pin on a DSP chip.

9.4.1.2 JTAG Bypass Register

The bypass register is a 1-bit register used to provide a simple, direct path from the TDI pin to the TDO pin. This is useful in boundary scan applications where many chips are serially connected together in a daisy-chain. Individual DSPs or other devices can be programmed with the BYPASS instruction so that individually they become pass-through devices during testing. This allows testing of a specific chip while still having all of the chips connected through the JTAG ports.

9.4.1.3 JTAG Instruction Register and Instruction Decoder

The TAP controller contains a 4-bit instruction register. The instruction is presented to an instruction decoder during the Update-IR state. See **TAP Controller** on page 9-59 for a description of the TAP controller operating states. The instruction decoder interprets and executes the instructions according to the conditions defined by the TAP controller state machine. The DSP56800 core implementation includes the three mandatory public instructions (BYPASS, SAMPLE / PRELOAD, and EXTEST) and six public instructions (CLAMP, HIGHZ, EXTEST_PULLUP, IDCODE, ENABLE_OnCE, and DEBUG_REQUEST).

The four bits (B3-B0) are used to decode 16 instructions as shown in **Table 9-14**.

Table 9-14 JTAG Instruction Register Encodings

B3-B0	Instruction
0000	EXTEST
0001	SAMPLE / PRELOAD
0010	IDCODE
0011	EXTEST_PULLUP
0100	HIGHZ
0101	CLAMP
0110	ENABLE_ONCE
0111	DEBUG_REQUEST
1111	BYPASS

All other encodings are reserved for future enhancements and will be decoded as BYPASS. The instruction register is reset to 0010 in the Test-Logic-Reset controller state. Therefore, the IDCODE instruction is selected on JTAG reset. In the Capture-IR state the two LSBs of the instruction shift register will be preset to 01, where the 1 is in the LSB location as required by the standard. The two most significant bits may either capture status or be set to 0. New instructions are shifted into the instruction shift register stage on Shift-IR state.

9.4.1.3.1 EXTEST—0000

The external test (EXTEST) instruction (0000) enables the boundary scan register between TDI and TDO, including cells for all digital device signals and associated control signals. The EXTAL and any Codec pins associated with analog signals are not included in the boundary scan register path. See the appropriate device user’s manual for details on the pins that have boundary scan cells.

In EXTEST the boundary scan register is capable of scanning user-defined values onto output pins, capturing values presented to input signals, and controlling the direction and value of bi-directional pins. EXTEST asserts internal system reset for the DSP system logic for the duration of EXTEST in order to force a predictable internal state while performing external boundary scan operations.

9.4.1.3.2 SAMPLE/PRELOAD—0001

The SAMPLE / PRELOAD instruction (0001) enables the boundary scan register between TDI and TDO. When this instruction is selected, the operation of the test logic shall have no effect on the operation of the on-chip system logic or on the flow of a signal between the system pin and the on-chip system logic as specified by the IEEE 1149.1 specification. This instruction provides two separate functions. First, it provides a means to obtain a snapshot of system data and control signals (SAMPLE). The snapshot occurs on the rising edge of TCK in the Capture-DR controller state. The data can be observed by shifting it transparently through the boundary scan register. In a normal system configuration many signals require external pull-ups to ensure proper system operation. Consequently, the same is true for the SAMPLE / PRELOAD functionality. The data latched into the boundary scan register during the Capture-DR controller state may not match the drive state of the package signal if the system-required pull-ups are not present within the test environment.

The second function of the SAMPLE / PRELOAD instruction is to initialize the boundary scan register output cells (PRELOAD) prior to selection of CLAMP, EXTEST, or EXTEST_PULLUP. This initialization ensures that known data will appear on the outputs when executing the EXTEST instruction. The data held in the shift register stage is transferred to the output latch on the falling edge of TCK in the Update-DR controller state. Data is not presented to the pins until the CLAMP, EXTEST, or EXTEST_PULLUP instruction is executed.

Note: Since there is no internal synchronization between the JTAG clock (TCK) and the system clock (CLK), the user must provide some form of external synchronization to achieve meaningful results when sampling system values via SAMPLE / PRELOAD.

9.4.1.3.3 IDCODE—0010

The IDCODE instruction (0010) enables the IDREGISTER between TDI and TDO. It is provided as a public instruction to allow the manufacturer, part number, and version of a component to be determined through the TAP. Once the IDCODE instruction is decoded, it will select the IDREGISTER which is a 32-bit test data register. Since the bypass register loads a logic 0 at the start of a scan cycle, whereas an IDREGISTER will load a constant logic 1 into its LSB, examination of the first bit of data shifted out of a component during a test data scan sequence immediately following exit from the Test-Logic-Reset controller state will show whether such a register is included in the design. When the IDCODE instruction is selected, the operation of the test logic shall have no effect on the operation of the on-chip system logic as required in the IEEE 1149.1a-1993 specification.

9.4.1.3.4 EXTEST_PULLUP—0011

The EXTEST_PULLUP instruction (0011) is provided as a public instruction to aid in fault diagnoses during boundary scan testing of a circuit board. This instruction functions similarly to EXTEST, with the only difference being the presence of a weak pull-up device on all input signals. The pull-up current will, given an appropriate charging delay, supply a deterministic logic 1 result on an open input. When this instruction is used in board-level testing with heavily-loaded nodes, it may require a charging delay greater than the two TCK periods needed to transition from the Update-DR state to the Capture-DR state. Two methods of providing an increase delay are available: traverse into the Run-Test/Idle state for extra TCK periods of charging delay or limit the maximum TCK frequency (slow down the TCK) so that 2 TCK periods are adequate. EXTEST_PULLUP asserts internal system reset for the DSP system logic for the duration of EXTEST_PULLUP in order to force a predictable internal state while performing external boundary scan operations.

9.4.1.3.5 HIGHZ—0100

The HIGHZ instruction (0100) enables the single-bit BYPASS register between TDI and TDO. It is provided as a public instruction in order to prevent having to drive the output signals back during circuit board testing. When the HIGHZ instruction is invoked, all output drivers are placed in an inactive-drive state. HIGHZ asserts internal system reset for the DSP system logic for the duration of HIGHZ in order to force a predictable internal state while performing external boundary scan operations.

9.4.1.3.6 CLAMP—0101

The CLAMP instruction (0101) enables the single-bit BYPASS register between TDI and TDO. It is provided as a public instruction. When the CLAMP instruction is invoked, the package output signals will respond to the preconditioned values within the update latches of the boundary scan register, even though the BYPASS register is enabled as the test data register. In-circuit testing can be facilitated by setting up guarding signal conditions that control the operation of logic not involved in the test with use of the SAMPLE/PRELOAD or EXTEST instructions. When the CLAMP instruction is executed, the state and drive of all signals remain static until a new instruction is invoked. A feature of the CLAMP instruction is that while the signals continue to supply the guarding inputs to the in-circuit test site, the BYPASS mode is enabled, thus minimizing overall test time. Data in the boundary scan cell remains unchanged until a new instruction is shifted in or the JTAG state machine is set to its reset state.

CLAMP asserts internal system reset for the DSP system logic for the duration of CLAMP in order to force a predictable internal state while performing external boundary scan operations.

9.4.1.3.7 ENABLE_ONCE—0110

The ENABLE_ONCE instruction (0110) enables the JTAG port to communicate with the OnCE state machine and registers. It is provided as a Motorola public instruction to allow the user to perform system debug functions. When the ENABLE_ONCE instruction is invoked, the TDI and TDO pins will be connected directly to the OnCE registers. The particular OnCE register connected between TDI and TDO is selected by the OnCE state machine and the OnCE instruction being executed. All communication with the OnCE instruction controller is done through the SELECT-DR-SCAN path of the JTAG state machine. Refer to **Section 2 Core Architecture Overview** for more information.

9.4.1.3.8 DEBUG_REQUEST—0111

The DEBUG_REQUEST instruction (0111) asserts a request to halt the core for entry to debug mode. It is typically used in conjunction with ENABLE_ONCE to perform system debug functions. It is provided as a Motorola public instruction. When the DEBUG_REQUEST instruction is invoked, the TDI and TDO pins will be connected to the BYPASS register. Refer to **Section 2 Core Architecture Overview** for more information.

9.4.1.3.9 BYPASS—1111

The BYPASS instruction (1111) enables the single-bit BYPASS register between TDI and TDO as shown in **Figure 9-18**. This creates a shift register path from TDI to the bypass register and finally to the TDO signal, circumventing the boundary scan register. This instruction is used to enhance test efficiency by shortening the overall path between TDI and TDO when no test operation of a component is required. In this instruction, the DSP system logic is independent of the test access port. When this instruction is selected, the test logic shall have no effect on the operation of the on-chip system logic as required in the IEEE 1149.1-1993a specification.

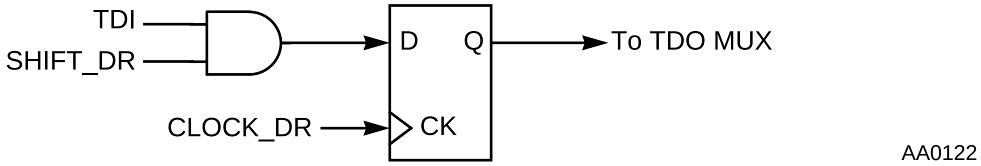


Figure 9-18 Bypass Register

9.4.1.4 TAP Controller

The TAP controller is a synchronous finite state machine that contains 16 states as illustrated in **Figure 9-19**. The TAP controller responds to changes at the TMS and TCK signals. Transitions from one state to another occur on the rising edge of TCK. The value shown adjacent to each state transition in this figure represents the signal present at TMS at the time of a rising edge at TCK.

The TDO pin remains in the high impedance state except during the Shift-DR or Shift-IR controller states. In these controller states, TDO will update on the falling edge of TCK. TDI is sampled on the rising edge of TCK.

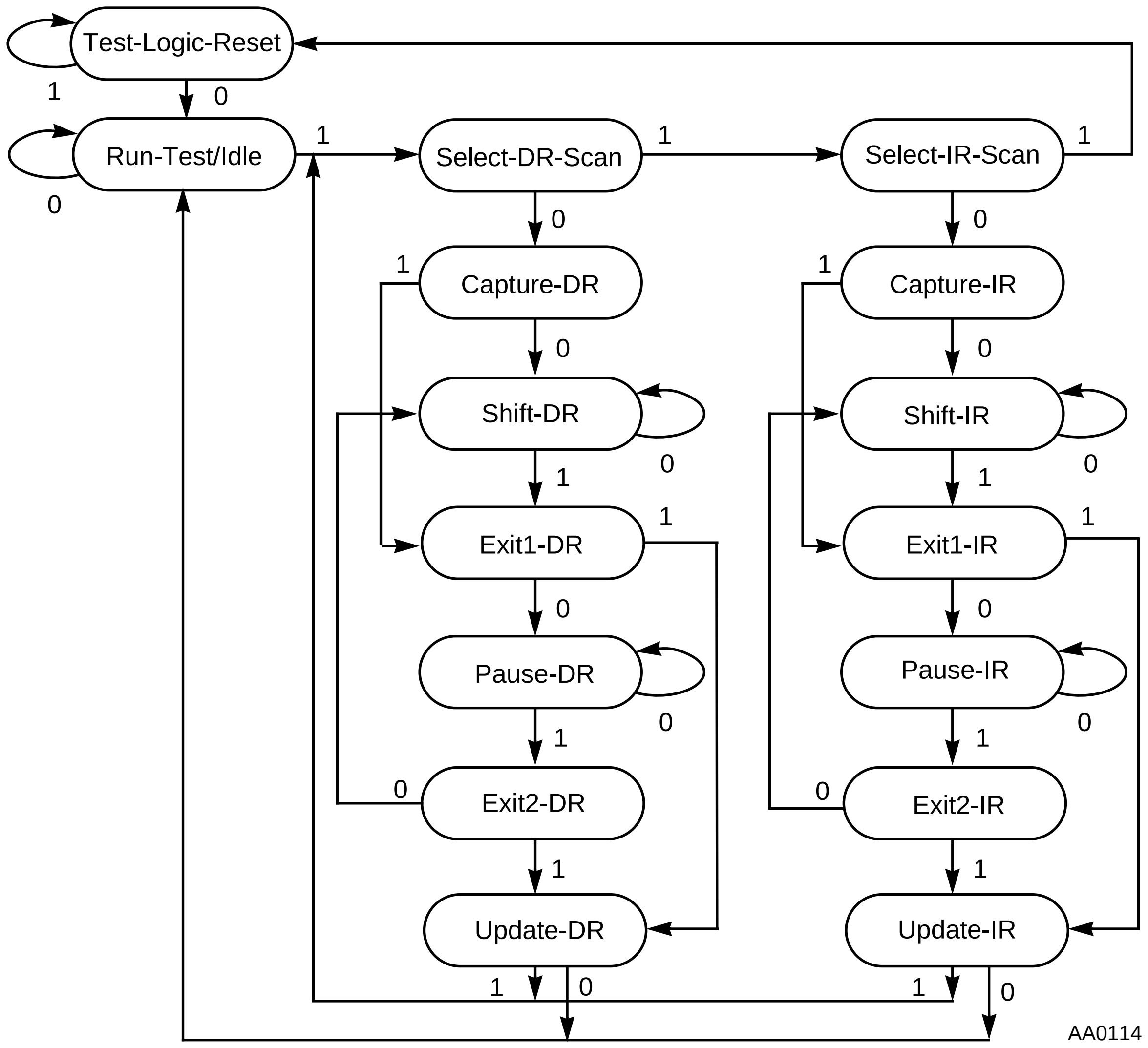


Figure 9-19 TAP Controller State Diagram

There are two paths through the 16-state machine. The SHIFT-IR_SCAN path is used to capture and load JTAG instructions into the instruction register. The SHIFT-DR_SCAN path is used to capture and load data into the other JTAG registers. The TAP controller will execute the last instruction decoded until a new instruction is entered at the Update-IR state or until the Test-Logic-Reset state is entered.

When using the JTAG port to access OnCE port registers, accesses are first enabled by shifting the ENABLE_ONCE instruction into the JTAG instruction register. Once this is selected, then the OnCE port registers and commands are read and written through the JTAG pins using the SHIFT-DR_SCAN path. Asserting the JTAG's TRST pin asynchronously forces the JTAG state machine into the Test-Logic-Reset state.

9.4.2 DSP56800 Restrictions

The control afforded by the output enable signals using the boundary scan register and the EXTEST instruction requires a compatible circuit-board test environment to avoid device-destructive configurations. The user must avoid situations in which the DSP56800 core output drivers are enabled into actively driven networks.

There are two constraints related to the JTAG interface. First, the TCK input does not include an internal pull-up resistor and should not be left unconnected to preclude mid-level inputs. The second constraint is to ensure that the JTAG test logic is kept transparent to the system logic by forcing TAP into the test-logic-reset controller state, using either of two methods. During power-up, $\overline{\text{TRST}}$ must be externally asserted to force the TAP controller into this state. After power-up is concluded, TMS must be sampled as a logic (one for five consecutive TCK rising edges. If TMS either remains unconnected or is connected to V_{DD}, then the TAP controller cannot leave the test-logic-reset state, regardless of the state of TCK.

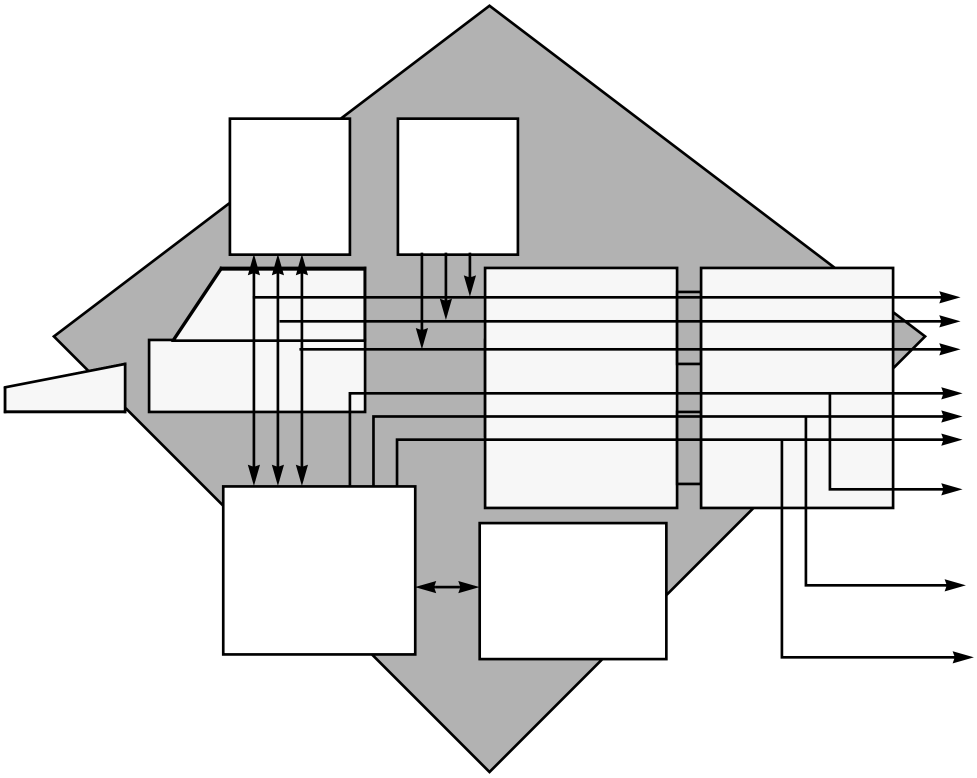
The DSP56800 core features a low-power stop mode, that is invoked using an instruction called STOP. The interaction of the JTAG interface with low-power stop mode is as follows:

1. The TAP controller must be in the Test-Logic-Reset state to either enter or remain in the low-power stop mode. Leaving the TAP controller Test-Logic-Reset state negates the ability to achieve low-power, but does not otherwise affect device functionality.
2. The TCK input is not blocked in low-power stop mode. To consume minimal power, the TCK input should be externally connected to V_{DD} or ground.
3. The TMS and TDI pins include on-chip pullup resistors. In low-power stop mode, these two pins should remain either unconnected or connected to V_{DD} to achieve minimal power consumption.

Since during stop state all DSP56800 core clocks are disabled, the JTAG interface provides the means of polling the device status (sampled in the capture-IR state). For an DSP56800 derivatives that do not include a debug request pin, the JTAG interface provides the software means of entering the debug mode by executing the DEBUG_REQUEST instruction.

SECTION 10

DEVELOPMENT TOOLS



10.1 INTRODUCTION

Motorola offers a full line of software and hardware digital signal processor (DSP) development tools to rapid development and debugging of user applications and algorithms on the DSP56800 Family. The development tools include the following:

- DSP56800 Family Assembler
- DSP56800 Family Linker/Librarian
- DSP56800 Family Simulator
- DSP56800 Family C Compiler with symbolic debugger
- DSP56800 Family Application Development System (ADS), including hardware and interface software

The DSP56800 Family Graphical User Interface (GUI) is included with the DSP56800 development tools. The GUI provides a multi-window graphical interface for the instruction simulator and application development system, giving the user source-level debug capability on assembly language and C language programs.

The DSP56800 Family Graphical User Interface provides display windows for the following:

- Hardware development and instruction simulator command and session windows
- Source Files
- Disassembled portions of memory
- Selected set of DSP core registers
- Selected set of on-chip peripheral registers
- Selected portion of memory
- Watch window
- C language call stack
- Breakpoint window

All of the commands are accessible through the GUI’s pull-down menus, dialog boxes, tool bars, windows, and buttons. Using these tools the user selects a desired operation such as setting a breakpoint in memory or displaying sections of memory. The interface then generates the corresponding command for the appropriate DSP56800 development tool. These commands are passed to the debugger via the GUI’s command window, and the output and other information is then displayed in the session window and other appropriate windows. The user may also enter commands directly into the command window, retaining direct control over the debugging session if desired. There is an expression calculator and many other unique features built into the GUI.

Figure 10-1 shows the placement of the development tools in the flow of development of a user application.

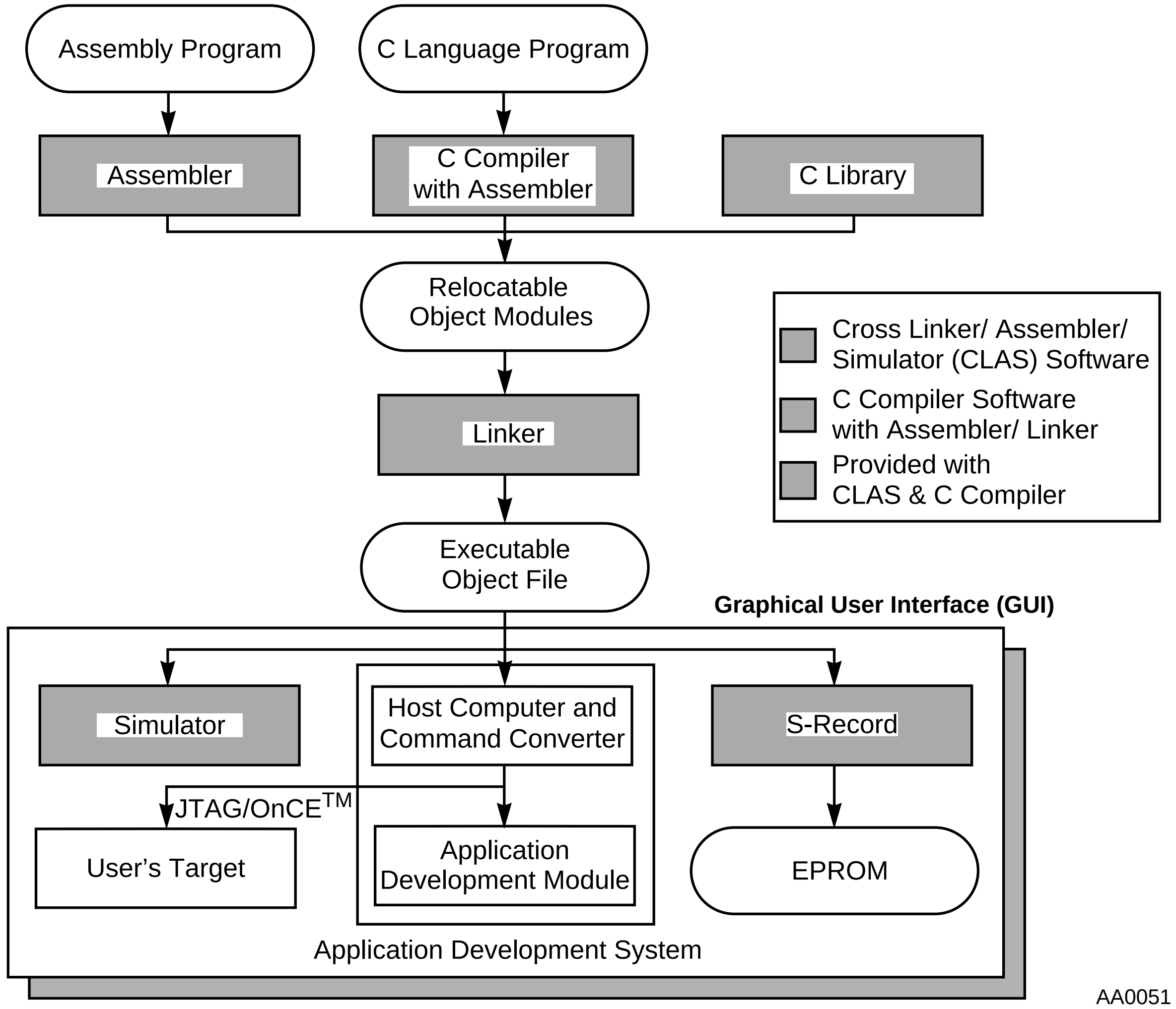

Assembly Program
C Language Program
Assembler
C Compiler with Assembler
C Library
Relocatable Object Modules
Linker
Executable Object File
Cross Linker/ Assembler/ Simulator (CLAS) Software
C Compiler Software with Assembler/ Linker
Provided with CLAS & C Compiler
Graphical User Interface (GUI)
Simulator
Host Computer and Command Converter
S-Record
JTAG/OnCE™
User's Target
Application Development Module
EPROM
Application Development System
AA0051

Figure 10-1 Development Flow

These tools can be ordered to operate on: ISA-BUS IBM PCs™, SBUS™ SUN-4 Workstations™, or Hewlett-Packard (HP) Series 7xx computers. Motorola's DSP development tools can be obtained through a local Motorola Semiconductor Sales Office or authorized distributor.

10.2 SOFTWARE DEVELOPMENT ENVIRONMENT

The software available from Motorola, the DSP56800CLASx software package, is written in C and consists of a relocatable macro cross assembler, linker, librarian, clock-by-clock multi-DSP-chip instruction simulator, and graphical user interface which are marketed as an integrated product. All software products run on IBM PCs, Hewlett-Packard Series 7xx computers, and SUN–4 Workstations.

The CLAS software packages are designed to provide flexibility to the programmer and a modular programming environment, giving full utilization to the DSP chips, a variety of data storage definitions, relocatability of generated code, symbolic debugging, and flexible linkages of object files. A library facility is included for creating archives of the final applications code.

10.2.1 Macro Cross Assembler

The DSP56800 Family Assembler (ASM56800) is a full-featured macro cross assembler that translates one or more source fields containing DSP instruction mnemonics, operands, and assembler directives into relocatable object modules that are relocated and linked by the Motorola DSP linker in the relocation mode. In the absolute mode, the assembler will generate absolute executable files. The assembler recognizes the full instruction set and all addressing modes of the DSP56800 family.

This assembler offers the usual complement of features found in modern assemblers, such as file inclusion, nested macros with support for macro libraries (via the MACLIB directive), and modular programming constructs ordinarily found only in higher level languages. A full set of assembler directives allows for assembly control, symbol definition, data definition / storage allocation, listing control and options, object file control, macros, conditional assembly, and structured programming.

The unique architecture and parallel operation of the DSP demands more advanced capabilities and programming aids that this assembler readily provides. These include built-in functions for common transcendental math computations such as sine, cosine, log, and square root functions; arbitrary expressions and modulo operations; and directives to define circular and bit-reversed data buffers. Moreover, the assembler incorporates extensive error checking and reporting to indicate programming violations peculiar to the digital signal processing environment or stemming from the advanced features of the DSP. These include errors for improper nesting of hardware DO loops and improper address boundaries for circular data buffers and bit-reversed buffers.

The assembler generates source code listings that include numbered source lines, optional titles and subtitles, optional instruction cycle counts, symbol table and cross-reference listings, and memory use reports.

The features of the DSP56800 Family Assembler include the following:

- Produces relocatable object modules compatible with the DSP linker program in the relocation mode.
- Produces absolute executable files compatible with the simulator program in the absolute mode.
- Supports full instruction set, memory spaces, and parallel data transfer fields of the DSP.
- Modular programming features including local labels, sections, and external definition / reference directives.
- Nested macro libraries.
- Complex expression evaluation, including boolean operators.
- Built-in functions for data conversion, string comparison, and common transcendental math operations.
- Directives to define circular and bit-reversed buffers.
- Extensive error checking and reporting.

10.2.2 Linker/Librarian

The DSP56800 Family Linker relocates and links relocatable object modules from the macro cross assembler to create an absolute executable file that can be loaded directly into the DSP56800 simulator or converted to Motorola S-record format for PROM burning.

The DSP56800 Family Librarian utility will merge multiple separate relocatable object modules into a single file, eliminating the need for reassembling known bug-free routines every time the mainline program is assembled.

10.2.3 Clock-by-Clock Instruction Simulator

The DSP56800 Family Simulator (SIM56800) is a software tool for developing programs and algorithms for the DSP. This program exactly emulates all of the functions (except for the JTAG/OnCE) of the DSP including the DSP core, all memory and register updates associated with program code execution, the entire internal and external memory space of the DSP, all on-chip peripheral operations, and all exception processing activity. This enables the simulator program to count clock and instruction cycles, providing an accurate measurement of code execution time that is so critical in digital signal processing applications.

The multi-DSP-chip simulator has the same look, feel and functions as the interface software provided with the ADS, making movement between the DSP chip's simulation and hardware environments easy.

The simulator program executes DSP object code generated by the linker or the simulator’s internal single-line assembler. The object code is first loaded into the simulated DSP memory map. Then, instruction execution can proceed in single-step mode (stopping after each instruction has been executed) or until a user-defined breakpoint is encountered. During program debug, the registers or memory locations may be displayed or changed.

The simulator package includes linkable object code libraries of simulator functions that were used to create the simulator. The libraries allow a customized simulator to be built and integrated with unique system simulations. Source code for some of the functions, such as the terminal I/O functions and external memory accesses, is provided to allow close simulation of the particular application.

The features of the DSP56800 Family Simulator include the following:

- Multiple DSP device simulation
- Source-level symbolic debug of assembly and C source programs
- Conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Instruction and cycle timing counters
- Session and/or command logging for later reference
- Input/output ASCII files for device peripherals
- Help file and help line display of simulator commands
- Macro command definition and execution
- Display enable/disable of registers and memory locations
- Hexadecimal/decimal/binary calculator

10.2.4 C Cross Compiler

The DSP56800 C Cross Compiler software package contains an optimizing C cross compiler and symbolic debugger and can be used with Motorola’s instruction simulator and application development system, assembler / linker / librarian, ANSI C libraries, and COFF and Motorola S-Record utilities. The C compiler is not available for Hewlett-Packard computers.

The DSP56800 Family C Compiler features include the following:

- Supports ISO / ANSI C, Strict ISO / ANSI C, and K&R C (pcc)
- Supports fractional data types in addition to standard C data types
- Supports pragmas for improving the compiler’s processing performance
- Supports intrinsics for modulo buffer support and other useful functions
- Assembly code support within the C code
- ISO C run-time library

The C compiler software package is supported on IBM PC and SUN-4 Workstations.

10.3 HARDWARE DEVELOPMENT ENVIRONMENT

Motorola DSP is continually developing evaluation and design tools to assist customers in evaluating new DSP products and adapting them to specific design requirements. The Application Development System (ADS) undergoes constant evaluation and development to provide the highest level of customer support.

The basic level of customer support is the evaluation board. Customer evaluation kits containing these boards are provided to evaluate specific DSP functionality. Each kit includes the following:

- An evaluation board
- Related DSP documentation such as data sheets, user’s manuals, and family manuals
- A user’s manual for the evaluation board
- A CD containing supporting software
- Documentation for recommended Motorola and third party devices that can interface with the evaluation board

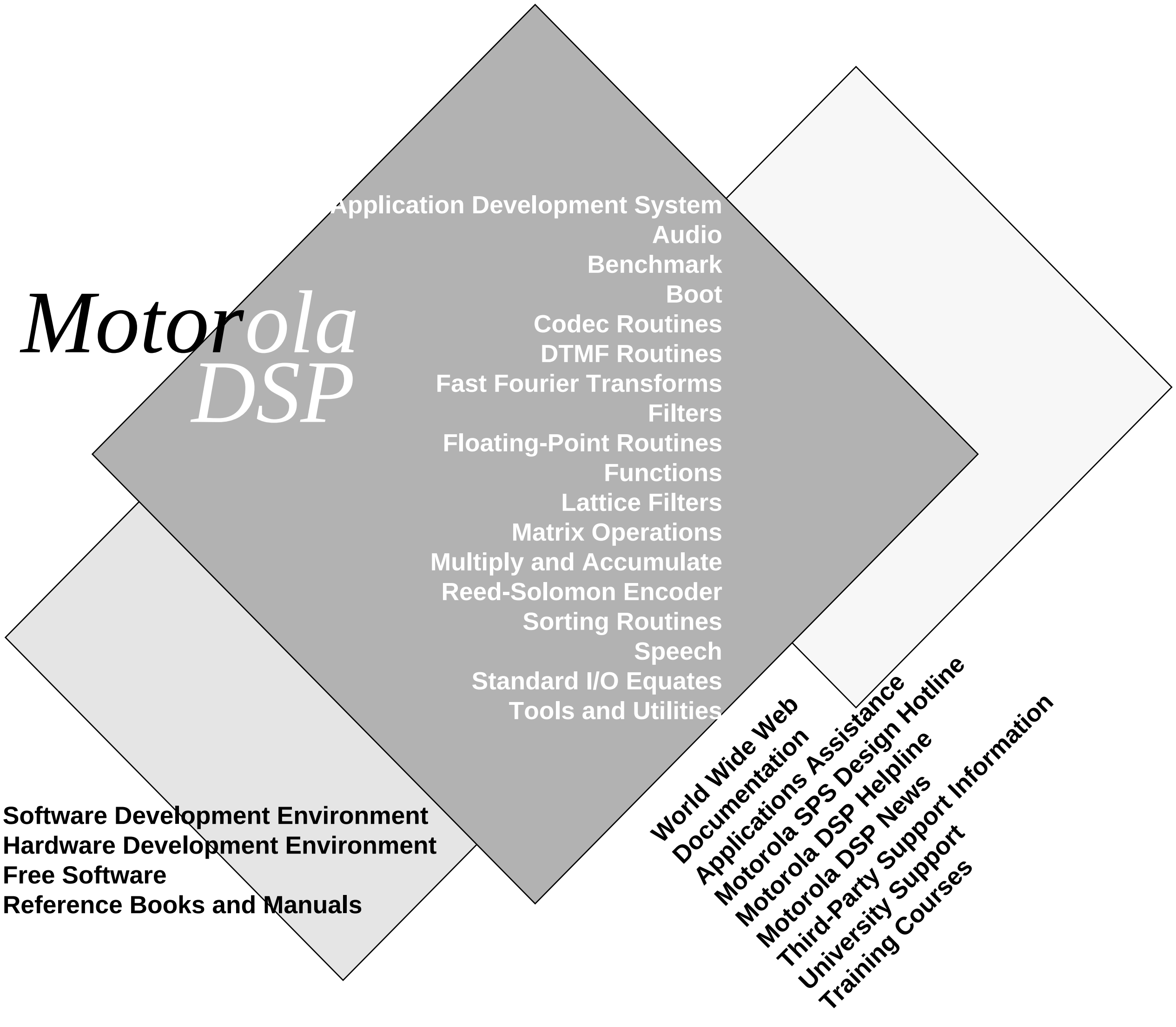
Motorola provides an upgrade path beyond chip evaluation for developing complex applications that integrate the DSP functionality. This can include the following:

- Interface hardware and software
- A command converter module
- An application development module that works with the evaluation board

For more information on Motorola’s hardware development environment, consult the Motorola DSP Home Page on the World Wide Web (see **Section 11 Additional Support**).

SECTION 11

ADDITIONAL SUPPORT



Motorola DSP

Application Development System
Audio
Benchmark
Boot
Codec Routines
DTMF Routines
Fast Fourier Transforms
Filters
Floating-Point Routines
Functions
Lattice Filters
Matrix Operations
Multiply and Accumulate
Reed-Solomon Encoder
Sorting Routines
Speech
Standard I/O Equates
Tools and Utilities

Software Development Environment
Hardware Development Environment
Free Software
Reference Books and Manuals

World Wide Web
Documentation
Applications Assistance
Motorola SPS Design Hotline
Motorola DSP Helpline
Motorola DSP News
Third-Party Support Information
University Support
Training Courses

11.1 OVERVIEW

User support from the conception of a design through completion is available from Motorola and third-party companies as shown in the following table:

Table 11-1 User Support Available

	Motorola	**Third Party**
Function	**Support Description**	**Support Description**
Design	• World Wide Web • Documentation • Applications Assistance • Training • Free Software	• Data Acquisition Packages • Filter Design Packages • Operating System Software • Simulator
Prototyping	• Software Development Environment • Hardware Development Environment	• Logic Analyzer with DSP56000/DSP56001 ROM Packages • In-Circuit Emulators • Data Acquisition Cards • DSP Development System Cards • Operating System Software • Debug Software
Design Verification	• Software Development Environment • Hardware Development Environment	• Data Acquisition Packages • Logic Analyzer with DSP56000/DSP56001 ROM Packages • Data Acquisition Cards • DSP Development System Cards • Application-Specific Development Tools • Debug Software

11.2 WORLD WIDE WEB

Motorola and Digital Signal Processors (DSP) Marketing maintain several World Wide Web (WWW) locations that you can access using third-party web browser software. The following web pages are good places to start:

- **Motorola DSP Home Page:**
  http://motserv.indirect.com/dsp/DSPhome.html
- **Motorola Product Groups:**
  http://motserv.indirect.com/home2/prodgroups/html/prod_groups.html
- **Motorola SPS Home Page:**
  http://motserv.indirect.com/home2/mothome.html

Note: You can get to both the Product Groups and the DSP home pages from the SPS home page. SPS Home Page → Product Groups→ DSP.

11.2.1 Motorola DSP Home Page

The Motorola DSP Home Page gives you access to the following information:

- **What's New**
  Product announcements, product updates, documentation updates.
- **Product Overviews**
  Description of features and functions of 16-bit, 24-bit, 32-bit, and peripherals.
- **University Program**
  Special donations of hardware and software.
- **Development Tools**
  Documentation available on-line for development tools.
- **Technical Documentation**
  On-line documentation for 16-bit, 24-bit, 32-bit, and peripherals. Link to Dr. Bub Archives for free software.

11.2.2 Motorola Product Groups

The Motorola Product Groups Home Page gives you access to the following information:

- Special Product Notices
- Advanced Microcontroller Division (AMCU)
- Analog Integrated Circuit (IC) Division (Linear)
- Application-Specific Integrated Circuits (ASIC) Bipolar and Complementary Metal Oxide Semiconductor (CMOS)
- Customer-Specific Integrated Circuit (CSIC) Microcontroller Division
- Metal Oxide Semiconductor (MOS) Digital-Analog IC Division
- RF Products - Communication Semiconductor Products Division (CSPD)
- **Digital Signal Processors (DSP)**
- Dynamic Memory Products Division
- Fast Static RAMs
- Hi-Performance Embedded Systems Division
- Logic IC Division
- Transient Voltage Suppressors (TVS)/Zeners and Reference Diodes
- Small Signal, FETs and Diodes
- Optoelectronic and Signal Products Division
- Power Products Division
- Surface Mount Information
- Sensor Products Division - Senseon
- Reduced Instruction Set Computers (RISC) Microprocessors Division

11.2.3 Motorola SPS Home Page

The Motorola Semiconductor Products Sector (SPS) Home Page gives you access to the following information:

- **Master Selection**
  Search for specific selection information (not browse)
- **Pricing**
  Search for specific price information (not browse)
- **Library**
  Search for specific library information (not browse)

- **On-Line Forms**
  - Literature Ordering.
    Request literature directly by number from a Literature Distribution Center (LDC).
  - Mfax.
    Request data sheets, application notes, bulletins, and selector guides to be sent to your fax machine.
  - Technical Support
    Send a request for assistance from the technical help line.
- **Books**
  Literature abstracts
- **Device Models**
  Spice models for selected semiconductors
- **Product Groups**
  Motorola Product Groups
- **Case Outlines**
  Mechanical drawings of packages
- **Other Services and Information**
  - Press Releases
  - Other Servers
  - Employment Pages
  - Mail
  - Sales
  - Consulting
  - Training
  - Rapid-NET Direct Ship
  - Motorola Financial Information
  - Powered by Motorola

11.3 DOCUMENTATION

DSP Marketing provides the following types of documentation:

- Product Briefs
- Data Sheets
- Family Manuals
- User’s Manuals
- Application Notes

These documents can be accessed in one or more of the following ways:

- View on-line
  (DSP Home Page → Technical Documentation)
- Order by number from Literature Distribution Center (LDC).
  - Fill out WWW on-line form
    (SPS Home Page → Literature Order)
  - Call Literature and Printing Services at (800) 441-2447.
- Receive via FAX (Mfax)
  - Fill out WWW on-line form
    (SPS Home Page → Mfax)
  - Call the Mfax automated service at (602) 244-6591.

11.4 APPLICATIONS ASSISTANCE

Applications assistance is available via:

- WWW
- Motorola SPS Design Hotline
- Motorola DSP Helpline
- Motorola DSP Newsletter
- Third-Party Support
- University Support
- Training

11.4.1 WWW

The World Wide Web is described in paragraph **World Wide Web** on page 11-4.

11.4.2 Motorola SPS Design Hotline

Information and assistance for all Motorola products is available through the Motorola Customer Support Center at the following number:

1-800-521-6274

11.4.3 Motorola DSP Helpline

Information and assistance for DSP applications is available through one of the following:

- Local Motorola field office
- Email
  dsphelp@dsp.sps.mot.com
- FAX
  (512) 891-4665

Note: To receive the fastest service, contact the field office first. If they cannot help you, send email or fax.

11.4.4 Motorola DSP Newsletter

The Motorola DSP News is a quarterly newsletter providing information on new products, application briefs, questions and answers, DSP product information, third-party product news, etc. This newsletter is free and is available upon request by sending a request for “DSPNEWSL/D” by one of the following methods:

- Order by FAX

(602) 994-6430

- Order online
  (DSP Home Page → Literature Order Form)
- Order from the Literature Distribution Center by phone:

(800) 441-2447

11.4.5 Third-Party Support Information

Information about third-party manufacturers who use and support Motorola DSP products is available by calling Motorola DSP Marketing at the following number:

(512) 891-3098

Third-party support includes:

- Filter design software
- Logic analyzer support
- Boards for VME, IBM-PC clones, MACII boards
- Development systems
- Data conversion cards
- Operating system software
- Debug software

Information is also available on the WWW (DSP Home Page → Development Tools) and in DSP News.

11.4.6 University Support

The Motorola University Support program helps DSP engineers of tomorrow experience first-hand the features of Motorola’s DSP products in university DSP laboratories using Motorola-donated DSP hardware and software.

Information concerning university support programs and university discounts for all Motorola DSP products is available by calling Motorola DSP Marketing at the following number:

(512) 891-3098

11.4.7 Training Courses

Training courses conducted by Motorola are offered in your plant or at the Phoenix, Arizona training facility. Other courses conducted by Motorola’s training consulting partners are available in other locations.

For more information about training, visit the Motorola WWW Training page (SPS Home Page → Other Info & Services → Training),

or call **Motorola** training at:

(602) 302-8008

or call **Arnewash, Incorporated** at:

(970) 223-1616

or call **Ascent Technology** at:

(800) 410-3601

11.5 SOFTWARE DEVELOPMENT ENVIRONMENT

The CLASx (**C**ompiler, **L**inker, **A**ssembler, **S**imulator) Software Development Environment is an integrated product written in C which comprises the following:

- C compiler
- Linker
- Relocatable macro cross assembler
- Clock-by-clock instruction simulator
- Librarian
- Graphical user interface

The CLASx Software Development Environment can be used on the following platforms:

- IBM™ PCs (386 or higher) running DOS 2.x and 3.x
- Macintosh™ II running MAC OS 7.0 or later
- SUN-4™ running Sun OS 4.1.x or Solaris 2.4
- Hewlett-Packard Series 7xx running HP-UX A.09.05

For more information about the CLASx Software Development Environment, see **Software Development Environment** on page 10-5.

11.6 FREE SOFTWARE

Motorola DSP Marketing provides free software associated with DSP products via the WWW. A partial list of the programs available is given in the following sections. New software will be posted on the web page as it is made available.

To download the free software that has been developed for Motorola DSPs, perform the following steps:

1. Connect to the WWW.
2. Go to the DSP Home Page.
3. Select Technical Documentation.

4. Select Dr. Bub Archives
5. Select the Macintosh (hqx) or PC (zip) file that you want to download.

11.6.1 Audio Software

The software in the following table is useful when designing audio with the DSP56000 and DSP56300 families.

Note: See Application Note APR2/D *Digital Stereo 10-Band Graphic Equalizer Using the DSP56001* for more information.

