DSP56800

Family Manual

16-Bit Digital Signal Controllers

DSP56800FM
Rev. 3.1
11/2005

freescale.com

freescale™
semiconductor

Contents

Chapter 1
Introduction

Chapter 2
Core Architecture Overview

Chapter 3
Data Arithmetic Logic Unit

Chapter 4
Address Generation Unit

Chapter 5
Program Controller

Chapter 6
Instruction Set Introduction

Chapter 7
Interrupts and the Processing States

Chapter 8
Software Techniques

Chapter 9
JTAG and On-Chip Emulation (OnCE™)

Appendix A
Instruction Set Details

Appendix B
DSC Benchmarks

List of Tables

List of Figures

List of Examples

About This Book

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 DSC algorithms
- Write code for general control tasks
- Write code for communication routines
- Write code for data manipulation algorithms

Audience

The information in this manual is intended to assist design and software engineers with integrating a DSP56800 Family device into a design and with developing application software.

Organization

Information in this manual is organized into chapters by topic. The contents of the chapters are as follows:

Chapter 1, "Introduction." This section introduces the DSP56800 core architecture and its application. It also provides the novice with a brief overview of digital signal processing.

Chapter 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.

Chapter 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.

Chapter 4, "Address Generation Unit." This section specifically describes the AGU architecture and its programming model, addressing modes, and address modifiers.

Chapter 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 Chapter 7, "Interrupts and the Processing States."

Chapter 6, "Instruction Set Introduction." This section presents an introduction to parallel moves and a brief description of the syntax, instruction formats, operand and 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."

Chapter 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 Chapter 9, "JTAG and On-Chip Emulation (OnCE™)."

Chapter 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.

Chapter 9, "JTAG and 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.

Appendix A, "Instruction Set Details." This section presents a detailed description of each DSP56800 Family instruction, its use, and its effect on the processor.

Appendix B, "DSP Benchmarks." DSP56800 Family benchmark example programs and results are listed in this appendix.

Suggested Reading

A list of DSC-related books is included here as an aid for the engineer who is new to the field of DSC:

Advanced Topics in Signal Processing, Jae S. Lim and Alan V. Oppenheim (Prentice-Hall: 1988).

Applications of Digital Signal Processing, A. V. Oppenheim (Prentice-Hall: 1978).

Digital Processing of Signals: Theory and Practice, Maurice Bellanger (John Wiley and Sons: 1984).

Digital Signal Processing, Alan V. Oppenheim and Ronald W. Schafer (Prentice-Hall: 1975).

Digital Signal Processing: A System Design Approach, David J. DeFatta, Joseph G. Lucas, and William S. Hodgkiss (John Wiley and Sons: 1988).

Discrete-Time Signal Processing, A. V. Oppenheim and R.W. Schafer (Prentice-Hall: 1989).

Foundations of Digital Signal Processing and Data Analysis, J. A. Cadzow (Macmillan: 1987).

Handbook of Digital Signal Processing, D. F. Elliott (Academic Press: 1987).

Introduction to Digital Signal Processing, John G. Proakis and Dimitris G. Manolakis (Macmillan: 1988).

Multirate Digital Signal Processing, R. E. Crochiere and L. R. Rabiner (Prentice-Hall: 1983).

Signal Processing Algorithms, S. Stearns and R. Davis (Prentice-Hall: 1988).

Signal Processing Handbook, C. H. Chen (Marcel Dekker: 1988).

Signal Processing: The Modern Approach, James V. Candy (McGraw-Hill: 1988).

Theory and Application of Digital Signal Processing, Lawrence R. Rabiner and Bernard Gold (Prentice-Hall: 1975).

Conventions

This document uses the following notational conventions:

- Bits within registers are always listed from most significant bit (MSB) to least significant bit (LSB).
- Bits within a register are formatted AA[n:0] when more than one bit is involved in a description. For purposes of description, the bits are presented as if they are contiguous within a register. However, this is not always the case. Refer to the programming model diagrams or to the programmer's sheets to see the exact location of bits within a register.
- When a bit is described as "set," its value is set to 1. When a bit is described as "cleared," its value is set to 0.
- Memory addresses in the separate program and data memory spaces are differentiated by a one-letter prefix. Data memory addresses are preceded by "X:" while program memory addresses have a "P:" prefix. For example, "P:$0200" indicates a location in program memory.
- Hex values are indicated with a dollar sign ($) preceding the hex value, as follows: $FFFB is the X memory address for the Interrupt Priority Register (IPR).
- Code examples are displayed in a monospaced font, as follows:

```
BFSET  #$0007,X:PCC ; Configure:                                  line 1

                    ; MISO0, MOSI0, SCK0 for SPI master           line 2

                    ; ~SS0 as PC3 for GPIO                        line 3
```

Definitions, Acronyms, and Abbreviations

The following terms appear frequently in this manual:

DSC	digital signal controller
JTAG	Joint Test Action Group
OnCE™	On-Chip Emulation
ALU	arithmetic logic unit
AGU	address generation unit

A complete list of relevant terms is included in the Glossary at the end of this manual.

Chapter 1
Introduction

The DSP56800 Digital Signal Controllers provide low cost, low power, mid-performance computing, combining DSC power and parallelism with MCU-like programming simplicity. The DSP56800 core is a general-purpose central processing unit, designed for both efficient digital signal processing and a variety of controller operations.