Table 11-2 Audio Software Available on the WWW

Document ID	DSP	Description	Size (K)
dge.asm	56000	Digital graphic equalizer source code	14.9
dge.lod	56000	Digital graphic equalizer hex file	2.7
dge.p	56000	Digital graphic equalizer s-record file	2.7
rvb1.asm	56000	Easy-to-read reverberation routine source code	17
rvb2.asm	56000	Same as rvb1.asm but optimized	15.4
stereo.hlp	56000	stereo.asm help file	0.6

11.6.2 Benchmark Programs

The programs in the following table are useful in many applications and are typically used to benchmark against other DSP chips.

Note: See the *DSP56000 Family Manual* for more details on benchmarking.

Table 11-3 Benchmark Programs Available on the WWW

Document ID	DSP	Description	Size (K)
1-56.asm	56000 56001	20 tap FIR filter source code	4.2
2-56.asm	56000 56001	64 tap FIR filter source code	4.2

Table 11-3 Benchmark Programs Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
3-56.asm	56000 56001	67 tap FIR filter source code	4.2
4-56.asm	56000 56001	8 pole 4 multiply cascaded canonic IIR filter source code	3.7
5-56.asm	56000 56001	8 pole 5 multiply cascaded canonic IIR filter source code	3.9
6-56.asm	56000 56001	8 pole cascaded transposed IIR filter source code	2.8
7-56.asm	56000 56001	Dot product source code	2
a-56.asm	56000 56001	Memory to memory FFT - 64 point source code	7.3
b-56.asm	56000 56001	Memory to memory FFT - 256 point source code	7.3
b11.asm	96002	Real multiply source code	0.9
b110.asm	96002	N complex updates source code	0.9
b110a.asm	96002	N complex updates source code	2.7
b111.asm	96002	Complex correlation or convolution (FIR filter) source code	2.7
b112.asm	96002	Nth order power series (real) source code	1.7
b113.asm	96002	2nd order real biquad IIR filter source code	1.4
b114.asm	96002	N cascaded real biquad IIR filters source code	3.9
b115a.asm	96002	Fast fourier transforms source code	3.2
b115b.asm	96002	Faster radix 2 decimation in time FFT source code	8.4
b115c.asm	96002	Radix 4 decimation frequency FFT source code	8.6
b116.asm	96002	LMS adaptive filter source code	5.8
b117.asm	96002	FIR lattice filter source code	3.5
b119.asm	96002	General lattice filter source code	4.3
b12.asm	96002	N real multipliers source code	1.3
b120.asm	96002	Normalized lattice filter source code	4.6
b123.asm	96002	N point 3x3 2d FIR convolution source code	7.4
b124.asm	96002	Table lookup w/linear interpolation between points source code	3.5
b125.asm	96002	Argument reduction source code	3.5
b126.asm	96002	Non-IEEE floating point division source code	2.1

Table 11-3 Benchmark Programs Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
b127.asm	96002	Multibit rotates source code	9.6
b128.asm	96002	Bit field extraction / insertion source code	8.6
b129.asm	96002	Newton-Raphson approximation for 1.0 / SQRT (x) source code	1.7
b13.asm	96002	Real update source code	1
b130.asm	96002	Newton-Raphson approximation for SQRT (x) source code	1.6
b131.asm	96002	Unsigned integer divide source code	3.3
b132.asm	96002	Signed integer divide source code	3.1
b133a.asm	96002	Graphics accept / reject, floating point version source code	2.6
b133b.asm	96002	Line accept / reject, floating point version source code	2.6
b133c.asm	96002	Line accept / reject, fixed point version source code	1.7
b133d.asm	96002	Four point polygon accept / reject source code	3.6
b133e.asm	96002	Four point polygon accept / reject (looped) source code	1.5
b134.asm	96002	Cascaded five coefficient transpose IIR filter source code	2.5
b135.asm	96002	3-dimensional graphics illumination source code	4.2
b136.asm	96002	Pseudorandom number generation source code	1.4
b137.asm	96002	Bezier cubic polynomial evaluation source code	3.5
b138a.asm	96002	Pack 4 bytes into a 32-bit word source code	1.1
b138b.asm	96002	Pack two 16-bit words into a single 32-bit word source code	0.9
b138c.asm	96002	Unpack a 32-bit word into 4 extended bytes source code	1.2
b138d.asm	96002	Unpack a 32-bit word into two 16-bit sign-extended bytes source code	1
b139.asm	96002	Nth order polynomial evaluation for 2 points source code	1.3
b14.asm	96002	N real updates source code	1.4
b140a.asm	96002	Graphics bit block transfer (BITBLT) source code	3.1
b140b.asm	96002	64-bit block transfer source code	3
b141.asm	96002	64x64 bit unsigned multiply source code	1.9

Table 11-3 Benchmark Programs Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
b142.asm	96002	Approximation of 1 / dl source code	1
b143a.asm	96002	Line drawing source code	5.4
b143b.asm	96002	Integer incremental line drawing algorithm source code	3.6
b144.asm	96002	Wire frame graphics rendering source code	54.2
b15.asm	96002	FIR filter w / data shift source code	1.5
b16.asm	96002	Real * complex correlation or convolution (FIR filter) source code	1.5
b17.asm	96002	Complex multiply source code	1.4
b18.asm	96002	N complex multiply source code	1.8
b19.asm	96002	Complex update source code	1.5
c-56.asm	56000 56001	Memory to memory FFT - 1024 point source code	10.3
d-56.asm	56000 56001	Port to memory FFT - 64 point source code	16.1
d2-56.asm	56000 56001	Port to memory FFT - 64 point source code	8
e-56.asm	56000 56001	Port to memory FFT - 256 point source code	1.6
e2-56.asm	56000 56001	Port to memory FFT - 256 point source code	8
f-56.asm	56000 56001	Port to memory FFT - 1024 point source code	16.1
f2-56.asm	56000 56001	Port to memory FFT - 104 point source code	11.2
magsqr.asm	56000 56001	Magnitude squared macro source code	0.5
read-me	56000 56001	Motorola benchmark descriptions text file	7.6
results.100	56000 56001	**Data text files**	1.8
results.75	56000 56001	**Data text files**	1.8
sincos.asm	56000 56001	Sine-cosine table generator for FFTs source code	1.2
sincos.hlp	56000 56001	Sine-cosine table generator help file	1.1

Table 11-3 Benchmark Programs Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
singen.asm	56000 56001	Generates “points” samples of a sine wave source code	0.9
sqrt3.asm	56000 56001	Full precision square root by polynomial approximation source code	1.4
sqrt3.hlp	56000 56001	Full precision square root help file	1
wbh4m.asm	56000 56001	Blackman-Harris 4 term minimum sidelobe window source code	0.7

11.6.3 Boot Software

The files and software listed in the following table are for booting from an EPROM.

Table 11-4 Boot Software Available on the WWW

Document ID	DSP	Description	Size (K)
boot.art	56000	Article on booting	6.7
boot.asm	56000	Construct a boot module source code	2.3
bood.lod	56000	Construct a boot module hex file	1
boot.lst	56000	Construct a boot module list file	4.1
boot.p	56000	Construct a boot module s-record file	1

11.6.4 Codec Routines

The programs in the following table perform code/decode analog -to-digital and digital-to-analog conversions.

Note: See the application note APR12/D *Twin CODEC Expansion Board for the DSP56000 Application Development System* for more information.

Table 11-5 Codec Routines Available on the WWW

Document ID	DSP	Description	Size (K)
linlog.asm	56000	Linear PCM to companded codec data conversion	4.8
linlog.hlp	56000	linlog.asm help file	1.7
loglin.asm	56000 56001	Companded codec to linear PCM data conversion	4.6
loglin.hlp	56000 56001	loglin.asm help file	1.5
loglint.asm	56000 56001	loglin.asm test program source code	2.2
loglint.hlp	56000 56001	loglint.asm help file	2

11.6.5 Demo Software

Table 11-6 Demo Software Available on the WWW

Document ID	DSP	Description	Size (K)
julia.asm	96000	Generates the Julia set	2.8

11.6.6 DTMF Routines

The programs in the following table are tone generation and detection codes for Dial Tone Multi-Frequency (DTMF) applications. The Goertzel algorithm is an optimized DTMF algorithm.

Table 11-7 DTMF Routines Available on the WWW

Document ID	DSP	Description	Size (K)
clear.cmd	56000	Clear command file	0.1
data.lod	56000	**Data** hex file	0.4
det.asm	56000	Subroutine used in IIR DTMF source code	5.9
dtmf.all	56000	Compilation of all DTMF source code routines	73.5

Table 11-7 DTMF Routines Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
dtmf.asm	56000	Main routine used in IIR DTMF source code	10.7
dtmf.mem	56000	DTMF routine memory file	0.5
dtmfmstr.asm	56000	Main routine for multi-channel DTMF source code	7.4
dtmfmstr.mem	56000	Multi-channel DTMF routine memory file	0.04
dtmftwo.asm	56000	DTMF receiver and generator main program source code	10.3
ex56.bat	56000	56000 assembler batch file	0.1
example.lst	56000	Goertzel algorithm list file	11.6
genxd.lod	56000	**X load file**	0.2
genyd.lod	56000	**Y load file**	0.2
goertzel.asm	56000	Goertzel routine source code	4.4
goertzel.lnk	56000	Goertzel routine link file	7
goertzel.lst	56000	Goertzel routine list file	11.6
load.cmd	56000	**Load** command file	0.04
read.me	56000	Instructions text file	0.7
sub.asm	56000	Subroutine linked for use in IIR DTMF source code	2.5
tstgoert.mem	56000	Goertzel routine memory file	0.4

11.6.7 Encoders

Table 11-8 Reed-Solomon Encoder Available on the WWW

Document ID	DSP	Description	Size (K)
bingray.asm	56000	Binary to Gray code conversion macro source code	0.6
bingrayt.asm	56000	bingray.asm test program source code	1
newc.c	56000	Reed-Solomon coder (C source code)	4.1
readme.rs	56000	Instructions for Reed-Solomon coding text file	5.2
rscd.asm	56000	Reed-Solomon coder for DSP56000 simulator source code	5.8
table1.asm	56000	Reed-Solomon coder include file	8
table2.asm	56000	Reed-Solomon coder include file	4

11.6.8 Fast Fourier Transforms

The Fast Fourier Transforms (FFTs) in the following table include complex and real FFTs.

Note: See the application note APR4/D *Implementation of Fast Fourier Transforms on Motorola's Digital Signal Processors*.

Table 11-9 Fast Fourier Transforms Available on the WWW

Document ID	DSP	Description	Size (K)
bergorde.asm	96000	Bergland order table generator source code	1.8
bergsinc.asm	96000	Bergland sine/cosine coefficient lookup table generator source code	0.9
bitrev.asm	56000	Converse bit reverse order to normal order in-place source code	1.1
bitrevtw.asm	56156	Sort sin and cosine coefficient look-up tables in bit reverse order for 56156 (source code)	1.4
cfft3n.asm	96000	Complex - Radix 2 decimation in-place FFT source code	3.2
cfft3nn.asm	96000	Complex - radix 2 decimation in-time FFT source code	13.2
cfft56.asm	56000	512 point non-in-place FFT source code	21.7
cfft96.asm	96000	Complex - radix 2 Cooley-Tukey decimation-in-time FFT source code	13.2
cfft96t.asm	96000	Complex - radix 2 Cooley-Tukey decimation-in-time FFT source code	2.3
dct1.asm	56000	Discrete cosine transform using FFT source code	5.5
dct1.hlp	56000	dct1.asm help file	1
dhit1.asm	56000	Routine to compute Hilbert transform in the frequency domain source code	1.9
dhit1.hlp	56000	dhit1.asm help file	1
fft2d256.asm	96000	256x256 complex FFT source code	
fft.asm	56000	Radix 2 decimation-in-time 512 point FFT source code	7.4
fftas.asm	56000	Radix 2 in-place decimation-in-time (smallest code size) with automatic scaling at each pass source code	3.8

Table 11-9 Fast Fourier Transforms Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
fftbf.asm	56002	Radix 2 in-place decimation-in-time (smallest code size) with block floating point on the 56002 source code	4.2
fftr2a.asm	56000	Radix 2, in-place, decimation-in-time FFT (smallest) source code	3.4
fftr2a.hlp	56000	fftr2a.asm help file	2.7
fftr2aa.asm	56000	Automatic scaling FFT source code	3.2
fftr2at.asm	56000	FFT test program(fftr2a.asm) source code	1
fftr2at.hlp	56000	fftr2at.asm help file	0.6
fftr2at.bak	56000	fftr2at.asm backup file	1
fftr2at.cld	56000	Automatic scaling **c program load** file	5.7
fftr2at.lst	56000	Automatic scaling list file	51.3
fftr2b.asm	56000	Radix 2, in-place, decimation-in-time FFT (faster) source code	4.3
fftr2bf.hlp	56000	fftr2bf.asm help file	1.6
fftr2b.hlp	56000	fftr2b.asm help file	3.7
fftr2c.asm	56000	Radix 2, in-place, decimation-in-time FFT (even faster) source code	6
fftr2c.hlp	56000	fftr2c.asm help file	3.2
fftr2cc.asm	56000	Radix 2, in-place decimation-in-time complex FFT macro source code	6.5
fftr2cc.hlp	56000	fftr2cc.asm help file	3.5
fftr2cn.asm	56000	Radix 2, decimation-in-time complex FFT macro with normally ordered input/output source code	6.6
fftr2cn.hlp	56000	fftr2cn.asm help file	2.5
fftr2cnt.asm	56000	**FFT to complex number conversion** source code	0.5
fftr2d.asm	56000	Radix 2, in-place, decimation-in-time FFT (using DSP56001 sine-cosine ROM tables) source code	3.7
fftr2d.hlp	56000	fftr2d.asm help file	3.5
fftr2dt.asm	56000	fftr2d.asm test program source code	1.3
fftr2dt.hlp	56000	fftr2dt.asm help file	0.6
fftr2e.asm	56000	1024 point, non-in-place, FFT (3.39ms) source code	9
fftr2e.hlp	56000	fftr2e.asm help file	5
fftr2et.asm	56000	fftr2e.asm test program source code	1
fftr2et.hlp	56000	fftr2et.asm help file	0.4

Table 11-9 Fast Fourier Transforms Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
fftr2en.asm	56000	1024 point, not-in-place, complex FFT macro with normally ordered input/output source code	9.8
fftr2en.hlp	56000	fftr2en.asm help file	4.9
fftr2bf.asm	56000	Radix-2, decimation-in-time FFT with block floating point source code	13.5
fftr2fn.asm	56000	Port to memory FFT - 1024 point source code	10.4
fftr2fnt.asm	56000	Test file source code for fftr2fn.asm	0.8
gen56.asm	56001	Input signal generator for FFT on 56001 source code	0.9
norm2ber.asm	96000	Convert normal order to Berlang order source code	0.5
rfft	96002	Bergland order table generator source code	22.1
rfft56t.asm	56000	Non-in-place FFT - 1024 point source code	11.7
rfft96b.asm	96002	Real-valued FFT source code	9.3
rfft96bt.asm	96002	Real input FFT source code	2.9
rfft96t.asm	96002	Test program source code	2.5
sincos.asm	56000	Sine-cosine table generator for FFTs source code	1.2
sincosf.asm	96002	Sine-cosine table generator source code	1.3
sincosr.asm	56000	Sine-cosine table generator for rfft56.asm source code	1.5
sincos.hlp	56000	sincos.asm help file	0.9
sinewave.asm	56000	Full-cycle sine wave table generator macro source code	1
sinewave.hlp	56000	sinewave.asm help file	1.4
split56.asm	56000	Amplifies coefficients of FFT by two source code	3.6
split96.asm	96002	Split N/2 complex FFT (hn) for N real FFT (Fn) source code	2.8

11.6.9 Filters

The programs in the following table include various Infinite Impulse Response (IIR), Finite Impulse Response (FIR), and lattice filter programs.

Note: See application note APR7/D *Implementing IIR/FIR Filters with Motorola's DSP56000/DSP56001* for more information.

Table 11-10 Filters Available on the WWW

Document ID	DSP	Description	Size (K)
fir.asm	56000	Direct form FIR filter source code	0.5
fir.hlp	56000	fir.asm help file	2.2
firt.asm	56000	fir.asm test program source code	1.2
iir1.asm	56000	Direct form second order all pole IIR filter source code	0.7
iir1.hlp	56000	iir1.asm help file	1.8
iir1t.asm	56000	iir1.asm test program source code	1.2
iir2.asm	56000	Direct form second order all pole IIR filter with scaling source code	0.8
iir2.hlp	56000	iir2.asm help file	2.3
iir2t.asm	56000	iir2.asm test program source code	1.3
iir3.asm	56000	Direct form arbitrary order all pole IIR filter source code	0.8
iir3.hlp	56000	iir3.asm help file	2.7
iir3t.asm	56000	iir3.asm test program source code	1.3
iir4.asm	56000	Second order direct canonic IIR filter (biquad IIR filter) source code	0.7
iir4.hlp	56000	iir4.asm help file	2.3
iir4t.asm	56000	iir4.asm test program source code	1.2
iir5.asm	56000	Second order direct canonic IIR filter with scaling (biquad IIR filter) source code	0.8
iir5.hlp	56000	iir5.asm help file	2.8
iir5t.asm	56000	iir5.asm test program source code	1.3
iir6.asm	56000	Arbitrary order direct canonic IIR filter source code	0.9
iir6.hlp	56000	iir6.asm help file	3
iir6t.asm	56000	iir6.asm test program source code	1.4
iir7.asm	56000	Cascaded biquad IIR filters source code	0.9
iir7.hlp	56000	Help for iir7.asm	3.9
iir7t.asm	56000	iir7.asm test program source code	1.4
latfir1.asm	56000	Lattice FIR filter macro source code	1.2
latfir1.hlp	56000	latfir1.asm help file	6.3
latfir1t.asm	56000	latfir1.asm test program source code	1.4

Table 11-10 Filters Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
latfir2.asm	56000	Lattice FIR filter macro (modified modulo count) source code	1.2
latfir2.hlp	56000	latfir2.asm help file	1.3
latfir2t.asm	56000	latfir2.asm test program source code	1.4
latgen.asm	56000	Generalized lattice FIR/IIR filter macro source code	1.3
latgen.hlp	56000	latgen.asm help file	5.5
latgent.asm	56000	latgen.asm test program source code	1.3
latiir.asm	56000	Lattice IIR filter macro source code	1.3
latiir.hlp	56000	latiir.asm help file	6.4
latiirt.asm	56000	latiir.asm test program source code	1.4
latnrm.asm	56000	Normalized lattice IIR filter macro source code	1.4
latnrm.hlp	56000	latnrm.asm help file	7.5
latnrmt.asm	56000	latnrm.asm test program source code	1.6
lms.hlp	56000	LMS Adaptive filter algorithm help file	5.8
p1	56200	Support software description	6.3
p2	56200	Adaptive filter interrupt driver flowchart	10.9
p3	56200	Adaptive filter interrupt driver program example	25.8
p4	56200	Polled I/O flowchart	10.4
p5	56200	Polled I/O program example	24.8
p6	56200	Dual FIR filter interrupt driver flowchart	9.5
p7	56200	Dual FIR filter interrupt driver program example	28.5
p8	56200	Dual FIR filter polled I/O, flowchart	9.7
p9	56200	Dual FIR filter polled I/O program example	28.5
transiir.asm	56000	Implements the transposed IIR filter source code	2
transiir.hlp	56000	transiir.asm help file	1

11.6.10 Floating Point Routines

The programs in the following table are miscellaneous floating-point algorithms for the DSP56000 and DSP56300 families.

Table 11-11 Floating Point Routines Available on the WWW

Document ID	DSP	Description	Size (K)
durbin.asm	56000	Solution for LPC coefficients source code	5.6
durbin.hlp	56000	durbin.asm help file	2.9
float.sha	56000	Floating point routines shell archive	86.5
fpabs.asm	56000	Floating point absolute value source code	2
fpadd.asm	56000	Floating point add source code	3.9
fpcalls.hlp	56000	Subroutine calling conventions help file	11.9
fpceil.asm	56000	Floating point CEIL subroutine source code	1.8
fpcmp.asm	56000	Floating point compare source code	2.6
fpdef.hlp	56000	Storage format and arithmetic representation definition help file	10.6
fpdiv.asm	56000	Floating point divide source code	3.8
fpfix.asm	56000	Floating to fixed point conversion source code	4
fpfloat.asm	56000	Fixed to floating point conversion source code	2
fpfloor.asm	56000	Floating point FLOOR subroutine source code	2.1
fpfrac.asm	56000	Floating point FRACTION subroutine source code	1.9
fpinit.asm	56000	Library initialization subroutine source code	2.3
fplist.asm	56000	Test file that lists all subroutines (source code)	1.6
fpmac.asm	56000	Floating point multiply-accumulate source code	2.7
fpmpy.asm	56000	Floating point multiply source code	2.3
fpneg.asm	56000	Floating point negate source code	2
fprevs.hlp	56000	Latest revisions of floating-point library help file	1.8
fpscale.asm	56000	Floating point scaling source code	2.1
fpsqrt.asm	56000	Floating point square root source code	2.9
fpsub.asm	56000	Floating point subtract source code	3.1

11.6.11 Functions

The programs in the following table are standard mathematical functions.

Table 11-12 Functions Available on the WWW

Document ID	DSP	Description	Size (K)
exp2.asm	56000	Exponential base 2 by polynomial approximation source code	0.9
exp2.asm	56000	Exponential base 2 by polynomial approximation source code	0.9
exp2.hlp	56000	exp2.asm help file	0.8
exp2t.asm	56000	exp2.asm test program source code	1
log2.asm	56000	Log base 2 by polynomial approximation source code	1.1
log2.hlp	56000	Help for log2.asm	0.7
log2nrm.asm	56000	Normalizing base 2 logarithm macro source code	2.2
log2nrm.hlp	56000	log2nrm.asm help file	0.7
log2nrmt.asm	56000	log2nrm.asm test program source code	1.1
log2t.asm	56000	log2.asm test program source code	1
rand1.asm	56000	Pseudo random sequence generator source code	2.4
rand1.hlp	56000	rand1.asm help file	0.7
sqrt1.asm	56000	Square Root by polynomial approximation, 7 bit accuracy source code	1
sqrt1.hlp	56000	sqrt1.asm help file	0.8
sqrt1t.asm	56000	sqrt1.asm test program source code	1.1
sqrt2.asm	56000	Square root by polynomial approximation, 10-bit accuracy source code	0.9
sqrt2.hlp	56000	sqrt2.asm help file	0.8
sqrt2t.asm	56000	sqrt2.asm test program source code	1
sqrt3.asm	56000	Full precision square root macro source code	1.4
sqrt3.hlp	56000	sqrt3.asm help file	0.8
sqrt3t.asm	56000	sqrt3.asm test program source code	1
tli.asm	56000	Linear table lookup / interpolation routine for function generation source code	3.2
tli.hlp	56000	tli.asm help file	1.5

11.6.12 Matrix Operations

The programs in the following table perform matrix operations.

Table 11-13 Matrix Operations Available on the WWW

Document ID	DSP	Description	Size (K)
8-56.asm	56000 56001	[2x2] [2x2] matrix multiply source code	3.6
9-56.asm	56000 56001	[3x3] [3x3] matrix multiply source code	3.3
b121.asm	96002	[1x3] [[3x3] and [1x4] [4x4] matrix multiply source code	3.7
b122.asm	96002	[1x3] [[3x3] and [nxn] [nxn] matrix multiply source code	4.3
matmul1.asm	56000	[1x3][3x3]=[1x3] matrix multiplication source code	1.8
matmul1.hlp	56000	matmul1.asm help file	0.5
matmul2.asm	56000	General matrix multiplication, C=AB source code	2.7
matmul2.hlp	56000	matmul2.asm help file	0.8
matmul3.asm	56000	General matrix multiply-accumulate, C=AB+Q source code	2.8
matmul3.hlp	56000	matmul3.asm help file	0.9

11.6.13 Multiply and Accumulate (MAC)

The programs in the following table perform double-precision multiply -and-accumulate operations.

Table 11-14 Multiply and Accumulate Operations Available on the WWW

Document ID	DSP	Description	Size (K)
dmac.asm	56000	Double precision multiplication and accumulate source code	2.9
sdm.asm	56000	Single x double multiplication using DP mode of 56K core source code	1.1
sdmac.asm	56002	Single x double MAC using DP mode of 56002 source code	1.3

11.6.14 Sorting Routines

Table 11-15 Sorting Routines Available on the WWW

Document ID	DSP	Description	Size (K)
sort1.asm	56001	Array sort by straight selection source code	1.3
sort1.hlp	56001	sort1.asm help file	1.9
sort1t.asm	56001	sort1.asm test program source code	0.7
sort2.asm	56001	Array sort by Heapsort method source code	2.2
sort2.hlp	56001	sort2.asm help file	2
sort2t.asm	56001	sort2.asm test program source code	0.7

11.6.15 Speech

Table 11-16 Speech Available on the WWW

Document ID	DSP	Description	Size (K)
adpcm.asm	56001	32 kbps CCITT ADPCM speech coder source code	12.1
adpcm.hlp	56001	adpcm.asm help file	14.8
adpcm.uue	56001	Binary to text UNIX-to-UNIX (UU) encode file	0.4
adpcmns.asm	56001	Nonstandard ADPCM source code	54.7
adpcmns.hlp	56001	adpcmns.asm help file	10
durbin1.asm	56001	Durbin Solution for PARCOR (LPC) coefficients source code	6.4
durbin1.hlp	56001	durbin1.asm help file	3.6
lgsol1.asm	56001	Leroux-Gueguen solution for PARCOR (LPC) coefficients source code	4.9
lgsol1.hlp	56001	lgsol1.asm help file	4

11.6.16 Standard I/O Equates

The programs in the following table are useful when writing standardized assembly code.

Table 11-17 Standard I/O Equates Available on the WWW

Document ID	DSP	Description	Size (K)
intequ.asm	56000	Standard interrupt equate file source code	1.1
intequlc.asm	56000	Lower case version of intequ.asm source code	1.1
ioequ.asm	56000	Motorola standard I/O equate file source code	8.8
ioequlc.asm	56000	Lower case version of ioequ.asm source code	8.9

11.6.17 Tools and Utilities

Note: The program dos4gw.exe solves memory problems when running Clasx and ADS software.

Table 11-18 Tools and Utilities Available on the WWW

Document ID	DSP	Description	Size (K)
cldlod.hqx	56000	**Convert c program load file (.cld) to load file (.lod) file, Macintosh compressed format. Includes source, executable, and makefile.**	37
cldlod.nxt	56000	**Convert c program load file (.cld) to load file (.lod) file, Next compressed format. Includes source, executable, and makefile.**	41
cldlod.rea	56000	**Read me text file.**	0.5
cldlod.sn3	56000	**Convert c program load file (.cld) to load file (.lod) file, Sun 3 compressed format. Includes source, executable, and makefile.**	114.7
cldlod.sn4	56000	**Convert c program load file (.cld) to load file (.lod) file, Sun compressed format. Includes source, executable, and makefile.**	131.1
cldlod.zip	56000	**Convert c program load file (.cld) to load file (.lod) file, PC compressed format. Includes source, executable, and makefile.**	25.4
dos4gw.exe	56000	**Convert c program load file (.cld) to load file (.lod) file, Macintosh compressed format.**	231.2
dspbug	56000	Ordering information for free debug monitor for DSP56000/DSP56001	882
parity.asm	56000	Parity calculation of a 24-bit number in accumulator A source code	1641

Table 11-18 Tools and Utilities Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
master2.asm	96000	Multi-device simulator source code	9.8
parity.hlp	56000	parity.asm help file	936
parityt.asm	56000	parity.asm test program source code	685
parityt.hlp	56000	parityt.asm help file	259
read.me	96000	Multi-device simulation read me text file	0.3
Slave2.asm	96000	Multi-device simulation source code	
sloader.asm	56000	Serial loader from the SCI port for the DSP56001 source code	3986
sloader.hlp	56000	sloader.asm help file	2598
sloader.p	56000	Serial loader s-record file for download to EPROM source code	736
srec.c	56000	Utility to convert DSP56000 OMF format to SREC (source code)	38975
srec.doc	56000	Manual page for srec.c.	7951
srec.h	56000	srec.c include file	3472
srec.exe	56000	IBM PC srec executable	22065

11.6.18 Viterbi

Viterbi is a Reed-Solomon decoder of Trellis encoding.

Note: See the application note APR6/D *Convolutional Encoding and Viterbi Decoding Using the DSP56001 with a V.32 Modem Trellis Example* for more information.

Table 11-19 Viterbi Routines Available on the WWW

Document ID	DSP	Description	Size (K)
bound.d	56000	Data file	1.3
decode.asm	56000	Viterbi decoder for V.32 source code file	10.7
encode.asm	56000	Convolutional decoder for V.32 source code file	0.9
read.me	56000	Usage instructions text file	

11.7 REFERENCE BOOKS AND MANUALS

A list of DSP-related books is included here as an aid for the engineer who is new to the field of DSP. This is a partial list of DSP references intended to help the new user find useful information in some of the many areas of DSP applications. Many of the books could be included in several categories, but are not repeated.

11.7.1 General DSP

Bellanger, Maurice. *Digital Processing Of Signals Theory And Practice.* New York, NY: John Wiley and Sons, 1984.

Cadzow, J. A. *Foundations Of Digital Signal Processing And Data Analysis.* New York, NY: MacMillan Publishing Company, 1987.

Candy, James V. *Signal Processing – The Modern Approach.* New York, NY: McGraw-Hill Company, Inc., 1988.

Chen, C.H. *Signal Processing Handbook.* New York, NY: Marcel Dekker, Inc., 1988.

Crochiere, R. E., and Rabiner, L. R. *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

DeFatta, David J., Lucas, Joseph G., and Hodgkiss, William S. *Digital Signal Processing: A System Design Approach.* New York, NY: John Wiley and Sons, 1988.

Elliott, D. F. *Handbook Of Digital Signal Processing*. San Diego, CA: Academic Press, Inc., 1987.

Lim, Jae S., and Oppenheim, Alan V. *Advanced Topics In Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Oppenheim, A. V. *Applications Of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Oppenheim, A. V., and Schafer, R.W. *Discrete-time Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989.

Oppenheim, Alan V., and Schafer, Ronald W. *Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

Proakis, John G., and Manolakis, Dimitris G. *Introduction To Digital Signal Processing.* New York, NY: Macmillan Publishing Company, 1988.

Stearns, S., and Davis, R. *Signal Processing Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Rabiner, Lawrence R., Gold, and Bernard. *Theory And Application Of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

11.7.2 Digital Audio and Filters

Antoniou, Andreas. *Digital Filters: Analysis And Design*. New York, NY: McGraw-Hill Company, Inc., 1979.

Chamberlin, H. *Musical Applications Of Microprocessors (Second Edition)*. Hasbrouck Heights, NJ: Hayden Book Co., 1985.

Haykin, Simon. *Introduction To Adaptive Filters*. New York, NY: MacMillan Publishing Company, 1984.

Jackson, Leland B. *Digital Filters And Signal Processing*. Higham, MA: Kluwer Academic Publishers, 1986.

Jayant, N. S., and Noll, Peter. *Digital Coding Of Waveforms*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Kuc, Roman. *Introduction To Digital Signal Processing*. New York, NY: McGraw-Hill Company, Inc., 1988.

Mulgrew, B., and Cowan, C. *Adaptive Filter And Equalizers*. Higham, MA: Kluwer Academic Publishers, 1988.

Roberts, Richard A., and Mullis, Clifford T. *Digital Signal Processing*. New York, NY: Addison-Welsey Publishing Company, Inc., 1987.

Strawn, John. *Digital Audio Signal Processing An Anthology*. William Kaufmann, Inc., 1985.

Watkinson, John. *The Art Of Digital Audio*. Stoneham. MA: Focal Press, 1988.

Widrow, B., and Stearns, S. D. *Adaptive Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

Williams, Charles S. *Designing Digital Filters*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986.

11.7.3 C Programming Language

American National Standards Institute. *Programming Language - C*. ANSI Document X3.159-1989. American National Standards Institute, inc., 1990.

Harbison, Samuel P., and Steele, Guy L. *C: A Reference Manual*. Prentice-Hall Software Series, 1987.

Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language*. Prentice-Hall, Inc., 1978.

11.7.4 Controls

Astrom, K., and Wittenmark, B. *Adaptive Control*. New York, NY: Addison-Welsey Publishing Company, Inc., 1989.

Astrom, K., and Wittenmark, B. *Computer Controlled Systems: Theory & Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Goodwin, G., and Sin, K. *Adaptive Filtering Prediction & Control*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Kuo, B. C. *Automatic Control Systems*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

Kuo, B. C. *Digital Control Systems*. New York, NY: Holt, Reinholt, and Winston, Inc., 1980.

Moroney, P. *Issues In The Implementation Of Digital Feedback Compensators*. Cambridge, MA: The MIT Press, 1983.

Phillips, C., and Nagle, H. *Digital Control System Analysis & Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

11.7.5 Graphics

Arnold, D. B., and Bono, P. R. *CGM And CGI*. New York, NY: Springer-Verlag, 1988.

Artwick, Bruce A. *Microcomputer Displays, Graphics, And Animation*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

Bono, P. R., and Herman, I. (Eds.). *GKS Theory And Practice*. New York, NY: Springer-Verlag, 1987.

Foley, J. D., and, Van Dam, A. *Fundamentals Of Interactive Computer Graphics*. Reading MA: Addison-Wesley Publishing Company Inc., 1984.

Hall, Roy. *Illumination And Color In Computer Generated Imagery*. New York, NY: Springer-Verlag.

Hearn, D. and Baker, M. Pauline. *Computer Graphics (Second Edition)*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986.

Morteson, Michael E. *Geometric Modeling*. New York, NY: John Wiley and Sons, Inc.

Newman, William M., and Sproull, Roger F. *Principles Of Interactive Computer Graphics*. New York, NY: McGraw-Hill Company, Inc., 1979.

Pixar. *The Renderman Interface*. San Rafael, CA. 94901.

Reid, Glenn C. (Adobe Systems, Inc.). *Postscript Language Program Design*. Reading MA: Addison-Wesley Publishing Company, Inc., 1988.

Rogers, David F. *Procedural Elements For Computer Graphics*. New York, NY: McGraw-Hill Company, Inc., 1985.

11.7.6 Image Processing

Barnsley, M. F., Devaney, R. L., Mandelbrot, B. B., Peitgen, H. O.,. Saupe, D., and Voss, R. F. *The Science Of Fractal Images*. New York, NY: Springer-Verlag.

Ekstrom, M. P. *Digital Image Processing Techniques*. New York, NY: Academic Press, Inc., 1984.

Gonzales, Rafael C., and Wintz, Paul. *Digital Image Processing (Second Edition)*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1977.

Pratt, William K. *Digital Image Processing*. New York, NY: John Wiley and Sons, 1978.

Rosenfeld, Azriel, and Kak, Avinash C. *Digital Picture Processing*. New York, NY: Academic Press, Inc., 1982.

11.7.7 Motorola DSP Manuals

Motorola. *DSP56000 Linker/librarian Reference Manual*. Motorola, Inc., 1991.

Motorola. *DSP56000 Macro Assembler Reference Manual*. Motorola, Inc., 1991.

Motorola. *DSP56000 Simulator Reference Manual*. Motorola, Inc., 1991.

Motorola. *DSP56000/DSP56001 User's Manual*. Motorola, Inc.,1990.

11.7.8 Numerical Methods

Berliout, P., and Bizard, P. *Algorithms (The Construction, Proof, And Analysis Of Programs)*. New York, NY: John Wiley and Sons, 1986.

Golub, G. H., and Van Loan, C. F. *Matrix Computations*. John Hopkins Press, 1983.

Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. *Numerical Recipes In C - The Art Of Scientific Programming*. Cambridge University Press, 1988.

Schroeder, Manfred R. *Number Theory In Science And Communication*. New York, NY: Springer-Verlag, 1986.

11.7.9 Pattern Recognition

Bracewell, R. N. *The Fast Fourier Transform And Its Applications*. New York, NY: McGraw-Hill Company, Inc., 1986.

Brigham, E. Oran. *The Fast Fourier Transform And Its Applications*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Duda, R. O., and Hart, P. E. *Pattern Classification And Scene Analysis*. New York, NY: John Wiley and Sons, 1973.

Gardner, William A. *Statistical Spectral Analysis, A Nonprobabilistic Theory*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

James, Mike. *Classification Algorithms*. New York, NY: Wiley-Interscience, 1985. Spectral Analysis:

11.7.10 Speech

Flanagan, J. L. *Speech Analysis, Synthesis, And Perception*. New York, NY: Springer-Verlag, 1972.

Honig, Michael L., and Messerschmitt, David G. *Adaptive Filters – Structures, Algorithms, And Applications*. Higham, MA: Kluwer Academic Publishers, 1984.

Jayant, N. S., and Noll, P. *Digital Coding Of Waveforms*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Markel, J. D. and Gray, A. H., Jr. *Linear Prediction Of Speech*. New York, NY: Springer-Verlag, 1976.

O'Shaughnessy, D. *Speech Communication – Human And Machine*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1987.

Rabiner, Lawrence R., and Schafer, R. W. *Digital Processing Of Speech Signals*. Englwood Cliffs, NJ: Prentice-Hall, Inc., 1978.

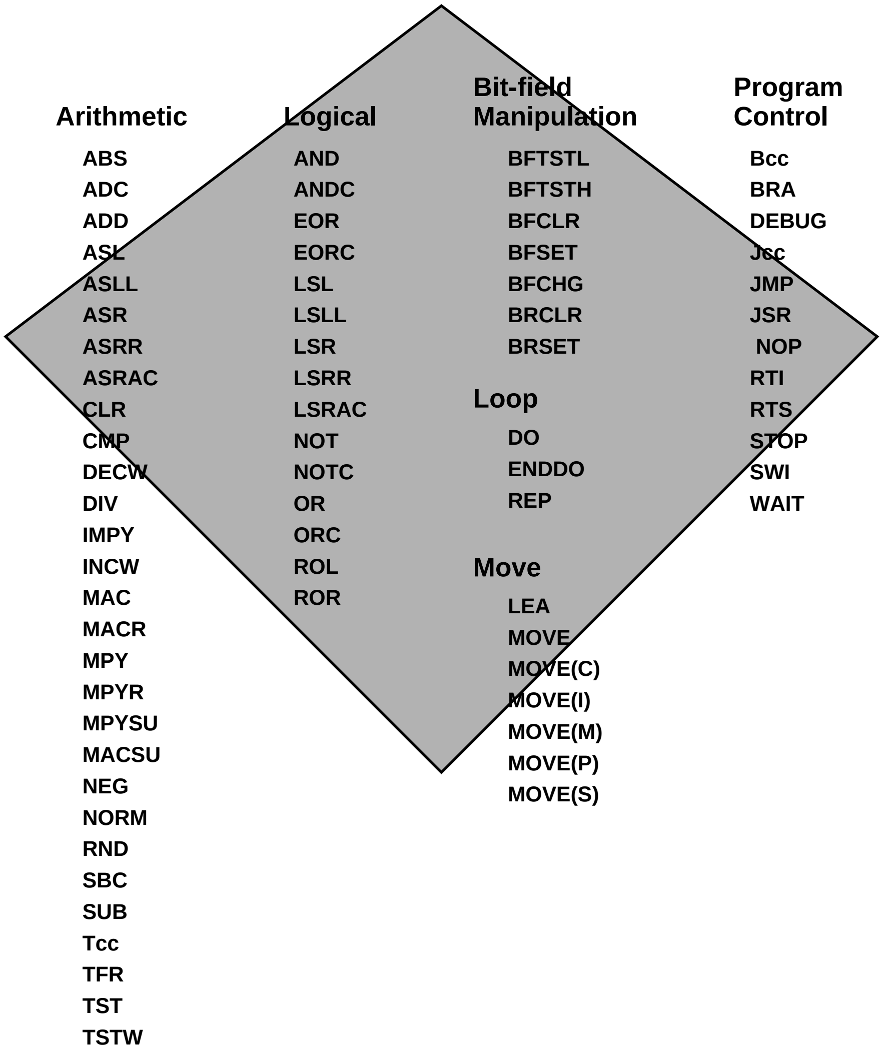
11.7.11 Telecommunications

Lee, Edward A., and Messerschmitt, David G. *Digital Communication*. Higham, MA: Kluwer Academic Publishers, 1988.

Proakis, John G. *Digital Communications*. New York, NY: McGraw-Hill Publishing Co., 1983.

APPENDIX A

INSTRUCTION SET DETAILS



Arithmetic

ABS
ADC
ADD
ASL
ASLL
ASR
ASRR
ASRAC
CLR
CMP
DECW
DIV
IMPY
INCW
MAC
MACR
MPY
MPYR
MPYSU
MACSU
NEG
NORM
RND
SBC
SUB
Tcc
TFR
TST
TSTW

Logical

AND
ANDC
EOR
EORC
LSL
LSLL
LSR
LSRR
LSRAC
NOT
NOTC
OR
ORC
ROL
ROR

Bit-field Manipulation

BFTSTL
BFTSTH
BFCLR
BFSET
BFCHG
BRCLR
BRSET

Loop

DO
ENDDO
REP

Move

LEA
MOVE
MOVE(C)
MOVE(I)
MOVE(M)
MOVE(P)
MOVE(S)

Program Control

Bcc
BRA
DEBUG
Jcc
JMP
JSR
NOP
RTI
RTS
STOP
SWI
WAIT

A.1 INTRODUCTION

This appendix contains detailed information about each instruction of the DSP56800 instruction set. It contains sections on notation, addressing modes, and condition codes. Also included is a section on instruction timing, which shows the number of program words and execution time of each instruction. Finally, the instruction set summary, which shows the syntax of all allowed DSP56800 instructions, is presented.

A.2 NOTATION

Each instruction description contains notations used to abbreviate certain operands and operations. The symbols and their respective descriptions are listed in **Table A-1** through **Table A-12**.

Table A-1 Data ALU Register Operands

Symbol	Description
X0	Input register X0 (16 bits)
Yn	Input register Y1 or Y0 (16 bits)
Y	Input register Y = Y1:Y0 (32 bits)
An	Accumulator registers A2 (4 bits), A1 (16 bits), or A0 (16 bits)
Bn	Accumulator registers B2 (4 bits), B1 (16 bits), or B0 (16 bits)
A	Accumulator A = A2:A1:A0 (36 bits)[1]
B	Accumulator B = B2:B1:B0 (36 bits)[1]
F	A or B accumulator
F1	MSP of an accumulator (A1 or B1)
DD	X0, Y0, or Y1 register
Note: 1. When this register is specified as a source operand in data move operands, limiting is performed where appropriate. When specified as a destination operand, sign extension and possibly zeroing are performed.	

Table A-2 Address ALU Register Operands

Symbol	Description
Rn	Address registers R0-R3 (16 bits each)
N	Address offset register N (16 bits)
M01	Address modifier register M01 (16 bits)

Table A-3 Program Control Registers

Symbol	Description
PC	Program counter register (16 bits)
MR	Mode register (8 bits)
CCR	Condition code register (8 bits)
SR	Status register, MR:CCR (16 bits)
OMR	Operating mode register (16 bits)
LA	Hardware loop address register (16 bits)
LC	Hardware loop counter register (13 bits)
HWS	Current top of the hardware DO loop stack (16 bits)

Table A-4 Address Operands

Symbol	Description
ea	Effective address
eax	Effective address for X bus
xxxx	Absolute address (16 bits)
xxxxx	Absolute address (19 bits)
pp	I/O short address (6 bits, one-extended)
aa	Absolute address (6 bits, zero-extended)
<...>	Specifies the contents of the specified address
X:	X memory reference
P:	Program memory reference

Table A-5 Miscellaneous Operands

Symbol	Description
S, Sn	Source operand register
D, Dn	Destination operand register
#xx	Immediate short data (7 bits for MOVE(I), 6 bits for DO / REP)
#xxxx	Immediate data (16 bits)
#iiii	16-bit immediate data mask
#ii00	8-bit immediate data mask in the upper byte
#00ii	8-bit immediate data mask in the lower byte

Table A-6 Unary Operators

Symbol	Description
$	Specifies that a value is represented in hexadecimal
~	Negation operator
PUSH	Push specified value onto the software stack operator
POP	Pull specified value from the software stack operator
READ	Read the top of the system stack (SS) operator
PURGE	Delete the top value on the hardware stack (HWS) operator
| |	Absolute value operator

Table A-7 Binary Operators

Symbol	Description
+	Addition operator
-	Subtraction operator
*	Multiplication operator
÷, /	Division operator
$\overline{X}$	“Logical complement of” operator
+	Logical inclusive OR operator
•	Logical AND operator
⊕	Logical exclusive OR operator
→	“Is transferred to” operator
↔	“Is transferred to or from” operator
:	Concatenation operator

Table A-8 Addressing Mode Operators

Symbol	Description
<<	I/O short or absolute short addressing mode force operator
>	Long addressing mode force operator
#	Immediate addressing mode operator
#>	Immediate long addressing mode force operator
#<	Immediate short addressing mode force operator

Table A-9 Status Register and Operating Mode Register Symbols

Symbol	Description
IM	Interrupt mask bit indicating the current interrupt priority level
LF	Loop flag bit indicating when a DO loop is in progress
NL	Nested loop flag bit indicating when a nested DO loop is in progress

Table A-10 Condition Code Register Symbols (Standard Definitions)

Symbol	Description
L	Limit bit indicating arithmetic overflow and/or data limiting
E	Extension bit indicating if the integer portion is in use
U	Unnormalized bit indicating if the result is unnormalized
N	Negative bit indicating if bit 35 (or 31) of the result is set
Z	Zero bit indicating if the result equals zero
V	Overflow bit indicating if arithmetic overflow has occurred in the result
C	Carry bit indicating if a carry or borrow occurred in the result

Table A-11 Instruction Timing Symbols

Symbol	Description
aio	Time required to access an I/O operand
ap	Time required to access a P memory operand
ax	Time required to access an X memory operand
axx	Time required to access X memory operands for double read
ea	Time or number of words required for an effective address
jx	Time required to execute part of a jump-type instruction
mv	Time or number of words required for a move-type operation

Table A-11 Instruction Timing Symbols (Continued)

Symbol	Description
mvb	Time required to execute part of a bit manipulation instruction
mvc	Time required to execute part of a MOVEC instruction
mvm	Time required to execute part of a MOVEM instruction
mvp	Time required to execute part of a MOVEP instruction
mvs	Time required to execute part of a MOVES instruction
rx	Time required to execute part of an RTS instruction
wp	Number of wait states used in accessing external P memory
wx	Number of wait states used in accessing external X memory

Table A-12 Other Symbols

Symbol	Description
( )	Optional letter, operand, or operation[1]
(...)	Any arithmetic or logical instruction which allows parallel moves
EXT	Extension register portion of an accumulator (A2 or B2)
LSB	Least significant bit
LSP	Least significant portion of an accumulator (A0 or B0)
LSW	Least significant word
MSB	Most significant bit
MSP	Most significant portion of an accumulator (A1 or B1)
MSW	Most significant word
r	Rounding constant
LIM	Limiting when reading a data ALU accumulator
<op>	Generic instruction (specifically defined within each section)
Note: 1.	For instructions that contain parentheses in the instructions name, such as DEC(W) or IMPY(16), the portion within the parenthesis is optional.

A.3 PROGRAMMING MODEL

The registers in the DSP56800 core programming model are shown in **Figure A-1**.

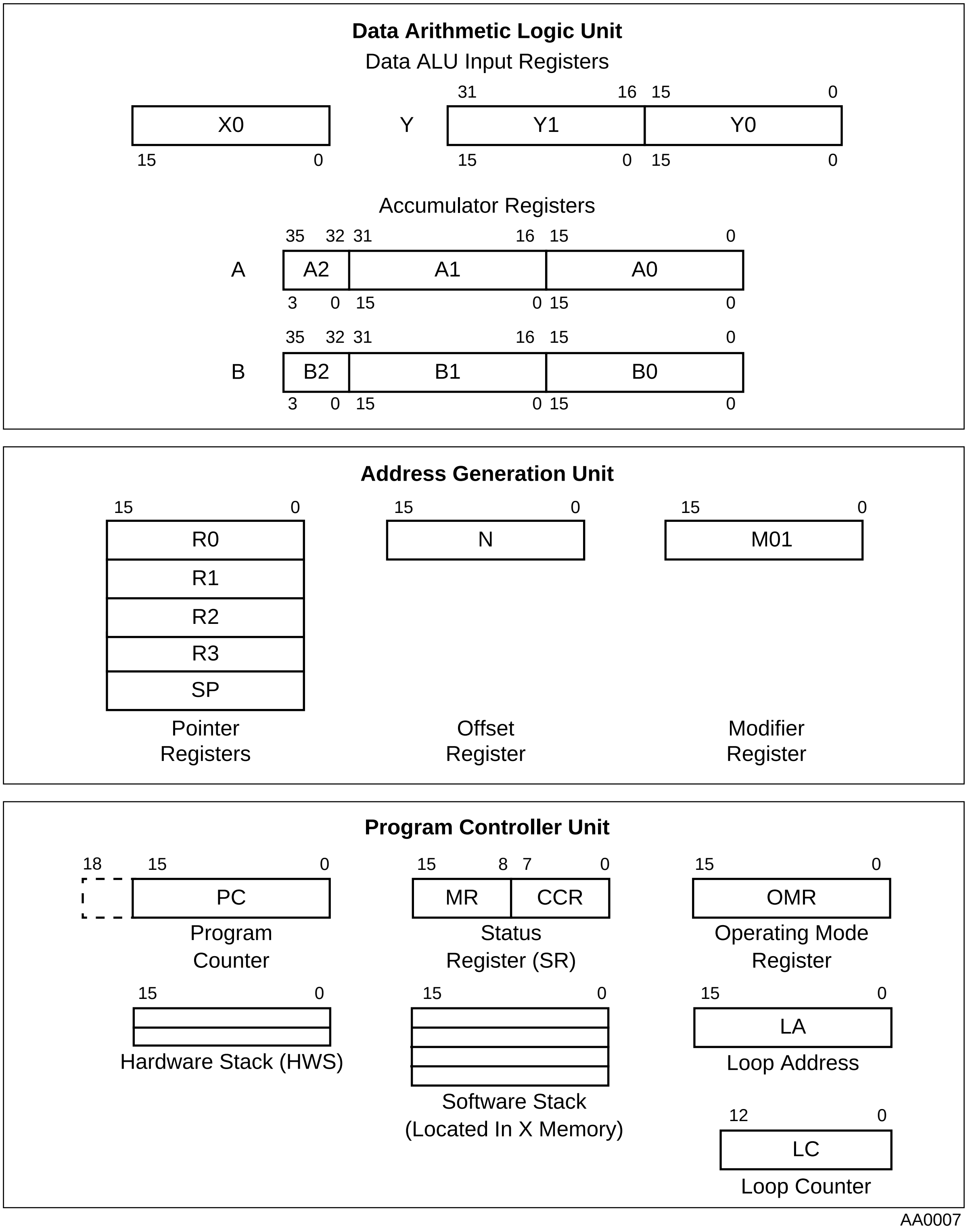


Figure A-1 DSP56800 Core Programming Model

A.4 ADDRESSING MODES

The addressing modes are grouped into three categories:

- Register direct—directly references the registers on the chip
- Address register indirect—uses an address register as a pointer to reference a location in memory
- Special—includes direct addressing, extended addressing, and immediate data

These addressing modes are described below and summarized in **Table 4-1 Address Register Indirect Addressing Modes Available** on page 4-11.

All address calculations are performed in the address ALU to minimize execution time and loop overhead. Addressing modes specify whether the operands are in registers, in memory or in the instruction itself (such as immediate data) and provide the specific address of the operands.

The register direct addressing mode can be subclassified according to the specific register addressed. The data registers include X0, Y1, Y0, Y, A2, A1, A0, B2, B1, B0, A and B. The control registers include HWS, LA, LC, OMR, SR, CCR and MR. The address registers include R0, R1, R2, R3, SP, N, and M01.

Address register indirect modes use an address register Rn (R0-R3) or the stack pointer (SP) to point to locations in X and P memory. The contents of the Rn is the effective address (ea) of the specified operand, except in the indexed by offset or indexed by displacement mode, where the effective address (ea) is (Rn+Nn) or (Rn+xxxx), respectively. Address register indirect modes use an address modifier register M01 to specify the type of arithmetic to be used to update the address register R0 and optionally R1. R2 and R3 always use linear arithmetic. If an addressing mode specifies the address offset register (N), it is used to update the corresponding Rn. This unique implementation is extremely powerful and allows the user to easily address a wide variety of DSP-oriented data structures. All address register indirect modes use at least one Rn and sometimes N and the modifier register (M01), and the double X memory read uses two address registers, one for the first X memory read and one for the second X memory read. Only R3 can be used for this second X memory read, and R3 is always updated using linear arithmetic.

The special addressing modes include immediate and absolute addressing modes as well as implied references to the program counter (PC), the software stack, the hardware stack (HWS) and the program (P) memory.

The addressing mode selected in the instruction word is further specified by the contents of the address modifier register M01. The modifier selects whether linear or modulo arithmetic is performed. The programming of this register is summarized in **Table 4-3 Addressing Mode Modifier Summary** on page 4-34.

A.5 CONDITION CODE COMPUTATION

The condition code portion of the status register consists of 7 defined bits:

- L—Limit
- E—Extension
- U—Unnormalized
- N—Negative
- Z—Zero
- V—Overflow
- C—Carry

The C,V,Z,N,U, and E bits are true condition code bits that reflect the condition of the result of a data ALU operation. These condition code bits are not affected by address ALU calculations or by data transfers over the CGDB. The N, Z, and V condition code bits are updated by the TSTW instruction, which can operate on both memory and registers. The L bit is a latching overflow bit which indicates that an overflow has occurred in the data ALU or that limiting has occurred when reading a data ALU register. This limiting occurs as the result of a data bus move operation with limiting accumulator data through the data limiters.

The computation of the C, V, N, and Z condition code bits is dependent on the OMR's CC bit. This bit establishes how the condition codes of an arithmetic or logic operation are interpreted, whether the result is interpreted as a 32-bit result with no extension (CC equals 1), or as a 36-bit result with extension (CC equals 0). The former interpretation is required when processing unsigned values, whereas the later can only be for signed values. This OMR bit must be set to 1 whenever the HI, HS, LO, or LS branch/jump conditions are used. The L, E, and U bits are computed the same regardless of the value of the CC bit.

Table A-13 gives the standard definitions of the condition codes. Exceptions to these are given in **Table A-15** on page A-14.

Table A-13 Condition Code Summary

Code	Description
L—Limit	L is set if the overflow bit V is set or if the data limiter has performed a limiting operation. L is not affected otherwise. The L bit is latched once it is set. The L bit is cleared only by the processor reset or an instruction that explicitly clears it. The L bit is affected by data movement operations which read the accumulator registers.
E—Extension	E is cleared if all the bits of the integer portion of the result are the same; that is, the bit patterns 00...00 or 11...11. E is set otherwise. E is computed using of the following bits: 35, 34, 33, 32, and 31. If E is cleared, then the MSP and LSP of the result contain all the significant bits; the high order integer portion is just sign extension. In this case the accumulator extension register can be ignored.
U—Unnormalized	U is set if the two MSBs of the MSP of the result are the same. U is cleared otherwise. U is computed as follows: $U = \sim(\text{Bit } 31 \oplus \text{Bit } 30)$
N—Negative	N is set if the MSB 35 of the result is set (bit 31 if the CC bit is set or if the destination is 16 bits). N is cleared otherwise.
Z—Zero	Z is set if the 36-bit result is zero (32-bit result if CC is set or 16-bit result if destination is 16 bits). Z is cleared otherwise.
V—Overflow	V is set if an arithmetic overflow occurs in the 36-bit result (32-bit result if the CC bit is set). This indicates that the result is not representable in the destination and the accumulator register has overflowed. Cleared otherwise. In saturation mode, an arithmetic overflow occurs in the 32-bit result. This indicates that the result is not representable in the accumulator register without the extension. The accumulator register has overflowed. V is cleared otherwise.
C—Carry	C is set if a carry is generated out of the most significant bit (MSB) of the result for an addition. Also set if borrow is generated in a subtraction. The carry or borrow is generated out of bit 35 of the result (bit 31 if the CC bit is set). C is cleared otherwise.

When generating condition codes for a 16-bit destination, the CC bit is ignored and condition codes are generated using the 16 bits of the result. Instructions in this category are ADD, AND, ASL, ASR, CMP, DECW, EOR, INCW, MAC, MACR, MPY, MPYR, OR, and SUB.

The condition codes for 16-bit destinations are computed as follows:

- N is set if bit 15 of the result is set.
- Z is set if bits 15-0 of the result are all cleared.
- V is set if overflow has occurred in the 16-bit result.
- C is set if a carry (borrow) has occurred out of bit 15 of the result.

Other instructions only generate results for a 16-bit destination such as the logical instructions. When generating condition codes for this case, the CC bit is ignored and condition codes are generated using the 16 bits of the result. Instructions in this category are LSL, LSR, NOT, ROL, and ROR. The rules for condition code generation are presented for the cases where the destination is a 16-bit register or 16 bits of a 36-bit accumulator.

The condition codes for logical instructions with 16-bit registers as destinations are computed as follows:

- N is set if bit 15 of the corresponding register is set.
- Z is set if bits 15-0 of the corresponding register are all cleared.
- V is always cleared.
- C—(Computation dependent on instruction.)

The condition codes for logical instructions with 36-bit accumulators as destinations are computed as follows:

- N is set if bit 31 of the corresponding accumulator is set.
- Z is set if bits 31-16 of the corresponding accumulator are all cleared.
- V is always cleared.
- C—(Computation dependent on instruction.)

Table A-15 details how each instruction affects the condition codes; **Table A-14** describes the notation used in **Table A-15**.

Table A-14 Condition Code Register Notation

Notation	Description
*	Set according to the standard definition by the result of the operation
-	Not affected by the operation
0	Cleared
1	Set
U	Undefined
?	Set according to the special computation definition by the result of the operation.

The condition code computation shown in **Table A-15** may differ from that defined on the opcode descriptions. This indicates that the standard definition may be used to generate the specific condition code result. For example, the Z flag computation for the CLR instruction is shown as the standard definition while the opcode description indicates that the Z flag is always set. **Table A-15** gives the chip implementation viewpoint while the opcode description give the user viewpoint. (Items in the “Notes” column of **Table A-15** are explained on page A-15.)

Table A-15 Condition Code Computations

Instruction	L	E	U	N	Z	V	C	Notes
ABS	*	*	*	*	*	*	-	
ADC	*	*	*	*	*	*	*	
ADD	*	*	*	*	*	*	*	
AND	*	-	-	?	?	0	-	7, 8
ANDC	-	-	-	-	-	-	?	4
ASL	*	*	*	*	*	?	?	1, 2
ASLL	-	-	-	*	*	-	-	
ASR	*	*	*	*	*	0	?	3
ASRAC	-	-	-	*	*	-	-	
ASRR	-	-	-	*	*	-	-	
Bcc	-	-	-	-	-	-	-	
BFCHG	-	-	-	-	-	-	?	4
BFCLR	-	-	-	-	-	-	?	5
BFSET	-	-	-	-	-	-	?	4
BFTSTH	-	-	-	-	-	-	?	4
BFTSTL	-	-	-	-	-	-	?	5
BRA	-	-	-	-	-	-	-	
BRCLR	-	-	-	-	-	-	?	5
BRSET	-	-	-	-	-	-	?	4
CLR	*	*	*	*	*	0	-	13
CMP	*	*	*	*	*	*	*	
DEBUG	-	-	-	-	-	-	-	
DECW	*	*	*	*	?	*	*	
DIV	*	-	-	-	-	?	?	1,2
DO	*	-	-	-	-	-	-	
ENDDO	-	-	-	-	-	-	-	
EOR	*	-	-	?	?	0	-	7, 8
EORC	-	-	-	-	-	-	?	4
ILLEGAL	-	-	-	-	-	-	-	
IMPY16	*	U	U	*	*	*	-	
INCW	*	*	*	*	?	*	*	
Jcc	-	-	-	-	-	-	-	
JMP	-	-	-	-	-	-	-	
JSR	-	-	-	-	-	-	-	
LEA	-	-	-	-	-	-	-	
LSL	*	-	-	?	?	0	?	7, 8, 9
LSLL	-	-	-	*	*	-	-	
LSR	*	-	-	?	?	0	?	7, 8, 10
LSRAC	-	-	-	*	*	-	-	
LSRR	-	-	-	*	*	-	-	
MAC	*	*	*	*	*	*	-	
MACR	*	*	*	*	*	*	-	
MACSU	*	*	*	*	*	*	-	
MOVE	*	-	-	-	-	-	-	
MOVE(C)	?	?	?	?	?	?	?	12
MOVE(I)	-	-	-	-	-	-	-	
MOVE(M)	*	-	-	-	-	-	-	
MOVE(P)	-	-	-	-	-	-	-	
MOVE(S)	*	-	-	-	-	-	-	
MPY	*	*	*	*	*	*	-	
MPYR	*	*	*	*	*	*	-	
MPYSU	*	*	*	*	*	*	-	
NEG	*	*	*	*	*	*	*	
NOP	-	-	-	-	-	-	-	
NORM	*	*	*	*	*	?	-	1
NOT	*	-	-	?	?	0	-	7, 8
NOTC	-	-	-	-	-	-	?	14
OR	*	-	-	?	?	0	-	7, 8
ORC	-	-	-	-	-	-	?	4
POP	?	?	?	?	?	?	?	15
REP	-	-	-	-	-	-	-	
RND	*	*	*	*	*	*	-	
ROL	*	-	-	?	?	0	?	7, 8, 9
ROR	*	-	-	?	?	0	?	7, 8, 10
RTI	-	?	?	?	?	?	?	11
RTS	-	-	-	-	-	-	-	
SBC	*	*	*	*	*	*	*	
STOP	-	-	-	-	-	-	-	
SUB	*	*	*	*	*	*	*	
SWI	-	-	-	-	-	-	-	
Tcc	-	-	-	-	-	-	-	
TFR	-	-	-	-	-	-	-	
TST	*	*	*	*	*	0	0	
TSTW	*	-	-	*	*	0	0	
WAIT	-	-	-	-	-	-	-	

Note:
1. V is set if an arithmetic overflow occurs in the 36- or 32-bit result. It is also set if the MSB of the destination operand is changed as a result of the left shift; V is cleared otherwise.
2. C is set if bit 35 or 31 of source operand is set and is cleared otherwise.
3. C is set if bit 0 of source operand is set and is cleared otherwise.
4. C is set if all bits specified by the mask are set and is cleared otherwise. Bits that are not set in the mask should be ignored. If a bit-field instruction is performed on the status register, all bits in this register selected by the bit-field's mask can be affected.
5. C is set if all bits specified by the mask are cleared and is cleared otherwise. Ignore bits which are not set in the mask. Note that if a bit-field instruction is performed on the status register, all bits in this register selected by the bit field's mask can be affected.
6. C is set if bit 35 of the result is cleared and is cleared otherwise.
7. N is set if bit 31 of the result is set and is cleared otherwise.
8. Z is set if bits 16-31 of result are zero and is cleared otherwise.
9. C is set if bit 31 of the source operand is set and is cleared otherwise.
10. C is set if bit 16 of the source operand is set and is cleared otherwise.
11. The "?" bit is set according to value pulled from stack.
12. If the SR is specified as a destination operand, the "?" bit is set according to the corresponding bit of the source operand. If SR is not specified as a destination operand, L is set if data limiting occurred. All bits marked with "?" are not affected otherwise.
13. Condition codes not affected if the operand is a 16-bit register.
14. Set if the value equals $FFF before the complement.
15. Condition codes will be affected if the register popped is the CCR.

See **Condition Code Generation** on page 3-32 for additional information on condition codes.

A.6 INSTRUCTION TIMING

This section describes how to calculate the DSP56800 instruction timing manually using the provided tables. Three complete examples are presented to illustrate the "layered" nature of the tables. Alternatively, the user can obtain the number of instruction program words and the number of oscillator clock cycles required for a given instruction by using the simulator; this is a simple and fast method of determining instruction timing information.

The number of words for an instruction is dependent on the instruction operation and its addressing mode. The symbols used in one table may reference subsequent tables to complete the instruction word count.

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

The tables in this section present the following information:

- **Table A-16** on page A-17 gives the number of instruction program words and the number of machine clock cycles for each instruction mnemonic.
- **Table A-17** on page A-18 gives the number of additional (if any) instruction words and additional (if any) clock cycles for each type of parallel move.
- **Table A-18** on page A-18 gives the number of additional (if any) clock cycles for each type of MOVEC operation.
- **Table A-19** on page A-18 gives the number of additional (if any) clock cycles for each type of MOVEM operation.
- **Table A-20** on page A-18 gives the number of additional (if any) clock cycles for each type of bit-field manipulation (BFCHG, BFCLR, BFSET, BFTSTH, BFTSTL, BRCLR, and BRSET) operation.
- **Table A-21** on page A-19 gives the number of additional (if any) clock cycles for each type of branch or jump (Bcc, Jcc, and JSR) operation.
- **Table A-22** on page A-19 gives the number of additional (if any) clock cycles for the RTS or RTI instruction.
- **Table A-23** on page A-19 gives the number of additional (if any) clock cycles for the TSTW instruction.
- **Table A-24** on page A-19 gives the number of additional (if any) instruction words and additional (if any) clock cycles for each effective addressing mode.
- **Table A-25** on page A-20 gives the number of additional (if any) clock cycles for external data, external program, and external I/O memory accesses.

The assumptions for calculating execution time are listed below:

- All instruction cycles are counted in machine clock cycles. Two machine clock cycles are equivalent to one instruction cycle.
- The instruction fetch pipeline is full.
- There is no contention for instruction fetches. Thus, external program instruction fetches are assumed not to have to contend with external data memory accesses.
- There are no wait states for instruction fetches done sequentially (as for non-change-of-flow instructions), but they are taken into account for change-of-flow instructions which flush the pipeline such as JMP, Jcc, RTS, etc.

Three examples presented at the end of this section illustrate the “layered” nature of the tables. (Items in the “Notes” column of **Table A-16** are explained on page A-18.)

Table A-16 Instruction Timing Summary

Mnemonic	Instruction Program Words	Osc. Clock Cycles	Notes
ABS	1	2+mv	
ADC	1	2	1
ADD	1+mva	2+mva	
AND	1	2	1
ANDC	2+ea	4+mvb	
ASL	1	2+mv	
ASLL	1	2	
ASR	1	2+mv	
ASRAC	1	2	
ASRR	1	2	
Bcc	1	4+jx	
BFCHG	2+ea	4+mvb	
BFCLR	2+ea	4+mvb	
BFSET	2+ea	4+mvb	
BFTSTH	2+ea	4+mvb	
BFTSTL	2+ea	4+mvb	
BRA	1	4+jx	
BRCLR	2+ea	8+mvb	
BRSET	2+ea	8+mvb	
CLR	1	2+mv	
CMP	1+mva	2+mva	
DEBUG	1	4	
DEC(W)	1	2+mv	
DIV	1	2	
DO	2	6	
ENDDO	1	2	
EOR	1	2	1
EORC	2+ea	4+mvb	
ILLEGAL	1	8	
IMPY(16)	1	2	
INC(W)	1	2+mv	
Jcc	2	4+jx	
JMP	2	6+jx	
JSR	2	6+jx	
LEA	1	2	
LSL	1	2	1
LSLL	1	2	1
LSR	1	2	1
LSRAC	1	2	1
LSRR	1	2	1
MAC	1	2+mv	
MACR	1	2+mv	
MACSU	1	2	
MOVE	1	2+mv	4
MOVE(C)	1+ea	2+mvc	
MOVE(I)	1	2	
MOVE(M)	1	8+mvm	
MOVE(P)	1	2+mv	
MOVE(S)	1	2+mv	
MPY	1	2+mv	
MPYR	1	2+mv	
MPYSU	1	2	
NEG	1	2+mv	
NOP	1	2	
NORM	1	2	
NOT	1	2	1
NOTC	2+ea	4+mvb	
OR	1	2	1
ORC	2+ea	4+mvb	
POP	1	2+mv	
REP	1	6	4
RND	1	2+mv	
ROL	1	2	1
ROR	1	2	1
RTI	1	10+rx	
RTS	1	10+rx	
SBC	1	2	1
STOP	1	n/a	2
SUB	1+mva	2+mva	
SWI	1	8	
Tcc	1	2	
TFR	1	2+mv	
TST	1	2+mv	
TSTW	1	2+tst	
WAIT	1	n/a	3

Note: 1. These arithmetic instructions do not permit a parallel move.
2. The STOP instruction disables the internal clock oscillator. After clock turn-on, an internal counter counts some 65,536 cycles before enabling the clock to the internal DSP circuits.
3. The WAIT instruction takes a minimum of 16 cycles to execute when an internal interrupt pending during the execution of the WAIT instruction.
4. This MOVE applies only to the case where two reads are performed in parallel from the X memory.

Table A-17 Parallel Move Timing

Parallel Move Operation	+ mv Words	+mv Cycles
No parallel data move	0	0
X: (X memory move)	0	ax
X: X: (XX memory move)	0	axx

Table A-18 MOVEC Timing Summary

MOVEC Operation	+ mvc Cycles
16-bit Immediate → Register	2
Register → Register	0
X Memory ↔ Register	ea + ax

Table A-19 MOVEM Timing Summary

MOVEM	+ mvm Cycles
Register ↔ P Memory	ap
Note: 1. The “ap” term represents the wait states spent when accessing the program memory during DATA read or write operations and does not refer to instruction fetches.	

Table A-20 Bit-field Manipulation Timing Summary

Bit-field Manipulation Operation	+ mvb Cycles
BFxxx[1] on X memory	ea + (2 * ax)
BFTSTx[2] on X memory	ea + ax
BFxxx[3] or BFTSTx on register	0
BRxxx with condition true	2 + ea + (2 * ax)
BRxxx with condition false	ea + (2 * ax)

BFxxx = BFCHG, BFCLR or BFSET
BFTSTx = BFTSTH, BFTSTL
BRxxx = BRSET,BRCLR	

Table A-21 Branch / Jump Instruction Timing Summary

Branch/Jump Instruction Operation	+ jx Cycles
Jcc, Bcc—condition true	2 + (2 * ap)
Jcc, Bcc—condition false	(2 * ap)
JMP, JSR	(2 * ap)

Note: All two-word jumps execute *three* program memory fetches to refill the pipeline but one of those fetches is sequential yet (the instruction word located at the jump instruction 2nd word address+1). If the jump instruction was fetched from a program memory segment with wait states, another “ap” should be added to account for that third fetch.

Table A-22 RTS Timing Summary

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

Note: The term “2 * ap” represents the two instruction fetches done by the RTI / RTS instruction to refill the pipeline. The ax term represents fetching the return address from the software stack when the stack pointer points to external X memory, and the 2*ax term includes both this fetch as well as the fetch of the SR as performed by the RTI and RTS instructions.

Table A-23 TSTW Timing Summary

TSTW Operation	+ tst Cycles
Register	0
X memory	ea + ax

Table A-24 Addressing Mode Timing Summary

Effective Addressing Mode	+ ea Words	+ ea Cycles
Address Register Indirect		
No Update	0	0
Post-increment by 1	0	0
Post-decrement by 1	0	0
Post addition by Offset Nn	0	0
Indexed by Offset Nn	0	2
Special		

Table A-24 Addressing Mode Timing Summary (Continued)

Effective Addressing Mode	+ ea Words	+ ea Cycles
Immediate Data	1	2
Immediate Short Data	0	0
Absolute Address	1	2
Absolute Short Address	0	0
I/O Short Address	0	0
Implicit	0	0
Indexed by Short Displacement	0	2
Indexed by Long Displacement	1	4

Table A-25 Memory Access Timing Summary

Access Type	X Memory Access	P Memory Access	I/O Access	+ ax Access	+ ap Cycle	+ aio Cycle	+ axx Cycle
X:	Int	—	—	0	—	—	—
X:	Ext	—	—	wx[1]	—	—	—
P:	—	Int	—	—	0	—	—
P:	—	Ext	—	—	wp[2]	—	—
IO:	—	—	Int	—	—	0	—
X:X:	Int:Ext	—	—	—	—	—	0
X:X:	Ext:Int	—	—	—	—	—	wx
X:X:	I/O:Int	—	—	—	—	—	0

Note: 1. wx—external X memory access wait states
2. wp—external X memory access wait states

Three examples using the tables above are listed on the following page:

Example A-1 Arithmetic Instruction with Two Parallel Reads

Problem

Calculate the number of DSP56800 instruction program words and the number of oscillator clock cycles required for the instruction

```
MACR X0,Y0,A   X:(R0)+,Y0   X:(R3)+,X0
```

where the following conditions are true:

- Operating mode register (OMR) = $02 (normal expanded memory map)
- Bus control register (BCR) = $20
- R0 address register = $C000 (external X memory)
- R3 address register = $0052 (internal X memory).

Solution

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

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in**Table A-16** on page A-17.

   According to **Table A-16** on page A-17, the MACR instruction will require 1 instruction program word and will execute in (2 + mv) oscillator clock cycles. The term “mv” represents the additional (if any) instruction program words and the additional (if any) oscillator clock cycles that may be required over and above those needed for the basic MACR instruction due to the parallel move portion of the instruction.
2. Evaluate the “mv” term using **Table A-17** on page A-18.

   The parallel move portion of the MACR instruction consists of an XX memory read. According to **Table A-17** on page A-18, the parallel move portion of the instruction will require mv = axx additional oscillator clock cycles. The term “axx” represents the number of additional (if any) oscillator clock cycles that are required to access two operands in the X memory.

Example A-1 Arithmetic Instruction with Two Parallel Reads (Continued)

3. Evaluate the “axx” term using **Table A-25** on page A-20.

   The parallel move portion of the MACR instruction consists of an XX Memory Read. According to **Table A-25** on page A-20, the term “axx” depends upon where the referenced X memory locations are located in the DSP56800 memory space. External X memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the DSP bus control register (BCR). Thus, assuming that the BCR contains the value $20, external X memory accesses require wx = 1 wait state or additional oscillator clock cycles. For this example the second X memory reference is assumed to be an internal reference while the first X memory reference is assumed to be an external reference. Thus, according to **Table A-25** on page A-20, the XX memory reference in the parallel move portion of the MACR instruction will require axx = wx = 1 additional oscillator clock cycle.

4. Compute the final results.

   Thus, based upon the assumptions given for **Table A-16** on page A-17, the instruction

   ```
   MACR X0,Y0,A  X:(R0)+,Y0  X:(R3)+,X
   ```

   will require 1 instruction program word and will execute in (2 + mv) = (2 + axx) = (2 + wx) = (2 + 1) = 3 oscillator clock cycles.

Note: If a similar calculation were to be made for a MOVEC, MOVEM, or one of the bit field manipulation instructions (BFCHG, BFCLR, BFSET or BFTST), the use of **Table A-17** on page A-18 would no longer be appropriate; the user would refer to **Table A-18** on page A-18, **Table A-19** on page A-18 or **Table A-20** on page A-18, respectively.

Example A-2 Jump Instruction

Problem

Calculate the number of DSP56800 instruction program words and the number of oscillator clock cycles required for the instruction

```
JEQ $2000
```

where the following conditions are true:

- OMR = $02 (normal expanded memory map)
- BCR = $04

Solution

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

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in **Table A-16** on page A-17.

   According to **Table A-16** on page A-17, the Jcc instruction will require two instruction program words and will execute in (4 + jx) oscillator clock cycles. The term “jx” represents the number of additional (if any) oscillator clock cycles required for a jump-type instruction.
2. Evaluate the “jx” term using **Table A-21** on page A-19.

   According to **Table A-21** on page A-19, the Jcc instruction will require 2 + jx additional oscillator clock cycles. If the “ea” condition is true, jx = 2 + 2 * ap, whereas jx = 2 * ap if the condition is false. The term “ap” represents the number of additional (if any) oscillator clock cycles that are required to access a P memory operand. Note that the “+ (2 * ap)” term represents the two program memory instruction fetches executed at the end of a one-word jump instruction to refill the instruction pipeline.

Example A-2 Jump Instruction (Continued)

3. Evaluate the “ap” term using **Table A-25** on page A-20.

   According to **Table A-25** on page A-20, the term “ap” depends upon where the referenced P memory location is located in the 16-bit DSP memory space. External memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the BCR. Thus, assuming that the BCR contains the value $02, external P memory accesses require wp = 2 wait states or additional oscillator clock cycles. For this example the P memory reference is assumed to be an external reference. Thus, according to **Table A-25** on page A-20, the Jcc instruction will use the value ap = wp = 2 oscillator clock cycles.

4. Compute the final results.

   Thus, based upon the assumptions given for **Table A-16** on page A-17, the instruction

   ```
   JEQ  $2000
   ```

   will require (1 + 1) = (1 + 1) = 2 instruction program word and will execute in (4 + jx) = (4 + ea + (2 * ap)) = (4 + ea + (2 * wp)) = (4 + 2 + (2 * 2)) = 10 oscillator clock cycles.

Example A-3 RTS Instruction

Problem

Calculate the number of DSP56800 instruction program words and the number of oscillator clock cycles required for the instruction

```
RTS
```

where the following conditions are true:

- OMR = $02 (normal expanded memory map)
- BCR = $41
- Return Address (on the stack) = $0100 (internal P memory)

Example A-3 RTS Instruction (Continued)

Solution

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

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in **Table A-16** on page A-17.

   According to **Table A-16** on page A-17, the RTS instruction will require 1 instruction program word and will execute in (10 + rx) oscillator clock cycles. The term “rx” represents the number of additional (if any) oscillator clock cycles required for an RTS instruction.
2. Evaluate the “rx” term using **Table A-22** on page A-19.

   According to **Table A-22** on page A-19, the RTS instruction will require rx = (2 * ap) additional oscillator clock cycles. The term “ap” represents the number of additional (if any) oscillator clock cycles that are required to access a P memory operand. The term “(2 * ap)” represents the two program memory instruction fetches executed at the end of an RTS instruction to refill the instruction pipeline.
3. Evaluate the “ap” term using **Table A-25** on page A-20.

   According to **Table A-25** on page A-20, the term “ap” depends upon where the referenced P memory location is located in the 16-bit DSP memory space. External memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the BCR. Thus, assuming that the BCR contains the value $0041, external P memory accesses require wp = 1 wait state or additional oscillator clock cycles. For this example the P memory reference is assumed to be an internal reference. This means that the return address ($0100) pulled from the system stack by the RTS instruction is in internal P memory. Thus, according to **Table A-25** on page A-20, the RTS instruction will use the value ap = 0 additional oscillator clock cycles.

Example A-3 RTS Instruction (Continued)

4. Compute the final results.

   Thus, based upon the assumptions given for **Table A-16** on page A-17, the instruction

   ```
   RTS
   ```

   will require one instruction program word and will execute in (10 + rx) = (10 + (2 * ap)) = (10 + (2 * 0)) = 10 oscillator clock cycles.

A.7 INSTRUCTION SET RESTRICTIONS

These items are restrictions on the DSP56800 instruction set:

- A NORM instruction cannot be immediately followed by an instruction which accesses X memory using the R0 pointer. In addition, NORM can only use the R0 address register.
- No bit-field operation (ANDC, ORC, NOTC, EORC, BFCHG, BFCLR, BFSET, BFTSTH, BFTSTL, BRCLR, or BRSET) can be performed on the HWS register.
- Only positive immediate values less than 8,192 can be moved to the LC register (13 bits).
- The following registers cannot be specified as the loop count for the DO or REP instruction: HWS, SR, OMR, or M01. Similarly, the immediate value of $0 is not allowed for the loop count of a DO instruction.
- A hardware DO loop may not cross a 64K page boundary.
- Any jump, branch, or branch on bit-field may not specify the instructions at LA or LA-1 of a hardware DO loop as their target addresses. Similarly, these instructions may not be located in the last two locations of a hardware DO loop (i.e., at LA or at LA-1).
- A REP instruction can not repeat on an instruction which accesses the P memory or on any multi-word instruction.
- The HI, HS, LO, and LS condition code expressions can only be used when the CC bit is set in the OMR register.
- The access performed using R3 and XAB2 / XDB2 cannot reference external memory. This access must always be made to internal memory.

- If a MOVE instruction changes the value in one of the address registers (R0-R3), then the contents of the register are not available for use until the second following instruction (i.e., the immediately following instruction should not use the modified register to access X memory or update an address). This also applies to the SP register and the M01 register. This also applies if a 16-bit immediate value is moved to the N register.
- If a bit-field instruction changes the value in one of the address registers (R0-R3), then the contents of the register are not available for use until the second following instruction (i.e., the immediately following instruction should not use the modified register to access X memory or update an address). This also applies to the SP, the N, and the M01 registers.
- For the case of nested hardware DO loops, it is required that there are at least two instructions after the pop of the LA and LC registers before the instruction at the last address of the outer loop.

A.8 INSTRUCTION DESCRIPTIONS

The following section describes each instruction in the DSP56800 Family instruction set in complete detail. The format of each instruction description is given in **Notation** on page A-3 at the beginning of this appendix. Instructions which allow parallel moves include the notation “(parallel move)” in both the “Assembler Syntax” and the “Operation” fields. The example given with each instruction discusses the contents of all the registers and memory locations referenced by the opcode/operand portion of that instruction though not those referenced by the parallel move portion of that instruction.

The “Parallel Move Descriptions” section that follows the MOVE instruction description give a complete discussion of parallel moves, including examples that discuss the contents of all the registers and memory locations referenced by the parallel move portion of an instruction.

Whenever an instruction uses an accumulator as both a destination operand for a data ALU operation and as a source for a parallel move operation, the parallel move operation will use the value in the accumulator prior to execution of any data ALU operation.

Whenever a bit in the condition code register is defined according to the standard definition as given in **Condition Code Computation** on page A-10, a brief definition will be given in normal text in the “Condition Code” section of that instruction description. Whenever a bit in the condition code register is defined according to a special definition for some particular instruction, the complete special definition of that bit will be given in the "Condition Code" section of that instruction in bold text to alert the user to any special conditions concerning its use.

Descriptions

ABS Absolute Value ABS

Operation:

|D|→D (parallel move)

Assembler Syntax:

ABS D (parallel move)

Description: Take the absolute value of the destination operand (D), and store the result in the destination accumulator.

Example:

```
ABS    A    X:(R0)+,Y0    ; take ABS value, move data into Y0, update R0
```

A Before Execution

F	FFFF	FFF2
A2	A1	A0

A After Execution

0	0000	000E
A2	A1	A0

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $F:FFFF:FFF2. Since this is a negative number, the execution of the ABS instruction takes the two's-complement of that value and returns $0:0000:000E.

Note: When the D operand equals $8:0000:0000 (-16.0 when interpreted as a decimal fraction), the ABS instruction will cause an overflow to occur since the result cannot be correctly expressed using the standard 36-bit, fixed-point, two's-complement data representation. Data limiting does not occur (i.e., A is not set to the limiting value of $7:FFFF:FFFF) but remains unchanged.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	**V**	C

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

ABS Absolute Value ABS

Instruction Fields:

OPERATION	OPERANDS
ABS	A B

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
ABS	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

ADC Add Long with Carry ADC

Operation:

S + C + D → D (no parallel move)

Assembler Syntax:

ADC S,D (no parallel move)

Description: Add the source operand (S) and C to the destination operand (D) and store the result in the destination accumulator. Long words (32 bits) may be added to the (36-bit) destination accumulator.

Usage: This instruction is typically used in multi-precision addition operations (see **Multi-precision Operations** on page 3-27) when it is necessary to add together two numbers that are larger than 32 bits (such as 64-bit or 96-bit addition).

Example:

ADC Y,A

Before Execution

	0	2000	8000
	A2	A1	A0
Y		2000	8000
		Y1	Y0
		SR	0301

After Execution

	0	4001	0001
	A2	A1	A0
Y		2000	8001
		Y1	Y0
		SR	0300

Explanation of Example:

Prior to execution, the 32-bit Y register, comprised of the Y1 and Y0 registers, contains the value $2000:8000, and the 36-bit accumulator contains the value $0:2000:8000. In addition, C is set to one. The ADC instruction automatically sign-extends the 32-bit Y registers to 36 bits and adds this value to the 36-bit accumulator. In addition, C is added into the LSB of this 36-bit addition. The 36-bit result is stored back in the A accumulator, and the condition codes are set correctly. The Y1:Y0 register pair is not affected by this instruction.

Note: C is set correctly for multi-precision arithmetic, using long word operands only when the extension register of the destination accumulator (A2 or B2) contains sign extension of bit 31 of the destination accumulator (A or B).

ADC Add Long with Carry ADC

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result is zero; cleared otherwise
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 35 of A or B result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERANDS
ADC	Y,A Y,B

Timing: 2 oscillator clock cycles
Memory: 1 program word

ADD Add ADD

Operation:

S + D → D (parallel move)

Assembler Syntax:

ADD S,D (parallel move)

Description: Add the source operand (S) to the destination operand (D) and store the result in the destination accumulator. Words (16 bits), long words (32 bits), and accumulators (36 bits) may be added to the destination.

Usage: This instruction can be used for both integer and fractional two’s-complement data.

Example:

```
ADD      X0,A    X:(R0)+,Y0      X:(R3)+,X0  ; 16-bit add, update Y0,X0,R0,R3
```

Before Execution

0	0100	0000
A2	A1	A0

X0	FFFF

After Execution

0	00FF	0000
A2	A1	A0

X0	FFFF

Explanation of Example:

Prior to execution, the16-bit X0 register contains the value $FFFF, and the 36-bit A accumulator contains the value $0:0100:0000. The ADD instruction automatically appends the 16-bit value in the X0 register with 16 LS zeros, sign-extends the resulting 32-bit long word to 36 bits, and adds the result to the 36-bit A accumulator. Thus, 16-bit operands are always added to the MSP of A or B (A1 or B1), with the result correctly extending into the extension register (A2 or B2). 16-bit operands can be added to the LSP of A or B (A0 or B0) by loading the 16-bit operand into Y0; this forms a 32-bit word by loading Y1 with the sign extension of Y0 and executing an ADD Y,A or ADD Y,B instruction. Similarly, the second accumulator can also be used as the source operand.

Note: C is set correctly using word or long word source operands if the extension register of the destination accumulator (A2 or B2) contains sign extension from bit 31 of the destination accumulator (A or B). C is always set correctly by using accumulator source operands.

ADD Add ADD

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 35 of A or B result

Instruction Fields:

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
ADD	X0,F Y1,F Y0,F A,B B,A (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

DATA ALU OPERATION		FIRST AND SECOND MEMORY READS		DESTINATIONS FOR MEMORY READS	
Operation	**Registers**	**Read1**	**Read2**	**Dest1**	**Dest2**
ADD	X0,A Y1,A Y0,A X0,B Y1,B Y0,B	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
				Y1	X0
				Valid destinations for Read1	Valid destinations for Read2

ADD Add ADD

Instruction Fields:

OPERATION	OPERANDS	COMMENTS	Cyc/Wd

ADD	X0,F
	Y1,F
	Y0,F
	F1,X0
	F1,Y1
	F1,Y0
	Y0,X0
	Y1,X0
	X0,Y0
	Y1,Y0
	X0,Y1
	Y0,Y1
		F = A or B
		F1 = A1 or B1
			(2/1)

	A,B B,A	—	(2/1)
	Y,A Y,B	F = A,B	(2/1)
	#xx,F #xx,DD	5-bit positive integer ranging from 0 to 31	(4/1)
	#xxxx,F #xxxx,DD	16-bit signed integer	(6/2)
	X:<aa>,F X:<aa>,DD	<aa>: First 64 locations of X memory	(4/1)
	F,X:<aa> DD,X:<aa>	—	(6/1)
	X:xxxx,F X:xxxx,DD	Long 16-bit absolute address	(6/2)
	F,X:xxxx DD,X:xxxx	—	(8/2)
	X:(SP-xx),F X:(SP-xx),DD	xx = [1 to 64]	(6/1)
	F,X:(SP-xx) DD,X:(SP-xx)	—	(8/1)

ADD Add ADD

Timing: 2 + mv oscillator clock cycles for ADD instructions with a single or dual parallel move.
Refer to previous tables for ADD instructions without a parallel move.