1.1 DSP56800 Family Architecture

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

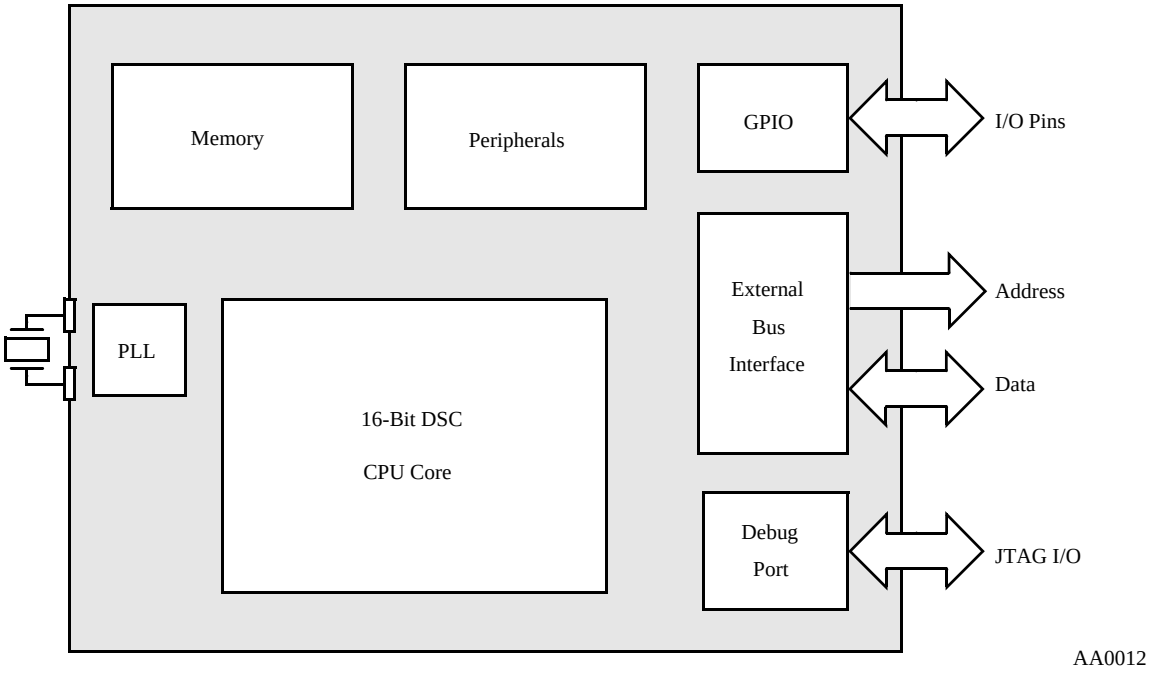


Figure 1-1. DSP56800-Based DSC Microcontroller Chip

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 DSCs. A software stack allows for unlimited interrupt and subroutine nesting, as well as support for structured programming techniques such as parameter passing

and the use of local variables. The veteran DSC programmer sees a powerful DSC 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 on page 1-1) such as serial ports, general-purpose timers, real-time and watchdog timers, different memory configurations (RAM, FLASH, or both), and general-purpose I/O (GPIO) ports.

On-Chip Emulation (OnCE™) capability is provided through a debug port conforming to the Joint Test Action Group (JTAG) standard. This provides real-time, embedded system debugging with on-chip emulation capability through the five-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 industry-standard debug support make the DSP56800 Family an excellent solution for real-time embedded control tasks. It is an excellent fit for wireless or wireline DSC 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 controller 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 on page 1-3 shows a block diagram of the DSP56800 core. The main features of the DSP56800 core include the following:

- Processing capability of up to 35 million instructions per second (MIPS) at 70 MHz
- Requires only 2.7–3.6 V of power
- Single-instruction cycle 16-bit x 16-bit parallel multiply-accumulator
- Two 36-bit accumulators including extension bits
- Single-instruction 16-bit barrel shifter
- Parallel instruction set with unique DSC addressing modes
- Hardware DO and REP loops
- Two external interrupt request pins
- Four 16-bit internal core data buses
- Three 16-bit internal address buses
- Instruction set that supports both DSC and controller functions
- Controller-style addressing modes and instructions for smaller code size
- Efficient C compiler and local variable support
- 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