Memory: 1 program word for ADD instructions with a single or dual parallel move.
Refer to previous tables for ADD instructions without a parallel move.

AND Logical AND AND

Operation:		**Assembler Syntax:**		
S•D → D	(no parallel move)	AND	S,D	(no parallel move)
S•D[31:16] → D[31:16]	(no parallel move)	AND	S,D	(no parallel move)

where • denotes the logical AND operator

Description: Logically AND the source operand (S) with the destination operand (D), and store the result in the destination. This instruction is a 16-bit operation. If the destination is a 36-bit accumulator, the source is ANDed with bits 31-16 of the accumulator. The remaining bits of the destination accumulator are not affected.

Usage: This instruction is used for the logical AND of two registers; the ANDC instruction is appropriate to AND a 16-bit immediate value with a register or memory location.

Example:

```
AND     X0,A                    ; AND X0 with A1
```

Before Execution

6	1234	5678
A2	A1	A0

X0	7F00

After Execution

6	1200	5678
A2	A1	A0

X0	7F00

Explanation of Example:

Prior to execution, the 16-bit X0 register contains the value $7F00, and the 36-bit A accumulator contains the value $6:1234:5678. The AND X0,A instruction logically ANDs the 16-bit value in the X0 register with bits 31-16 of the A accumulator (A1) and stores the 36-bit result in the A accumulator. Bits 35-32 in the A2 register and bits 15-0 in the A0 register are not affected by this instruction.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of A or B result is set
Z — Set if bits 31-16 of A or B result are zero
V — Always cleared

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

AND Logical AND AND

Instruction Fields:

OPERATION	OPERANDS	COMMENTS

AND	X0,F
	Y1,F
	Y0,F
	F1,X0
	F1,Y1
	F1,Y0
	Y0,X0
	Y1,X0
	X0,Y0
	Y1,Y0
	X0,Y1
	Y0,Y1
		F = A or B
		F1 = A1 or B1

Timing: 2 oscillator clock cycles
Memory: 1 program word

ANDC Logical AND, Immediate ANDC

Operation:	**Assembler Syntax:**	
#xxxx•X:<ea> → X:<ea>	ANDC	#iiii,X:<ea>
##xxxx•D → D	ANDC	#iiii,D

where • denotes the logical AND operator

Description: Logically AND a 16-bit immediate data value with the destination operand, and store the results back into the destination. C is also modified as described below. This instruction performs a read-modify-write operation on the destination and requires two destination accesses.

Example:

```
ANDC    #$5555,X:<<$A000     ; AND with immediate data
```

Before Execution		**After Execution**	
X:$A000	C3FF	X:$A000	4155
SR	0301	SR	0300

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$A000 contains the value $C3FF. Execution of the instruction tests the state of the bits 4, 8, 9 in X:$FFE2, clears C (because not all the CCR bits were set), and then clears the bits.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	N	Z	V	C

For destination operand SR:

? — Cleared as defined in the field and if specified in the field

For other destination operands:

L — Set if data limiting occurred during 36-bit source move

C — Set if the all bits specified by the mask are set
Clear if the *not* all bits specified by the mask are set

Note: This instruction is the same as a BFCLR instruction with the 16-bit immediate mask inverted (one’s-complement). This instruction will disassemble as a BFCLR instruction with the mask bits inverted.

ANDC Logical AND, Immediate ANDC

Instruction Fields:

OPERATION		OPERAND	COMMENTS	Cyc/Wd
ANDC	**#iiii**	X:(R2+xx)	xx = [0 to 63]	(6/2)
		X:(SP-xx)	xx = [1 to 64]	(6/2)
		X:<aa>	First 64 words of X memory	(4/2)
		X:<pp>	Last 64 words of X memory	(4/2)
		X:xxxx	16-bit absolute address	(6/3)
		Any register except the HWS	—	(4/2)

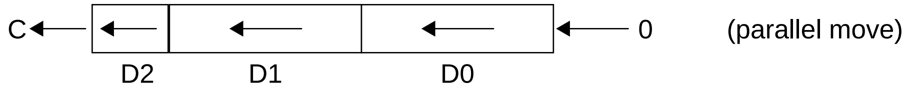
Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

ASL Arithmetic Shift Left ASL

Assembler Syntax:

```
ASL        D                          (parallel move)
```

Operation:



Description: Arithmetically shift the destination operand (D) one bit to the left, and store the result in the destination accumulator. The MSB of the destination prior to the execution of the instruction is shifted into C and a zero is shifted into the LSB of the destination.

Example:

```
ASL        A        X:(R3)+N,Y0     ; multiply A by 2, update R3,Y0
```

Before Execution

A	0123	0123
A2	A1	A0

SR	0300

After Execution

4	0246	0246
A2	A1	A0

SR	0373

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $A:0123:0123. Execution of the ASL A instruction shifts the 36-bit value in the A accumulator one bit to the left and stores the result back in the A accumulator. C is set by the operation because bit 35 of A was set prior to the execution of the instruction. The V bit of CCR (bit 1) is also set because bit 35 of A has changed during the execution of the instruction. The U bit of CCR (bit 4) is set because the result is not normalized, the E bit of CCR (bit 5) is set because the signed integer portion of the result is in use, and the L bit of CCR (bit 6) is set because an overflow has occurred.

ASL Arithmetic Shift Left ASL

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if bit 35 of A or B result is changed due to left shift
C — Set if bit 35 of A or B was set prior to the execution of the instruction

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERAND
ASL	A B X0 Y1 Y0

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
ASL	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

ASLL Multi-Bit Arithmetic Left Shift ASLL

Operation:

S1 << S2 → D (no parallel move)

Assembler Syntax:

ASLL S1,S2,D (no parallel move)

Description: Arithmetically shift the first 16-bit source operand (S1) to the left, by the value contained in the lowest 4 bits of the second source operand (S2), and store the result in the destination register. If the destination is a 36-bit accumulator, correctly sign-extend into the extension register (A2 or B2), and place zero in the LSP (A0 or B0).

Example:

```
ASLL      Y1,X0,A
```

Before Execution

A2	A1	A0
0	3456	3456

Y1	AAAA
X0	0004

After Execution

A2	A1	A0
F	AAA0	0000

Y1	AAAA
X0	0004

Explanation of Example:

Prior to execution, the Y1 register contains the value to be shifted ($AAAA) and the X0 register contains the amount by which to shift ($0004). The contents of the destination register are not important prior to execution because they have no effect on the calculated value. The ASLL instruction arithmetically shifts the value $AAAA four bits to the left and places the result in the destination register A. Since the destination is an accumulator, the extension word (A2) is filled with sign extension, and the LSP (A0) is set to zero.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	V	C

N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero

Note: If CC bit is set N is undefined and Z is set if the LSBs 31-0 are zero.

ASLL Multi-Bit Arithmetic Left Shift ASLL

Instruction Fields:

OPERATION	OPERANDS	COMMENTS

ASLL	Y0,Y0,F
	Y1,Y0,F
	A1,Y0,F
	B1,Y1,F
	Y1,X0,F
	Y0,X0,F
	Y0,Y0,DD
	Y1,Y0,DD
	A1,Y0,DD
	B1,Y1,DD
	Y1,X0,DD
	Y0,X0,DD
		Multi-bit Logical and Arithmetic Shifting
		First register is value to be shifted; second register is the shift amount (uses 4 LSBs)
		F = A,B
		DD = X0,Y1,Y0

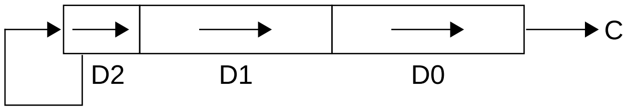
Timing: 2 oscillator clock cycles
Memory: 2 program words

ASR Arithmetic Shift Right ASR

Assembler Syntax:

ASR D (parallel move)

Operation:



(parallel move)

Description: Arithmetically shift the destination operand (D) one bit to the right and store the result in the destination accumulator. The LSB of the destination prior to the execution of the instruction is shifted into C and the MSB of the destination is held constant.

Example:

```
ASR     B      X:(R2)+,Y0      ; divide B by 2, update R3, load R3
```

Before Execution

A	A864	A865
B2	B1	B0

SR	0300

After Execution

D	5432	5432
B2	B1	B0

SR	0329

Explanation of Example:

Prior to execution, the 36-bit B accumulator contains the value $A:A864:A865. Execution of the ASR B instruction shifts the 36-bit value in the B accumulator one bit to the right and stores the result back in the B accumulator. C is set by the operation because bit 0 of A was set prior to the execution of the instruction. The N bit of CCR (bit 3) is also set because bit 35 of the result in A is set. The E bit of CCR (bit 5) is set because the signed integer portion of B is used by the result.

ASR Arithmetic Shift Right ASR

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if data limiting has occurred during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Always cleared
C — Set if bit 0 of A or B was set prior to the execution of the instruction

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERAND
ASR	A B X0 Y1 Y0

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
ASR	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2 + mv oscillator clock cycles
Memory: 1 program words

ASRAC Arithmetic Right Shift with Accumulate ASRAC

Operation:		**Assembler Syntax:**	
S1 >> S2 + D → D	(no parallel move)	ASRAC S1,S2,D	(no parallel move)

Description: Arithmetically shift the first 16-bit source operand (S1) to the right by the value contained in the lowest 4 bits of the second source operand (S2), and accumulate the result with the value in the destination register. If the destination is a 36-bit accumulator, correctly sign-extend into the extension register (A2 or B2).

Usage: This instruction is typically used for multi-precision arithmetic right shifts.

Example:

```
ASRAC    Y1,X0,A                    ; 16-bit add, update X1,X0,R0,R3
```

Before Execution

A2	A1	A0
0	0000	0099

Y1	C003
X0	0004

After Execution

A2	A1	A0
F	FC00	3099

Y1	C003
X0	0004

Explanation of Example:

Prior to execution, the Y1 register contains the value to be shifted ($C003), the X0 register contains the amount by which to shift ($0004), and the destination accumulator contains $0:0000:0099. The ASRAC instruction arithmetically shifts the value $C003 four bits to the right and accumulates this result with the value already in the destination register A. Since the destination is an accumulator, the extension word (A2) is filled with sign extension.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	V	C

N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero

See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

ASRAC Arithmetic Right Shift w/ Accumulate ASRAC

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
ASRAC	Y0,Y0,F Y1,Y0,F A1,Y0,F B1,Y1,F Y1,X0,F Y0,X0,F	Multi-bit Shifting with Accumulation First register is the value to be shifted; second register is the shift amount (uses 4 LSBs) F = A,B

Timing: 2 oscillator clock cycles
Memory: 2 program words

ASRR Multi-Bit Arithmetic Right Shift ASRR

Operation:

S1 >> S2 → D (no parallel move)

Assembler Syntax:

ASRR S1,S2,D (no parallel move)

Description: Arithmetically shift the first 16-bit source operand (S1) to the right by the value contained in the lowest 4 bits of the second source operand (S2), and store the result in the destination register. If the destination is a 36-bit accumulator, correctly sign-extend into the extension register (A2 or B2), and place zero in the LSP (A0 or B0).

Example:

```
ASRR    Y1,X0,A                 ; left shift of 16-bit Y1 by X0
```

Before Execution

0	1234	5678
A2	A1	A0

Y1	AAAA
X0	0004

After Execution

F	FAAA	0000
A2	A1	A0

Y1	AAAA
X0	0004

Explanation of Example:

Prior to execution, the Y1 register contains the value to be shifted ($AAAA), and the X0 register contains the amount by which to shift ($0004). The contents of the destination register are not important prior to execution because they have no effect on the calculated value. The ASRR instruction arithmetically shifts the value $AAAA 4 bits to the right and places the result in the destination register A. Since the destination is an accumulator, the extension word (A2) is filled with sign extension, and the LSP (A0) is set to zero.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	V	C

N — Set if bit 35 of A or B result is set

Z — Set if A or B result equals zero

ASRR Multi-Bit Arithmetic Right Shift ASRR

Instruction Fields:

OPERATION	OPERANDS	COMMENTS

ASRR	Y0,Y0,F
	Y1,Y0,F
	A1,Y0,F
	B1,Y1,F
	Y1,X0,F
	Y0,X0,F
	
	Y0,Y0,DD
	Y1,Y0,DD
	A1,Y0,DD
	B1,Y1,DD
	Y1,X0,DD
	Y0,X0,DD
		Multi-bit Logical and Arithmetic Shifting
		
		First register is value to be shifted; second register is the shift amount (uses 4 LSBs)
		
		F = A,B
		DD = X0,Y1,Y0

Timing: 2 oscillator clock cycles
Memory: 2 program words

Bcc Branch Conditionally Bcc

Operation:

If cc, then PC + label → PC
else PC + 1 → PC

Assembler Syntax:

Bcc aa

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 offset is a 7-bit sized value that is sign-extended to 16 bits.

The term "cc" specifies the following:

"cc" Mnemonic		Condition
CC (HS*)	— carry clear (higher or same)	C=0
CS (LO*)	— carry set(lower)	C=1
EQ	— equal	Z=1
GE	— greater than or equal	$N \oplus V=0$
GT	— greater than	$Z+(N \oplus V)=0$
HI*	— higher	$\overline{C} \bullet \overline{Z} = 1$
LE	— less than or equal	$Z+(N \oplus V)=1$
LS*	— lower or same	C+Z = 1
LT	— less than	$N \oplus V=1$
NE	— not equal	Z=0
NN	— not normalized	$Z+(\overline{U} \bullet \overline{E})=0$
NR	— normalized	$Z+(\overline{U} \bullet \overline{E})=1$

* Only available when CC bit set in the OMR

where: $\overline{X}$ denotes the logical complement of X,
+ denotes the logical OR operator,
• denotes the logical AND operator,
⊕ denotes the logical exclusive OR operator

Example:

```
          BNE    LABEL          ; branch to label if "greater-than"
          INCW   A
          INCW   A
LABEL
          ADD    B,A
```

Explanation of Example:

In this example, if the Z bit is zero when executing the BNE instruction, program execution skips the two INCW instructions and continues with the ADD instruction. If the specified condition is not true, no branch is taken, the program counter is incremented by one, and program execution continues with the first INCW instruction. The Bcc instruction uses a PC-relative offset of 2 for this example.

Restrictions:
- — A Bcc instruction used within a DO loop cannot begin at the LA or LA-1 within that DO loop.
- — A Bcc instruction cannot be repeated using the REP instruction.

Bcc Branch Conditionally Bcc

Condition Codes Affected:

The condition codes are tested but not modified by this instruction.

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
Bcc	$xx	7-bit PC relative offset [-64,63]

Timing: 4 + jx oscillator clock cycles
Memory: 1 program word

BFCHG Test Bit-field and Change BFCHG

Operation:	**Assembler Syntax:**
$\overline{\text{(<bit-field> of destination)}}$ → (<bit-field> of destination)	BFCHG #iiii,X:<ea>
$\overline{\text{(<bit-field> of destination)}}$ → (<bit-field> of destination)	BFCHG #iiii,D

Description: Test all selected bits of the destination operand. If all selected bits are set, C is set; otherwise, C is cleared. Then complement the selected bits, and store the result in the destination memory location. The bits to be tested are selected by a 16-bit immediate value in which every bit set is to be tested and changed. This instruction performs a read-modify-write operation on the destination memory location or register and requires two destination accesses.

Usage: This instruction is very useful in performing I/O and flag bit manipulation.

Example:

```
BFCHG    #$0310,X:<<$FFE2        ; test and change bits 4, 8, 9 in a peripheral register
```

Before Execution		**After Execution**	
X:$FFE2	0010	X:$FFE2	0300
SR	0001	SR	0000

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $0010. Execution of the instruction tests the state of the bits 4, 8, 9 in X:$FFE2, does not set C (because all of the CCR bits were not set), and then complements the bits.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	**C**

For destination operand SR:

? — Changed if specified in the field

For other destination operands:

L — Set if data limiting occurred during 36-bit source move

C — Set if the all bits specified by the mask are set
Clear if *not* all bits specified by the mask are set

Note: If all bits in the mask are set to zero, the instruction executes two NOPs and sets C.

BFCHG Test Bit-field and Change BFCHG

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
BFCHG #iiii	X:(R2+xx)	xx = [0 to 63]	(6/2)
	X:(SP-xx)	xx = [1 to 64]	(6/2)
	X:<aa>	First 64 words of X memory	(4/2)
	X:<pp>	Last 64 words of X memory	(4/2)
	X:xxxx	16-bit absolute address	(6/3)
	Any register except the HWS	—	(4/2)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

BFCLR Test Bit-field and Clear BFCLR

Operation:	Assembler Syntax:	
0 →(<bit-field> of destination)	BFCLR	#iiii,X:<ea>
0 →(<bit-field> of destination)	BFCLR	#iiii,D

Description: Test all selected bits of the destination operand. If all selected bits are set, C is set; otherwise, C is cleared. Then clear the selected bits, and store the result in the destination memory location. The bits to be tested are selected by a 16-bit immediate value in which every bit set is to be tested and cleared. This instruction performs a read-modify-write operation on the destination memory location or register and requires two destination accesses.

Usage: This instruction is very useful in performing I/O and flag bit manipulation.

Example:

```
BFCLR    #$0310,X:<<$FFE2       ; test and clear bits 4, 8, 9 in an on-chip peripheral register
```

Before Execution		After Execution	
X:$FFE2	7F95	X:$FFE2	7C85
SR	0001	SR	0000

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $7F95. Execution of the instruction tests the state of the bits 4, 8, 9 in X:$FFE2, clears C (because not all the CCR bits were clear), and then clears the bits.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	**C**

For destination operand SR:

? — Cleared as defined in the field and if specified in the field

For other destination operands:

L — Set if data limiting occurred during 36-bit source move

C — Set if the all bits specified by the mask are set
Clear if *not* all bits specified by the mask are set

Note: If all bits in the mask are set to zero, the instruction executes two NOPs and sets C.

BFCLR Test Bit-field and Clear BFCLR

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
BFCLR #iiii	X:(R2+xx)	xx = [0 to 63]	(6/2)
	X:(SP-xx)	xx = [1 to 64]	(6/2)
	X:<aa>	First 64 words of X memory	(4/2)
	X:<pp>	Last 64 words of X memory	(4/2)
	X:xxxx	16-bit absolute address	(6/3)
	Any register except the HWS	—	(4/2)

Timing: Refer to table above in Instruction Fields.

Memory: Refer to table above in Instruction Fields.

Descriptions

BFSET Test Bit-field and Set BFSET

Operation:	**Assembler Syntax:**	
1 → (<bit-field> of destination)	BFSET	#iiii,X:<ea>
1 → (<bit-field> of destination)	BFSET	#iiii,D

Description: Test all selected bits of the destination operand. If all selected bits are set, C is set; otherwise, C is cleared. Then set the selected bits, and store the result in the destination memory location. The bits to be tested are selected by a 16-bit immediate value in which every bit set is to be tested and set. This instruction performs a read-modify-write operation on the destination memory location or register and requires two destination accesses.

Usage: This instruction is very useful in performing I/O and flag bit manipulation.

Example:

```
BFSET    #$F400,X:<<$FFE2
```

Before Execution		**After Execution**	
X:$FFE2	8921	X:$FFE2	FD21
SR	0000	SR	0000

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $8921. Execution of the instruction tests the state of bits 10, 12, 13, 14, 15 in X:$FFE2, does not set C (because all the CCR bits were not set), and then sets the bits.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	**C**

For destination operand SR:

? — Set as defined in the field and if specified in the field

For other destination operands:

L — Set if data limiting occurred during 36-bit source move

C — Set if the all bits specified by the mask are set
Clear if *not* all bits specified by the mask are set

Note: If all bits in the mask are set to zero, the instruction executes two NOPs and sets C.

BFSET Test Bit-field and Set BFSET

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
BFSET #iiii	X:(R2+xx)	xx = [0 to 63]	(6/2)
	X:(SP-xx)	xx = [1 to 64]	(6/2)
	X:<aa>	First 64 words of X memory	(4/2)
	X:<pp>	Last 64 words of X memory	(4/2)
	X:xxxx	16-bit absolute address	(6/3)
	Any register except the HWS	—	(4/2)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

BFTSTH Test Bit-field High BFTSTH

Operation:

Test <bit-field> of destination for 1s
Test <bit-field> of destination for 1s

Assembler Syntax:

BFTSTH	#iiii,X:<ea>
BFTSTH	#iiii,D

Description: Test all selected bits of the destination operand. If all selected bits are set, C is set; otherwise, C is cleared. The bits to be tested are selected by a 16-bit immediate value in which every bit set is to be tested. This instruction performs two destination accesses.

Usage: This instruction is very useful for testing I/O and flag bits.

Example:

```
BFTSTH   #$0310,X:<<$FFE2       ; test high bits 4, 8, 9 in an on-chip peripheral register
```

Before Execution		**After Execution**	
X:$FFE2	0FF0	X:$FFE2	0FF0
SR	0000	SR	0001

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $0FF0. Execution of the instruction tests the state of bits 4, 8, 9 in X:$FFE2 and sets C (because all the CCR bits were set).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	**C**

L — Set if data limiting occurred during 36-bit source move
C — Set if the all bits specified by the mask are set
Clear if not all bits specified by the mask are set

Note: If all bits in the mask are set to zero, the instruction executes two NOPs and sets C.

BFTSTH Test Bit-field High BFTSTH

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
BFTSTH #iiii	X:(R2+xx)	xx = [0 to 63]	(6/2)
	X:(SP-xx)	xx = [1 to 64]	(6/2)
	X:<aa>	First 64 words of X memory	(4/2)
	X:<pp>	Last 64 words of X memory	(4/2)
	X:xxxx	16-bit absolute address	(6/3)
	Any register except the HWS	—	(4/2)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

BFTSTL Test Bit-field Low BFTSTL

Operation:	Assembler Syntax:	
Test <bit-field> of destination for 0s	BFTSTL	#iiii,X:<ea>
Test <bit-field> of destination for 0s	BFTSTL	#iiii,D

Description: Test all selected bits of the destination operand. If all selected bits are clear, C is set; otherwise, C is cleared. The bits to be tested are selected by a 16-bit immediate value in which every bit set is to be tested. This instruction performs two destination accesses.

Usage: This instruction is very useful for testing I/O and flag bits.

Example:

```
BFTSTL   #$0310,X:<<$FFE2       ; test low bits 4, 8, 9 in an on-chip peripheral register
```

Before Execution

X:$FFE2	18EC
SR	0000

After Execution

X:$FFE2	18EC
SR	0001

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $18EC. Execution of the instruction tests the state of bits 4, 8, 9 in X:$FFE2 and sets C (because all the CCR bits were cleared).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	**C**

L — Set if data limiting occurred during 36-bit source move
C — Set if the all bits specified by the mask are cleared
Clear if not all bits specified by the mask are cleared

Note: If all bits in the mask are set to zero, the instruction executes two NOPs and sets C.

BFTSTL Test Bit-field Low BFTSTL

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
BFTSTL #iiii	X:(R2+xx)	xx = [0 to 63]	(6/2)
	X:(SP-xx)	xx = [1 to 64]	(6/2)
	X:<aa>	First 64 words of X memory	(4/2)
	X:<pp>	Last 64 words of X memory	(4/2)
	X:xxxx	16-bit absolute address	(6/3)
	Any register except the HWS	—	(4/2)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

BRA Branch BRA

Operation:

PC+label → PC

Assembler Syntax:

BRA aa

Description: Branch to the location in program memory at PC + displacement. The PC contains the address of the next instruction. The displacement is a 7-bit signed value that is sign-extended to form the PC-relative offset.

Example:

```
          BRA     LABEL
          INCW    A
          INCW    A
LABEL
          ADD     B,A
```

Explanation of Example:

In this example, program execution skips the two INCW instructions and continues with the ADD instruction. The BRA instruction uses a PC-relative offset of 2 for this example.

Condition Codes Affected:

The condition codes are not affected by this instruction.

Restrictions:

— A BRA instruction used within a DO loop cannot begin at the LA or LA-1 within that DO loop.

— A BRA instruction cannot be repeated using the REP instruction.

BRA Branch BRA

Instruction Fields:

OPERATION	OPERAND	COMMENTS
BRA	$xx	7-bit PC-relative offset [-64,63]

Timing: 4 + jx oscillator clock cycles
Memory: 1program word

BRCLR Branch if Bits Clear BRCLR

Operation:	Assembler Syntax:	
Branch if <bit-field> of destination is all 0s	BRCLR	#iiii,X:<ea>,aa
Branch if <bit-field> of destination is all 0s	BRCLR	#iiii,D,aa

Description: Test all selected bits of the destination operand. If all the selected bits are clear, C is set, and program execution continues at the location in program memory at PC + displacement. Otherwise, C is cleared, and execution continues with the next sequential instruction. The bits to be tested are selected by a 16-bit immediate value in which every bit set is to be tested.

Usage: This instruction is useful in performing I/O flag polling.

Example:

```
         BRCLR #$0310,X:<<$C000,LABEL
         INCW   A
         INCW   A
LABEL
         ADD    B,A
```

Before Execution		After Execution	
X:$FFE2	18EC	X:$FFE2	0FF0
SR	0000	SR	0001

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $18EC. Execution of the instruction tests the state of bits 4, 8, 9 in X:$FFE2 and sets C (because all the CCR bits were clear). Since C is set, then program execution is transferred to the address offset from the current program counter by the displacement specified in the instruction.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	**C**

L — Set if data limiting occurred during 36-bit source move
C — Set if the all bits specified by the mask are cleared
Clear if not all bits specified by the mask are cleared

Note: If all bits in the mask are set to zero, C is set, and the branch is taken.

BRCLR Branch if Bits Clear BRCLR

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
BRCLR #iiii	X:(R2+xx)	xx = [0 to 63]	(10+jx / 3)
	X:(SP-xx)	xx = [1 to 64]	(10+jx / 3)
	X:<aa>	First 64 words of X memory	(8+jx / 3)
	X:<pp>	Last 64 words of X memory	(8+jx / 3)
	X:xxxx	16-bit absolute address	(10+jx / 4)
	Any register except the HWS	—	(8+jx / 3)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

BRSET Branch if Bits Set BRSET

Operation:

Branch if <bit-field> of destination is all 1s
Branch if <bit-field> of destination is all 1s

Assembler Syntax:

```
BRSET       #iiii,X:<ea>,aa
BRSET       #iiii,D,aa
```

Description: Test all selected bits of the destination operand. If all the selected bits are set, C is set, and program execution continues at the location in program memory at PC + displacement. Otherwise, C is cleared, and execution continues with the next sequential instruction. The bits to be tested are selected by a 16-bit immediate value in which every bit set is to be tested.

Usage: This instruction is useful in performing I/O flag polling.

Example:

```
          BRSET #$0310,X:<<$C000,LABEL
          INCW   A
          INCW   A
LABEL
          ADD    B,A
```

Before Execution		After Execution	
X:$FFE2	0FF0	X:$FFE2	0FF0
SR	0000	SR	0001

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $0FF0. Execution of the instruction tests the state of bits 4, 8, 9 in X:$FFE2 and sets C (because all the CCR bits were set). Since C is set, then program execution is transferred to the address offset from the current program counter by the displacement specified in the instruction.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	**C**

L — Set if data limiting occurred during 36-bit source move
C — Set if the all bits specified by the mask are set
Clear if not all bits specified by the mask are set

Note: If all bits in the mask are set to zero, C is set and the branch is taken.

BRSET Branch if Bits Set BRSET

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
BRSET #iiii	X:(R2+xx)	xx = [0 to 63]	(10+jx/3)
	X:(SP-xx)	xx = [1 to 64]	(10+jx/3)
	X:<aa>	First 64 words of X memory	(8+jx/3)
	X:<pp>	Last 64 words of X memory	(8+jx/3)
	X:xxxx	16-bit absolute address	(10+jx/4)
	Any register except the HWS	—	(8+jx/3)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

CLR Clear Accumulator CLR

Operation:

0 → D (parallel move)

Assembler Syntax:

CLR D (parallel move)

Description: Clear the destination register.

Example:

```
CLR      A      A,X:(R0)+      ; save A into X-data memory before clearing it
```

A Before Execution

2	3456	789A
A2	A1	A0

A After Execution

0	0000	0000
A2	A1	A0

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $2:3456:789A. Execution of the CLR A instruction clears the 36-bit A accumulator to zero.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if data limiting has occurred during parallel move
E — Always cleared if destination is a 36-bit accumulator
U — Always set if destination is a 36-bit accumulator
N — Always cleared if destination is a 36-bit accumulator
Z — Always set if destination is a 36-bit accumulator
V — Always cleared if destination is a 36-bit accumulator

Note: The condition codes are only affected if the destination of the CLR instruction is one of the two 36-bit accumulators (A or B).

CLR Clear Accumulator CLR

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
CLR	X0, Y1, Y0, A1, B1, R0-R3, N	Identical to move #0, <reg>. Does NOT set condition codes

OPERATION	OPERAND	COMMENTS
CLR	A B	Clears 36-bit accumulator and SETS condition codes

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
CLR	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

CMP Compare CMP

Operation:

D - S (parallel move)

Assembler Syntax:

CMP S,D (parallel move)

Description: Subtract the two operands, and update the CCR. The result of the subtraction operation is not stored.

Usage: This instruction can be used for both integer and fractional two's complement data.

Note: This instruction subtracts 36-bit operands. When a word is specified as the source, it is sign-extended and zero-filled to form a valid 36-bit operand. In order for C to be set correctly as a result of the subtraction, the destination must be properly sign-extended. The destination can be *improperly* sign-extended by writing A1 or B1 explicitly prior to executing the compare so that A2 or B2, respectively, may not represent the correct sign extension. This note particularly applies to the case in which the source is extended to compare 16-bit operands, such as X0 with A1.

Example:

```
CMP     Y0,A    X0,X:(R1)+N     ; compare Y0 and A, save X0, update R1
```

Before Execution

0	0020	0000
A2	A1	A0

Y0	0024
SR	0300

After Execution

0	0020	0000
A2	A1	A0

Y0	0024
SR	0319

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $0:0020:0000, and the 16-bit Y0 register contains the value $0024. Execution of the CMP Y0,A instruction automatically appends the 16-bit value in the Y0 register with 16 LS zeros, sign-extends the resulting 32-bit long word to 36 bits, subtracts the result from the 36-bit A accumulator, and updates the CCR (leaving the A accumulator unchanged).

CMP Compare CMP

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of the result is in use
U — Set if result is not normalized
N — Set if bit 35 of the result is set
Z — Set if result equals zero
V — Set if overflow has occurred in result
C — Set if a carry (or borrow) occurs from bit 35 of the result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

CMP Compare CMP

Instruction Fields:

OPERATION	OPERANDS	COMMENTS	Cyc/Wd

CMP	X0,F
	Y1,F
	Y0,F
	F1,X0
	F1,Y1
	F1,Y0
	Y0,X0
	Y1,X0
	X0,Y0
	Y1,Y0
	X0,Y1
	Y0,Y1
		F = A or B
		F1 = A1 or B1
			(2/1)

	A,B B,A	—	(2/1)
	#xx,F #xx,DD	5-bit positive integer ranging from 0 to 31	(4/1)
	#xxxx,F #xxxx,DD	16-bit signed integer	(6/2)
	X:<aa>,F X:<aa>,DD	<aa>: First 64 locations of X memory	(4/1)
	X:xxxx,F X:xxxx,DD	Long 16-bit absolute address	(6/2)
	X:(SP-xx),F X:(SP-xx),DD	xx = [1 to 64]	(6/1)

CMP Compare CMP

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
CMP	X0,F Y1,F Y0,F A,B B,A (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

DEBUG Enter Debug Mode DEBUG

Operation:

Enter the debug processing state

Assembler Syntax:

DEBUG

Description: Enter the debug processing state if the PWD bit is clear in the OnCE port's OCR register, and wait for OnCE commands. If this bit is not clear, then the processor simply executes two NOPs and continues program execution.

Condition Codes Affected:

No condition codes are affected.

DEBUG Enter Debug Mode DEBUG

Instruction Fields:

OPERATION
DEBUG

Timing: 4 oscillator clock cycles
Memory: 1 program word

DEC(W) Decrement Word DEC(W)

Operation:

D2:D1-1 → D2:D1 (parallel move)

Assembler Syntax:

DECW D (parallel move)

Description: Decrement a 16-bit destination or the two upper portions (A2:A1 or B2:B1) of a 36-bit accumulator. If the destination is a 36-bit accumulator, leave the LSP (A2 or B2) unchanged.

Usage: This instruction is typically used when processing integer data.

Example:

```
DECW     A          X:(R2)+,X0       ; Decrement the 20 MSBs of A; update R2,X0
```

A Before Execution

0	0001	0033
A2	A1	A0

A After Execution

0	0000	0033
A2	A1	A0

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $0:0001:0033. Execution of the DECW A instruction decrements by one the upper 20 bits of the A accumulator.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of the result is in use
U — Set if result is unnormalized
N — Set if bit 35 of the result is set
Z — Set if the 20 MSBs of the result are all zeros
V — Set if overflow has occurred in result
C — Set if a carry (or borrow) occurs from bit 35 of the result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERANDS
DEC(W)	A B X0 Y1 Y0

DEC(W) Decrement Word DEC(W)

Instruction Fields:

OPERATION	OPERANDS	COMMENTS	Cyc/Wd
DEC(W)	X:<aa>	<aa>: First 64 locations of X memory	(6/1)
	X:xxxx	Long 16-bit absolute address	(8/2)
	X:(SP-xx)	xx = [1 to 64]	(8/1)

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
DEC(W)	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

DIV Divide Iteration DIV

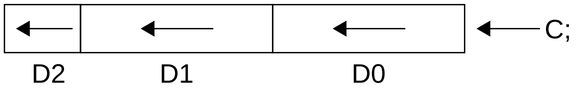
Assembler Syntax:

DIV S,D (no parallel move)

Operation:

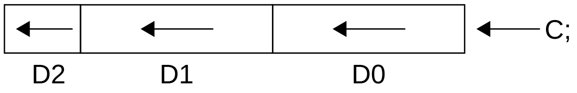
If $D[35] \oplus S[15] = 1$

Then



$D1 + S \longrightarrow D1$

Else



$D1 - S \longrightarrow D1$

Description: This instruction is a divide iteration used to calculate one bit of the result of a division. After the correct number of iterations, this will divide the destination operand (D)—dividend or numerator—by the source operand (S)—divisor or denominator—and store the result in the destination accumulator. *The 32-bit dividend must be a positive value that is correctly sign-extended to 36 bits and is stored in the full 36-bit destination accumulator.The 16-bit divisor is a signed value and is stored in the source operand.* (Division of signed numbers is handled using the techniques documented in **Division** on page 8-17.) This instruction can be used for both integer and fractional division. Each DIV iteration calculates one quotient bit using a nonrestoring division algorithm (see the description that follows). After execution of the first DIV instruction, the destination operand holds both the partial remainder and the formed quotient. The partial remainder occupies the high-order portion of the destination accumulator D and is a signed fraction. The formed quotient occupies the low order portion of the destination accumulator D (A0 or B0) and is a positive fraction. One bit of the formed quotient is shifted into the LSB of the destination accumulator at the start of each DIV iteration. The formed quotient is the true quotient if the true quotient is positive. If the true quotient is negative, the formed quotient must be negated. *For fractional division, valid results are obtained only when* $|D| < |S|$. This condition ensures that the magnitude of the quotient is less than one (i.e., is fractional) and precludes division by zero.

The DIV instruction calculates one quotient bit based on the divisor and the previous partial remainder. To produce an N-bit quotient, the DIV instruction is executed N times, where N is the number of bits of precision desired in the quotient ($1 \leq N \leq 16$). Thus, for a full precision (16- bit) quotient, 16 DIV iterations are required. In general, executing the DIV instruction N times produces an N-bit quotient and a 32-bit remainder, which has (32 - N) bits of precision and whose N MSBs are zeros. The partial remainder is not a true remainder and must be corrected (due to the non-restoring nature of the division algorithm) before it may be used. Therefore, once the divide is complete, it is necessary to reverse the last DIV operation and restore the remainder to obtain the true remainder.

The DIV instruction uses a non-restoring division algorithm that consists of the following operations:

1. Compare the source and destination operand sign bits. An exclusive OR operation is performed on bit 35 of the destination operand and bit 15 of the source operand.
2. Shift the partial remainder and the quotient. The 36-bit destination accumulator is shifted one bit to the left. C is moved into the LSB (bit 0) of the accumulator.

DIV Divide Iteration DIV

3. Calculate the next quotient bit and the new partial remainder. The 16-bit source operand (signed divisor) is either added to, or subtracted from, the MSP of the destination accumulator (A1 or B1), and the result is stored back into the MSP of the destination accumulator. If the result of the exclusive OR operation described previously was one (i.e., the sign bits were different), the source operand S is added to the accumulator. If the result of the exclusive OR operation was zero (i.e., the sign bits were the same), the source operand S is subtracted from the accumulator. Due to the automatic sign extension of the 16-bit signed divisor, the addition or subtraction operation correctly sets C with the next quotient bit.

Example:

The “DIV” iteration instruction can be used in one of several different division algorithms, depending on the needs of an application. **Division** on page 8-17 shows the correct usage of this instruction for fractional and integer division routines and discusses in detail issues related to division. In addition, several examples are provided in this section. The division routine is greatly simplified if both operands are positive, or if it is not necessary to also calculate a remainder.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	**V**	**C**

L — Set if overflow bit V is set

V — Set if the MSB of the destination operand is changed as a result of the instruction’s left shift operation

C — Set if bit 35 of the result is cleared

Instruction Fields:

DATA ALU OPERATION	
Operation	**Operands**
DIV	X0,A Y1,A Y0,A X0,B Y1,B Y0,B

Timing: 2 oscillator clock cycles

Memory: 1 program word

DO Start Hardware Do Loop DO

Operation upon Executing DO Instruction:	**Assembler Syntax:**
HWS[0] → HWS[1]; #xx → LC PC → HWS[0]; LF → NL; expr → LA 1→ LF	DO #xx,expr
HWS[0] → HWS[1]; S → LC PC → HWS[0]; LF → NL; expr → LA 1→ LF	DO S,expr

Operation when Loop Completes (End-of-loop Processing):

NL → LF
HWS[1] → HWS[0]; 0 → NL

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 (shown previously as "expr"). No overhead other than the execution of this DO instruction is required to set up this loop. DO loops can receive their loop count as an immediate value or as a variable stored in an on-chip register. When executing a DO loop, the instructions are actually fetched each time through the loop. Therefore, a DO loop can be interrupted.

During the first instruction cycle, the DO instruction's source operand is loaded into the 13-bit LC register and the second location in the HWS receives the contents of the first location. The LC register stores the remaining number of times the DO loop will be executed and can be accessed from inside the DO loop as a loop count variable subject to certain restrictions. The DO instruction allows all registers on the DSP core to specify the number of loop iterations, except for the following: M01, HWS, OMR, and SR. If immediate short data is instead used to specify the loop count, the 6 LSBs of the LC register are loaded from the instruction, and the upper 7 MSBs are cleared.

During the second instruction cycle, the current contents of the PC are pushed onto the HWS. The DO instruction's destination address (shown as "expr") is then loaded into the LA register. This 16-bit operand is located in the instruction's 16-bit absolute address extension word (as shown in the opcode section). The value in the PC pushed onto the HWS is the address of the first instruction following the DO instruction (i.e., the first actual instruction in the DO loop). At the bottom of the loop, when it is necessary to return to the top for another loop pass, this value is read (i.e., copied but not pulled) from the top of the HWS and loaded into the PC.

During the third instruction cycle, the LF is set. The PC is 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 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.

During the end-of-loop processing, the NL bit is written into the LF, and the NL bit is cleared. The contents of the second HWS location are written into the first HWS location. Instruction fetches now continue at the address of the instruction that follows the last instruction in the DO loop.

DO Start Hardware Do Loop DO

DO loops can also be nested as shown in **Loops** on page 8-25. 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.

Note: The assembler calculates the end-of-loop address to be loaded into LA by evaluating the end-of-loop “expr” and subtracting one. This is done to accommodate the case in which 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.

Note: The LF is cleared by a hardware reset.

Note: Due to pipelining, if an address register (R0-R3, SP, or M01) is changed using a move-type instruction (LEA, Tcc, MOVE, MOVEC, 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 due to the loop construct.

Note: If the A or B accumulator is specified as a source operand, and the data from the accumulator indicates that extension is used, the value to be loaded into the LC register will be limited to a 16-bit maximum positive or negative saturation constant. If positive saturation occurs, the limiter places $7FFF onto the bus, and the lower 13 bits of this value are all ones. The thirteen ones are loaded into the LC register as the maximum unsigned positive loop count allows. If negative saturation occurs, the limiter places $8000 onto the bus, and the lower 13 bits of this value are all 0s. The thirteen 0s are loaded into the LC register, specifying a loop count of zero. The A and B accumulators remain unchanged.

Note: If LC is zero upon entering the DO loop, the loop is executed 2^{13} times. To avoid this, use the software technique outlined in **Loops on page 8-25**.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	C

LF — Set when a DO loop is in progress
L — Set if data limiting occurred

DO Start Hardware Do Loop DO

Restrictions: The end-of-loop comparison previously described occurs at instruction fetch time. That is, LA is compared with PC when the instruction at the LA-2 is being executed. Therefore, instructions that 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 the LA-2, LA-1, or LA specifies one of the program controller registers SR, SP, LA, LC, or (implicitly) PC as a destination register. Similarly, the HWS register may not be specified as a source or destination register in an instruction starting at the LA-2, LA-1, or LA. Additionally, the HWS 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 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 the LA-2, LA-1 and LA

DO
MOVEC from HWS
MOVEC to LA, LC, SR, SP, or HWS
Any bit-field instruction on the Status Register (SR)
Two word instructions that read LC, SP, or HWS

At the LA-1

ENDDO
Single word instructions that read LC, SP, or HWS

At the LA

Any two-word instruction (This restriction applies to the situation in which the DSP simulator’s single-line assembler is used to change the last instruction in a DO loop from a one-word instruction to a two-word instruction.)

Bcc, Jcc	BRSET, BRCLR
BRA, JMP	REP
JSR	RTI, RTS
WAIT, STOP	

Similarly, since the DO instruction accesses the program controller registers, the DO instruction must not be immediately preceded by any of the following instructions:

Immediately before DO:

MOVEC to HWS
MOVEC from HWS

Other Restrictions:

DO HWS,xxxx

JSR to (LA) whenever the LF is set.

A DO instruction cannot be repeated using the REP instruction.

DO Start Hardware Do Loop DO

Example:

```
            DO      #cnt1, END            ; begin DO loop
            MOVE    X:(R0),A
            REP     #cnt2                 ; nested REP loop
            ASL     A                     ; repeat this instruction
            MOVE    A,X:(R0)+             ; last instruction in DO loop
END                 :                     ; (outside DO loop)
```

Explanation of Example:

This example illustrates a DO loop with a REP loop nested within the DO loop. In this example, “cnt1” values are fetched from memory; each is left shifted by “cnt2” counts and is stored back in memory. The DO loop executes “cnt1” times while the ASL instruction inside the REP loop executes (“cnt1” * “cnt2”) times. The END label is located at the first instruction past the end of the DO loop, as mentioned previously.

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
DO	#xx	6-bit unsigned short immediate data
	X0,Y1,Y0, R0,R1,R2,R3, N, LC, LA, A2,B2,A1,B1, A0,B0,A,B	Any register (uses 13 LSBs of register) except M01, SR, OMR, and the HWS

Timing: 6 oscillator clock cycles
Memory: 2 program words

ENDDO End Current DO Loop ENDDO

Operation:

NL → LF
HWS[1] → HWS[0]; 0 → NL

Assembler Syntax:

ENDDO

Description: Terminate the current hardware DO loop immediately. Normally, a hardware DO loop is terminated when the last instruction of the loop is executed and the current LC equals one, but this instruction can terminate a loop before normal completion. If the value of the current DO LC is needed, it must be read before the execution of the ENDDO instruction. Initially, the LF is restored from the NL bit, and the top-of-loop address is purged from the HWS. The contents of the second HWS location are written into the first HWS location, and the NL bit is cleared.

Example:

```
             DO       Y0,ENDLP            ; execute loop ending at ENDLP (Y0) times
                      :
             MOVEC    LC,A                ; get current value of loop counter (LC)
             CMP      Y1,A                ; compare loop counter with value in Y1
             JNE      CONTINU             ; go to ONWARD if LC not equal to Y1
             ENDDO                        ; LC equal to Y1, restore all DO registers
             JMP      ENDLP               ; go to NEXT
CONTINU               :                   ; LC not equal to Y1, continue DO loop
                      :                   ; (last instruction in DO loop)
ENDLP        MOVE     #$1234,X0           ; (first instruction AFTER DO loop)
```

Explanation of Example:

This example illustrates the use of the ENDDO instruction to terminate the current DO loop. The value of the LC is compared with the value in the Y1 register to determine if execution of the DO loop should continue. The ENDDO instruction updates certain program controller registers but does not automatically jump past the end of the DO loop. Thus, if this action is desired, a JMP/BRA instruction (i.e., JMP NEXT as shown previously) must be included after the ENDDO instruction to transfer program control to the first instruction past the end of the DO loop.

Note: The ENDDO instruction updates the program controller registers appropriately but does not automatically jump past the end of the loop. This must be done explicitly by the programmer if desired.

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:

MOVEC to SR or HWS
MOVEC from HWS
Any bit-field instruction on the SR

Also, the ENDDO instruction cannot be the next to last instruction in a DO loop (at the LA-1).

ENDDO End Current DO Loop ENDDO

Condition Codes Affected:

The condition codes are not affected by this instruction.

Instruction Fields:

OPERATION
ENDDO

Timing: 2 oscillator clock cycles
Memory: 1 program word

EOR Logical Exclusive OR EOR

Operation:		**Assembler Syntax:**		
$S \oplus D \rightarrow D$	(no parallel move)	EOR	S,D	(no parallel move)
$S \oplus D[31:16] \rightarrow D[31:16]$	(no parallel move)	EOR	S,D	(no parallel move)

where ⊕ denotes the logical exclusive OR operator

Description: Logically exclusive OR the source operand (S) with the destination operand (D), and store the result in the destination. This instruction is a 16-bit operation. If the destination is a 36-bit accumulator, the source is exclusive ORed with bits 31-16 of the accumulator. The remaining bits of the destination accumulator are not affected.

Usage: This instruction is used for the logical exclusive OR of two registers. If it is desired to exclusive OR a 16-bit immediate value with a register or memory location, then the EORC instruction is appropriate.

Example:

```
EOR     Y1,B                    ;Exclusive OR Y1 with B1
```

Before Execution

	5	5555	6789
	B2	B1	B0
Y1		FF00	

After Execution

	5	AA55	6789
	B2	B1	B0
Y1		FF00	

Explanation of Example:

Prior to execution, the 16-bit Y1 register contains the value $FF00, and the 36-bit B accumulator contains the value $5:5555:6789. The EOR Y1,B instruction logically exclusive OR's the 16-bit value in the Y1 register with bits 31-16 of the B accumulator (B1) and stores the 36-bit result in the B accumulator. The lower word of the accumulator (B0) and the extension byte (B2) are not affected by the operation.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of A or B result is set
Z — Set if bits 31-16 of A or B result are zero
V — Always cleared

EOR Logical Exclusive OR EOR

Instruction Fields:

OPERATION	OPERANDS	COMMENTS

EOR	X0,F
	Y1,F
	Y0,F
		F = A or B
		F1 = A1 or B1

	F1,X0 F1,Y1 F1,Y0	
	Y0,X0 Y1,X0 X0,Y0 Y1,Y0 X0,Y1 Y0,Y1	

Timing: 2 oscillator clock cycles
Memory: 1 program word

EORC Logical Exclusive OR Immediate EORC

Operation:

#xxxx ⊕ X:<ea> → X:<ea>
#xxxx ⊕ D → D

where ⊕ denotes the logical exclusive OR operator

Assembler Syntax:

```
EORC     #iiii,X:<ea>
EORC     #iiii,D
```

Description: Logically exclusive OR a 16-bit immediate data value with the destination operand (D) and store the results back into the destination. C is also modified as described below. This instruction performs a read-modify-write operation on the destination and requires two destination accesses.

Example:

```
EORC     #$0FF0,X:<<$FFE2     ; Exclusive OR with immediate data
```

Before Execution		**After Execution**	
X:$FFE0	5555	X:$FFE0	5AA5
SR	0000	SR	0000

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $0010. Execution of the instruction tests the state of the bits 4, 8, 9 in X:$FFE2, does not set C (because all of the CCR bits were not set), and then complements the bits.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	N	Z	V	**C**

For destination operand SR:
? — Changed if specified in the field
For other destination operands:
C — Set if the all bits specified by the mask are set

Note: This instruction is the same as a BFCHG instruction and uses the same 16-bit immediate mask. This instruction will disassemble as a BFCHG instruction.

EORC Logical Exclusive OR Immediate EORC

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
EORC #iiii	X:(R2+xx)	xx = [0 to 63]	(6/2)
	X:(SP-xx)	xx = [1 to 64]	(6/2)
	X:<aa>	First 64 words of X memory	(4/2)
	X:<pp>	Last 64 words of X memory	(4/2)
	X:xxxx	16-bit absolute address	(6/3)
	Any register except the HWS	—	(4/2)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

ILLEGAL Illegal Instruction Interrupt ILLEGAL

Operation:	**Assembler Syntax:**	
Begin illegal instruction exception routine	ILLEGAL	(no parallel move)

Description: Normal instruction execution is suspended and illegal instruction exception processing is initiated. The interrupt priority level bits (I1 and I0) are set to 11 in the status register. The purpose of the illegal interrupt is to force the DSP into an illegal instruction exception for test purposes. Executing an ILLEGAL instruction is a fatal error; the exception routine should indicate this condition and cause the system to be restarted.

If the ILLEGAL instruction is in a DO loop at the LA and the instruction at the LA-1 is being interrupted, then LC will be decremented twice due to the same mechanism that causes LC to be decremented twice if JSR, REP,… are located at the LA.

Since REP is uninterruptable, repeating an ILLEGAL instruction results in the interrupt not being taken until after completion of the REP. After servicing the interrupt, program control will return to the address of the second word following the ILLEGAL instruction. Of course, the ILLEGAL interrupt service routine should abort further processing, and the processor should be reinitialized.

Usage: The ILLEGAL instruction provides a means for testing the interrupt service routine executed upon discovering an illegal instruction. This allows a user to verify that the interrupt service routine can correctly recover from an illegal instruction and restart the application. The ILLEGAL instruction is not used in normal programming.

Example:

```
ILLEGAL
```

Explanation of Example: See the previous description.

Condition Codes Affected:

The condition codes are not affected by this instruction.

ILLEGAL Illegal Instruction Interrupt ILLEGAL

Instruction Fields:

OPERATION
ILLEGAL

Timing: 8 oscillator clock cycles
Memory: 1 program word

IMPY(16) Integer Multiply IMPY(16)

Operation:

(S1*S2) → D1
sign-extend D2; leave D0 unchanged

Assembler Syntax:

IMPY16 S1,S2,D (no parallel move)

Description: Perform an integer multiplication on the two 16-bit signed integer source operands (S1 and S2) and store the lowest 16 bits of the integer product in the upper word (D1) of the destination accumulator (D), leaving the lower word (D0) unchanged, and sign-extending the extension register (D2).

Usage: This instruction is useful in general computing when it is necessary to multiply 2 integers and the nature of the computation can guarantee that the result fits in a 16-bit destination. In this case, it is better to place the result in the MSP (A1 or B1) of an accumulator, because more instruction have access to this portion than to the other portions of the accumulator.

Note: No overflow control or rounding is performed during integer multiply instructions. The result is always a 16-bit signed integer result that is sign-extended to 24 bits.

Example:

```
IMPY      Y0,X0,A                    ; form product
```

Before Execution

F	AAAA	789A
A2	A1	A0
	X0	0003
	Y0	0004

After Execution

0	000C	789A
A2	A1	A0
	X0	0003
	Y0	0004

Explanation of Example:

Prior to execution, the data ALU registers X0 and Y0 contain, respectively, two 16-bit signed integer values ($0003 and $0004). The contents of the destination accumulator are not important prior to execution. Execution of the IMPY X0,Y0,A instruction integer multiplies X0 and Y0 and stores the result ($000C) in A1. A0 remains unchanged, and A2 is sign-extended.

IMPY(16) Integer Multiply IMPY(16)

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	**E**	**U**	**N**	**Z**	**V**	C

E — Not defined
U — Not defined
N — Set if bit 35 of the result is set
Z — Set if the 20 MSBs of the result equal zero
V — Set if overflow occurs in the 16-bit result

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
IMPY	Y0,Y0,F Y1,Y0,F A1,Y0,F B1,Y1,F Y1,X0,F Y0,X0,F	Multiplication result may not be inverted F = A or B
	Y0,Y0,DD Y1,Y0,DD A1,Y0,DD B1,Y1,DD Y1,X0,DD Y0,X0,DD	Multiplication result may not be inverted DD = X0,Y1,Y0

Timing: 2 oscillator clock cycles
Memory: 1 program word

INC(W) Increment Word INC(W)

Operation:

D2:D1+1 → D2:D1 (parallel move)

Assembler Syntax:

INCW D (parallel move)

Description: Increment a 16-bit destination (D) or the two upper portions (A2:A1 or B2:B1) of a 36-bit accumulator. If the destination is a 36-bit accumulator, leave the LSP (A0 or B0) unchanged.

Usage: This instruction is typically used when processing integer data.

Example:

```
INCW      A         X:(R0),X0        ; Increment the 20 MSBs of A; update X0
```

A Before Execution

0	0001	0033
A2	A1	A0

A After Execution

0	0002	0033
A2	A1	A0

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $0:0001:0033. Execution of the INCW A instruction increments by one the upper 20 bits of the A accumulator.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of the result is in use
U — Set if result is unnormalized
N — Set if bit 35 of the result is set
Z — Set if the 20 MSBs of the result are all zeroes
V — Set if overflow has occurred in result
C — Set if a carry (or borrow) occurs from bit 35 of the result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERANDS
INC(W)	A B X0 Y1 Y0

INC(W) Increment Word INC(W)

Instruction Fields:

OPERATION	OPERANDS	COMMENTS	Cyc/Wd
INC(W)	X:<aa>	<aa>: First 64 locations of X memory	(6/1)
	X:xxxx	Long 16-bit absolute address	(8/2)
	X:(SP-xx)	xx = [1 to 64]	(8/1)

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
INC(W)	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word

Jcc Jump Conditionally Jcc

Operation:

If cc, then label → PC
else PC+1 → PC

Assembler Syntax:

Jcc xxxxx

Description: If the specified condition is true, program execution continues at the effective address specified in the instruction. If the specified condition is false, the PC is incremented and program execution continues sequentially. The effective address is a 19-bit absolute address.

The term “cc” specifies the following:

“cc” Mnemonic		Condition
CC (HS*)	— carry clear (higher or same)	C=0
CS (LO*)	— carry set (lower)	C=1
EQ	— equal	Z=1
GE	— greater than or equal	N ⊕ V=0
GT	— greater than	Z+(N ⊕ V)=0
HI*	— higher	xxxxx
LE	— less than or equal	Z+(N ⊕ V)=1
LS*	— lower or same	xxxxxx
LT	— less than	N ⊕ V=1
NE	— not equal	Z=0
NN	— not normalized	$Z+(\overline{U}\bullet\overline{E})=0$
NR	— normalized	$Z+(\overline{U}\bullet\overline{E})=1$
* Only available when CC bit set in the OMR		

where: $\overline{X}$ denotes the logical complement of X,
+ denotes the logical OR operator,
• denotes the logical AND operator,
⊕ denotes the logical exclusive OR operator

Example:

```
          JCS     LABEL          ; jump to label if carry bit is set
          INCW    A
          INCW    A
LABEL
          ADD     B,A
```

Explanation of Example:

In this example, if C is one when executing the JCS instruction, program execution skips the two INCW instructions and continues with the ADD instruction. If the specified condition is not true, no jump is taken, the program counter is incremented by one, and program execution continues with the first INCW instruction. The Jcc instruction uses a 19-bit absolute address for this example.

Restrictions:

— A Jcc instruction used within a DO loop cannot begin at the LA or LA-1 within that DO loop.

— A Jcc instruction cannot be repeated using the REP instruction.

Jcc Jump Conditionally Jcc

Condition Codes Affected:

The condition codes are tested but not modified by this instruction.

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
Jcc	$xxxxx	19-bit absolute address

Timing: 4 + jx oscillator clock cycles
Memory: 2 program words

JMP Jump JMP

Operation:

label → PC

Assembler Syntax:

JMP xxxxx

Description: Jump to program memory at the location given by the instruction's effective address. The effective address is a 19-bit absolute address.

Example:

```
JMP     LABEL
```

Explanation of Example:

In this example, program execution is transferred to the address represented by label. The DSP core supports up to 19-bit program addresses.

Condition Codes Affected:

The condition codes are not affected by this instruction.

Restrictions:

— A JMP instruction used within a DO loop cannot begin at the LA within that DO loop.

— A JMP instruction cannot be repeated using the REP instruction.

JMP Jump JMP

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
JMP	$xxxxx	19-bit absolute address

Timing: 6 oscillator clock cycles
Memory: 1 program word

JSR Jump to Subroutine JSR

Operation:

SP+1 → SP
PC → X:(SP)
SP+1 → SP
SR → X:(SP)
xxxxx → PC

Assembler Syntax:

JSR xxxxx

Description: Jump to subroutine in program memory at the location given by the instruction’s effective address. The effective address is a 19-bit absolute address.

Example:

```
JSR        LABEL                    ; jump to absolute address indicated by “LABEL”
```

Explanation of Example:

In this example, program execution is transferred to the subroutine at the address represented by label. The DSP core supports up to 19-bit program addresses.

Condition Codes Affected:

The condition codes are not affected by this instruction.

Restrictions:

— A JSR instruction used within a DO loop cannot begin at the LA within that DO loop.

— A JSR instruction used within a DO loop cannot specify the LA as its target.

— A JSR instruction cannot be repeated using the REP instruction.

JSR Jump to Subroutine JSR

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
JSR	$xxxxx	19-bit absolute address

Timing: 8 oscillator clock cycles
Memory: 1 program word

LEA Load Effective Address LEA

Operation:

ea → D (no parallel move)

Assembler Syntax:

LEA ea

Description: The address calculation specified is executed and the resulting effective address (ea) is stored in the destination register (D). The source address register and the update mode used to compute the updated address are specified by the effective address. The source address register specified in the effective address is not updated. All update addressing modes may be used. The new register contents are available for use by the immediately following instruction.

Example:

```
LEA     (R0)+N0               ; update R0 using (R0)+N0
```

	Before Execution		After Execution
R0	8001	R0	8C02
N	0C01	N	0C01
M01	1000	M01	1000

Explanation of Example:

Prior to execution, the 16-bit address register R0 contains the value $8001, the 16-bit address register N contains the value $0C01, and the 16-bit modulo register M01 contains the value $1000. Execution of the LEA (R0)+N instruction adds the contents of the R0 register to the contents of the N register and stores the resulting updated address in the R0 address register. The addition is performed using modulo arithmetic since it is done with the R0 register and M01 is not equal to $FFFF. No wraparound occurs during the addition because the result falls within the boundaries of the modulo buffer.

Condition Codes Affected:

The condition codes are not affected by this instruction.

LEA Load Effective Address LEA

Instruction Fields:

OPERATION	OPERAND
LEA	(Rn)+ (Rn)- (Rn)+N
	(SP)+ (SP)- (SP)+N

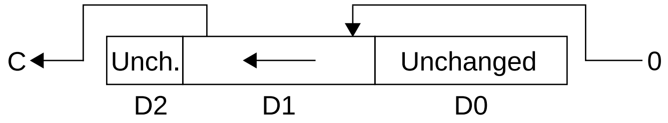
Timing: 2 oscillator clock cycles
Memory: 1 program word

LSL Logical Shift Left LSL

Assembler Syntax:

LSL D

Operation:



(no parallel move)

Description: Logically shift 16 bits of the destination operand (D) one bit to the left, and store the result in the destination. If the destination is a 36-bit accumulator, the result is stored in the MSP of the accumulator (A1 or B1), and the remaining portions of the accumulator (A2, B2, A0, B0) are not modified.The MSB of the destination (bit 31 if the destination is a 36-bit accumulator) prior to the execution of the instruction is shifted into C and zero is shifted into the LSB of D1 (bit 16 if the destination is a 36-bit accumulator).

Example:

```
LSL      B                        ; multiply B1 by 2
```

Before Execution

6	8000	00AA
B2	B1	B0

SR	0300

After Execution

6	0000	00AA
B2	B1	B0

SR	0305

Explanation of Example:

Prior to execution, the 36-bit B accumulator contains the value $6:8000:00AA. Execution of the LSR B instruction shifts the 16-bit value in the B1 register one bit to the left and stores the result back in the B1 register. C is set by the operation because bit 31 of A1 was set prior to the execution of the instruction. The Z bit of CCR (bit 2) is also set because the result in A1 is zero.

LSL Logical Shift Left LSL

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	**N**	**Z**	**V**	**C**

L — Set if overflow has occurred in result
N — Set if bit 31 of A or B result is set
Z — Set if A1 or B1 result equals zero
V — Always cleared
C — Set if bit 31 of A or B was set prior to the execution of the instruction

Instruction Fields:

OPERATION	OPERAND
LSL	A B X0 Y1 Y0

Timing: 2 oscillator clock cycles
Memory: 1 program word

Descriptions

LSLL Multi-Bit Logical Left Shift LSLL

Operation:			**Assembler Syntax:**		
S1 << S2 →	D	(no parallel move)	LSLL	S1,S2,D	(no parallel move)

Description: Logically shift the first 16-bit source operand (S1) to the left by the value contained in the lowest 4 bits of the second source operand (S2) and store the result in the destination register (D). The destination must always be a 16-bit register.

Example:

```
LSLL    Y1,X0,Y1                 ; left shift of 16-bit Y1 by X0
```

Before Execution		**After Execution**	
Y1	AAAA	Y1	AAA0
X0	0004	X0	0004

Explanation of Example:

Prior to execution, the Y1 register contains the value to be shifted ($AAAA) and the X0 register contains the amount to shift by ($0004). The contents of the destination register are not important prior to execution because they have no effect on the calculated value. The LSLL instruction logically shifts the value $AAAA 4 bits to the left and places the result in the destination register Y1.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	V	C

N — Set if bit 35 of A or B result is set
Z — Set if the result in D is zero

LSLL Multi-Bit Logical Left Shift LSLL

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
LSLL	Y0,Y0,DD Y1,Y0,DD A1,Y0,DD B1,Y1,DD Y1,X0,DD Y0,X0,DD	Multi-bit Logical Shifting First register is value to be shifted; second register is the shift amount (uses 4 LSBs) DD = X0,Y1,Y0

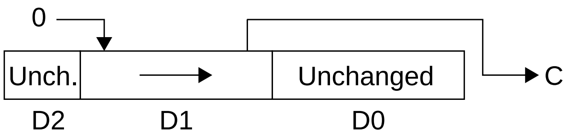
Timing: 2 oscillator clock cycles
Memory: 2 program words

LSR Logical Shift Right LSR

Assembler Syntax:

LSR D

Operation:



(no parallel move)

Description: Logically shift 16 bits of the destination operand (D) one bit to the right and store the result in the destination. If the destination is a 36-bit accumulator, the result is stored in the MSP of the accumulator (A1 or B1), and the remaining portions of the accumulator (A2, B2, A0, B0) are not modified.The LSB of the destination (bit 16 if the destination is a 36-bit accumulator) prior to the execution of the instruction is shifted into C, and zero is shifted into the MSB of D1 (bit 31if the destination is a 36-bit accumulator).

Example:

```
LSR     B                          ; divide B1 by 2 (B1 considered unsigned)
```

Before Execution

F	0001	00AA
B2	B1	B0

SR	0300

After Execution

F	0000	00AA
B2	B1	B0

SR	0305

Explanation of Example:

Prior to execution, the 36-bit B accumulator contains the value $F:0001:00AA. Execution of the LSR B instruction shifts the 16-bit value in the B1 register one bit to the right and stores the result back in the B1 register. C is set by the operation because bit 0 of B1 was set prior to the execution of the instruction. The Z bit of CCR (bit 2) is also set because the result in B1 is zero.

LSR Logical Shift Right LSR

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	**N**	**Z**	**V**	**C**

L — Set if data limiting has occurred during parallel move
N — Always cleared
Z — Set if A1 or B1 result equals zero
V — Always cleared
C — Set if bit 16 of A or B was set prior to the execution of the instruction

Instruction Fields:

OPERATION	OPERAND
LSR	A B X0 Y1 Y0

Timing: 2 oscillator clock cycles
Memory: 1 program word

LSRAC Logical Right Shift with Accumulate LSRAC

Operation:

S1 >> S2 + D →D (no parallel move)

Assembler Syntax:

LSRAC S1,S2,D (no parallel move)

Description: Logically shift the first 16-bit source operand (S1) to the right, by the value contained in the lowest 4 bits of the second source operand (S2) and accumulate the result with the value in the destination register (D). If the destination is a 36-bit accumulator, correctly sign-extend into the extension register (A2 or B2).

Usage: This instruction is used for multi-precision logical right shifts.

Example:

```
LSRAC    Y1,X0,A                    ;16-bit add, update X1,X0,R0,R3
```

Before Execution

0	0000	0099
A2	A1	A0
	Y1	C003
	X0	0004

After Execution

0	0C00	3099
A2	A1	A0
	Y1	C003
	X0	0004

Explanation of Example:

Prior to execution, the Y1 register contains the value to be shifted ($C003), the X0 register contains the amount by which to shift ($0004), and the destination accumulator contains $0:000:0099. The LSRAC instruction logically shifts the value $C003 4 bits to the right and accumulates this result with the value already in the destination register A. Since the destination is an accumulator, the extension word (A2) is filled with sign extension.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	V	C

N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero

See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

LSRAC Logical Right Shift with Accumulate LSRAC

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
LSRAC	Y0,Y0,F Y1,Y0,F A1,Y0,F B1,Y1,F Y1,X0,F Y0,X0,F	Used when performing a multi-bit shift of a multi-precision value First register is the value to be shifted; second register is the shift amount (uses 4 LSBs) F = A,B

Timing: 2 oscillator clock cycles
Memory: 2 program words

LSRR Multi-Bit Logical Right Shift LSRR

Operation:

S1 >> S2 → D (no parallel move)

Assembler Syntax:

LSRR S1,S2,D (no parallel move)

Description: Logically shift the first 16-bit source operand (S1) to the right by the value contained in the lowest 4 bits of the second source operand (S2), and store the result in the destination register (D). If the destination is a 36-bit accumulator, correctly sign-extend into the extension register (A2 or B2), and place zero in the LSP (A0 or B0).

Example:

```
LSRR    Y1,X0,A                 ; right shift of 16-bit Y1 by X0
```

Before Execution

A2	A1	A0
0	3456	3456

Y1	AAAA
X0	0004

After Execution

A2	A1	A0
0	0AAA	0000

Y1	AAAA
X0	0004

Explanation of Example:

Prior to execution, the Y1 register contains the value to be shifted ($AAAA), and the X0 register contains the amount by which to shift ($0004). The contents of the destination register are not important prior to execution because they have no effect on the calculated value. The LSRR instruction logically shifts the value $AAAA 4 bits to the right and places the result in the destination register (A). Since the destination is an accumulator, the extension word (A2) is filled with sign extension, and the LSP (A0) is set to zero.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	V	C

N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero

LSRR Multi-Bit Logical Right Shift LSRR

Instruction Fields:

OPERATION	OPERANDS	COMMENTS

LSRR	Y0,Y0,F
	Y1,Y0,F
	A1,Y0,F
	B1,Y1,F
	Y1,X0,F
	Y0,X0,F
	Y0,Y0,DD
	Y1,Y0,DD
	A1,Y0,DD
	B1,Y1,DD
	Y1,X0,DD
	Y0,X0,DD
		Multi-bit Logical Shifting
		First register is value to be shifted; second register is the shift amount (uses 4 LSBs)
		F = A,B
		DD = X0,Y1,Y0

Timing: 2 oscillator clock cycles
Memory: 2 program words

MAC Multiply-Accumulate MAC

Operation:	Assembler Syntax:	
D + S1 * S2 → D (no parallel move)	MAC (±)S1,S2,D	(no parallel move)
D + S1 * S2 → D (one parallel move)	MAC S1,S2,D	(one parallel move)
D + S1 * S2 → D (two parallel reads)	MAC S1,S2,D	(two parallel reads)

Description: Multiply the two signed 16-bit source operands (S1 and S2) and add/subtract the product to/from the specified 36-bit destination accumulator (D). The "-" sign option is used to negate the specified product prior to accumulation. This option is not available when a single parallel move or two parallel read operations are performed.

Usage: This instruction is used for multiplication and accumulation of fractional data or integer data when a full 32-bit product is required (see **Integer Multiplication** on page 3-23). When the destination is a 16-bit register, this instruction is useful only for fractional data.

Example:

```
MAC      X0,Y1,A              X:(R2)+,Y1X:(R3)+,X0
```

	A2	A1	A0	X0	Y1
Before Execution	0	0003	0003	4000	0AA0
After Execution	0	0553	0003	4000	0AA0

Explanation of Example:

Prior to execution, the 16-bit X0 register contains the value $4000, the 16-bit Y1 register contains the value $0AA0, and the 36-bit A accumulator contains the value $0:0003:0003. Execution of the MAC X0,Y1,A instruction multiplies the 16-bit signed value in the X0 register by the 16-bit signed value in Y1, adds the resulting 32-bit product to the 36-bit A accumulator, and stores the result ($0:0553:0003) into the A accumulator. In parallel, X0 and Y1 are updated with new values fetched from the data memory, and the two address registers (R2 and R3) are post-incremented by one.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	C

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

MAC Multiply-Accumulate MAC

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
MAC	Y0,Y0,F Y1,Y0,F A1,Y0,F B1,Y1,F Y1,X0,F Y0,X0,F	Allows multiplication result to be inverted if desired F = A,B
	Y0,Y0,DD Y1,Y0,DD A1,Y0,DD B1,Y1,DD Y1,X0,DD Y0,X0,DD	Multiplication result may not be inverted DD = X0,Y1,Y0

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
MAC	X0,F Y1,F Y0,F A,B B,A (F = A,B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

DATA ALU OPERATION		FIRST AND SECOND MEMORY READS		DESTINATIONS FOR MEMORY READS	
Operation	**Registers**	**Read1**	**Read2**	**Dest1**	**Dest2**
MAC	X0,A Y1,A Y0,A X0,B Y1,B Y0,B	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
				Y1	X0
				Valid destinations for Read1	Valid destinations for Read2

Timing: 2 + mv oscillator clock cycles for MAC instructions with a parallel move.
Refer to previous table for MAC instructions without a parallel move.

Memory: 1 program word for MAC instructions with a parallel move.
Refer to previous table for MAC instructions without a parallel move.

MACR Multiply Accumulate and Round MACR

Operation:	Assembler Syntax:		
D + S1 * S2 + r → D (no parallel move)	MACR	(±)S1,S2,D	(no parallel move)
D + S1 * S2 + r → D (one parallel move)	MACR	S1,S2,D	(one parallel move)
D + S1 * S2 + r → D (two parallel reads)	MACR	S1,S2,D	(two parallel reads)

Description: Multiply the two signed 16-bit source operands (S1 and S2), add/subtract the product to/from the specified 36-bit destination accumulator (D), and round the result using the specified rounding. The rounded result is stored in the destination accumulator. (Refer to RND for more complete information on the convergent rounding process.) The "-" sign option is used to negate the specified product prior to accumulation. This option is not available when a single parallel move or two parallel reads are performed. The default sign option is "+".

Usage: This instruction is used for multiplication, accumulation, and rounding of fractional data.

Example:

```
MACR     -X0,Y1,A
```

Before Execution

A2	A1	A0
0	0003	8000

X0	4000
Y1	C000

After Execution

A2	A1	A0
0	2004	0000

X0	4000
Y1	C000

Explanation of Example:

Prior to execution, the 16-bit X0 register contains the value $4000, the 16-bit Y1 register contains the value $C000, and the 36-bit A accumulator contains the value $0:0003:8000. Execution of the MACR -X0,Y1,A instruction multiplies the 16-bit signed value in the X0 register by the 16-bit signed value in Y1 and subtracts the resulting 32-bit product to the 36-bit A accumulator, rounds the result, and stores the result ($0:2004:0000) into the A accumulator. In this example, the default rounding (convergent rounding) is performed.

MACR Multiply-Accumulate and Round MACR

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	C

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
MACR	Y0,Y0,F Y1,Y0,F A1,Y0,F B1,Y1,F Y1,X0,F Y0,X0,F	Allows multiplication result to be inverted if desired F = A or B
	Y0,Y0,DD Y1,Y0,DD A1,Y0,DD B1,Y1,DD Y1,X0,DD Y0,X0,DD	Multiplication result may not be inverted DD = X0,Y1,Y0

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
MACR	X0,F Y1,F Y0,F A,B B,A (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

MACR Multiply-Accumulate and Round MACR

DATA ALU OPERATION		FIRST AND SECOND MEMORY READS		DESTINATIONS FOR MEMORY READS	
Operation	**Registers**	**Read1**	**Read2**	**Dest1**	**Dest2**
MACR	X0,A Y1,A Y0,A X0,B Y1,B Y0,B	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
				Y1	X0
				Valid destinations for Read1	Valid destinations for Read2

MACR Multiply-Accumulate and Round MACR

Timing: 2 + mv oscillator clock cycles for MACR instructions with a parallel move.
Refer to previous table for MACR instructions without a parallel move.

Memory: 1 program word for MACR instructions with a parallel move.
Refer to previous table for MACR instructions without a parallel move.

MACSU Multiply-Accumulate Signed x Unsigned MACSU

Operation:

D + S1 * S2 → D (S1 signed, S2 unsigned)

Assembler Syntax:

MACSU S1,S2,D (no parallel move)

Description: Multiply the two 16-bit source operands (S1 and S2) and add the product to the specified 36-bit destination accumulator (D). S1 can be unsigned, but S2 is always considered unsigned. This mixed arithmetic multiply-accumulate does not allow a parallel move and can be used for multi-precision multiplications.

Usage: In addition to single-precision multiplication of a signed times unsigned value and accumulation, this instruction is also used for multi-precision multiplications, as shown in **Multi-precision Multiplication** on page 3-27.

Example:

```
MACSU   X0,Y0,A
```

Before Execution

A2	A1	A0
0	0000	0099

X0	3456
Y0	8000

After Execution

A2	A1	A0
0	3456	0099

X0	3456
Y0	8000

Explanation of Example:

The 16-bit X0 register contains the value $3456 and the 16-bit Y0 register contains the value $8000. Execution of the MACSU X0,Y0,A instruction multiplies the 16-bit signed value in the X0 register by the 16-bit unsigned value in Y0, then adds the result to the A accumulator and stores the signed result back into the A accumulator. If this were a MAC instruction, Y0 ($8000) would equal -1.0, and the multiplication result would be $F:CBAA:0000. Since this is a MACSU instruction, Y0 is considered unsigned and equals +1.0. This gives a multiplication result of $0:3456:0000.

MACSU Multiply-Accumulate Signed x Unsigned MACSU

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	**E**	**U**	**N**	**Z**	**V**	C

E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERANDS	COMMENTS

MACSU	Y0,Y0,F
	Y0,Y1,F
	Y0,A1,F
	Y1,B1,F
	X0,Y1,F
	X0,Y0,F
	
	Y0,Y0,DD
	Y0,Y1,DD
	Y0,A1,DD
	Y1,B1,DD
	X0,Y1,DD
	X0,Y0,DD
		For these two instructions, the first operand is treated as signed and the second as unsigned.
		
		Multiplication result may not be inverted
		
		F = A,B
		DD = X0,Y1,Y0

Timing: 2 oscillator clock cycles
Memory: 1 program word

MOVE Introduction to DSP56800 Moves MOVE

Description: The DSP56800 Family instruction set contains a powerful set of moves, resulting not only in better DSP performance, but in simpler, more efficient general-purpose computing. The powerful set of controller and DSP moves results not only in ease of programming, but in more efficient code, that, in turn, results in reduced power consumption for an application. This description gives an introduction to all of the different types of moves available on the DSP56800 architecture. It covers all of the variations of the MOVE instruction, as well as all of the parallel moves. There are eight types of moves available on the DSP56800:

- Any register ↔ any register
- Any register ↔ X data memory
- Any register ↔ on-chip peripheral register
- Immediate data → any register
- Immediate data → X data memory
- Immediate data → on-chip peripheral register
- Register ↔ program memory
- One X data memory access in parallel with an arithmetic operand (single parallel move)
- Two X data memory reads in parallel with an arithmetic operand (dual parallel read)
- Two X data memory reads in parallel with no arithmetic operand specified (MOVE only)
- Conditional register transfer (transfer only if condition is true)
- Register transfer through the data ALU

The move types above are discussed in detail under the following DSP56800 instructions:

MOVE:

- One X data memory access in parallel with an arithmetic operand (single parallel move)
- Two X data memory reads in parallel with an arithmetic operand (dual parallel read)
- Two X data memory reads in parallel with no arithmetic operand specified (MOVE only)

MOVE(C):

- Any register ↔ any register
- Any register ↔ X data memory
- Any register ↔ on-chip peripheral register

MOVE(I):

- Immediate data → any register
- Immediate data → X data memory

MOVE(M):

- Two X data memory reads in parallel with no arithmetic operand specified

MOVE(P):

- Register ↔ on-chip peripheral register
- Immediate data → on-chip peripheral register

MOVE(S):

- Register ↔ first 64 locations of X data memory
- Immediate data → first 64 locations of X data memory

Tcc:

- Conditional register transfer (transfer only if condition is true)

TFR:

- Register transfer through the data ALU

MOVE Introduction to DSP56800 Moves MOVE

Two types of parallel moves are permitted—register-to-memory moves and dual memory-to-register moves. Both types of parallel moves use a restricted subset of all available DSP56800 addressing modes, and the registers available for the move portion of the instruction are also a subset of the total set of DSP core registers. These subsets include the registers and addressing modes most frequently found in high performance numeric computation and DSP algorithms. Also, the parallel moves allow a move to occur *only* with an arithmetic operation in the data ALU. A parallel move is not permitted, for example, with a JMP, LEA, or BFSET instruction.

Since the on-chip peripheral registers are accessed as locations in X data memory, there are many move instructions which can access these peripheral registers. Also, the case of “No Move Specified” for arithmetic operations optionally allows a parallel move.

When a 36-bit accumulator (A or B) is specified as a source operand (S), there is a possibility that the data may be limited. If the data out of the accumulator indicates that the accumulator extension bits are in use, and the data is to be moved into a 16-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 16-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 36-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the CCR is latched (i.e., is sticky).

When a 36-bit accumulator (A or B) is specified as a destination operand (D), any 16-bit source data to be moved into that accumulator is automatically extended to 36 bits by sign-extending the MSB of the source operand (bit 15) and appending the source operand with 16 LS zeros. The automatic sign-extension and zeroing features may be circumvented by specifying the destination register to be one of the individual 16-bit accumulator registers (A1 or B1).

The MOVE, MOVE(C), MOVE(I), MOVE(M), MOVE(P), and MOVE(S) descriptions are found on the following pages. Detailed descriptions of the two parallel move types are covered under the MOVE instruction. The Tcc and TFR descriptions are covered in their respective sections.

MOVE Parallel Move—Single Parallel Move MOVE

Operation:		**Assembler Syntax:**	
<op>	X:<ea> → D	<op>	X:<ea>,D
<op>	S → X:<ea>	<op>	S,X:<ea>

where <op> refers to any arithmetic instruction which allows parallel moves. Examples include ADD, DECW, MACR, NEG, SUB, TFR, etc.

Description: Perform a data ALU operation and, in parallel, move the specified register from/to X data memory. Two indirect addressing modes may be used (post-increment by one and post-increment by the offset register).

Seventeen data ALU instructions allow the capability of specifying an optional single parallel move. These data ALU instructions have been selected for optimal performance on the critical sections of frequently-used DSP algorithms. A summary of the different data ALU instructions, registers used for the memory move, and addressing modes available for the single parallel move is shown in **Table 6-10 Data ALU Instructions—Single Parallel Move** on page 6-19.

If the arithmetic operation of the instruction specifies a given source register (S) or destination register (D), that same register or portion of that register may be used as a source in the parallel data bus move operation. This allows data to be moved in the same instruction in which it is being used as a source operand by a data ALU operation. That is, *duplicate sources* are allowed within the same instruction. Examples of duplicate sources include:

```
ADD    A,B   A,X:(R2)+    ; A register allowed as source of parallel move
ADD    A,B   X:(R2)+,A    ; A register allowed as destination of parallel move
```

If the arithmetic operation portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 36-bit A or B accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0/B0, A1/B1, A2/B2, or A/B as its destination. That is, *duplicate destinations* are *not* allowed within the same instruction. Examples of duplicate destinations include:

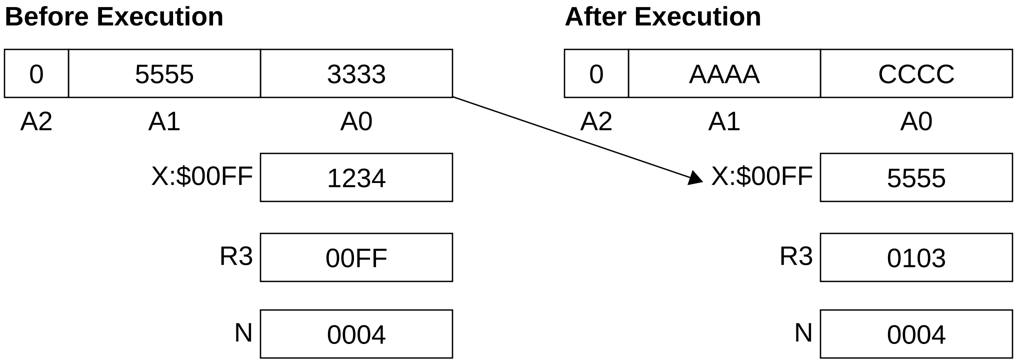
```
ADD    B,A   X:(R2)+,A    ; NOT ALLOWED--A register used twice as a destination
ASL    A     X:(R2)+,A    ; NOT ALLOWED--A register used twice as a destination
```

Exceptions: — TST, CMP, CMPM allow both the accumulator and its lower portion (A and A0, B and B0) to be the parallel move destination even if this accumulator is used by the data ALU operation. These instructions do not have a true destination.

MOVE Parallel Move—Single Parallel Move MOVE

Example:

```
ASL      A      A,X:(R3)+N      ; save old value of A in X:(R3); A*2 → A; update R3
```



Explanation of Example:

Prior to execution, the 16-bit R3 address register contains the value $00FF, the A accumulator contains the value $0:5555:3333, and the 16-bit X memory location X:$00FF contains the value $1234. Execution of the parallel move portion of the instruction, A,X:(R3)+, uses the R3 address register to move the contents of the A1 register before left shifting into the 16-bit X memory location (X:$00FF). R3 is then updated by the value in the N register.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	C

L — Set if data limiting has occurred during parallel move

DATA ALU OPERATION		**PARALLEL MEMORY READ or WRITE**	
Operation	**Registers**	**Mem Access**	**Src/Dest**
OPERATION	**OPERANDS**	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2 + mv oscillator clock cycles for all instructions of this type

Memory: 1 program word for all instructions of this type

MOVE Parallel Move—Dual Parallel Reads MOVE

Operation:			**Assembler Syntax:**		
<op>	X:<ea> → D1	X:<ea> → D2	<op>	X:<ea>,D1	X:<ea>,D2
MOVE	X:<ea> → D1	X:<ea> → D2	MOVE	X:<ea>,D1	X:<ea>,D2

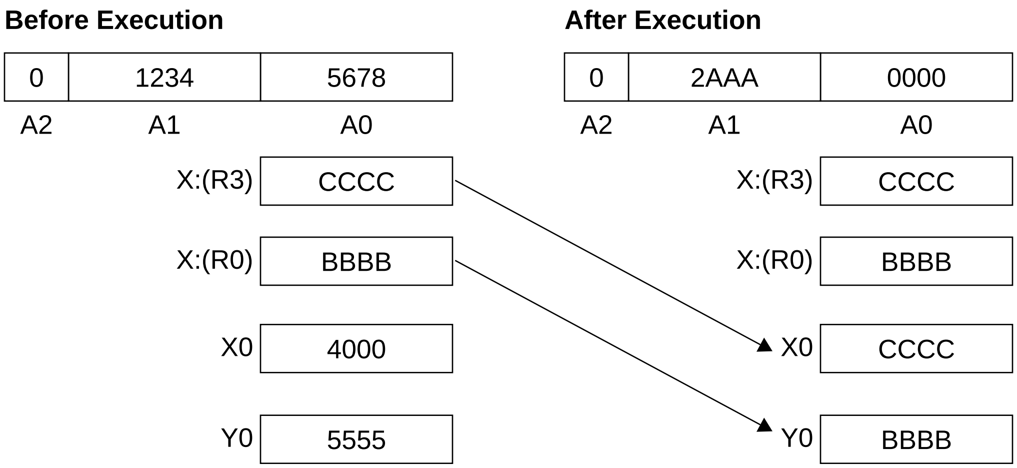
where <op> refers to a limited set of arithmetic instructions which allow double parallel reads.

Description: Read two 16-bit word operands from X memory. Two independent effective addresses (ea) can be specified where one of the effective addresses uses the R0 or R1 address register, while the other effective address must use address register R3. Two parallel address updates are then performed for each effective address. The address update on R3 is only performed using linear arithmetic, and the address update on R0 or R1 is performed using linear or modulo arithmetic.

Six data ALU instructions (ADD, MAC, MACR, MPY, MPYR, and SUB) allow the capability of specifying an optional dual memory read. In addition, MOVE can also be specified. These data ALU instructions have been selected for optimal performance on the critical sections of frequently used DSP algorithms. A summary of the different data ALU instructions, registers used for the memory move, and addressing modes available for the dual parallel read is shown in **Table 6-9 Data ALU Instructions—Dual Parallel Read** on page 6-18. When the MOVE instruction is selected, only the dual memory accesses occur—no arithmetic operation is performed.

Example:

```
MPYR X0,Y0,A    X:(R0)+,Y0       X:(R3)+,X0
```



Explanation of Example:

Prior to execution, the 16-bit X0 register contains the value $4000, and the 16-bit Y0 register contains the value $5555. Execution of the parallel move portion of the instruction X:(R0)+,Y0 X:(R3)+,X0 moves the 16-bit value in the X memory location X:(R0) into the register Y0, moves the 16-bit X memory location X:(R3) into the register X0, and post-increments by one the 16-bit values in the R0 and R3 address registers. The multiplication is performed with the old values of X0 and Y0, and the result is convergently rounded before storing it in the accumulator.

Note: The second X data memory parallel read using the R3 address register can never access off-chip memory or on-chip peripherals. It can only access on-chip X data memory.