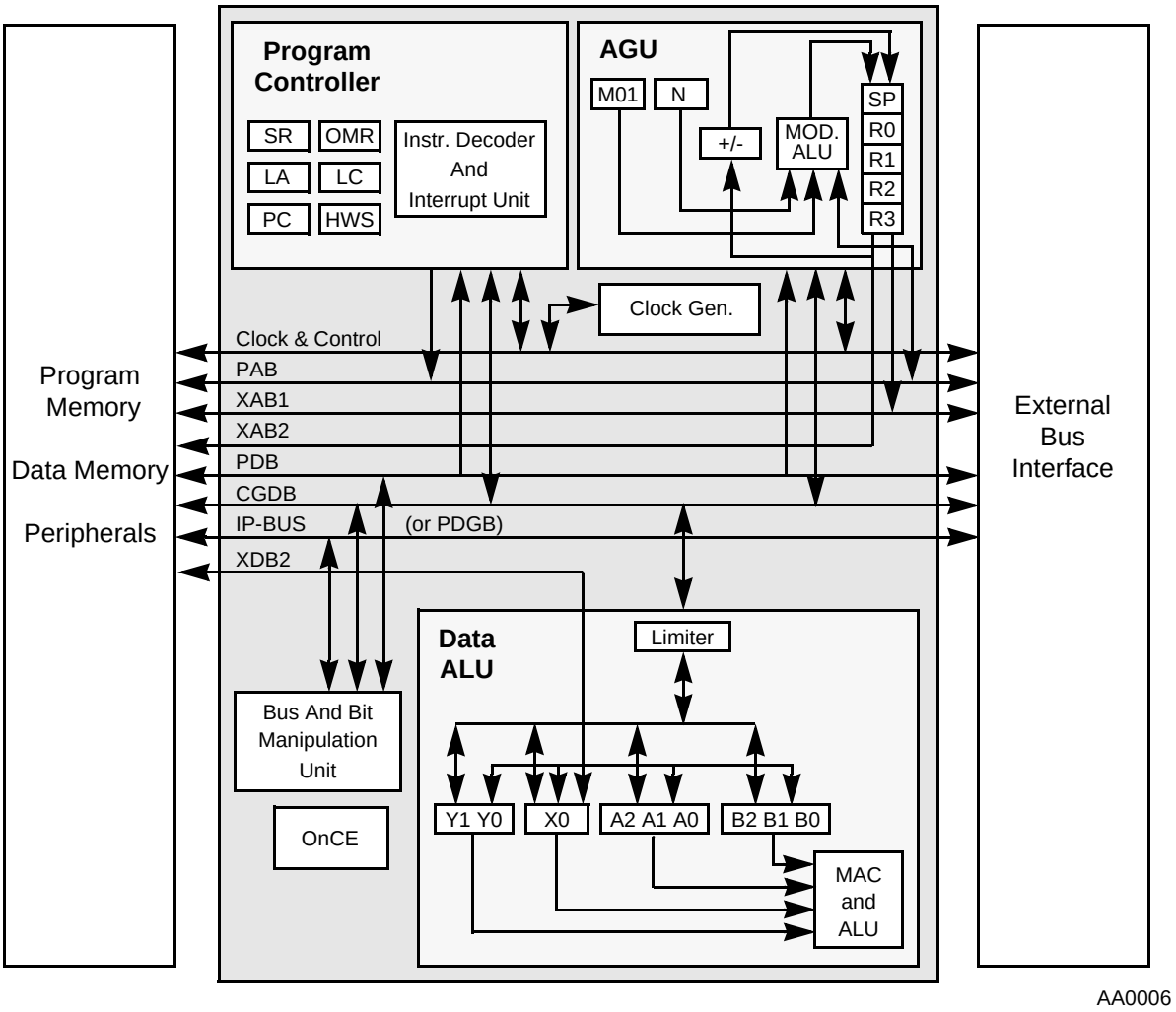


Figure 1-2. DSP56800 Core Block Diagram

1.1.2 Peripheral Blocks

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

- Program FLASH and RAM modules
- Bootstrap FLASH for program RAM parts
- Data FLASH and RAM modules
- Phase-locked loop (PLL) module
- General purpose Quad Timers
- Computer operating properly (COP) module
- Serial Communication Interfaces (SCIs)
- Synchronous serial interface module (SSI)
- Serial peripheral interface (SPI)
- Quadrature Decoders
- Controller Area Network (CAN) Modules

- Multiple channels Pulse Width Modulation (PWM) Modules
- External Memory Interface (EMI)
- Multiple channels Analog-to-Digital Converters (ADC)
- Programmable general-purpose I/O (dedicated & shared)
- JTAG/OnCE port for debugging

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 family of Freescale DSCs. An example of a chip (56F807) built with this core is shown in Figure 1-3.

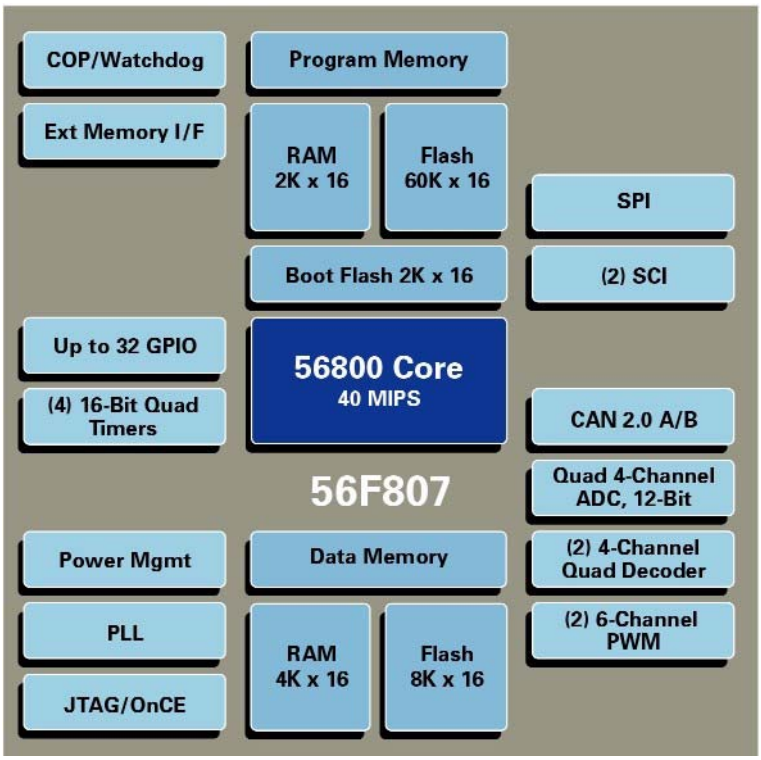


Figure 1-3. Example of Chip Built Around the DSP56800 Core

1.2 Introduction to Digital Signal Processing

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

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

Figure 1-4 on page 1-6 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 by 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.