MOVE Parallel Move—Dual Parallel Reads MOVE

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	C

L — Set if data limiting has occurred during parallel move

Instruction Fields:

DATA ALU OPERATION		FIRST AND SECOND MEMORY READS		DESTINATIONS FOR MEMORY READS	
Operation	**Registers**	**Read1**	**Read2**	**Dest1**	**Dest2**
OPERATION	**OPERANDS**	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
				Y1	X0
				Valid destinations for Read1	Valid destinations for Read2

DATA ALU OPERATION	FIRST AND SECOND MEMORY READS		DESTINATIONS FOR MEMORY READS	
Operation	**Read1**	**Read2**	**Dest1**	**Dest2**
MOVE	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
			Y1	X0
			Valid destinations for Read1	Valid destinations for Read2

Timing: 2 + mv oscillator clock cycles for all instructions of this type
Memory: 1 program word for all instructions of this type

MOVE(C) Move Control Register MOVE(C)

Operation:	Assembler Syntax:	
X:<ea>→ D	MOVE(C)	X:<ea>,D
S1→ X:<ea>	MOVE(C)	S,X:<ea>
S → D	MOVE(C)	S,D

Description: Move the contents of the specified source (control) register (S) to the specified destination or move the specified source to the specified destination (control) register (D). The control registers S and D consist of the AGU registers, data ALU registers, and the program controller registers. These registers may be moved to or from any other register or location in X data memory.

If the HWS is specified as a destination operand, the contents of the first HWS location are copied into the second and the LF and NL bits are updated accordingly. If the HWS is specified as a source operand, the contents of the second HWS location are copied into the first and the LF and NL bits are updated accordingly. This allows more efficient manipulation of the HWS.

When a 36-bit accumulator (A or B) is specified as a source operand, there is a possibility that the data may be limited. If the data out of the shifter indicates that the accumulator extension register is in use, and the data is to be moved into a 16-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 16-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 36-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the CCR is latched (i.e., is sticky).

When a 36-bit accumulator (A or B) is specified as a destination operand, any 16-bit source data to be moved into that accumulator is automatically extended to 36 bits by sign-extending the MSB of the source operand (bit 15) and appending the source operand with 16 LS zeros. The automatic sign-extension and zeroing features may be circumvented by specifying the destination register to be one of the individual 16-bit accumulator registers (A1 or B1).

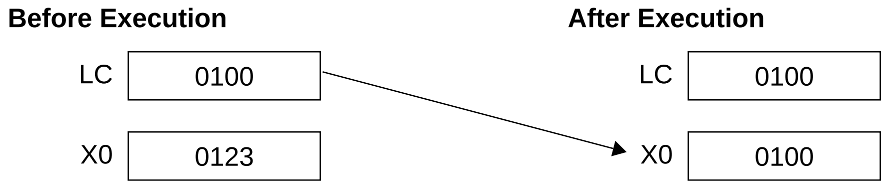
Note: Due to pipelining, if an address register (Rn, SP, or M01) is changed with a MOVE or bit-field instruction, the new contents will not be available for use as a pointer until the second following instruction. If the SP is changed, no PUSH or POP instructions are permitted until the second following instruction.

Note: If the N address register is changed with a MOVE instruction, this register's contents *will* be available for use on the immediately following instruction. In this case the instruction which writes the N address register will be stretched one additional instruction cycle. This is true for the case when the N register is used by the immediately following instruction; if N is not used, then the instruction is not stretched an additional cycle. If the N address register is changed with a bit-field instruction, the new contents *will not* be available for use until the second following instruction.

MOVE(C) Move Control Register MOVE(C)

Example:

```
MOVE(C)         LC,X0           ; move the LC register into the X0 register
```

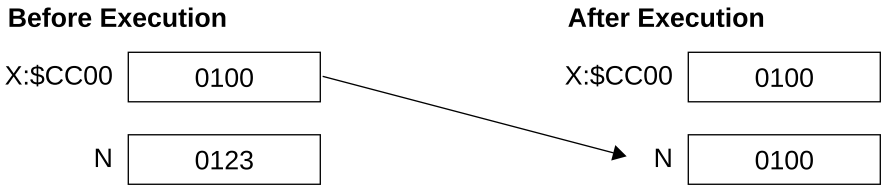


Explanation of Example:

Execution of the MOVE(C) instruction moves the contents of the program controller’s 16-bit LC register into the data ALU’s 16-bit X0 register.

Example:

```
MOVE(C)         X:$CC00,N       ; move X data memory value into the N register
```

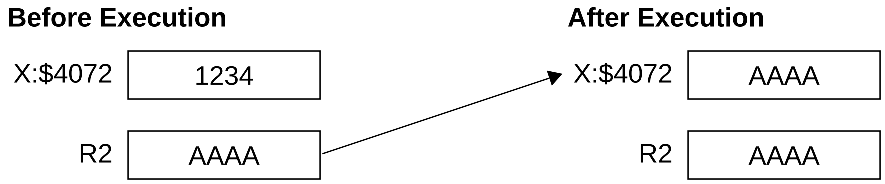


Explanation of Example:

Execution of the MOVE(C) instruction moves the contents of the X data memory at location $CC00 into the AGU’s 16-bit N register.

Example:

```
MOVE(C)         R2,X:(R3+$3072) ; move R2 register into X data memory
```



Explanation of Example:

Prior to execution, the contents of R3 is $1000. Execution of the MOVE(C) instruction moves the AGU’s 16-bitR2 register contents into the X data memory at the location $4072.

Restrictions:

— A MOVE(C) instruction used within a DO loop which specifies the HWS as the source or the SR or HWS as the destination cannot begin at the LA-2, LA-1, or LA within that DO loop.

— A MOVE(C) instruction which specifies the HWS as the source or as the destination cannot be used immediately before a DO instruction.

— A MOVE(C) instruction which specifies the HWS as the source or the SR or HWS as the destination cannot be used immediately before an ENDDO instruction.

— A MOVE(C) instruction which specifies the SR, HWS, or SP as the destination cannot be used immediately before an RTI or RTS instruction.

— A MOVE(C) HWS,HWS instruction is illegal and cannot be used.

MOVE(C) Move Control Register MOVE(C)

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

If D is the SR:

L — Set according to bit 6 of the source operand
E — Set according to bit 5 of the source operand
U — Set according to bit 4 of the source operand
N — Set according to bit 3 of the source operand
Z — Set according to bit 2 of the source operand
V — Set according to bit 1 of the source operand
C — Set according to bit 0 of the source operand

If D1 and D2 are not SR:

L — Set if data limiting has occurred during move

MOVE(C) Move Control Register MOVE(C)

Instruction Fields:

OPERATION	Src/Dest	Src/Dest	COMMENTS	Cyc/Wd
MOVE(C)	X:(Rn) X:(Rn)+ X:(Rn)- X:(Rn)+N X:(SP) X:(SP)+ X:(SP)- X:(SP)+N	Any register	—	(2/1)
	X:xxxx	Any register	16-bit absolute address	(4/2)

	X:(Rn+N)
	X:(SP+N)
		Any register	—	(4/1)

	X:(Rn+xxxx)
	X:(SP+xxxx)
		Any register	Signed 16-bit index	(6/2)

	X:(R2+xx)
	X:(SP-xx)
		X0, Y1, Y0,
		A, B, A1, B1
		R0-R3, N
			—	(4/1)

	Any register	Any register	—	(2/1)

Timing: 2 + mvc oscillator clock cycles
Memory: 1 + ea program words

MOVE(I) Move Immediate MOVE(I)

Operation:	Assembler Syntax:	
#xx → D	MOVE(I)	#xx,D
#xxxx → D	MOVE(I)	#xxxx,D
#xxxx → X:<ea>	MOVE(I)	#xxxx,X:<ea>

Description: The 7-bit signed immediate operand is stored in the lowest 7 bits of the destination (D), and the upper bits are filled with sign extension. The destination can be any register, X data memory location, or on-chip peripheral register.

Example:

```
MOVE(I)  #<$FFC7,X0              ; moves negative value into X0 since bit 6 is 1
```

Before Execution		After Execution	
X0	1234	X0	FFC7

Explanation of Example:

Prior to execution, X0 contains the value $1234. Execution of the instruction moves the value $FFC7 into X0.

Example:

```
MOVE(I)  #$C33C,X:$A009          ; moves 16-bit value directly into a memory location
```

Before Execution		After Execution	
X:$A009	1234	X:$A009	C33C

Explanation of Example:

Prior to execution, the X data memory location $A009 contains the value $1234. Execution of the instruction moves the value $C33C into this memory location.

Note: The MOVE(P) and MOVE(S) instructions also provide a mechanism for loading 16-bit immediate values directly into the last 64 and first 64 locations, respectively, in X data memory.

Condition Codes Affected:

The condition codes are not affected by this instruction.

MOVE(I) Move Immediate MOVE(I)

Instruction Fields:

OPERATION	DESTINATION	COMMENTS	Cyc/Wd
MOVE(I) #xx,	X0, Y1, Y0, A, B, A1, B1 R0-R3, N	Signed 7-bit integer data (data is put in the LSP)	(2/1)
MOVE(I) #xxxx,	Any register	Signed 16-bit immediate data	(4/2)

MOVE(I) #xxxx,	X:(R2+xx)
	X:(SP-xx)
		Signed 16-bit immediate data	(6/2)

MOVE(I) #xxxx,	X:xxxx	Signed 16-bit immediate data	(6/3)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

MOVE(M) Move Program Memory MOVE(M)

Operation:

P:<ea> → D
S→ P:<ea>

Assembler Syntax:

MOVE(M)	P:<ea>,D
MOVE(M)	S,P:<ea>

Description: Move the specified register from/to the specified program memory location. The source register (S) and destination registers (D) are data ALU registers.

When a 36-bit accumulator (A or B) is specified as a source operand, there is a possibility that the data may be limited. If the data out of the shifter indicates that the accumulator extension register is in use, and the data is to be moved into a 16-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 16-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 36-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the CCR is latched (i.e., is sticky).

When a 36-bit accumulator (A or B) is specified as a destination operand, any 16-bit source data to be moved into that accumulator is automatically extended to 36 bits by sign-extending the MSB of the source operand (bit 15) and appending the source operand with 16 LS zeros. The automatic sign-extension and zeroing features may be circumvented by specifying the destination register to be one of the individual 16-bit accumulator registers (A1 or B1).

Example:

```
MOVE(M)         P:(R2)+N,A      ; move P:(R2) into the A accumulator, update R2 with N
```

Before Execution

A	1234	5678
A2	A1	A0

P:$0077	0116
R2	$0077

After Execution

0	0116	0000
A2	A1	A0

P:$0077	0116
R2	$007A

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $A:1234:5678, R2 contains the value $0077, the N register contains the value $0003, and the 16-bit program memory location P:(R2) contains the value $0116. Execution of the MOVE(M) instruction moves the 16-bit program memory location P:(R2) into the 36-bit A accumulator. R2 is then post-incremented by N.

MOVE(M) Move Program Memory MOVE(M)

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	C

L — Set if data limiting has occurred during the move

Instruction Fields:

OPERATION	Src/Dest	Src/Dest
MOVE(M)	P:(Rn)+ P:(Rn)+N	X0, Y1, Y0, A, B, A1, B1 R0-R3, N

Timing: 8 + mvm oscillator clock cycles
Memory: 1 program words

MOVE(P) Move Peripheral Data MOVE(P)

Operation:

X:<pp> → D
S → X:<pp>
#xxxx → X:<pp>

Assembler Syntax:

MOVE(P)	X:<pp>,D
MOVE(P)	S,X:<pp>
MOVE(P)	#xxxx,X:<pp>

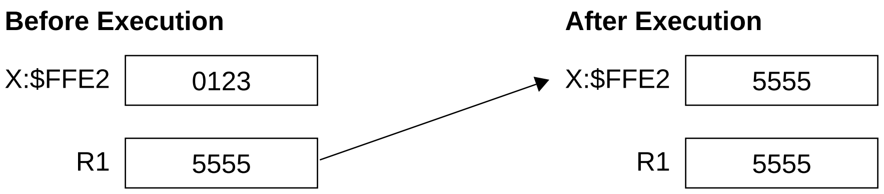
Description: Move the specified operand from/to the on-chip peripheral registers, located in the last 64 locations of the X data memory map. The 6-bit I/O short absolute address is one-extended to generate a 16-bit I/O peripheral address.

When a 36-bit accumulator (A or B) is specified as a source operand, there is a possibility that the data may be limited. If the data out of the shifter indicates that the accumulator extension register is in use, and the data is to be moved into a 16-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 16-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 36-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the CCR is latched (i.e., is sticky).

When a 36-bit accumulator (A or B) is specified as a destination operand, any 16-bit source data to be moved into that accumulator is automatically extended to 36 bits by sign-extending the MSB of the source operand (bit 15) and appending the source operand with 16 LS zeros. The automatic sign-extension and zeroing features may be circumvented by specifying the destination register to be one of the individual 16-bit accumulator registers (A1 or B1).

Example:

```
MOVEP   R1,X:<$FFE2             ; write to on-chip peripheral register at X:$FFE2
```

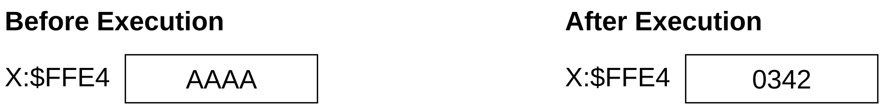


Explanation of Example:

Prior to execution, the 16-bit on-chip peripheral register located at $FFE2 contains the value $0123. Execution of the MOVE(P) R1,X:<$FFE2 instruction moves the value $5555 contained in the R1 register into the peripheral register.

Example:

```
MOVEP   #$0342,X:<$24           ; moves 16-bit value directly into a peripheral register
```



Explanation of Example:

Prior to execution, the on-chip peripheral register located at X data memory location $FFE4 contains the value $AAAA. The MOVEP one-extends the value $24 to form the peripheral address $FFE4. Execution of the instruction moves the value $0342 into this location.

MOVE(P) Move Peripheral Data MOVE(P)

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	C

L — Set if data limiting has occurred during move

Note: It is also possible to access the last 64 locations in the X data memory map using the MOVE(C) instruction, which can directly access these locations either using the address register indirect addressing modes or the absolute address addressing mode which specifies a 16-bit absolute address.

Instruction Fields:

OPERATION	Src/Dest	Src/Dest	COMMENTS
MOVE(P)	X:<pp>	X0, Y1, Y0, A, B, A1, B1 R0-R3, N	<pp>: last 64 locations of X memory

OPERATION	DESTINATION	COMMENTS	Cyc/Wd
MOVE(P) #xxxx,	X:<pp>	Signed 16-bit immediate data	(4/2)

Timing: 2 + mvp oscillator clock cycles
Memory: 1 program word

MOVE(S) Move Absolute Short MOVE(S)

Operation:	Assembler Syntax:	
X:<aa> → D	MOVE(S)	X:<aa>,D
S → X:<aa>	MOVE(S)	S,X:<aa>
#xxxx → X:<aa>	MOVE(S)	#xxxx,X:<aa>

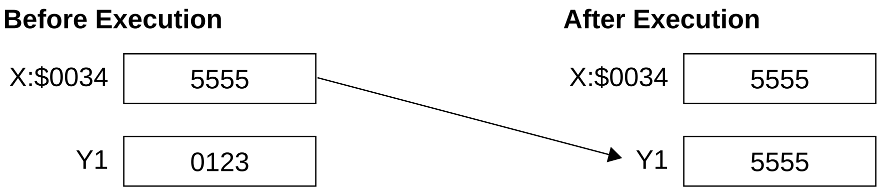
Description: Move the specified operand from/to the first 64 memory locations in X data memory. The 6-bit absolute short address is zero-extended to generate a 16-bit X data memory address.

When a 36-bit accumulator (A or B) is specified as a source operand, there is a possibility that the data may be limited. If the data out of the shifter indicates that the accumulator extension register is in use, and the data is to be moved into a 16-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 16-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 36-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the CCR is latched (i.e., is sticky).

When a 36-bit accumulator (A or B) is specified as a destination operand, any 16-bit source data to be moved into that accumulator is automatically extended to 36 bits by sign-extending the MSB of the source operand (bit 15) and appending the source operand with 16 LS zeros. The automatic sign-extension and zeroing features may be circumvented by specifying the destination register to be one of the individual 16-bit accumulator registers (A1 or B1).

Example:

```
MOVES   X:<$0034,Y1              ; write to X:$0034
```



Explanation of Example:

Prior to execution, X:$0034 contains the value $5555 and Y1 contains the value $0123. Execution of the instruction moves the value $5555 into the Y1 register.

Example:

```
MOVES   #$0342,X:<$24            ; moves 16-bit value directly into memory location
```

Before Execution
After Execution
X:$0024
AAAA
X:$0024
0342

Explanation of Example:

Prior to execution, the contents of the X data memory location $0024 contains the value $AAAA. The MOVES zero-extends the value $24 to form the memory address $0024. Execution of the instruction moves the value $0342 into this location.

MOVE(S) Move Absolute Short MOVE(S)

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	C

L — Set if data limiting has occurred during move

Note: It is also possible to access the first 64 locations in the X data memory using the MOVE(C) instruction, which can directly access these locations either using the address register indirect addressing modes or the absolute address addressing mode which specifies a 16-bit absolute address.

Instruction Fields:

OPERATION	Src/Dest	Src/Dest	COMMENTS
MOVE(S)	X:<aa>	X0, Y1, Y0, A, B, A1, B1 R0-R3, N	<aa>: First 64 locations of X memory

OPERATION	DESTINATION	COMMENTS	Cyc/Wd
MOVE(S) #xxxx,	X:<aa>	Signed 16-bit immediate data	(4/2)

Timing: 2 + mvs oscillator clock cycles
Memory: 1 program word

MPY Signed Multiply MPY

Operation:

<u>+</u> S1 * S2 → D (no parallel move)
S1 * S2 → D (one parallel move)
S1 * S2 → D (two parallel reads)

Assembler Syntax:

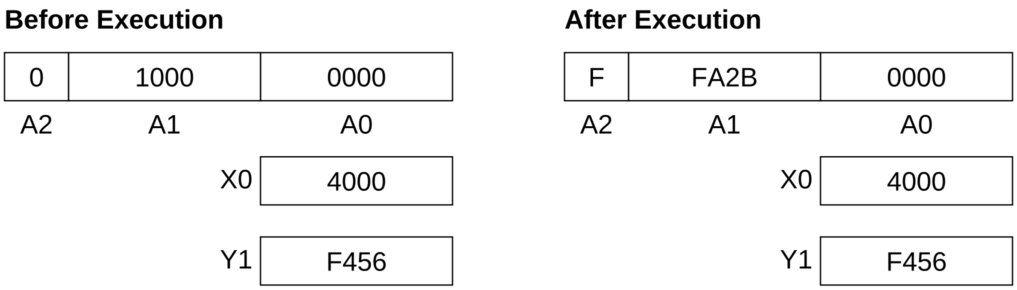
MPY	(<u>+</u>)S1,S2,D	(no parallel move)
MPY	S1,S2,D	(one parallel move)
MPY	S1,S2,D	(two parallel reads)

Description: Multiply the two signed 16-bit source operands (S1 and S2), and store the product in the specified 36-bit destination accumulator (D). The "-" sign option is used to negate the specified product. This option is not available when a single parallel move or two parallel read operations are performed or when D is the 16-bit X0, Y1, or Y0.

Usage: This instruction is used for multiplication of fractional data or integer data when a full 32-bit product is required (see **Integer Multiplication** on page 3-23). When the destination is a 16-bit register, this instruction is useful only for fractional data.

Example:

```
MPY       X0,Y1,A                  ; multiply X0 by Y1
```



Explanation of Example:

Prior to execution, the 16-bit X0 register contains the value $4000 (0.5), the 16-bit Y1 register contains the value $F456 (-0.0911255), and the 36-bit A accumulator contains the value $00:1000:0000 (0.125). Execution of the MPY X0,Y1,A instruction multiplies the 16-bit signed value in the X0 register by the 16-bit signed value in Y1 and stores the result ($F:FA2B:0000) into the A accumulator (X0 * Y1 = -0.045562744140625).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	C

L — Set if limiting (parallel move) or overflow (result) has occurred
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

MPY Signed Multiply MPY

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
MPY	Y0,Y0,F Y1,Y0,F A1,Y0,F B1,Y1,F Y1,X0,F Y0,X0,F	Allows multiplication result to be inverted if desired. F = A,B
	Y0,Y0,DD Y1,Y0,DD A1,Y0,DD B1,Y1,DD Y1,X0,DD Y0,X0,DD	Multiplication result may not be inverted. DD = X0,Y1,Y0

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
MPY	X0,F Y1,F Y0,F A,B B,A (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

DATA ALU OPERATION		FIRST & SECOND MEMORY READS		DESTINATIONS FOR MEMORY READS	
Operation	**Registers**	**Read1**	**Read2**	**Dest1**	**Dest2**
MPY	X0,A Y1,A Y0,A X0,B Y1,B Y0,B	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
				Y1	X0
				Valid destinations for Read1	Valid destinations for Read2

Timing: 2 + mv oscillator clock cycles for MPY instructions with a parallel move.
Refer to previous table for MPY instructions without a parallel move.

Memory: 1 program word for MPY instructions with a parallel move.
Refer to previous table for MPY instructions without a parallel move.

MPYR Signed Multiply and Round MPYR

Operation:

± S1 * S2 + r → D (no parallel move)
± S1 * S2 + r → D (one parallel move)
S1 * S2 + r → D (two parallel reads)

Assembler Syntax:

MPYR	(±)S1,S2,D	(no parallel move)
MPYR	S1,S2,D	(one parallel move)
MPYR	S1,S2,D	(two parallel reads)

Description: Multiply the two signed 16-bit source operands (S1 and S2), round the result using the specified rounding, and store it in the specified 36-bit destination accumulator (D). (Refer to RND for more complete information on the convergent rounding process.) The “-” sign option is used to negate the specified product. This option is not available when a single parallel move or two parallel reads are performed or when D is the 16-bit X0, Y1, or Y0. The default sign option is “+”.

Usage: This instruction is used for multiplication and rounding of fractional data.

Example:

```
MPYR     -X0,Y1,A              ; multiply X0 by Y1 and negate the product
```

Before Execution

A2	A1	A0
0	1000	1234

X0	4000
Y1	F456

After Execution

A2	A1	A0
F	FE8B	0000

X0	4000
Y1	F456

Explanation of Example:

Prior to execution, the 16-bit X0 register contains the value $4000 (0.5), the 16-bit Y1 register contains the value $F456 (-0.0911255), and the 36-bit A accumulator contains the value $00:1000:1234 (0.125002169981599). Execution of the MPYR -X0,Y1,A instruction multiplies the 16-bit signed value in the X0 register by the 16-bit signed value in Y1, rounds the result, and stores the result ($FF:FE8B:0000) into the A accumulator (-X0 * Y1 = -0.011383056640625). In this example, the default rounding (convergent rounding) is performed.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	C

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

MPYR Signed Multiply and Round MPYR

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
MPYR	Y0,Y0,F Y1,Y0,F A1,Y0,F B1,Y1,F Y1,X0,F Y0,X0,F	Allows multiplication result to be inverted if desired. F = A,B
	Y0,Y0,DD Y1,Y0,DD A1,Y0,DD B1,Y1,DD Y1,X0,DD Y0,X0,DD	Multiplication result may not be inverted. DD = X0,Y1,Y0

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
MPYR	X0,F Y1,F Y0,F A,B B,A (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

DATA ALU OPERATION		FIRST & SECOND MEMORY READS		DESTINATIONS FOR MEMORY READS	
Operation	**Registers**	**Read1**	**Read2**	**Dest1**	**Dest2**
MPYR	X0,A Y1,A Y0,A X0,B Y1,B Y0,B	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
				Y1	X0
				Valid destinations for Read1	Valid destinations for Read2

Timing: 2 + mv oscillator clock cycles for MPYR instructions with a parallel move.
Refer to previous table for MPYR instructions without a parallel move.

Memory: 1 program word for MPYR instructions with a parallel move.
Refer to previous table for MPYR instructions without a parallel move.

MPYSU Signed Unsigned Multiply MPYSU

Operation:

S1 * S2 → D (S1 signed, S2 unsigned)

Assembler Syntax:

MPYSU S1,S2,D (no parallel move)

Description: Multiply the two 16-bit source operands (S1 and S2), and store the product in the specified 36-bit destination accumulator (D). S1 can be unsigned; S2 is always considered unsigned. This mixed arithmetic multiply does not allow a parallel move and can be used for multi-precision multiplications.

Usage: In addition to single-precision multiplication of a signed value times unsigned value, this instruction is also used for multi-precision multiplications, as shown in **Multi-precision Multiplication** on page 3-27.

Example:

```
MPYSU   X0,Y0,A
```

Before Execution

0	0000	0000
A2	A1	A0

X0	3456
Y0	8000

After Execution

0	3456	0000
A2	A1	A0

X0	3456
Y0	8000

Explanation of Example:

The 16-bit X0 register contains the value $3456, and the 16-bit Y1 register contains the value $8000.

Execution of the MPYSU X0,Y0,A instruction multiplies the 16-bit signed value in the X0 register by the 16-bit unsigned value in Y0 and stores the signed result into the A accumulator. If this was a MPY instruction, Y0 ($8000) would equal -1.0, and the multiplication result would be $F:CBAA:0000. Since this is a MPYSU instruction, Y0 is considered unsigned and equals +1.0. This gives a multiplication result of $0:3456:0000.

MPYSU Signed Unsigned Multiply MPYSU

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	**E**	**U**	**N**	**Z**	**V**	C

E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERANDS	COMMENTS

MPYSU	Y0,Y0,F
	Y0,Y1,F
	Y0,A1,F
	Y1,B1,F
	X0,Y1,F
	X0,Y0,F
	Y0,Y0,DD
	Y0,Y1,DD
	Y0,A1,DD
	Y1,B1,DD
	X0,Y1,DD
	X0,Y0,DD
		For these two instructions, the first operand is treated as signed and the second as unsigned.
		Multiplication result may not be inverted.
		F = A,B
		DD = X0,Y1,Y0

Timing: 2 oscillator clock cycles
Memory: 1 program word

NEG Negate Accumulator NEG

Operation:			**Assembler Syntax:**		
0 - D →	D	(parallel move)	NEG	D	(parallel move)

Description: The destination operand (D) is subtracted from zero, and the result is stored in the destination accumulator.

Usage: This instruction is used for negating a 36-bit accumulator. It can also be used to negate a 16-bit value loaded in the MSP of an accumulator if the LSP of the accumulator is $0000 (see **Unsigned Load of an Accumulator** on page 8-10).

Example:

```
NEG      B        X1,X:(R3)+       ; 0-B → B, save X1, update R3
```

Before Execution

0	1234	5678
B2	B1	B0

SR	0300

After Execution

F	EDCB	A988
B2	B1	B0

SR	0309

Explanation of Example:

Prior to execution, the 36-bit B accumulator contains the value $0:1234:5678. The NEG B instruction takes the two's-complement of the value in the B accumulator and stores the 36-bit result back in the B accumulator.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a borrow is generated from the MSB of the result

See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

NEG Negate Accumulator NEG

Instruction Fields:

OPERATION	OPERAND
NEG	A B

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
NEG	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

NOP No Operation NOP

Operation:

PC+1 → PC

Assembler Syntax:

NOP

Description: Increment the PC. Pending pipeline actions, if any, are completed. Execution continues with the instruction following the NOP.

Example:

```
NOP                    ;increment the program counter
```

Explanation of Example:

The NOP instruction increments the PC and completes any pending pipeline actions.

Condition Codes Affected:

The condition codes are not affected by this instruction.

NOP No Operation NOP

Instruction Fields:

OPERATION
NOP

Timing: 2 oscillator clock cycles
Memory: 1 program word

NORM Normalize Accumulator Iteration NORM

Operation:

If $(\overline{E} \bullet U \bullet \overline{Z} = 1)$ then ASL D and Rn - 1 → Rn
else if (E = 1) then ASR D and Rn + 1→ Rn
else NOP

where $\overline{X}$ denotes the logical complement of X, and
where • denotes the logical AND operator

Assembler Syntax:

NORM R0,D

Description: Perform one normalization iteration on the specified destination operand (D), update the address register R0 based upon the results of that iteration, and store the result back in the destination accumulator. This is a 36-bit operation. If the accumulator extension is not in use, the accumulator is unnormalized, and the accumulator is not zero, then the destination operand is arithmetically shifted one bit to the left, and the specified address register is decremented by one. If the accumulator extension register is in use, the destination operand is arithmetically shifted one bit to the right, and the specified address register is incremented by one. If the accumulator is normalized or zero, a NOP is executed, and the specified address register is not affected. Since the operation of the NORM instruction depends on the E, U, and Z CCR bits, these bits must correctly reflect the current state of the destination accumulator prior to executing the NORM instruction. The L and V bits in the CCR will be cleared unless they have been improperly set up prior to executing the NORM instruction.

Example:

```
TST     A
REP     #31                    ;maximum number of iterations (31) needed
NORM    R0,A                   ;perform one normalization iteration
```

Before Execution

0	0000	8000
A2	A1	A0

R0	0000

After Execution

0	4000	0000
A2	A1	A0

R0	FFF1

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $0:0000:8000, and the 16-bit R0 address register contains the value $0000. The repetition of the NORM R0,A instruction normalizes the value in the 36-bit accumulator and stores the resulting number of shifts performed during that normalization process in the R0 address register. A negative value reflects the number of left shifts performed, while a positive value reflects the number of right shifts performed during the normalization process. In this example, fifteen left shifts are required for normalization.

NORM Normalize Accumulator Iteration NORM

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	v	**E**	**U**	**N**	**Z**	**V**	C

L — Set if overflow has occurred in A or B result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if bit 35 is changed as a result of a left shift

See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

OPERATION	OPERANDS
NORM	R0,A R0,B

Timing: 2 oscillator clock cycles
Memory: 1 program word

Descriptions

NOT Logical Complement NOT

Operation:			Assembler Syntax:		
$\overline{D} \rightarrow D$		(no parallel move)	NOT	D	(no parallel move)
$\overline{D}[31:16] \rightarrow$	D[31:16]	(no parallel move)	NOT	D	(no parallel move)

where the bar over the D ($\overline{D}$) denotes the logical NOT operator

Description: Take the one’s-complement of the destination operand (D), and store the result in the destination. This instruction is a 16-bit operation. If the destination is a 36-bit accumulator, the one’s-complement is performed on bits 31-16 of the accumulator. The remaining bits of the destination accumulator are not affected.

Example:

```
NOT A    A,X:(R2)+              ; save A1 and take the 1’s complement of A1
```

Before Execution

5	1234	5678
A2	A1	A0

SR	0300

After Execution

5	EDCB	5678
A2	A1	A0

SR	0300

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $5:1234:5678. The NOT A instruction takes the one’s-complement of bits 31-16 of the A accumulator (A1) and stores the result back in the A1 register. The remaining A accumulator bits are not affected.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of A or B result is set
Z — Set if bits 31-16 of A or B result are zero
V — Always cleared

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

NOT Logical Complement NOT

Instruction Fields:

OPERATION	OPERAND
NOT	A B X0 Y1 Y0

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

Descriptions

NOTC Logical Complement with Carry NOTC

Operation:

$\overline{\text{X:<ea>}} \rightarrow \text{X:(ea)}$
$\overline{\text{D}} \rightarrow \text{D}$

Assembler Syntax:

```
NOTC     X:<ea>
NOTC     D
```

Description: Take the one’s complement of the destination operand (D), and store the result in the destination. This instruction is a 16-bit operation. If the destination is a 36-bit accumulator, the one’s-complement is performed on bits 31-16 of the accumulator. The remaining bits of the destination accumulator are not affected. C is also modified as described below.

Example:

```
NOTC     R2
```

	Before Execution		After Execution
R2	CAA3	R2	355C
SR	3456	SR	3456

Explanation of Example:

Prior to execution, the R2 register contains the value $CAA3. Execution of the instruction complements the value in R2. C is modified as described below.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	N	Z	V	**C**

For destination operand SR:

? — Changed if specified in the field

For other destination operands:

C — Set if the value equals $FFFF before the complement

NOTC Logical Complement with Carry NOTC

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
NOTC	X:(R2+xx)	xx = [0 to 63]	(6/2)
	X:(SP-xx)	xx = [1 to 64]	(6/2)
	X:<aa>	First 64 words of X memory	(4/2)
	X:<pp>	Last 64 words of X memory	(4/2)
	X:xxxx	16-bit absolute address	(6/3)
	Any register except the HWS	—	(4/2)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

OR Logical Inclusive OR OR

Operation: **Assembler Syntax:**

Operation		Assembler Syntax		
S + D → D	(no parallel move)	OR	S,D	(no parallel move)
S + D[31:16] → D[31:16]	(no parallel move)	OR	S,D	(no parallel move)

where + denotes the logical inclusive OR operator

Description: Logically OR the source operand (S) with the destination operand (D), and store the result in the destination. This instruction is a 16-bit operation. If the destination is a 36-bit accumulator, the source is ORed with bits 31-16 of the accumulator.The remaining bits of the destination accumulator are not affected.

Usage: This instruction is used for the logical OR of two registers. If it is desired to OR a 16-bit immediate value with a register or memory location, then the ORC instruction is appropriate.

Example:

```
OR        Y1,B                     ; OR Y1 with B
```

Before Execution

0	1234	5678
B2	B1	B0

Y1	FF00

After Execution

0	FF34	5678
B2	B1	B0

Y1	FF00

Explanation of Example:

Prior to execution, the 16-bit Y1 register contains the value $FF00, and the 36-bit B accumulator contains the value $0:1234:5678. The OR Y1,B instruction logically OR's the 16-bit value in the Y1 register with B1, and stores the 36-bit result in the B accumulator.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of A or B result is set
Z — Set if bits 31-16 of A or B result are zero
V — Always cleared

OR Logical Inclusive OR OR

Instruction Fields:

OPERATION	OPERANDS	COMMENTS

OR	X0,F
	Y1,F
	Y0,F
		F = A or B
		F1 = A1 or B1

	F1,X0 F1,Y1 F1,Y0	
	Y0,X0 Y1,X0 X0,Y0 Y1,Y0 X0,Y1 Y0,Y1	

Timing: 2 mv oscillator clock cycles
Memory: 1 program word

ORC Logical Inclusive OR Immediate ORC

Operation:		**Assembler Syntax:**	
#xxxx + X:<ea> → X:<ea>		ORC	#iiii,X:<ea>
#xxxx + D → D		ORC	#iiii,D

where + denotes the logical inclusive OR operator

Description: Logically OR a 16-bit immediate data value with the destination operand (D), and store the results back into the destination. C is also modified as described below. This instruction performs a read-modify-write operation on the destination and requires two destination accesses.

Example:

```
ORC       #$5555,X:<<$7C30       ; OR with immediate data
```

	Before Execution		**After Execution**
X:$7C30	00AA	X:$7C30	55FF
SR	0300	SR	0300

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$7C30 contains the value $00AA. Execution of the instruction tests the state of bits 10, 12, 13, 14, 15 in X:$FFE2, does not set C (because all these bits were not set), and then sets the bits.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	N	Z	V	**C**

For destination operand SR:

? — Set as defined in the field and if specified in the field

For other destination operands:

C — Set if the all bits specified by the mask are set

Note: This instruction is the same as a BFSET instruction and uses the same 16-bit immediate mask. This instruction will disassemble as a BFSET instruction**.**

ORC Logical Inclusive OR Immediate ORC

Instruction Fields:

OPERATION	OPERAND	COMMENTS	Cyc/Wd
ORC #iiii	X:(R2+xx)	xx = [0 to 63]	(6/2)
	X:(SP-xx)	xx = [1 to 64]	(6/2)
	X:<aa>	First 64 words of X memory	(4/2)
	X:<pp>	Last 64 words of X memory	(4/2)
	X:xxxx	16-bit absolute address	(6/3)
	Any register except the HWS	—	(4/2)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

POP Pop from Stack POP

Operation:

X:(SP) → D
SP-1 → SP

Assembler Syntax:

POP D

Description: Read one location from the software stack into a destination register(D), and post-decrement the SP.

Example:

```
POP        LC
```

Before Execution		After Execution	
X:$0100	AAAA	X:$0100	AAAA
LC	0099	LC	AAAA
SP	0100	SP	00FF

Explanation of Example:

Prior to execution, the LC register contains the value $0099, and the SP contains the value $0100. The POP instruction reads from the location in X data memory pointed to by the SP and places this value in the LC register. The SP is then decremented after the read from memory.

Condition Codes Affected:

The condition codes are not affected by this instruction.

POP Pop from Stack POP

Instruction Fields:

OPERATION	DESTINATION	COMMENTS
POP	Any register	The PUSH instruction is now a 2-word, 2-cycle macro
	(No register specified)	A single stack location can also be popped with no destination specified

Timing: 2 oscillator clock cycles
Memory: 1 program word

REP Repeat Next Instruction REP

Operation:	Assembler Syntax:
LC → TEMP; #xx → LC Repeat next instruction until LC = 1 TEMP → LC	REP #xx
LC → TEMP; S → LC Repeat next instruction until LC = 1 TEMP → LC	REP S

Description: Repeat the single word instruction immediately following the REP instruction the specified number of times. The value specifying the number of times the given instruction is to be repeated is loaded into the 13-bit LC register. The contents of the 13-bit LC register are treated as unsigned (i.e., always positive). The single word instruction is then executed the specified number of times, decrementing the LC after each execution until LC equals one. When the REP instruction is in effect, the repeated instruction is fetched only one time, and it remains in the instruction register for the duration of the loop count. Thus, the REP instruction is not interruptible. The contents of the LC register upon entering the REP instruction are stored in an internal temporary register and are restored into the LC register upon exiting the REP loop. *If LC is set equal to zero, the instruction is not repeated and execution continues with the instruction immediately following the instruction that was to be repeated*. The instruction's effective address specifies the address of the value that is to be loaded into the LC.

The REP instruction allows all registers on the DSP core to specify the number of loop iterations except for the following: M01, HWS, OMR, and SR. If immediate short data is instead used to specify the loop count, the 6 LSBs of the LC register are loaded from the instruction and the upper 7 MSBs are cleared.

Note: If the A or B accumulator is specified as a source operand, and the data out of the accumulator indicates that extension is in use, the value to be loaded into the LC register will be limited to a 16-bit maximum positive or negative saturation constant. If positive saturation occurs, the limiter places $7FFF onto the bus, and the lower 13 bits of this value are all ones. The thirteen ones are loaded into the LC register as the maximum unsigned positive loop count allowed. If negative saturation occurs, the limiter places $8000 onto the bus, and the lower 13 bits of this value are all 0s. The thirteen 0s are loaded into the LC register, specifying a loop count of zero. The A and B accumulators remain unchanged.

Note: The REP instruction and the REP loop may not be interrupted once in progress until completion of the REP loop.

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 or move word instruction
DO
BRCLR, BRSET
MOVEM
REP,
RTS
SWI, DEBUG
Bcc, Jcc
BRA, JMP
JSR
RTI
STOP, WAIT
Tcc

Also, a REP instruction cannot be the last instruction in a DO loop (at the LA). The assembler will generate an error if any of the above instructions are found immediately following a REP instruction.

REP Repeat Next Instruction REP

Example:

```
REP     X0              ; repeat (X0) times
INCW    Y1              ; increment the Y1 register
```

	Before Execution		After Execution
X0	0003	X0	0003
Y1	0000	Y1	0003
LC	00A5	LC	00A5

Explanation of Example:

Prior to execution, the 16-bit X0 register contains the value $0003, and the 16-bit LC register contains the value $00A5. Execution of the REP X0 instruction takes the lower 13 bits of the value in the X0 register and stores it in the 13-bit LC register. Then, the single word INCW instruction immediately following the REP instruction is repeated $0003 times. The contents of the LC register before the REP loop are restored upon exiting the REP loop.

Example:

```
REP     X0              ; repeat (X0) times
INCW    Y1              ; increment the Y1 register
ASL     Y1              ; multiply the Y1 register by 2
```

	Before Execution		After Execution
X0	0000	X0	0000
Y1	0005	Y1	000A
LC	00A5	LC	00A5

Explanation of Example:

Prior to execution, the 16-bit X0 register contains the value $0000, and the 16-bit LC register contains the value $00A5. Execution of the REP X0 instruction takes the lower 13 bits of the value in the X0 register and stores it in the 13-bit LC register. Since the loop count is zero, the single word INCW instruction immediately following the REP instruction is skipped and execution continues with the ASL instruction. The contents of the LC register before the REP loop are restored upon exiting the REP loop.

REP Repeat Next Instruction REP

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	C

L — Set if data limiting occurred using A or B as source operands

REP Repeat Next Instruction REP

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
REP	$xx	6-bit unsigned short immediate data
	X0,Y1,Y0, R0,R1,R2,R3, N, LC, LA, A2,B2,A1,B1, A0,B0,A,B	Any register (uses 13 LSBs of register) except M01, SR, OMR, and the HWS

Timing: If the argument equals zero, 8 + mv oscillator clock cycles;
if the argument does not equal zero, 6 + mv oscillator clock cycles

Memory: 1 program word

RND Round Accumulator RND

Operation:			**Assembler Syntax:**		
D + r →	D	(parallel move)	RND	D	(parallel move)

Description: Round the 36-bit value in the specified destination operand (D), store the result in the EXT and MSPs of the destination accumulator (A2:A1 or B2:B1), and clear the LSP of the accumulator. This instruction uses the rounding technique selected by the R bit in the OMR. When the R bit in OMR is cleared (default mode), convergent rounding is selected; when the R bit is set, two's-complement rounding is selected. The rounding constant added into bit 15 of the destination. Refer to **Rounding** on page 3-37 for more information about the rounding modes.

Example:

```
RND     A       ; round A accumulator into A2:A1, zero A0
```

	Before Execution			**After Execution**		
I	5	1236	789A	5	1236	0000
	A2	A1	A0	A2	A1	A0

	Before Execution			**After Execution**		
II	0	1236	8000	0	1236	0000
	A2	A1	A0	A2	A1	A0

	Before Execution			**After Execution**		
III	0	1235	8000	0	1236	0000
	A2	A1	A0	A2	A1	A0

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $5:1236:789A for Case I, the value $0:1236:8000 for Case II and the value $0:1235:8000 for Case III. Execution of the RND A instruction rounds the value in the A accumulator into the MSP of the A accumulator (A1) and then zeros the LSP of the A accumulator (A0). The example is given assuming that the convergent rounding is selected. Case II is the special case that distinguishes convergent rounding from the two's-complement rounding, since it clears the LSB of the MSP after the rounding operation is performed.

RND Round Accumulator RND

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	C

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: If the CC bit is set and bit 31 of the result is set, then N is set. If the CC bit is set and bits 31-0 of the result equals zero, then Z is set. The rest of the bits are unaffected by the setting of the CC bit.

Instruction Fields:

OPERATION	OPERANDS
RND	A B

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
RND	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

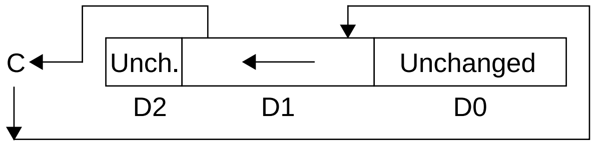
Descriptions

ROL Rotate Left ROL

Assembler Syntax:

```
ROL        D
```

Operation:



(parallel move)

Description: Logically shift 16-bits of the destination operand (D) one bit to the left, and store the result in the destination. If the destination is a 36-bit accumulator, the result is stored in the MSP of the accumulator (A1 or B1), and the remaining portions of the accumulator (A2, B2, A0, B0) are not modified.The MSB of the destination (bit 31 if the destination is a 36-bit accumulator) prior to the execution of the instruction is shifted into C, and the previous value of C is shifted into the LSB of the destination (bit 16 if the destination is a 36-bit accumulator).

Example:

```
ROL        A                          ; rotate A1 left one bit
```

Before Execution

F	0000	00AA
B2	B1	B0

SR	0001

After Execution

F	0001	00AA
B2	B1	B0

SR	0000

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $F:0001:00AA. Execution of the ROL A instruction shifts the 16-bit value in the A1 register one bit to the left, shifting bit 31 into C, rotating C into bit 16, and storing the result back in the A1 register.