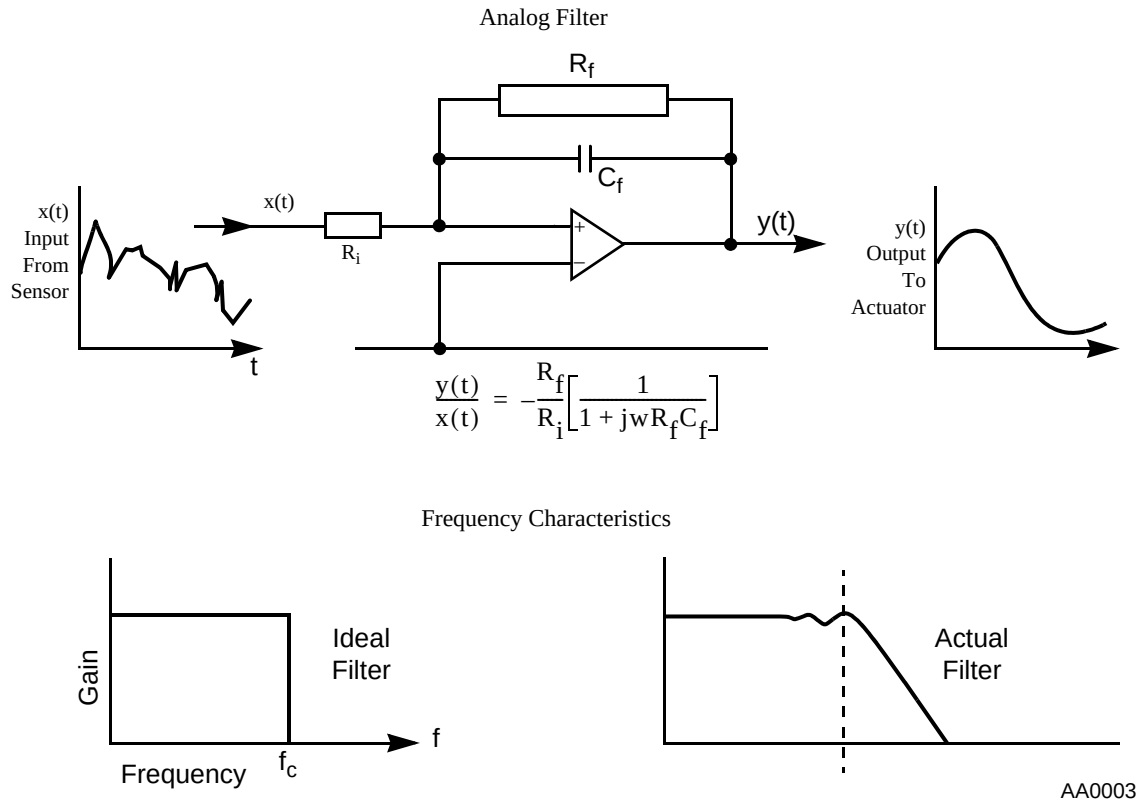


Figure 1-4. Analog Signal Processing

The equivalent circuit using a DSC is shown in Figure 1-5 on page 1-7. This application requires an analog-to-digital (A/D) converter and digital-to-analog (D/A) converter in addition to the DSC. Even with these additional parts, the component count can be lower using a DSC due to the high integration available with current components.

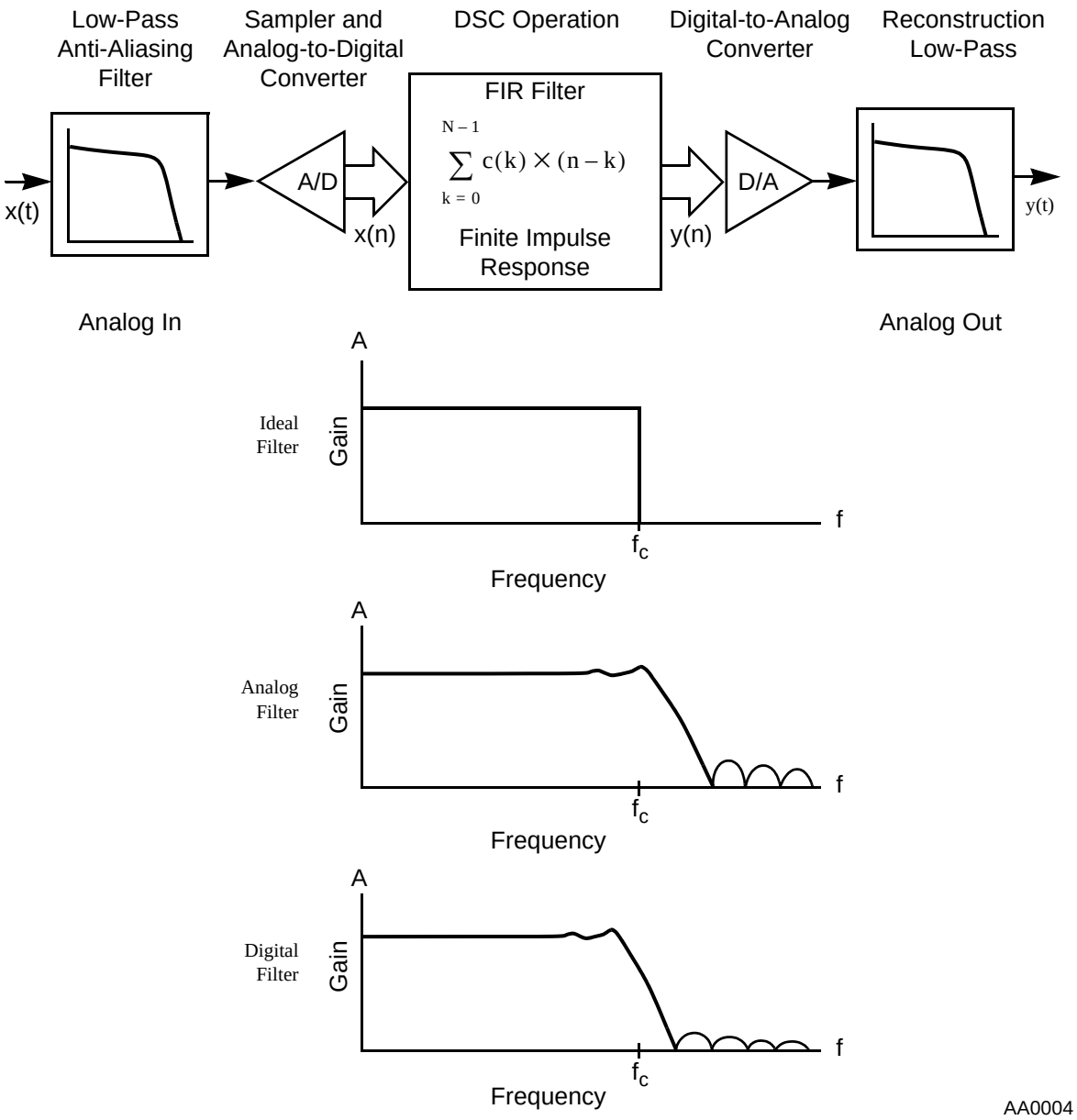


Figure 1-5. Digital Signal Processing

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 DSC.

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

The DSC 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 DSC 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 DSC architecture to efficiently execute commonly used DSC benchmarks and controller code in minimal time.