ROL Rotate Left ROL

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	**N**	**Z**	**V**	**C**

N — Set if bit 31 of A or B result is set
Z — Set if bits 31-16 of A or B result are zero
V — Always cleared
C — Set if bit 31 of A or B was set prior to the execution of the instruction

Instruction Fields:

OPERATION	OPERAND
ROL	A B X0 Y1 Y0

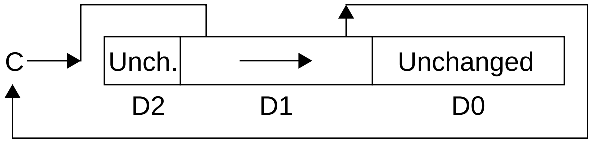
Timing: 2 mv oscillator clock cycles
Memory: 1 program word

ROR Rotate Right ROR

Assembler Syntax:

ROR D

Operation:



(parallel move)

Description: Logically shift 16-bits of the destination operand (D) one bit to the right, and store the result in the destination. If the destination is a 36-bit accumulator, the result is stored in the MSP of the accumulator (A1 or B1), and the remaining portions of the accumulator (A2, B2, A0, B0) are not modified. The LSB of the destination (bit 16 if the destination is a 36-bit accumulator) prior to the execution of the instruction is shifted into C and the previous value of C is shifted into the MSB of the destination (bit 31 if the destination is a 36-bit accumulator).

Example:

```
ROR     B                          ;rotate B1 right one bit
```

Before Execution

F	0001	00AA
B2	B1	B0

SR	0000

After Execution

F	0000	00AA
B2	B1	B0

SR	0005

Explanation of Example:

Prior to execution, the 36-bit B accumulator contains the value $F:0001:00AA. Execution of the ROR B instruction shifts the 16-bit value in the B1 register one bit to the right, shifting bit 16 into C, rotating C into bit 31, and storing the result back in the B1 register.

ROR Rotate Right ROR

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	**V**	**C**

N — Set if bit 31 of A or B result is set
Z — Set if bits 31-16 of A or B result are zero
V — Always cleared
C — Set if bit 16 of A or B was set prior to the execution of the instruction

Instruction Fields:

OPERATION	OPERAND
ROR	A B X0 Y1 Y0

Timing: 2 mv oscillator clock cycles
Memory: 1 program word

RTI Return from Interrupt RTI

Operation:

X:(SP) → SR; SP-1→ SP
X:(SP) → PC; SP-1→ SP

Assembler Syntax:

RTI

Description: Pull the SR and the PC from the software stack. The previous PC and SR are lost.

Example:

```
RTI                 ; pull the SR and PC registers from the stack
```

Before Execution		**After Execution**	
X:$0100	$1300	X:$0100	$1300
X:$00FF	$754C	X:$00FF	$754C
SR	$0309	SR	1300
SP	$0100	SP	$00FE

Explanation of Example:

The RTI instruction pulls the 16-bit PC and the 16-bit SR from the stack and updates the system SP. Program execution continues at $754C. This 19-bit address is formed using bits P2-P0 in the SR and the 16-bits in the PC.

Restrictions: — Due to pipelining in the program controller and the fact that the RTI instruction accesses certain program controller registers, the RTI instruction must not be immediately preceded by any of the following instructions:

MOVE(C) to the SP
Any bit-field instruction performed on the SR

— An RTI instruction cannot be the last instruction in a DO loop (at the LA).
— An RTI instruction cannot be repeated using the REP instruction.

RTI Return from Interrupt RTI

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

LF — Set according to the value pulled from the stack
P2 — Set according to the value pulled from the stack
P1 — Set according to the value pulled from the stack
P0 — Set according to the value pulled from the stack
I1 — Set according to the value pulled from the stack
I0 — Set according to the value pulled from the stack
L — Set according to the value pulled from the stack
E — Set according to the value pulled from the stack
U — Set according to the value pulled from the stack
N — Set according to the value pulled from the stack
Z — Set according to the value pulled from the stack
V — Set according to the value pulled from the stack
C — Set according to the value pulled from the stack

Instruction Fields:

OPERATION
RTI

Timing: 10 + rx oscillator clock cycles
Memory: 1 program word

RTS Return from Subroutine RTS

Operation:

X:(SP) → P2-P0 bits in the SR; SP-1→ SP
X:(SP) → PC; SP-1→ SP

Assembler Syntax:

RTS

Description: Pull the P2-P0 bits in the SR and the PC from the software stack. The previous PC is lost. Bits 15-13 and 9-0 of the SR are not affected.

Example:

```
RTS                    ; pull the SR’s P2-P0 bits and the PC from the stack
```

Before Execution		**After Execution**	
X:$0100	$1300	X:$0100	$1300
X:$00FF	$754C	X:$00FF	$754C
SR	$0009	SR	1009
SP	$0100	SP	$00FE

Explanation of Example:

The RTS instruction pulls the 16-bit PC and bits P2-P0 of the SR from the software stack and updates the SP. Program execution continues $754C. This 19-bit address is formed using bits P2-P0 in the SR and the 16-bits in the PC.

Restrictions: — Due to pipelining in the program controller and the fact that the RTS instruction accesses certain program controller registers, the RTS instruction must not be immediately preceded by any of the following instructions:

MOVE(C) to the SP

— An RTS instruction cannot be the last instruction in a DO loop (at the LA)
— An RTS instruction cannot be repeated using the REP instruction

RTS Return from Subroutine RTS

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	**P2**	**P1**	**P0**	I1	I0	*	L	E	U	N	Z	V	C

P2 — Receives the value in bit 12 of the first location pulled from the stack
P1 — Receives the value in bit 11 of the first location pulled from the stack
P0 — Receives the value in bit 10 of the first location pulled from the stack

Instruction Fields:

OPERATION
RTS

Timing: 10 + rx oscillator clock cycles
Memory: 1 program word

Descriptions

SBC Subtract Long with Carry SBC

Operation:

D - S - C → D (no parallel move)

Assembler Syntax:

SBC S,D (no parallel move)

Description: Subtract the source operand (S) and C of the CCR from the destination operand (D), and store the result in the destination accumulator. Long words (32 bits) are subtracted from the 36-bit destination accumulator.

Usage: This instruction is typically used in multi-precision subtraction operations (see **Multi-precision Addition and Subtraction** on page 3-27) when it is necessary to subtract together two numbers that are larger than 32-bits, such as 64-bit or 96-bit subtraction.

Example:

SBC Y,A

Before Execution

0	4000	0000
A2	A1	A0
Y	3FFF	FFFE
	Y1	Y0
	SR	0301

After Execution

0	0000	0001
A2	A1	A0
Y	3FFF	FFFE
	Y1	Y0
	SR	0310

Explanation of Example:

Prior to execution, the 32-bit Y register (comprised of the Y1 and Y0 registers) contains the value $3FFF:FFFE, and the 36-bit accumulator contains the value $0:4000:0000. In addition, C is set to one. The SBC instruction automatically sign-extends the 32-bit Y registers to 36-bits and subtracts this value from the 36-bit accumulator. In addition, C is subtracted from the LSB of this 36-bit addition. The 36-bit result is stored back in the A accumulator, and the conditions codes are set correctly. The Y1:Y0 register pair is not affected by this instruction.

Note: C is set correctly for multi-precision arithmetic using long word operands only when the extension register of the destination accumulator (A2 or B2) contains sign extension of bit 31 of the destination accumulator (A or B).

SBC Subtract Long with Carry SBC

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero. Cleared otherwise
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 35 of A or B result

Instruction Fields:

OPERATION	OPERANDS
SBC	Y,A Y,B

Timing: 2 oscillator clock cycles
Memory: 1 program word

STOP Stop Instruction Processing STOP

Operation:

Enter the stop processing state

Assembler Syntax:

STOP

Description: Enter the stop processing state. All activity in the processor is suspended until the $\overline{\text{RESET}}$ pin is asserted, the $\overline{\text{IRQA}}$ pin is asserted, or an on-chip peripheral asserts a signal to exit the stop processing state. The stop processing state is a very low-power standby mode where all clocks to the DSP core are gated off as well as the clocks to many of the on-chip peripherals such as serial ports. It is still possible for timers to continue to run in stop state. In these cases the timers can be individually powered down at the peripheral itself for lower power consumption. The clock oscillator can also be disabled for lowest power consumption.

When the exit from the stop state is caused by a low level on the $\overline{\text{RESET}}$ pin, then the processor enters the reset processing state. The time to recover from the stop state using $\overline{\text{RESET}}$ will depend on a clock stabilization delay controlled by the stop delay (SD) bit in the OMR.

When the exit from the stop state is caused by a low level on the $\overline{\text{IRQA}}$ pin, then the processor will service the highest priority pending interrupt and will not service the $\overline{\text{IRQA}}$ interrupt unless it is highest priority. The interrupt will be serviced after an internal delay counter counts 524,284 clock phases (i.e., $[2^{19}-4]$T) or 28 clock phases (i.e., $[2^{5}-4]$T) delay if the SD bit is set to one. During this clock stabilization count delay, all peripherals and external interrupts are cleared and re-enabled/arbitrated at the start of the 17T period following the count interval. The processor will resume program execution at the instruction following the STOP instruction that caused the entry into the stop state after the interrupts have been serviced or, if no interrupt was pending, immediately after the delay count plus 17T. If the $\overline{\text{IRQA}}$ pin is asserted when the STOP instruction is executed the internal delay counter will be started. Refer to **Stop Processing State** on page 7-26 for details on the stop mode.

Restrictions:

— A STOP instruction cannot be repeated using the REP instruction

— A STOP instruction cannot be the last instruction in a DO loop (i.e., at the LA)

Example:

```
STOP      ; enter low-power standby mode
```

Explanation of Example:

The STOP instruction suspends all processor activity until the processor is reset or interrupted as previously described. The STOP instruction puts the processor in a low-power standby mode. No new instructions are fetched until the processor exits the STOP processing state.

Condition Codes Affected:

The condition codes are not affected by this instruction.

STOP Stop Instruction Processing STOP

Instruction Fields:

OPERATION
STOP

Timing: The STOP instruction disables internal distribution of the clock. The time to exit the stop state depends on the value of the SD bit.

Memory: 1 program word

SUB Subtract SUB

Operation:			Assembler Syntax:		
D - S →	D	(parallel move)	SUB	S,D	(parallel move)
D - S →	D	(two parallel reads)	SUB	S,D	(two parallel reads)

Description: Subtract the source operand (S) from the destination operand (D), and store the result in the destination operand. Words (16 bits), long words (32 bits) and accumulators (36 bits) may be subtracted from the destination.

Usage: This instruction can be used for both integer and fractional two's-complement data.

Example:

```
SUB      X0,A    X:(R2)+N,Y1     ; 16-bit subtract, load X0, update R2
```

Before Execution

0	0058	1234
A2	A1	A0

X0	0003

After Execution

0	0055	1234
A2	A1	A0

X0	3456

Explanation of Example:

Prior to execution, the16-bit X0 register contains the value $0003 and the 36-bit A accumulator contains the value $0:0058:1234. The SUB instruction automatically appends the 16-bit value in the X0 register with 16 LS zeros, sign-extends the resulting 32-bit long word to 36 bits, and subtracts the result from the 36-bit A accumulator. Thus, 16-bit operands are always subtracted from the MSP of A or B (A1 or B1) with the results correctly extending into the extension register (A2 or B2). 16-bit operands can be subtracted from the LSP of A or B (A0 or B0) by loading the 16-bit operand into Y0, forming a 32-bit word by loading Y1 with the sign extension of Y0, and executing a SUB Y,A or SUB Y,B instruction. Similarly, the second accumulator can also be used for the source operand.

Note: C is set correctly using word or long word source operands if the extension register of the destination accumulator (A2 or B2) contains sign extension from bit 31 of the destination accumulator (A or B). C is always set correctly using accumulator source operands.

SUB Subtract SUB

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 35 of A or B result

See **16-bit Destinations** on page 3-34 for cases with X0, Y0, or Y1 as D.
See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

Instruction Fields:

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
SUB	X0,F Y1,F Y0,F A,B B,A (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1

DATA ALU OPERATION		FIRST & SECOND MEMORY READS		DESTINATIONS FOR MEMORY READS	
Operation	**Registers**	**Read1**	**Read2**	**Dest1**	**Dest2**
SUB	X0,A Y1,A Y0,A X0,B Y1,B Y0,B	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	X:(R3)+ X:(R3)-	Y0	X0
				Y1	X0
				Valid destinations for Read1	Valid destinations for Read2

SUB Subtract SUB

OPERATION	OPERANDS	COMMENTS	Cyc/Wd
SUB	X0,F Y1,F Y0,F F,X0 F,Y1 F,Y0 Y0,X0 Y1,X0 X0,Y0 Y1,Y0 X0,Y1 Y0,Y1	F = A,B	(2/1)
	A,B B,A	—	(2/1)
	Y,F	F = A,B	(2/1)
	#xx,F #xx,DD	5-bit positive integer ranging from 1 to 31	(4/1)
	#xxxx,F #xxxx,DD	16-bit signed integer	(6/2)
	X:<aa>,F X:<aa>,DD	<aa>: First 64 locations of X memory	(4/1)
	X:xxxx,F X:xxxx,DD	Long 16-bit absolute address	(6/2)
	X:(SP-xx),F X:(SP-xx),DD	xx = [1 to 64]	(6/1)

SUB Subtract SUB

Timing: 2 + mv oscillator clock cycles for SUB instructions with a parallel move.
Refer to previous tables for SUB instructions without a parallel move.

Memory: 1 program word for SUB instructions with a parallel move.
Refer to previous tables for SUB instructions without a parallel move.

SWI Software Interrupt SWI

Operation:

Begin SWI exception processing

Assembler Syntax:

SWI

Description: Suspend normal instruction execution, and begin SWI exception processing. The interrupt priority level, specified by the I1 and I0 bits in the SR, is set to the highest interrupt priority level upon entering the interrupt service routine.

Example:

```
SWI                               ; begin SWI exception processing
```

Explanation of Example:

The SWI instruction suspends normal instruction execution and initiates SWI exception processing.

Restrictions: — A SWI instruction cannot be repeated using the REP instruction

Condition Codes Affected:

The condition codes are not affected by this instruction.

SWI Software Interrupt SWI

Instruction Fields: none

OPERATION
SWI

Timing: 8 oscillator clock cycles
Memory: 1 program word

Tcc Transfer Conditionally Tcc

Operation:	Assembler Syntax:
If cc, then S → D	Tcc S,D
If cc, then S → D and R0 → R1	Tcc S,D R0,R1

Description: Transfer data from the specified source register (S) to the specified destination accumulator (D) if the specified condition is true. If a second source register R0 and a second destination register R1 are also specified, transfer data from address register R0 to address register R1 if the specified condition is true. If the specified condition is false, a NOP is executed.

Usage: When used after the CMP instruction, the Tcc instruction can perform many useful functions such as a “maximum value” or “minimum value” function. The desired value is stored in the destination accumulator. If address register R0 is used as an address pointer into an array of data, the address of the desired value is stored in the address register R1. The Tcc instruction may be used after any instruction and allows efficient searching and sorting algorithms.

The term “cc” specifies the following:

“cc” Mnemonic		Condition
CC (HS*)	— carry clear (higher or same)	C=0
CS (LO*)	— carry set (lower)	C=1
EQ	— equal	Z=1
GE	— greater than or equal	N ⊕ V=0
GT	— greater than	Z+(N ⊕ V)=0
LE	— less than or equal	Z+(N ⊕ V)=1
LT	— less than	N ⊕ V=1
NE	— not equal	Z=0
* Only available when CC bit set in the OMR		

where: + denotes the logical OR operator,
⊕ denotes the logical exclusive OR operator

Note: This instruction is considered to be a move-type instruction. Due to pipelining, if an address register (R0 or R1 for the Tcc instruction) is changed using a move-type instruction, 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).

Tcc Transfer Conditionally Tcc

Example:

```
CMP     X0,A                 ; compare X0 and A (sort for minimum)
TLT     X0,A     R0,R1       ; transfer X0 → A and R0 → R1 if X0 < A
```

Explanation of Example:

In this example, the contents of the 16-bit X0 register are transferred to the 36-bit A accumulator, and the contents of the 16-bit R0 address register are transferred to the 16-bit R1 address register if the specified condition is true. If the specified condition is not true, a NOP is executed.

Condition Codes Affected:

The condition codes are tested but not modified by this instruction.

Instruction Fields:

OPERATION	DATA ALU	AGU
Tcc	A,B B,A X0,A Y1,A Y0,A X0,B Y1,B Y0,B	(no transfer) R0,R1

Timing: 2 oscillator clock cycles
Memory: 1 program word

TFR Transfer Data ALU Register TFR

Operation:

S → D (parallel move)

Assembler Syntax:

TFR S,D (parallel move)

Description: Transfer data from the specified source data ALU register (S) to the specified destination data ALU accumulator (D). TFR uses the internal data ALU data paths, and thus data does not pass through the data limiter. This allows the full 36-bit contents of one of the accumulators to be transferred into the other accumulator without data limiting. The TFR instruction only affects bits L in the CCR, which can be set by data movement associated with the instruction's parallel move operations.

Usage: This instruction is very similar to a MOVE instruction but has two uses. First, it can be used to perform a 36-bit transfer of one accumulator to another. Second, when used with a parallel move, this instruction allows a register move and a memory move to occur simultaneously in one instruction, which executes in one instruction cycle.

Example:

```
TFR     B,A     X:(R0)+,Y1      ; move B to A and
                                ; update Y1, R0
```

Before Execution

3	0123	0123
A2	A1	A0
A	CCCC	EEEE
B2	B1	B0

After Execution

A	CCCC	EEEE
A2	A1	A0
A	CCCC	EEEE
B2	B1	B0

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $3:0123:012,3 and the 36-bit B accumulator contains the value $A:CCCC:EEEE. Execution of the TFR B,A instruction moves the 36-bit value in B into the 36-bit A accumulator.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	E	U	N	Z	V	C

L — Set if data limiting has occurred during parallel move

TFR Transfer Data ALU Register TFR

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
TFR	X0,A Y1,A Y0,A X0,B Y1,B Y0,B	TFR is not used when dest is not an accumulator, because since there is no parallel move, it is just as simple to use a MOVE instruction.
	A,B B,A	TFR differs from a MOVE because it transfers 36 bits.

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
TFR	X0,F Y1,F Y0,F A,B B,A F = A,B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2 + mv oscillator clock cycles
Memory: 1 program word

Descriptions

TST Test Accumulator TST

Operation:

S - 0 (parallel move)

Assembler Syntax:

TST S (parallel move)

Description: Compare the specified source accumulator (S) with zero, and set the condition codes accordingly. No result is stored although the condition codes are updated.

Example:

```
TST     A     X:(R0)+N,B     ; set condition codes for the value in A, update B and R0
```

Before Execution

8	0203	0000
A2	A1	A0

SR	0300

After Execution

8	0203	0000
A2	A1	A0

SR	0338

Explanation of Example:

Prior to execution, the 36-bit A accumulator contains the value $8:0203:0000, and the 16-bit SR contains the value $0300. Execution of the TST A instruction compares the value in the A register with zero and updates the CCR accordingly. The contents of the A accumulator are not affected.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if data limiting has occurred during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of A or B result is set
Z — Set if A or B result equals zero
V — Always cleared
C — Always cleared

See **36-bit Destinations—CC bit set** on page 3-33 and **20-bit Destinations—CC bit set** on page 3-34 for the case when the CC bit is set.

TST Test Accumulator TST

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
TST	A B	Perform a test on all 36-bits of accumulator.

DATA ALU OPERATION		PARALLEL MEMORY READ or WRITE	
Operation	**Registers**	**Mem Access**	**Src/Dest**
TST	A B	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A1 B1 A B

Timing: 2+mv oscillator clock cycles
Memory: 1 program word

TSTW Test Register or Memory TSTW

Operation:		**Assembler Syntax:**		
S - 0	(no parallel move)	TSTW	S	(no parallel move)

Description: Compare 16-bits of the specified source register or memory location with zero, and set the condition codes accordingly. No result is stored although the condition codes are updated.

Example:

```
TSTW    X:$0007                 ; set condition codes using X:$0007
```

Before Execution		**After Execution**	
X:$0007	FC00	X:$0007	FC00
SR	0300	SR	0308

Explanation of Example:

Prior to execution,location X:$0007contains the value $FC00 and the 16-bit SR contains the value $0300. Execution of the instruction compares the value in the X0 register with zero and updates the CCR accordingly. The contents of location X:$0007 is not affected.

Note: This instruction does not set the same set of condition codes that the TST instruction does. Both instructions correctly set the V, N, Z, and C bits, but TST sets the E bit whereas TSTW does not. This is a 16-bit test operation when done on an accumulator (A or B), where limiting is performed if appropriate when reading the accumulator.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	P2	P1	P0	I1	I0	*	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of A or B result is set
Z — Set if result equals zero
V — Always cleared

TSTW Test Register or Memory TSTW

Instruction Fields:

OPERATION	SOURCE	COMMENTS	Cyc/Wd
TSTW	x:<aa> x:<pp>	—	(2/1)

	X:(R2+xx)
	X:(SP-xx)
		—	(4/1)

	X:xxxx	—	(4/2)
	X:(Rn) X:(Rn)+ X:(Rn)- X:(Rn)+N	—	(2/1)
	X:(Rn+N)	—	(4/1)
	X:(Rn+xxxx)	—	(6/2)
	X:(SP) X:(SP)+ X:(SP)- X:(SP)+N	—	(2/1)
	X:(SP+N)	—	(4/1)
	X:(SP+xxxx)	—	(6/2)
	Any register except the HWS	Test only 16-bit when an accumulator register is specified.	(2/1)
	(Rn)-	Test a Rn register and then post-decrement it.	(2/1)

Timing: 2 oscillator clock cycles

Memory: 1 program word

WAIT Wait for interrupt WAIT

Operation: Disable clocks to the processor core, and enter the wait processing state.

Assembler Syntax: WAIT

Description: Enter the wait processing state. The internal clocks to the processor core and memories are gated off, and all activity in the processor is suspended until an unmasked interrupt occurs. The clock oscillator and the internal I/O peripheral clocks remain active.

When an unmasked interrupt or external (hardware) processor reset occurs, the processor leaves the wait state and begins exception processing of the unmasked interrupt or reset condition.

Restrictions:

— A WAIT instruction cannot be the last instruction in a DO loop (at the LA).

— A WAIT instruction cannot be repeated using the REP instruction.

Example:

```
WAIT                ; enter low power mode, wait for interrupt
```

Explanation of Example:

The WAIT instruction suspends normal instruction execution and waits for an unmasked interrupt or external reset to occur. No new instructions are fetched until the processor exits the wait processing state.

Condition Codes Affected:

The condition codes are not affected by this instruction.

WAIT Wait for interrupt WAIT

Instruction Fields:

OPERATION
WAIT

Timing: If an internal interrupt is pending during the execution of the WAIT instruction, the WAIT instruction takes a minimum of 32T cycles to execute.

If no internal interrupt is pending when the WAIT instruction is executed, the period that the DSP is in the wait state is the period before the interrupt or reset causing the DSP to exit the wait state plus a minimum of 28T cycles to a maximum of 31T cycles (see the appropriate data sheet).

Memory: 1 program word

MACROS MACROS

Description: The instructions presented in this section do not represent actual DSP56800 instructions, but rather represent useful instruction macros for standard operations that are built from simple sequences of DSP56800 instructions. See **Section 8 Software Techniques** for additional information.

MACROS MACROS

Bcc Macro—Branch Conditionally Bcc

Operation:

If (cc), then PC + label → PC
else PC+1 → PC

Assembler Syntax:

```
Bcc     aa
```

Description: If the specified condition (cc) is true, program execution continues at location PC + displacement. The PC contains the address of the next instruction. If the condition is false, the PC is incremented and program execution continues sequentially. The offset is a 7-bit value that is sign-extended to 16 bits.

The term “cc” specifies the following:

“cc” Mnemonic		Condition
VC	— overflow cleared	V = 0
VS	— overflow set	V = 1
PL	— plus	N = 0
MI	— minus	N = 1
RCLR	— all selected bits cleared	all bits = 0
RSET	— all selected bits set	all bits = 1
EC	— extension cleared	E = 0
ES	— extension set	E = 1
LC	— limit cleared	L = 0
LS	— limit set	L = 1

Macro Expansion:

```
BFTST_  <mask>,<dst>
BCS     <label>
```

Example:

```
        BVS     LABEL           ; branch to label if “greater-than”
        INCW    A
        INCW    A
LABEL
        ADD     B,A
```

Explanation of Example:

In this example, if the V bit is set when executing the BVS instruction, program execution skips the two INCW instructions and continues with the ADD instruction. If the specified condition is not true, no branch is taken, the PC is incremented by one, and program execution continues with the first INCW instruction. The Bcc instruction uses a PC-relative offset of 2 in this example.

Bcc Macro—Branch Conditionally Bcc

Instruction Fields:

OPERATION	OPERANDS	COMMENTS
Bcc	$xx	7-bit PC-relative offset [-64, 63]

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

Jcc Macro—Jump Conditionally Jcc

Operation:		**Assembler Syntax:**	
If (cc), then PC + label	→ PC	Jcc	#iiii, X:<ea>, aa
else PC+1	→ PC	Jcc	#iiii, D, aa

Description: If the specified condition (cc) is true, program execution continues at the effective address specified in the instruction. If the specified condition is false, the PC is incremented and the program execution continues sequentially. The effective addresss is a 19-bit absolute addresss.

The term "cc" specifies the following:

"cc" Mnemonic		Condition
VC	— overflow cleared	V = 0
VS	— overflow set	V = 1
PL	— plus	N = 0
MI	— minus	N = 1
RCLR	— all selected bits cleared	all bits = 0
RSET	— all selected bits set	all bits = 1
EC	— extension cleared	E = 0
ES	— extension set	E = 1
LC	— limit cleared	L = 0
LS	— limit set	L = 1

Macro Expansion:

```
BFTST_   <mask>,<dst>
JCS      <label>
```

Example:

```
JRCLR #$000F, X:<<$FFE2, LABEL
```

Before Execution		**After Execution**	
X:$FFE2	0FF0	X:$FFE2	0FF0
SR	0000	SR	0001

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $0FF0. Execution of the instruction tests the state of bits 0-3 in X:$FFE2 and sets C (because all of these were cleared).

Jcc Macro—Jump Conditionally Jcc

Instruction Fields:

OPERATION		OPERAND	COMMENTS	Cyc/Wd
Jcc	**#iiii**	X:(R2+xx)	xx = [0 to 63]	(10+jx / 4)
		X:(SP-xx)	xx = [1 to 64]	(10+jx / 4)
		X:<aa>	First 64 words of X memory	(8+jx / 4)
		X:<pp>	Last 64 words of X memory	(8+jx / 4)
		X:xxxx	16-bit absolute address	(10+jx / 5)
		Any register except the HWS	—	(8+jx / 4)

Timing: Refer to table above in Instruction Fields.
Memory: Refer to table above in Instruction Fields.

Descriptions

PUSH Macro—Push onto Stack PUSH

Operation:

SP+1 → SP
S → X:(SP)

Assembler Syntax:

PUSH S

Description: Pre-increment the SP, and write one location to the software stack from a source register.

Example:

```
PUSH      LC
```

	Before Execution		After Execution
x:$0100	AAAA	X:$0100	0099
LC	0099	LC	0099
SP	00FF	SP	0100

Explanation of Example:

Prior to execution, the LC register contains the value $0099, and the SP contains the value $00FF. The PUSH instruction increment SP to $0100 and write the X data memory pointed to by the new SP with the contents of the LC register.

Condition Codes Affected:

The condition codes are not affected by this instruction.

PUSH Macro—Push onto Stack PUSH

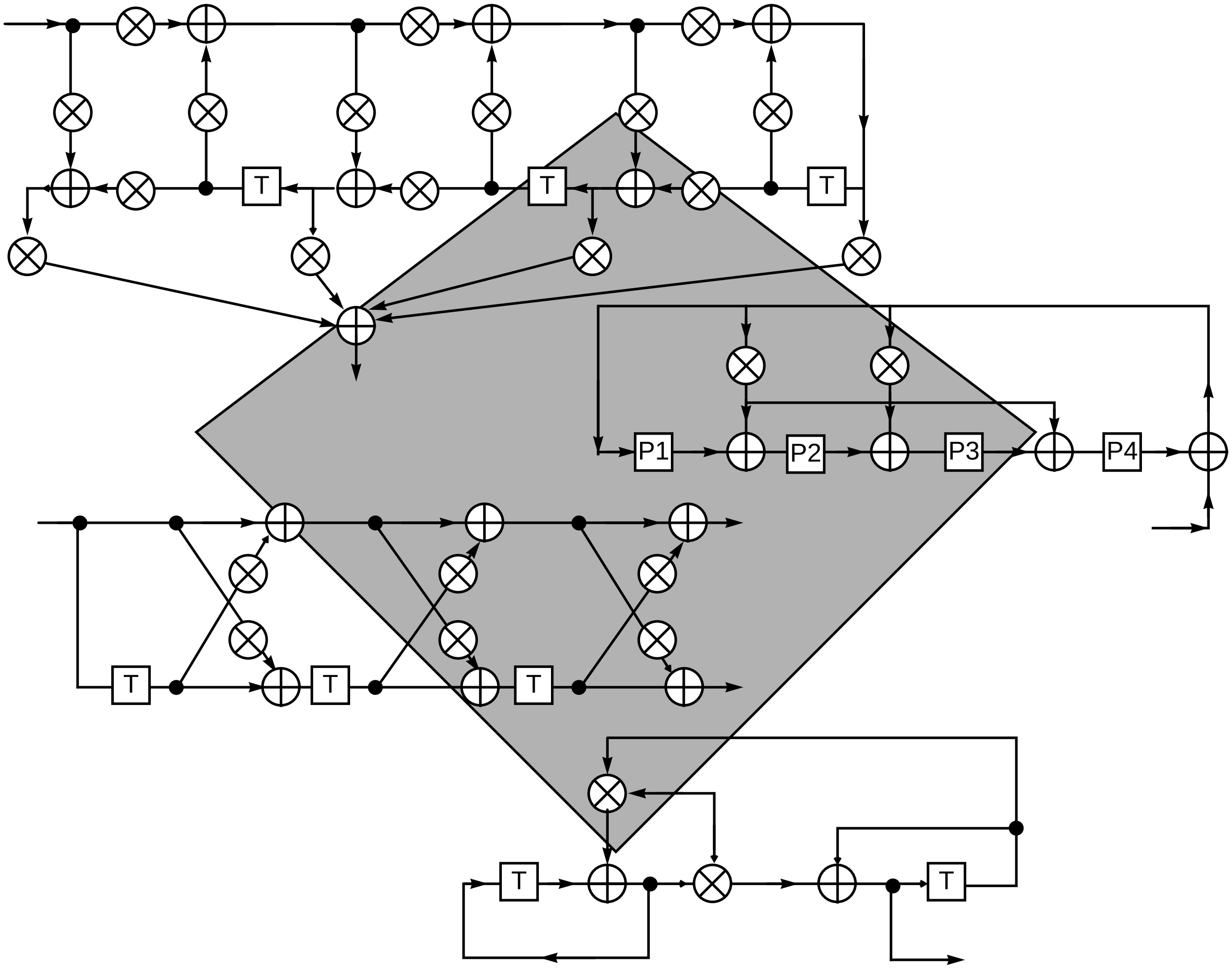
Instruction Fields:

OPERATION	DESTINATION	COMMENTS
PUSH	Any register	The PUSH instruction is a 2 word, 2 cycle macro.

Timing: 4 + mv oscillator clock cycles
Memory: 2 program words

APPENDIX B

DSP BENCHMARKS



B.1 INTRODUCTION

The following benchmarks illustrate the source code syntax and programming techniques for the Motorola DSP. The assembly language source is organized into five columns, as shown in **Example B-1**.

Example B-1 Source Code Layout

Label[1]	**Opcode**[2]	**Operands**[3]	**Data Bus**[4]		**Comment**[5]
`FIR`	`MAC`	`Y0,X0,A`	`X:(R0)+,Y0`	`X:(R3)+,X0`	`;Do each tap`

Note: 1. Used for program entry points and end-of-loop indication.
2. Indicates the data ALU, addres ALU, or program controller operation to be performed. This column must also be included in the source code.
3. Specifies the operands to be used by the opcode.
4. Specifies an optional data transfer over the data bus and the addressing mode to be used.
5. Used for documentation purposes and does not affect the assembled code.

Table B-1 shows the number of program words and instruction cycles for each benchmark.

Table B-1 Benchmark Summary

Benchmark	**Execution Time (# Icyc)**	**Program Length (# Words)**
Real Correlation Or Convolution (FIR Filter)	1N	9
N Complex Multiplies	6N	15
Complex Correlation Or Convolution (Complex FIR)	5N	15
Nth Order Power Series (Real, Fractional Data)	1N	13
N Cascaded Real Biquad IIR Filters (Direct Form II)	6N	16
N Radix 2 FFT Butterflies	12N	21
LMS Adaptive Filter: Single Precision	3N	18
LMS Adaptive Filter: Double Precision	6N	21
LMS Adaptive Filter: Double Precision Delayed	5N	27
Vector Multiply-Accumulate	2N	12
Energy in a Signal	1N	7
[3x3][1x3] Matrix Multiply	20	20
[NxN][NxN] Matrix Multiply	$N^3 + 8N^2$	30
N Point 3x3 2-D FIR Convolution	$13N^2 + 11^N$	41

Table B-1 Benchmark Summary (Continued)

Benchmark	Execution Time (# Icyc)	Program Length (# Words)
Sine Wave Generation: Double Integration Technique	2N	13
Sine Wave Generation: Second Order Oscillator	5N	16
Array Search: Index of the Highest Signed Value	4N	10
Array Search: Index of the Highest Positive Value	2N	10
Proportional Integrator Differentiator (PID) Algorithm	6	6
Autocorrelation Algorithm	$(p + 1)^2 (N - p / 2)$	23

B.2 BENCHMARK CODE

The source code below lists all the defines for the benchmarks.

```
          page 132
          opt  cc
; define section
AD        EQU  0
BD        EQU  $100
bd        EQU  $100
C         EQU  $200
c         EQU  $200
D         EQU  $300
N         EQU  100
AR        EQU  $300
AI        EQU  $400
OUTPUT    EQU  $500
output    EQU  $FFF1
INPUT     EQU  $501
input     EQU  $FFF1
W         EQU  0
w         EQU  0
H         EQU  0
XM        EQU  0
state     equ  0
```

```
ntaps     equ  $10
k         equ  0
n         equ  32
p         equ  10
mask      equ  10
image     equ  $40
dividend  equ  .25
divisor   equ  .5
paddr     equ  0
qaddr     equ  4
w1        equ  0
w2        equ  10
s         equ  0
tablebase equ  0
lpc       equ  8
frame     equ  0
cor       equ  $100
shift     equ  $80   ; shift constant
table     equ  $180  ; base address of a-law table
          org  p:$40
```

B.2.1 Real Correlation Or Convolution (FIR Filter)

```
; c(n) = SUM(I=0,...,N-1) { a(I) * b(n-I) }

      opt  cc
      MOVE #AD,R0                        ; 2   2
      MOVE #BD,R3                        ; 2   2
      CLR  A       X:(R0)+,Y0            ; 1   1
      MOVE         X:(R3)+,X0            ; 1   1
      REP  #N                            ; 1   3
      MAC  Y0,X0,A X:(R0)+,Y0 X:(R3)+,X0 ; 1   1
      RND  A                             ; 1   1
;                                        ________
;                             Total:      9   1N+11
```

B.2.2 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) Y1=ar
; ci(I) = ar(I) * bi(I) + ai(I) * br(I) Y0=ai      X0=br, bi
    opt   cc
    MOVE  #AD,R0                                   ; 2  2
    MOVE  #C-1,R2                                  ; 2  2
    MOVE  #BD,R3                                   ; 2  2
    MOVE              X:(R2),B                     ;           dummy move !!
    DO    #NUM,END_DO8                             ; 2  3
    MOVE              X:(R0)+,Y1        X:(R3)+,X0 ; 1  1      get ar,br
    MPY   Y1,X0,A     B,X:(R2)+                    ; 1  1      ar*br,
                                                   ;           store imag
    MOVE              X:(R0)+,Y0                   ; 1  1      get ai
    MPY   Y0,X0,B     X:(R3)+,X0                   ; 1  1      ai*br, get bi
                                                   ;           get bi
    MACR  -Y0,X0,A                                 ; 1  1      ar*br-ai*bi
    MACR  Y1,X0,B     A,X:(R2)+                    ; 1  1      ar*bi+ai*br,
END_DO8                                                        store real
    MOVE  B,X:(R2)+                                ; 1  1
;                                                  ________
;                                       Total:     15   6N+11
```

B.2.3 Complex Correlation Or Convolution (Complex FIR)

```
; 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)                       Y0=ar Y1=br
; { ar(I) * br(n-I) - ai(I) * bi(n-I) }
; ci(n) = SUM(I=0,...,N-1)                       Y0=ai X0=bi
; { ar(I) * bi(n-I) + ai(I) * br(n-I) }
        opt  cc
        MOVE #AD,R0                        ; 2   2
        MOVE #BD,R3                        ; 2   2
        CLR  A          X:(R0)+,Y0         ; 1   1     ar
        CLR  B          X:(R3)+,Y1         ; 1   1     br
        DO   #N,END_DOB                    ; 2   3
        MAC  Y0,Y1,A    X:(R3)+,X0         ; 1   1     ar*br ,ai,bi
        MAC  Y0,X0,B    X:(R0)+,Y0         ; 1   1     ar*bi
        MAC  Y0,Y1,B    X:(R3)+,Y1         ; 1   1     ar*bi+ai*br,ar
        MAC  -Y0,X0,A                      ; 1   1     ar*br-ai*bi
        MOVE            X:(R0)+,Y0         ; 1   1
END_DOB
        RND  A                             ; 1   1
        RND  B                             ; 1   1
;                                          _______
;                                 Total:   15    5N+11
```

B.2.4 Nth Order Power Series (Real, Fractional Data)

```
; c = SUM(I=0,...,N) { a(I) * b**I }
; = [[[a(n) *b+a(n-1)] *b+a(n-2)]*b+a(n-3)].....
        opt   cc
        MOVE  #BD,R1                        ; 2   2
        MOVE  #AD,R0                        ; 2   2
        MOVE               X:(R1),Y0        ; 1   1     b
        MOVE  Y0,Y1                         ; 1   1     b
        MOVE               X:(R0)+,A        ; 1   1     get a(n)
        MOVE               X:(R0)+,B        ; 1   1     get a(n-1)
        DO    #NUM/2,END_DOC                ; 2   3
        MAC   A1,Y0,B      X:(R0)+,A        ; 1   1     get a(n-2), etc.
        MAC   B1,Y1,A      X:(R0)+,B        ; 1   1     get a(n-3), etc.
END_DOC
        RND   A                             ; 1   1
;                                           _______
;                                    Total: 13    1N+12
```

B.2.5 N Cascaded Real Biquad IIR Filters (Direct Form II)

Many digital filter design packages generate coefficients for direct form II IIR filters. Often, these coefficients are greater in magnitude than 1.0. This implementation is suitable for IIR filters with coefficients greater in magnitude than 1.0 because it allows the user to simply divide all coefficients generated by 2.