As shown in Figure 1-6, the key attributes of a DSC 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 DSC

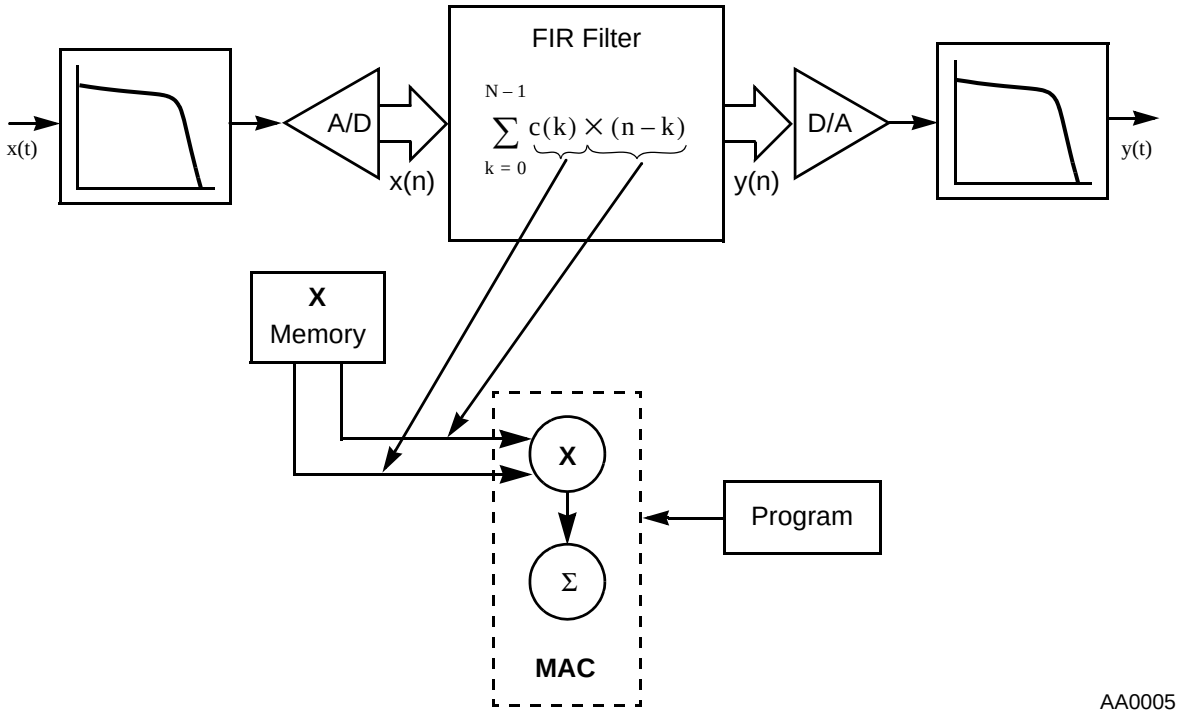


Figure 1-6. Mapping DSC Algorithms into Hardware

The multiply-accumulation (MAC) operation is the fundamental operation used in DSC. The DSP56800 Family of processors has a dual Harvard architecture optimized for MAC operations. Figure 1-6 on page 1-8 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 DSC can perform two memory moves, a multiply and an accumulate, and two address updates in a single operation. As a result, many DSC benchmarks execute very efficiently for a single-multiplier architecture.

1.3 Summary of Features

The high throughput of the DSP56800 Family 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 DSC 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 DSCs 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**— Freescale'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 For the Latest Information

For the latest electronic version of this document, as well as other DSC documentation (including user's manuals, product briefs, data sheets, and errata) please consult the inside front cover of this manual for contact information for the following services:

- Freescale DSC World Wide Web site
- Freescale DSC Helpline

The DSC Web site maintain the most current specifications, documents, and drawings.

Chapter 2
Core Architecture Overview

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 DSC integrated circuit family members can be designed with different on-chip and off-chip memory sizes and on-chip peripheral requirements. This chapter 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 chapters.

2.1 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 multiplication in a first instruction, for the AGU to generate up to two addresses for a second instruction, and for 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

Figure 2-1 on page 2-2 shows a block diagram of the CPU architecture.

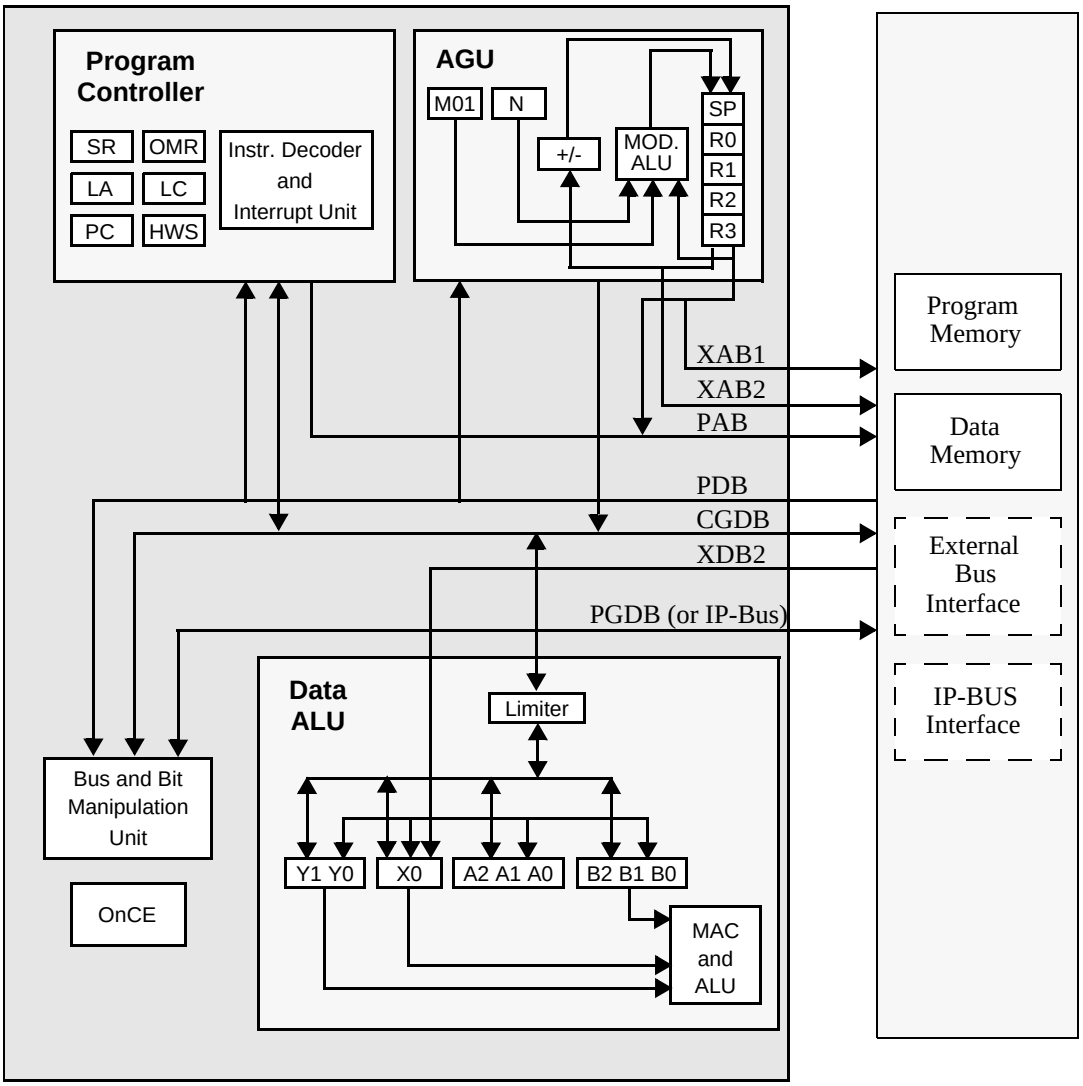


Figure 2-1. DSP56800 Core Block Diagram

Note that Figure 2-1 illustrates two methods for connecting peripherals to the DSP56800 core: using the Freescale-standard IP-BUS interface or via a dedicated Peripheral Global Data Bus (PGDB). The interface method used to connect to peripherals is dependent on the specific DSP56800-based device being used. The latest products have chosen the IP-BUS interface. Consult your device user's manual for more information on peripheral interfacing.

2.1.1 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 36-bit accumulator registers (A and B)
  - 16-bit registers (A0 and B0)
  - 16-bit registers (A1 and B1)
  - 4-bit 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-accumulation (with positive or negative accumulation), addition, subtraction, shifting, and logical operations in one instruction cycle. Arithmetic operations are done using two'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 X memory data bus two (XDB2) as a 16-bit operand. Refer to Chapter 3, "Data Arithmetic Logic Unit," for a detailed description of the data ALU.

2.1.2 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 X memory address bus one (XAB1) or program address bus (PAB) and one for X memory address bus two (XAB2). The ALU can directly address 65,536 locations on the XAB1 or XAB2 and 65,536 locations on the PAB, totaling 131,072 sixteen-bit data words. It supports a complete set of addressing modes. Its arithmetic unit can perform 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 Chapter 4, “Address Generation Unit,” for a detailed description of the AGU.

2.1.3 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)

In addition to the tasks listed above, the program controller also controls the memory map and operating mode. The operating mode and memory map are programmable via the OMR, and are established after reset by external interface 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 Chapter 5, “Program Controller.” For more details on program looping, refer to Section 5.3, “Program Looping,” on page 5-14 and Section 8.6, “Loops,” on page 8-20. For information on reset and interrupts, refer to Chapter 7, “Interrupts and the Processing States.”

2.1.4 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 internal data buses together without introducing delays. This allows data to be moved from program to data memory, for example. The bus unit is also used to transfer data to the IP-Bus (or PGDB) on those devices that use it to connect to on-chip peripherals.

The bit-manipulation unit performs bit-field manipulations on X (data) 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 (that is, the mask can only test up to 8 bits at a time).

Note that when the IP-BUS (or PGDB) interface is used, peripheral registers may be memory mapped into any data (X) memory address range and are accessed with standard X-memory reads and writes. If the peripheral registers are mapped to the last 64 locations in X memory, these can be accessed with a special memory addressing mode (see Section 4.2.4.3, “I/O Short Address (Direct Addressing): <pp>,” on page 4-23).

2.1.5 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 DSC 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 Chapter 9, “JTAG and On-Chip Emulation (OnCE™),” for a detailed description of the JTAG/OnCE port. Consult your development system’s documentation for information on debugging using the JTAG/OnCE port interface.

2.1.6 Address Buses

Addresses are provided to the internal X data memory on two unidirectional 16-bit buses, X memory address bus one (XAB1) and X memory address bus two (XAB2). Program memory addresses are provided on the 16-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.

2.1.7 Data Buses

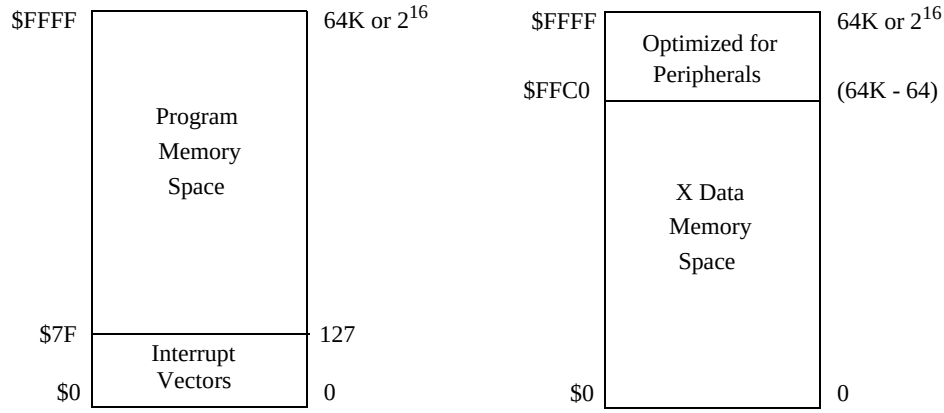
Inside the chip, data is transferred using the following:

- Bidirectional 16-bit buses:
  - Core global data bus (CGDB)
  - Program data bus (PDB)
  - IB-BUS or Peripheral Global data bus (PGDB) — dependent on chip implementation
- One unidirectional 16-bit bus: X memory 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 occur using the CGDB, except transfers to and from peripherals on DSP56800-based devices that implement the IP-BUS or PGDB peripheral data bus. 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 any register less than 16 bits wide is read, the unused bits are read as zeros. Reserved and unused bits should always be written with zeros to insure future compatibility.