```
; w(n)/2 = x(n)/2 - (a1/2) * w(n-1) - (a2/2) * w(n-2)
; y(n)/2 = w(n)/2 + (b1/2) * w(n-1) + (b2/2) * w(n-2)
; D High Memory Order - w(n-2)1,w(n-1)1,w(n-2)2,w(n-1)2,...
; D Low  Memory Order - (a2/2)1,(a1/2)1,(b2/2)1,(b1/2)1,(a2/2)2,...
; This version uses two pointers.
        opt  cc
        MOVE #W,R0                            ; 2   2
        MOVE #C,R3                            ; 2   2
        MOVE #-1,N                            ; 1   1
        MOVE             x:input,A            ; 1   1
        ASR  A                       X:(R3)+,X0 ; 1   1     X0=a2/2
        MOVE             X:(R0)+,Y0           ; 1   1     Y0=wn-2
        DO   #N,END_DOE                       ; 2   3
        MAC  Y0,X0,A     X:(R0)+N,Y1 X:(R3)+,X0 ; 1   1     y1=wn-1
        MAC  Y1,X0,A     Y1,X:(R0)+           ; 1   1
        ASL  A                       X:(R3)+,X0 ; 1   1     X0= b2/2
        ASR  A           A,X:(R0)+            ; 1   1     X0=b1/2
        MAC  Y0,X0,A                 X:(R3)+,X0 ; 1   1
        MAC  Y1,X0,A     X:(R0)+,Y0  X:(R3)+,X0 ; 1   1
END_DOE
;                                             _______
;                                Total:       16    6N+11
```

B.2.6 N Radix 2 FFT Butterflies

This is a decimation in time (DIT), in-place algorithm. **Figure B-1** gives a graphic overview and memory map.

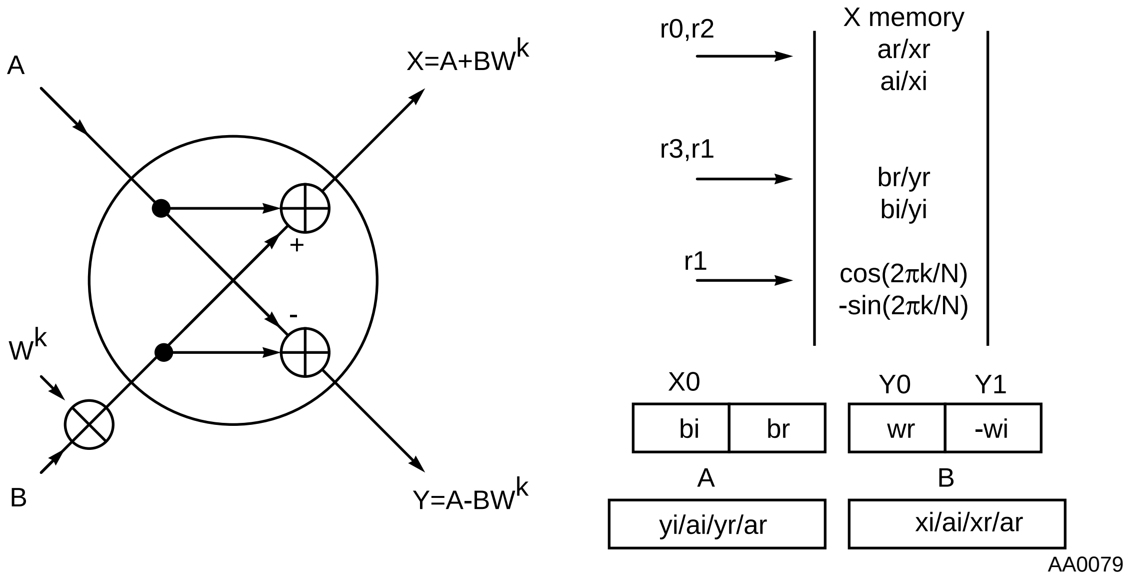


Figure B-1 N Radix 2 FFT Butterflies Memory Map

```
; Twiddle Factor  Wk= wr + jwi =  cos(2πk/N) +j sin(2πk/N)  pointed by R1
; - saved on each pass
; xr = ar + wr * br - wi * bi
; xi = ai + wi * br + wr * bi
; yr = ar - wr * br + wi * bi = 2 * ar - xr
; yi  = ai - wi * br -  wr * bi = 2 * ai - xi
   opt    cc
   move              x:(r1)+,y0  x:(r3)+,x0    ; y0=wr ; x0=br
   move              x:(r0),b                  ;b=ar
   move              x:(r1)+n,y1               ; y1=wi
; save r1, update r1 to point last bi/yi
   move              #0,n                      ; emulate X:(Rn) adr mode
   do     #n,end_bfly                    ;2    3
   push   x0                             ;1    1            push br
   mac    y0,x0,b     x:(r3)+,x0          ;1    1            b=ar+wrbr
   macr   -y1,x0,b                        ;1    1            b=xr
   move               a,x:(r1)+           ;1    1
   move               x:(r0)+,a           ;1    1            a=ar
   asl    a           b,x:(r2)+           ;1    1            a=2ar-xr=yr
   sub    b,a         x:(r0)+n,b          ;1    1
   move               a,x:(r1)+           ;1    1            b=ai
   mac    y0,x0,b     x:(r0)+,a           ;1    1            b=ai+wrbi
   pop    x0                              ;1    1            pop br
   macr   y1,x0,b     x:(r3)+,x0          ;1    1            b=xi ;a=ai
   asl    a           b,x:(r2)+           ;1    1            a=2ai-xi=yi
   sub    b,a         x:(r0)+n,b          ;1    1            b=ar
end_bfly
   move               #xx,n               ;1    1
   move               b,x:(r1)+n          ;1    1            save last yi
; save r1,
; update r1 to point twiddle factors      ________
;                                 Total:  17    13N+9
```

B.2.7 LMS Adaptive Filter

Figure B-2 gives a graphical reprentation of this implementation of the LMS adaptive filter.

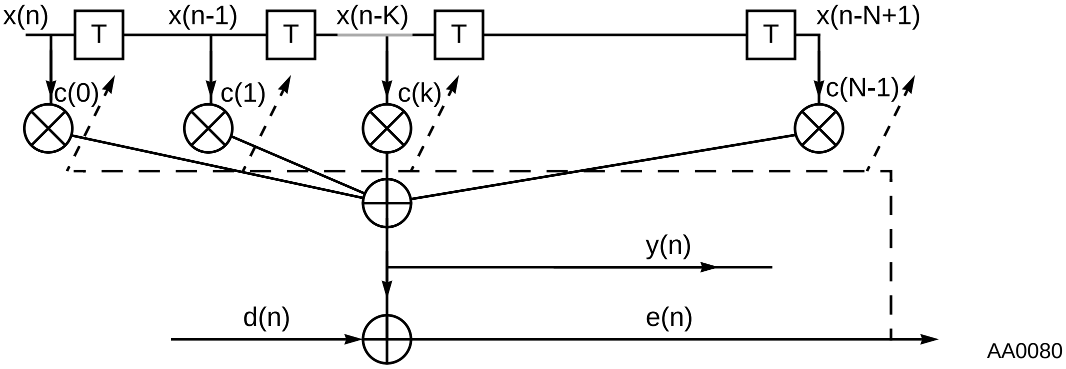


Figure B-2 LMS Adaptive Filter Graphic Representation

The following three LMS adaptive filter benchmarks are provided:

- Single precision
- Double precision
- Double precision delayed

```
; Notation and symbols:
; x(n) - Input sample at time n.
; d(n) - Desired signal at time n.
; y(n) - FIR filter output at time n.
; H(n) - Filter coefficient vector at time n.
; H={c0,c1,c2,,...,ck,...,c(N-1)}
; X(n) - Filter state variable vector at time N.
; X={x(n),x(n-1),....,x(n-N+1)}
; Mu - Adaptation gain.
; N - Number of coefficient taps in the filter.
; True LMS Algorithm      Delayed LMS Algorithm
; Get input sample        Get input sample
; Save input sample       Save input sample
; Do FIR                    Do FIR
; Get d(n), find e(n)       Update coefficients
; Update coefficients       Get d(n), find e(n)
; Output y(n)               Output y(n)
; Shift vector X            Shift vector X
; System equations:
; e(n)=d(n)-H(n)X(n)        e(n)=d(n)-H(n)X(n)         (FIR filter and error)
; H(n+1)=H(n)+uX(n)e(n)     H(n+1)=H(n)+uX(n-1)e(n-1)  (Coefficient update)
```

The references for this code include:

- "Adaptive Digital Filters and Signal Analysis," Maurice G. Bellanger , Marcel Deker, Inc. New York and Basel
- "The DLMS Algorithm Suitable for the Pipelined Realization of Adaptive Filters," Proc. IEEE ASSP Workshop, Academia Sinica, Beijing, 1986

Note: The sections of code shown describe how to initialize all registers, filter an input sample, and perform 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.

B.2.7.1 Single Precision

Figure B-3 shows a memory map for this implementation of the single-precision LMS adaptive filter.

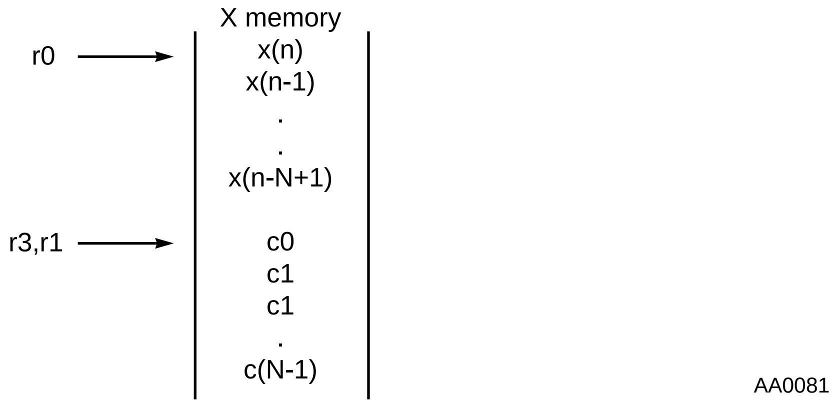


Figure B-3 LMS Adaptive Filter—Single Precision Memory Map

```
   opt    cc
   move   #XM,r0                         ; start of X
   move   #N-1,m0                        ; modulo N
   move   #-2,n                          ; adjustment for filtering
   movep  x:input,y0                     ; get input sample
   move   #H,r3                          ; 2   2     coefficients
   clr    a       y0,x:(r0)+             ; 1   1     save x(n)
   move                       x:(r3)+,x0 ; 1   1     get c0
   rep    #N-1                           ; 1   3     do fir
   mac    y0,x0,a x:(r0)+,y0  x:(r3)+,x0 ; 1   1
   macr   y0,x0,a                        ; 1   1     last tap
   movep                      a,x:output ; 1   1     output fir if desired
; (Get d(n), subtract fir output, multiply by "u", put the result in y1.
; This section is application dependent.)
   move   #H,r3                          ; 2   2     coefficients
   move   r3,r1                          ; 1   1     coefficients
   move           x:(r0)+,y0             ; 1   1     get x(n)
   move                       x:(r3)+,a  ; 1   1     a=c0
   do   #ntaps,_coefupdate               ; 2   3     update coef.
   macr   y1,y0,a x:(r0)+,y0  x:(r3)+,x0 ; 1   1
   tfr   x0,a     a,x:(r1)+              ; 1   1     copy c,
_coefupdate
   move           x:(r0)+n,y0            ; 1   1     update r0
;                                        ________
;                           Total:       18    3N+18
```

B.2.7.2 Double Precision

Figure B-4 shows a memory map for this implementation of the double-precision LMS adaptive filter.

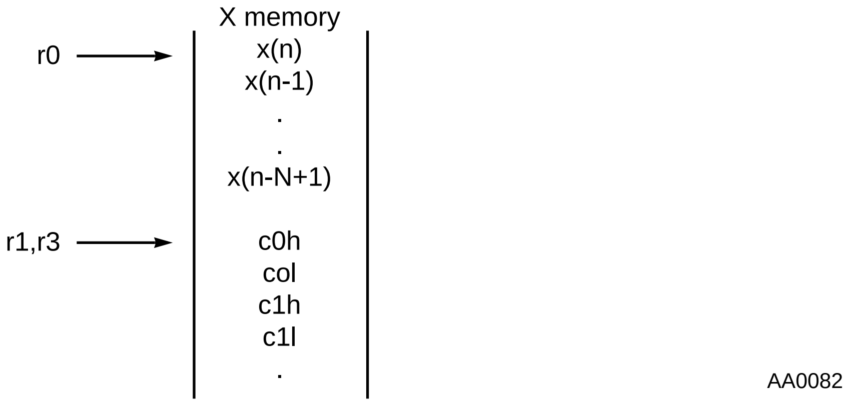


Figure B-4 LMS Adaptive Filter—Double Precision Memory Map

```
     opt    cc
     move   #XM,r0                             ; start of X
     move   #N-1,m0                            ; modulo N
     move   #2,n
     movep  x:input,y0                         ; get input sample
     move   #H,r3                              ; 1   1    ; coefficients
     clr    a         y0,x:(r0)+               ; 1   1    ; save x(n)
     move                         x:(r3)+n,x0 ; 1   1    ; get c0
     rep    #N-1                               ; 1   3    ; do fir
     mac    x0,y0,a   x:(r0)+,y0  x:(r3)+n,x0 ; 1   1    ; mac; next x
     macr   x0,y0,a                            ; 1   1    ; last tap
     movep            a,x:output               ; output fir if desired

; (Get d(n), subtract fir output, multiply by "u", put the result in x0.
; This section is application dependent.)
     move   #H,r3                              ; 2   2    ; coefficients
     move   r3,r1                              ; 1   1    ; coefficients
     move             x:(r0)+,y0               ; 1   1    ; get x(n)
     move                         x:(r3)+,a    ; 1   1    ; a1=c0h
     move                         x:(r3)+,a0   ; 1   1    ; a0=col
     do    #ntaps,_coefupdat                 ; 2   3    ; update coef.
     mac   x0,y0,a   x:(r0)+,y0              ; 1   1    u e(n) x(n)+c;
                                                        fetch x(n)
     move            a,x:(r1)+               ; 1   1    save updated c()h
     move            a0,x:(r1)+              ; 1   1    ; save updated c()l
     move            x:(r3)+,a               ; 1   1    ; fetch next c()h
     move            x:(r3)+,a0              ; 1   1    ; fetch next c()l
_coefupdat
     move  #-2,n                             ; 1   1    ; adjustment for
                                                        ; filtering
     move            x:(r0)+n,y0             ; 1   1    ; update r0
;                                            _______
;                                Total:      21    6N+18
```

B.2.7.3 Double Precision Delayed

Figure B-5 shows a memory map for this implementation of the double-precision delayed LMS adaptive filter.

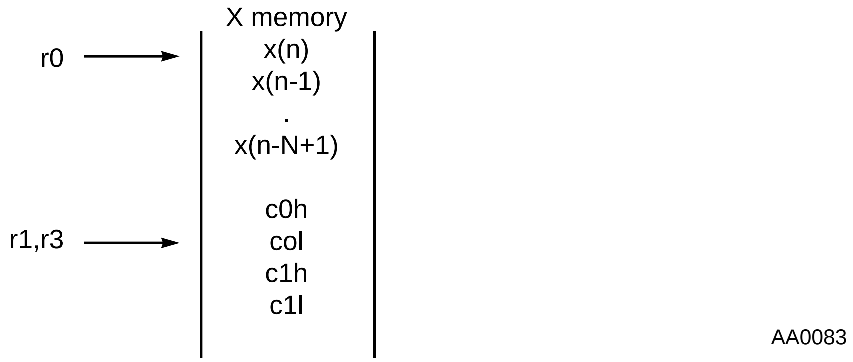


Figure B-5 LMS Adaptive Filter—Double Precision Delayed Memory Map

```
; Delayed LMS algorithm with matched coefficient and data vectors
; Algorithm runs in 5N (2 coeffs processed in each 10 cycle loop)
; Data Sample is stored in Y0 and Y1.
; Coefficient is stored in X0
; Loop Gain * Error is stored in X:(R2) (will be placed in X0).
; FIR operation done in B.
; Coeff update operation done in A.
; FIR sum = a  = a +c(k)old*x(n-k)
; c(k)new  = b = c(k)old -mu*eold *x(n-k-1)
   opt     cc
   move    #state,r0                        ; 2  2
   move    #ntaps,m0                        ; 2  2
   move    #c,r3                            ; 2  2
   move    #c-2,r1                          ; 2  2
   move    #0,n                             ; 1  1    emulate (Rn) adr
                                                      mode
   clr     b        x:(r0)+,y0              ; 1  1    y0 = x(n)
   move             x:(r0)+,y1   x:(r3)+,x  ; 1  1    y1= x(n-1), x0=c0h
                                 0
   do      #ntaps/2,end_lms2                ; 2  3
   mac     y0,x0,b  a,x:(r1)+               ; 1  1
   move             a0,x:(r1)+                 ; 1  1
   tfr      x0,a    x:(r2)+n,x0                ; 1  1
   move                          x:(r3)+,a0    ; 1  1    a0=ckl
   macr     x0,y1,a x:(r0)+,y0   x:(r3)+,x0    ; 1  1    x0=c(k+1)h
   mac      x0,y1,b a,x:(r1)+                  ; 1  1
   move             a0,x:(r1)+                 ; 1  1
   tfr      x0,a    x:(r2)+n,x0                ; 1  1
   move                          x:(r3)+,a0    ; 1  1
   macr     x0,y0,a x:(r0)+,y1   x:(r3)+,x0    ; 1  1
end_lms2
   move             a,x:(r1)+                  ; 1  1
   move             a0,x:(r1)+                 ; 1  1
   lea              (r0)-                      ; 1  1
   lea              (r0)-                      ; 1  1
;                                              _______
;                                Total:        27   5N+18
```

B.2.8 Vector Multiply-Accumulate

This code multiples a vector by a scalar and adds the result to another vector. The Y0 register holds the scalar value. **Figure B-6** gives a graphical overview and memory map for the vector multiply-accumulate code.

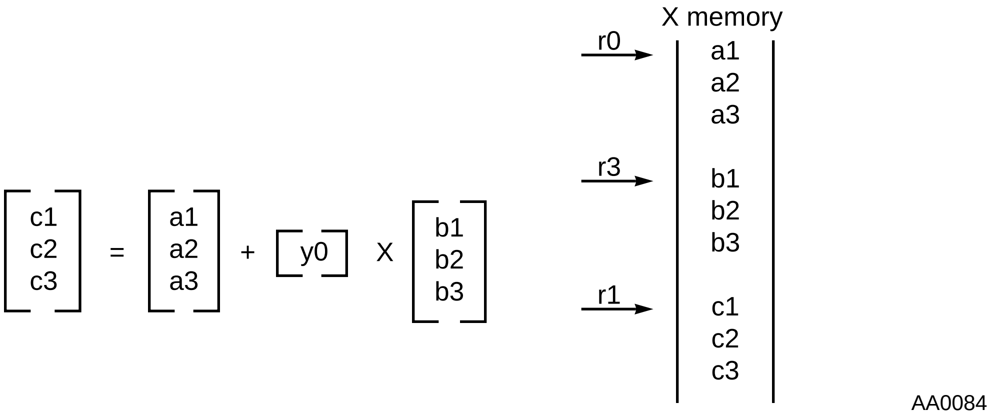


Figure B-6 Vector Multiply-Accumulate Memory Map

```
   opt     cc
   move    #ad,r0                            ; 2   2     point to vec a
   move    #bd,r3                            ; 2   2     point to vec b
   move    #cd,r1                            ; 2   2     point to vec c
   clr     a                     x:(r3)+,x0  ; 1   1
   move                x:(r0)+,a             ; 1   1
   do      #NUM,_vmac                        ; 2   3
   mac     y0,x0,a     x:(r0)+,y1  x:(r3)+,x0 ; 1   1
   tfr     y1,a        a,x:(r1)+             ; 1   1
_vmac
;                                            _______
;                                Total:      12    2N+11
```

B.2.9 Energy in a Signal

This code calculates the energy in a signal by summing together the square of each sample.

```
      opt  cc
      move #ad,r0                             ; 2  2    point to signal a
      nop                                     ; 1  1
      clr  a          x:(r0)+,a               ; 1  1
      do   #NUM,_energy                       ; 2  3
      mac  y0,y0,a    x:(r0)+,y0              ; 1  1
_energy
;                                             _______
;                                  Total:     7    1N+7
```

B.2.10 [3x3][1x3] Matrix Multiply

Figure B-7 gives a graphical overview and memory map for a [3x3][1x3] matrix multiply.

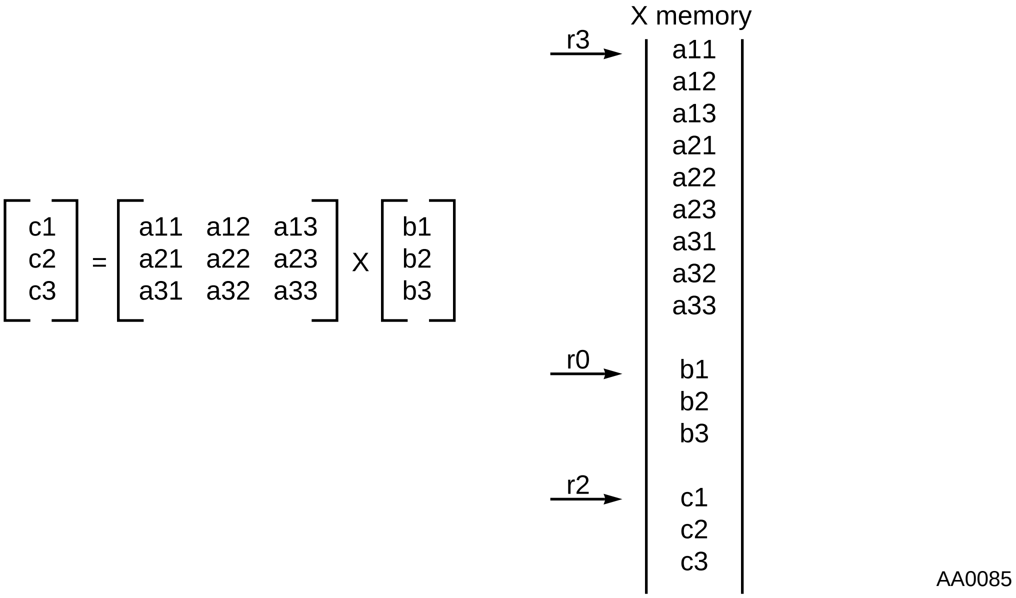


Figure B-7 [3x3][1x3] Matrix Multiply Memory Map

```
opt    cc
move   #AD,r3                                 ; 2   2     point to mat a
move   #bd,r0                                 ; 2   2     point to vec b
move   #2,m0                                  ; 1   1     addrb mod 3
move   #c,r2                                  ; 2   2     point to vec c
move             x:(r0)+,y0    x:(r3)+,x0     ; 1   1     y0=a11; x0=b1
mpy    y0,x0,a   x:(r0)+,y0    x:(r3)+,x0     ; 1   1     a11*b1
mac    y0,x0,a   x:(r0)+,y0    x:(r3)+,x0     ; 1   1     +a12*b2
macr   y0,x0,a   x:(r0)+,y0    x:(r3)+,x0     ; 1   1     +a13*b3
move             a,x:(r2)+                    ; 1   1     store c1
mpy    y0,x0,a    x:(r0)+,y0   x:(r3)+,x0     ; 1   1     a21*b1
mac    y0,x0,a   x:(r0)+,y0    x:(r3)+,x0     ; 1   1     +a22*b2
macr   y0,x0,a   x:(r0)+,y0    x:(r3)+,x0     ; 1   1     +a23*b3
```

```
      move            a,x:(r2)+                   ; 1   1     store c2
      mpy    y0,x0,a  x:(r0)+,y0   x:(r3)+,x0     ; 1   1     a31*b1
      mac    y0,x0,a  x:(r0)+,y0   x:(r3)+,x0     ; 1   1     +a32*b2
      macr   y0,x0,a                              ; 1   1     +a33*b3->c3
      move            a,x:(r2)+                   ; 1   1     store c3
;                                                 _______
;                                  Total:         20     20
```

B.2.11 [NxN][NxN] Matrix Multiply

The matrix multiplications are for square NxN matrices (all elements are in row-major format). **Figure B-8** gives a graphical overview and memory map of an [NxN][NxN] matrix multiply.

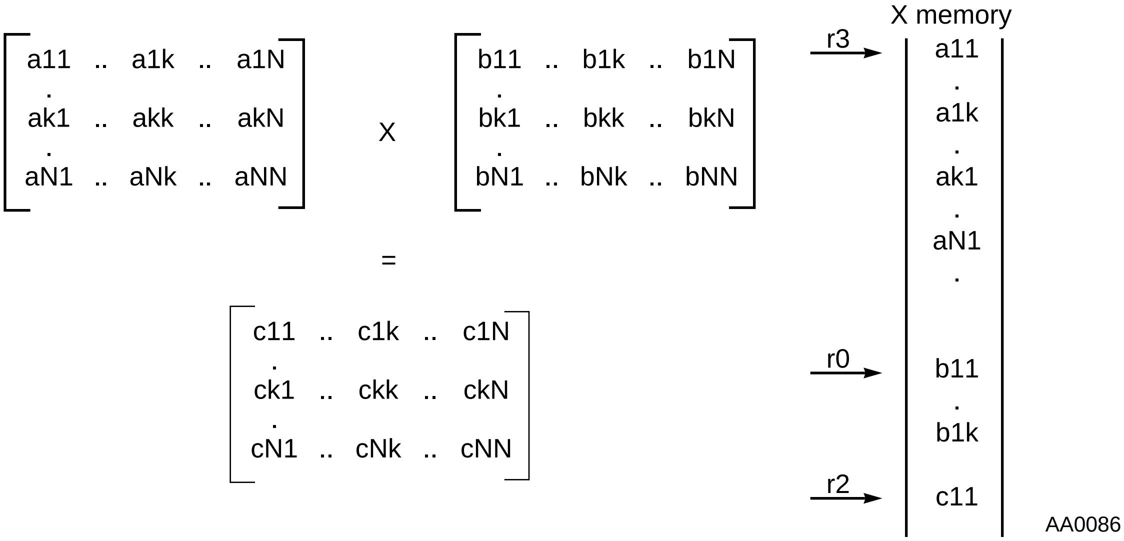


Figure B-8 [NxN][NxN] Matrix Multiply

```
opt  cc
move #ad,r0                                    ; 2  2    point to A
move r0,y1                                     ; 1  1    point to current column
move #bd,r3                                    ; 2  2    point to B
move #c,r2                                     ; 2  2    output mat C
move #N,b                                      ; 2  2    array size
move b,n                                       ; 1  1
push lc                                        ; 1  1
push la                                        ; 1  1
do   n,erows                                   ; 2  3    do rows
push lc                                        ; 1  1
push la                                        ; 1  1
do   n,ecols                                   ; 2  3    do columns
```

```
      move y1,r0                              ; 1  1    copy  row A
      move r1,r3                              ; 1  1    copy col B
      clr  a         x:(r0)+,y0               ; 1  1    clr sum & pipe
      move                        x:(r3)+n,x0 ; 1  1
      rep  #N-1                               ; 1  3    sum
      mac  y0,x0,a   x:(r0)+,y0   x:(r3)+n,x0 ; 1  1
      macr y0,x0,a                x:(r3)+,y0  ; 1  1    finish, next col
      move a,x:(r2)+                          ; 1  1    save output
ecols pop  la                                 ; 1  1
      pop  lc                                 ; 1  1
      add  y1,b                               ; 1  1    next row A
      move b,y1                               ; 1  1
      move #bd,r1                             ; 2  2    first element B
erows
      pop  la                                 ; 1  1
      pop  lc                                 ; 1  1
;                                             _______

;                    Words:       Cycles:
;          Total:    30           ((9+(N-1))N+10)N+12)= N^3+8N^2+10N+17
```

B.2.12 N Point 3x3 2-D FIR Convolution

The two dimensional FIR uses a 3x3 coefficient mask as shown in **Figure B-9**.

$$\begin{bmatrix} c11 & c12 & c13 \\ c21 & c22 & c23 \\ c31 & c32 & c33 \end{bmatrix}$$

AA0087

Figure B-9 3x3 Coefficient Mask

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 (see **Figure B-10**).

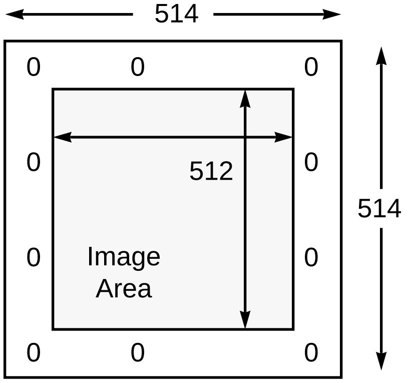


Figure B-10 Image Stored as 514x514 Array

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 d memory. For example:

- Image(1,1) maps to index 0
- Image(1,514) maps to index 513
- Image(2,1) maps to index 514

See **Table B-2** for the definiations of r0, r2, and r3.

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.

Table B-2 Variable Descriptions

Variable	Description		
r0	image(n,m)	image(n,m+1)	image(n,m+2)
	image(n+514,m)	image(n+514,m+1)	image(n+514,m+2)
	image(n+2*514,m)	image(n+2*514,m+1)	image(n+2*514,m+2)
r2	output image		
r3	FIR coefficients		

```
opt     cc
move    #coeffs,r3                          ; 2    2      pt to coef.
move    #image,r0                           ; 2    2      top boundary
move    #512,y1                             ; 2    2
move    #-1029,r1                           ; 2    2
move    #output,r2                          ; 2    2      output image
move               x:(r0)+,y0   x:(r3)+,x0  ; 1    1      y0=im(1,1 ),  x0=c11
move    y1,n                                ; 1    1      row i to i+1 adjust

push    lc                                  ; 1    1
push    la                                  ; 1    1
do      y1,rows                             ; 2    3
push    lc                                  ; 1    1
push    la                                  ; 1    1
do      y1,cols                             ; 2    3
mpy     y0,x0,a    x:(r0)+,y0   x:(r3)+,x0  ; 1    1      im(1,1)*c11
mac     y0,x0,a    x:(r0)+n,y0  x:(r3)+,x0  ; 1    1      +im(1,2)*c12
mac     y0,x0,a    x:(r0)+,y0   x:(r3)+,x0  ; 1    1      +im(1,3)*c13
mac     y0,x0,a    x:(r0)+,y0   x:(r3)+,x0  ; 1    1      +im(2,1)*c21
mac     y0,x0,a    x:(r0)+n,y0  x:(r3)+,x0  ; 1    1      +im(2,2)*c22
move    r1,n                                ; 1    1      row i to i-2 adjust
mac     y0,x0,a    x:(r0)+,y0   x:(r3)+,x0  ; 1    1      +im(2,3)*c23
```

```
        mac     y0,x0,a     x:(r0)+,y0    x:(r3)+,x0    ; 1    1    +im(3,1)*c31
        mac     y0,x0,a     x:(r0)+n,y0   x:(r3)+,x0    ; 1    1    +im(3,2)*c32
        move    #0,r3                                   ; 1    1    back to 1st coeff
        move    y1,n                                    ; 1    1    row i to i+1 adjust
        macr    y0,x0,a     x:(r0)+,y0    x:(r3)+,x0    ; 1    1    +im(3,3)*c33
        move    a,x:(r2)+                               ; 1    1
cols
        pop     la                                      ; 1    1
        pop     lc                                      ; 1    1
;  adjust pointers for frame boundary
        lea     (r0)+                                   ; 1    1    adjust r0
        lea     (r0)+                                   ; 1    1
        lea     (r2)+                                   ; 1    1    adjust r2
        lea     (r2)+                                   ; 1    1
rows
        pop     la                                      ; 1    1
        pop     lc                                      ; 1    1
                                                        ; ________
;                                         Total:        41     13N^2+11N+16
;                                                              Kernel: 13
```

B.2.13 Sine Wave Generation

The following two sine wave generation benchmarks are provided:

- Double integration technique
- Second order oscillator

B.2.13.1 Double Integration Technique

Figure B-11 gives a graphical overview of the double integration technique.

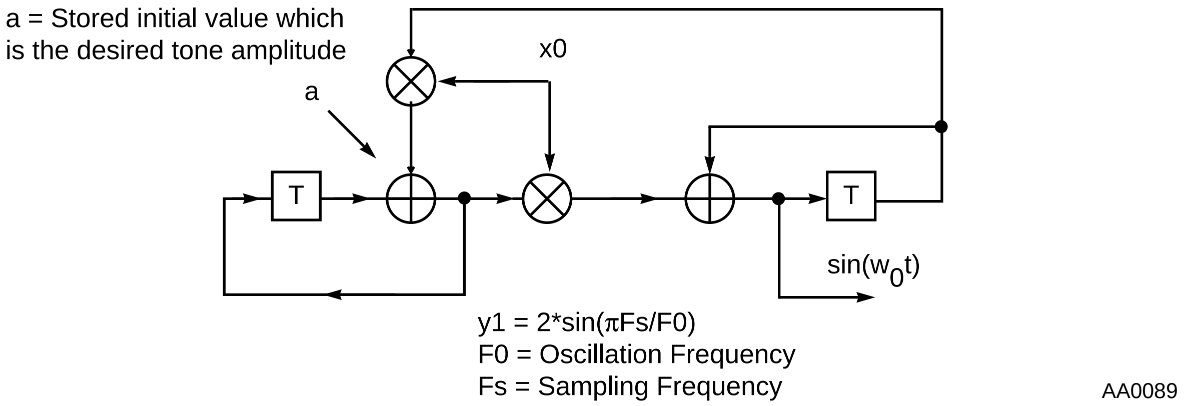

a = Stored initial value which
is the desired tone amplitude
a
x0
T
T
$\sin(w_0t)$
y1 = 2*sin(πFs/F0)
F0 = Oscillation Frequency
Fs = Sampling Frequency
AA0089

Figure B-11 Sine Wave Generator—Double Integration Technique

```
    opt    cc
    clr    b                                ; 1   1
    move   #$4000,a                         ; 2   2
    move   #0,n                             ; 1   1

    move   #$4532,y1                        ; 2   2
    move   #$1,r1                           ; 1   1
    move   y1,y0                            ; 1   1

    do     x0,loop1                         ; 2   3
    mac    y1,b1,a      b,x:(r1)+n          ; 1   1
    mac    -y0,a1,b                         ; 1   1
loop1
```

```
    move               b,x:(r1)             ; 1  1
;                                           _______
;                                   Total:  13   2N+12
```

B.2.13.2 Second Order Oscillator

Figure B-12 gives a graphical overview of a second order oscillator.

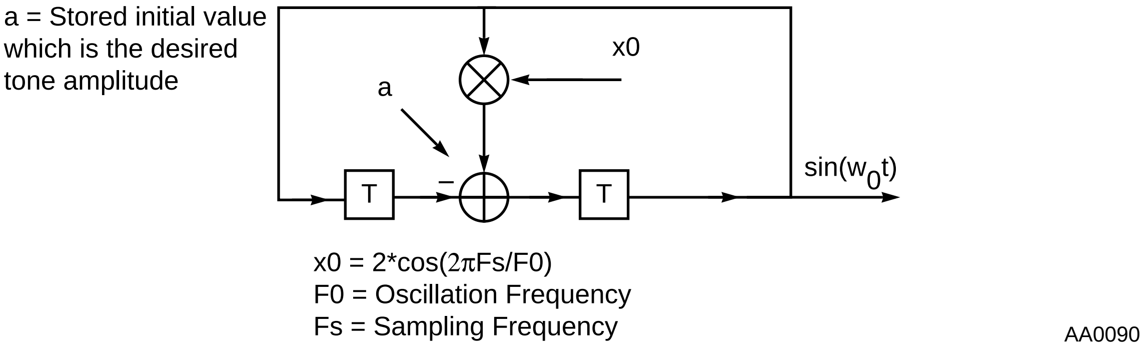


Figure B-12 Sine Wave Generator—Second Order Oscillator

```
      opt     cc
      clr     a                        ; 1    1
      move    #$4000,y1                ; 2    2
      move    #$6d4b,y0                ; 2    2
      move    #$1,r1                   ; 1    1
      move    #tmp,r0                  ; 1    1
      move    #0,n                     ; 1    1
      do      x0,loop2                 ; 2    3
      mac     -y1,y0,a                 ; 1    1
      neg     a           y1,x:(r1)+n  ; 1    1
      mac     y1,y0,a                  ; 1    1
      move                a,x:(r0)+n   ; 1    1   temp storage for swap
      tfr     y1,a        x:(r0)+n,y1  ; 1    1
loop2
      move                y1,x:(r1)    ; 1    1
;                                      ______
;                         Total:       16     5N+12
```

B.2.14 Array Search

The following two array search benchmarks are provided:

- Index of the highest signed value
- Index of the highest positive value

B.2.14.1 Index of the Highest Signed Value

```
         opt   cc
         move  #AD,r0                       ; 2   2
         clr   a          x:(r0)+,b         ; 1   1

         do    #N,end_lp3                   ; 2   3
         abs   b                            ; 1   1
         cmp   b,a                          ; 1   1
         tle   b,a        r0,r1             ; 1   1
         move  x:(r0)+,b                    ; 1   1
end_lp3
         lea   (r1)-                        ; 1   1
         lea   (r1)-                        ; 1   1
;                                           ________
;                                    Total: 10    4N+8   (worst case)
```

B.2.14.2 Index of the Highest Positive Value

```
        opt     cc
        move    #AD,r0                              ; 2     2
        clr     a            x:(r0)+,x0             ; 1     1

        do      #N/2,end_lp3                        ; 2     3
        cmp     x0,a         x:(r0)+,y0             ; 1     1
        tle     x0,a         r0,r1                  ; 1     1
        cmp     y0,a         x:(r0)+,x0             ; 1     1
        tle     y0,a         r0,r1                  ; 1     1
end_lp3
        lea     (r1)-                               ; 1     1
        lea     (r1)-                               ; 1     1
;                                                   _______
;                                            Total: 10      2N+8    (worst case)
```

B.2.15 Proportional Integrator Differentiator (PID) Algorithm

The proportional integrator differentiator (PID) algorithm is the most commonly used algorithm in control applications. **Figure B-13** gives a graphical overview and memory map of this implementation of a proportional integrator differentiator.

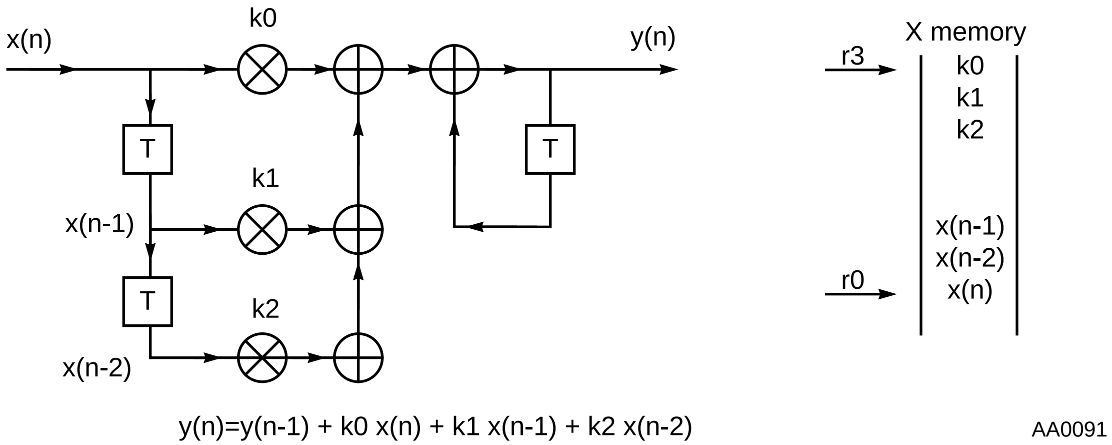


Figure B-13 Proportional Integrator Differentiator Algorithm

```
;       y(n) = y(n-1) + k0 x(n) + k1 x(n-1) + k2 x(n-2)
        opt   cc
        move  #s+2,r0                         ;
        move  #2,m0                           ; r0 mod 3
        move  #k,r3                           ;

        move           x:(r0)+,b              ; 1     1 get y(n-1)
        move           x:(r0)+,y0  x:(r3)+,y0 ; 1     1 get x(n-2),k2
        mac   x0,y0,b  x:(r0)+,y0  x:(r3)+,y0 ; 1     1 get x(n-1),k1
        mac   y0,x0,b              x:(r3)+,x0 ; 1     1 get k0
        movep          x:input,b              ; 1     1 get x(n)
        macr  y0,x0,b                         ; 1     1
        move           b,x:(r0)               ; 1     1 save y(n)
        movep          b,x:output             ; 1     1 y(n) in b
;                                             _________
;                          Total:             8       8
;    A faster version of the PID
;    y(n) = y(n-1) + k0 x(n) + k1 x(n-1) + k2 x(n-2)
     opt   cc
     move  #s+2,r0                               ;
     move  #2,m0                                 ; r0 mod 3
     move  #k,r3                                 ;
;    B accumulator holds y(n-1), Y1 holds the K0 coefficient
     move             x:(r0)+,y0  x:(r3)+,y0 ; 1    1 get x(n-2),k2
     mac   x0,y0,b    x:(r0)+,y0  x:(r3)-,x0 ; 1    1 get x(n-1),k1
     mac   y0,x0,b                           ; 1    1
     movep            x:input,b              ; 1    1 save x(n)
     macr  y0,x0,b    b,x:(r0)+              ; 1    1 save x(n)
     movep            b,x:output             ; 1    1 y(n) in b
;                                            _________
;                                            6      6
```

B.2.16 Autocorrelation Algorithm

```
     move  #cor,r1                          ; 2    2
     move  #frame,r2                        ; 2    2
     do    #lpc+1,_loop1                    ; 2    3
     move  r2,r3                            ; 1    1
     clr   b                                ; 1    1
     move  #frame,r0                        ; 2    2
     lea   (r2)+                            ; 1    1
     move  lc,y1                            ; 1    1
     move  #>N-(p+1),a                      ; 2    2
     add   y1,a       x:(r0)+,y0 x:(r3)+,x0 ; 1    1
     rep   a                                ; 1    3
     mac   y0,x0,b    x:(r0)+,y0 x:(r3)+,x0 ; 1    1
     move  b0,x:(r1)+                       ; 1    1
     move  b1,x:(r1)+                       ; 1    1
_loop1                                      ; ______
;                                           23     (p+1)^2(N-p/2)+15(p+1) +6
```

GLOSSARY

A/D	analog-to-digital
ADM	application development module
ADS	application development system
AGU	address generation unit
ALU	arithmetic logic unit

See **Notation** on page A-3 for notations and symbols not listed here.

A/D	analog-to-digital
ADM	application development module
ADS	application development system
AGU	address generation unit
ALU	arithmetic logic unit
AS	accumulator shifter
BCR	bus control register
BE1-BE0	breakpoint enable bits
BK4-BK0	breakpoint configuration bits
BS1-BS0	breakpoint selection bits
C	carry bit
CC	condition code

CCR	condition code register
CID	chip identification register
COFF	common object file format
COP	computer operating properly
COPDIS	COP timer disable
CPU	central processing unit
CMOS	complementary metal oxide semiconductor
D/A	digital-to-analog
DE	$\overline{\text{DE}}$ pint output enable bit
DRM	debug request mask bit
DSP	digital signal processor
E	extension bit
EM1-EM0	event modifier bits

EX	external X memory
EXT	extension register
CGDB	core global data bus
FH	FIFO halt bit
FIFO	first-in-last-out
GPIO	general-purpose input/output
GUI	graphical user interface
HBO	hardware breakpoint occurrence
HI	high
HS	high or same
HWS	hardware stack
I1, I0	interrupt mask bits
IC	integrated circuit

JTAG	Joint Test Access Group
I/O	input/output
IPR	interrupt priority register
IPL	interrupt priority level
K&R	Kernighan and Ritchie
L	limit bit
LA	loop address register
LC	loop counter register
LF	loop flag bit
LIFO	last-in-first-out
LO	low
LS	least significant; low or same
LSB	least significant bit

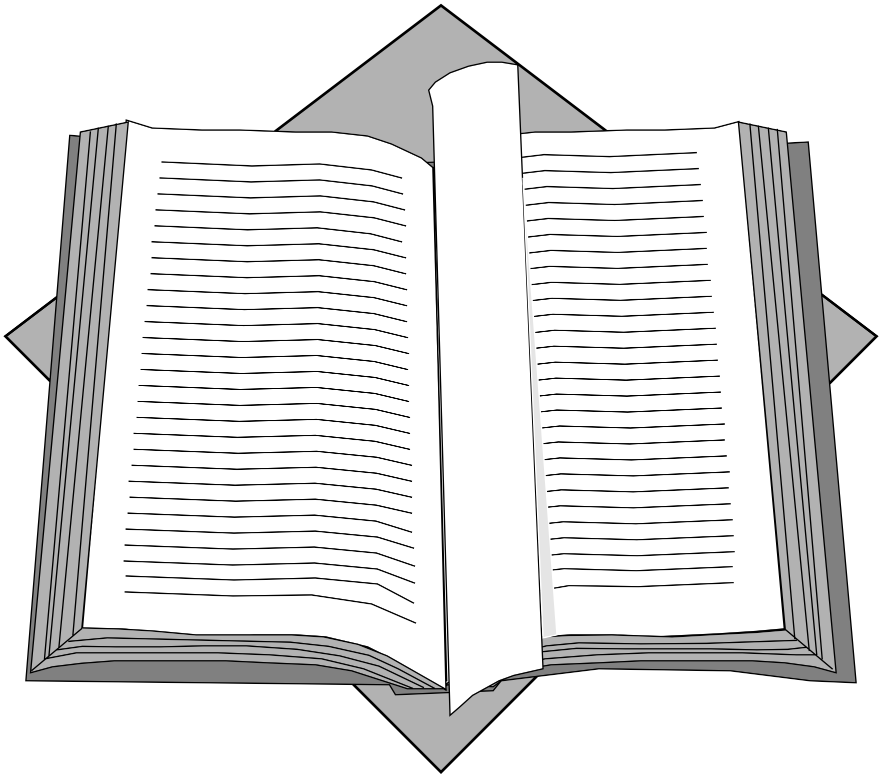
LSP	least significant portion
MA, MB	operating modes
MAC	multiply / accumulate
MCU	microcontroller unit
MIPS	million instructions per second
MO1	modifier register
MR	mode register
MS	most significant
MSB	most significant bit
MSP	most significant portion
N	offset register
NL	nested looping bit
OBAR	OnCE breakpoint address register

OCMDR	OnCE command register
OCNTR	OnCE breakpoint counter
ODEC	OnCE decoder
OISR	OnCE input shift register
OMAC	OnCE memory address comparator
OMAL	OnCE breakpoint address latch
OMR	operating mode register
OPABDR	OnCE PAB decode register
OPABER	OnCE PAB execute register
OPABFR	OnCE PAB fetch register
OPDBR	OnCE PDB register
OPGDBR	Once PGDB register
OS1, OS0	OnCE status bits

OSR	OnCE status register
OnCE™	On-Chip Emulation unit
P2-P0	program counter extension
PAB	program address bus
PC	program counter
PGDB	peripheral global data bus
PWD	power down mode bit
PLL	phase-locked loop
R	rounding
Rn	address registers (R0-R3)
SBO	software breakpoint occurrence
SD	stop delay
SP	stack pointer

SPI	serial peripheral interface
SR	status register
SSI	synchronous serial interface
TAP	test access port
TO	trace occurrence
V	overflow bit
WWW	World Wide Web
X	external
XAB1	external address bus one
XAB2	external address bus two
XDB2	external data bus two
XP	X/P memory bit

INDEX



OnCE™ is a trademark of Motorola, Inc.

HP-UX™ is a trademark of Hewlett-Packard.

IBM™ is a trademark of International Business Machines.

Macintosh™ is a trademark of Apple Computer, Inc.

SBUS™, SUN-4™, and SunOS™ are trademarks of Sun Microsystems, Inc.

Windows™ is a trademark of Microsoft Corporation.

Order this document by DSP56800FM / AD

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not authorized for use as components in life support devices or systems intended for surgical implant into the body or intended to support or sustain life. Buyer agrees to notify Motorola of any such intended end use whereupon Motorola shall determine availability and suitability of its product or products for the use intended. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Employment Opportunity / Affirmative Action Employer.