2.2 Memory Architecture

The DSP56800 has a dual Harvard memory architecture, with separate program and data memory spaces. Each address space supports up to 2^{16} (65,536) memory words. Dedicated address and data buses for each address space allow for simultaneous accesses to both program memory and data memory. There is also a support for a second read-only data path to data memory. In DSP56800 Family devices that implement this second bus, it is possible to initiate two simultaneous data read operations, allowing for a total of three parallel memory accesses.



NOTE: The placement of the peripheral space is dependent on the specific system implementation for the DSP56800 core. When the IP-BUS interface is used, peripheral registers may be memory mapped into any data (X) memory address range and are accessed with standard X-memory reads and writes.

Figure 2-2. DSP56800 Memory Spaces

Locations $0 through $007F in the program memory space are available for reset and interrupt vectors. Peripheral registers are located in the data memory address space as memory-mapped registers. This peripheral space can be located anywhere in the data address space, although the address range $FFC0–$FFFF provides faster access when using an addressing mode optimized for this region; however, the location of the peripheral space is dependent on the specific peripheral bus implementation of the DSP56800 core. See Section 4.2.4.3, “I/O Short Address (Direct Addressing): <pp>,” on page 4-23 for more information.

2.3 Blocks Outside the DSP56800 Core

The following blocks are optionally found on DSP56800-based DSC chips and are considered peripheral and memory blocks, not part of the DSP56800 core. These and other blocks are described in greater detail in the appropriate chip-specific user's manual. Figure 2-3 shows an example DSP56800-based device. Note that this device uses the Freescale IP-BUS interface to connect to peripherals. Other chips may use the PGDB peripheral bus.

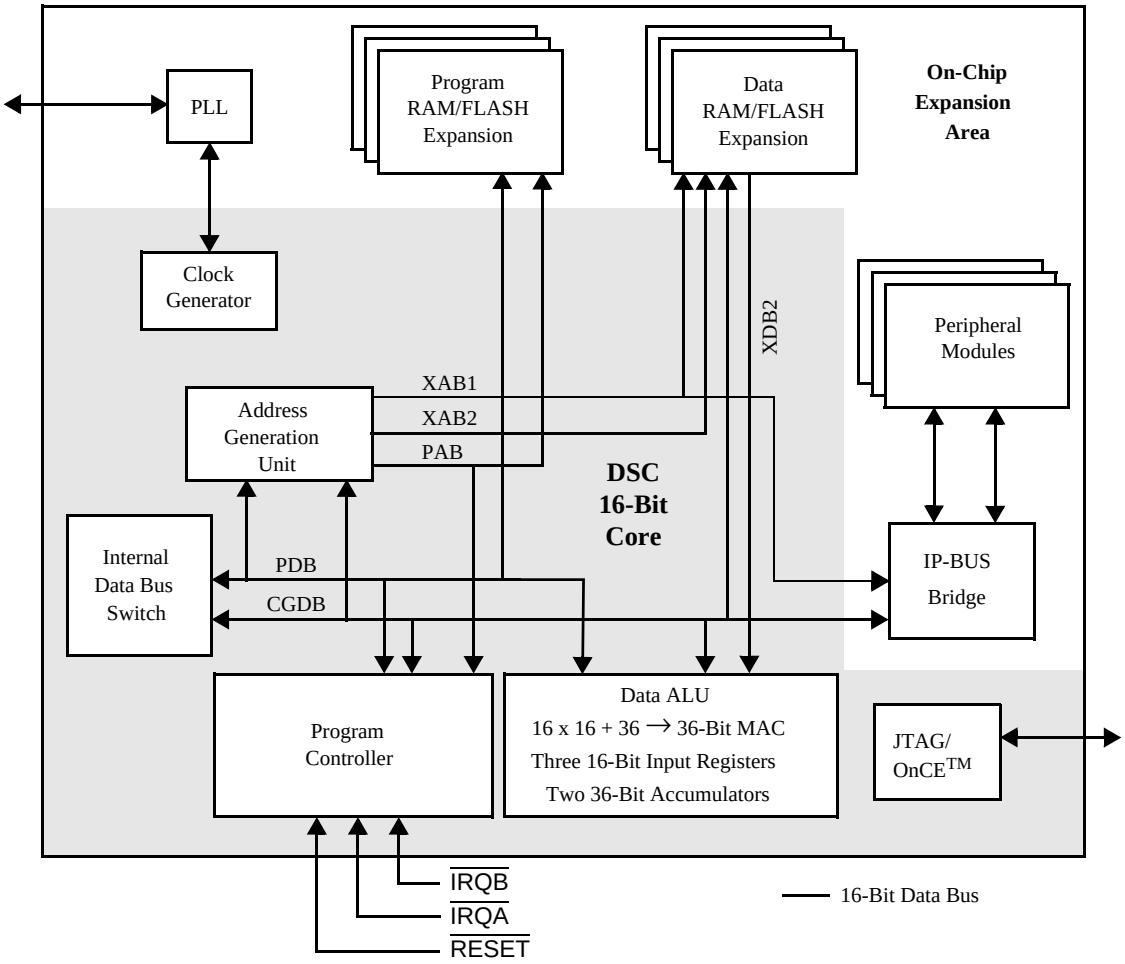


Figure 2-3. Sample DSP56800-Family Chip Block Diagram

2.3.1 External Data Memory

External data memory (data RAM, data FLASH, or both) 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. Depending upon the particular on-chip peripherals found on a device, some portion of the data address space may be reserved for peripheral registers, and not be accessible as external data memory. A total of 65,536 memory locations can be addressed.

2.3.2 Program Memory

Program memory (program RAM, program FLASH, or both) 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 FLASH is usually found on chips that have on-chip program RAM. The bootstrap FLASH is used for initially loading application code into the on-chip program RAM so it can be run from there. Refer to Section 5.1.9.1, “Operating Mode Bits (MB and MA) — Bits 1–0,” on page 5-10 and to the user’s manual of the particular DSC chip for a description of the different bootstrapping modes.

2.3.4 IP-BUS Bridge

Some devices based on the DSP56800 architecture connect to on-chip peripherals using the Freescale-standard IP-BUS interface. These devices contain an IP-BUS bridge unit, which allows peripherals to be accessed using the CGDB data bus and XAB1 address bus. Peripheral registers are memory-mapped into the data address space. Consult the appropriate DSP56800-based device User’s Manual for more information on peripheral interfacing for a particular chip.

2.3.5 Phase Lock Loop (PLL)

The phase lock loop (PLL) allows the DSC 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 2-4 on page 2-9. There may also be other important registers that are not included in the DSP56800 core, but mapped into the data address space. These include registers for peripheral devices and other functions that are not bound into the core.

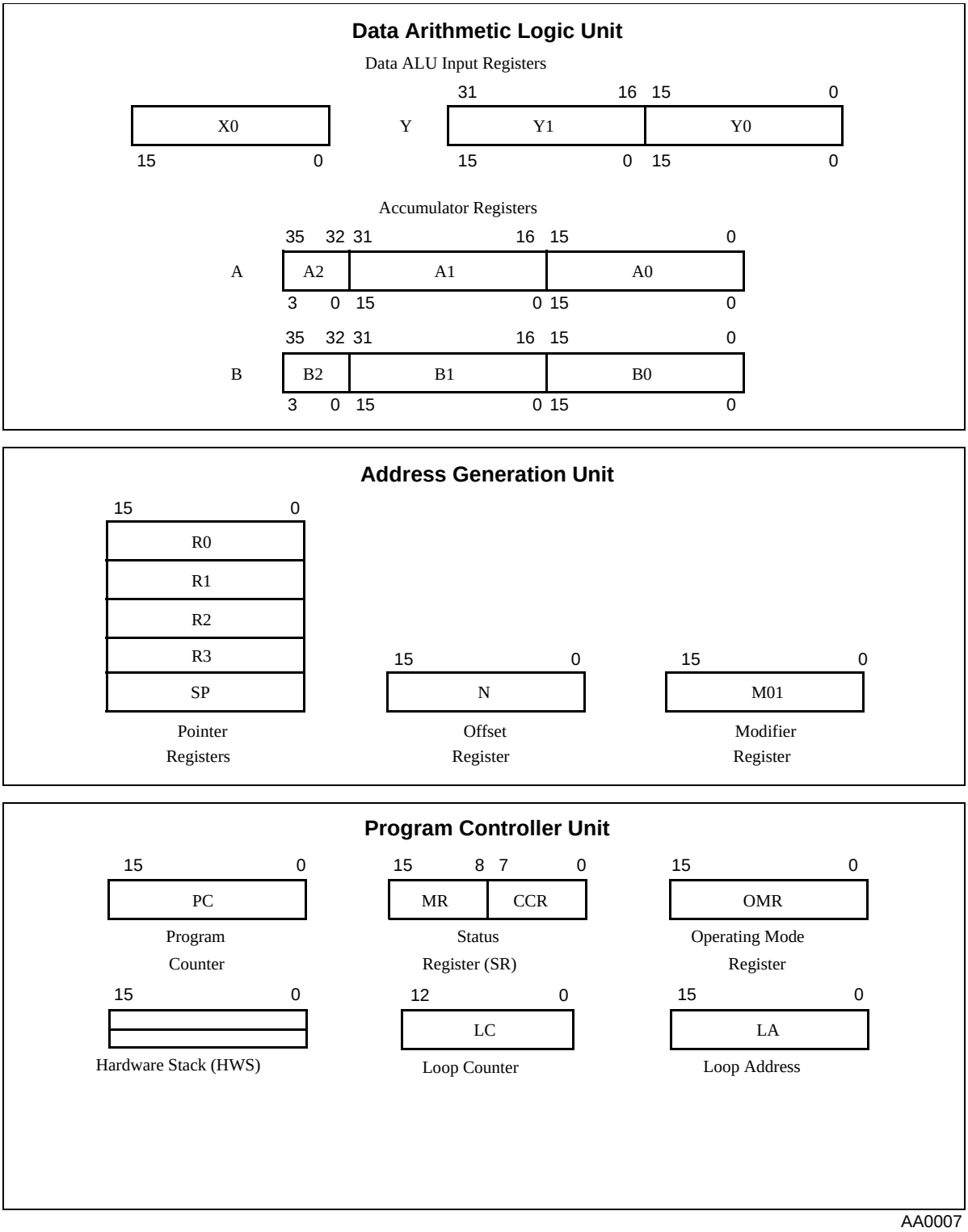


Figure 2-4. DSP56800 Core Programming Model

Chapter 3
Data Arithmetic Logic Unit

This chapter 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 36-bit accumulator registers (A and B)
  - 16-bit registers (A0 and B0)
  - 16-bit registers (A1 and B1)
  - 4-bit 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 operations) 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 negated product (with or without rounding)
- Multiplication and accumulation (with or without rounding)
- Multiplication and accumulation with negated 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)

3.1 Overview and Architecture

The major components of the data ALU are the following:

- Three 16-bit input registers (X0, Y0, and Y1)
- Two 36-bit accumulator registers (A and B)
  - 16-bit registers (A0 and B0)
  - 16-bit registers (A1 and B1)
  - 4-bit 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 on page 3-3, and its corresponding programming model is shown in Figure 3-2 on page 3-4. 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 portions, and least significant portions of the 36-bit accumulator, respectively. Instructions can access the register as a whole or by these individual portions (see Section 3.1.2, “Data ALU Accumulator Registers,” on page 3-4 and Section 3.2, “Accessing the Accumulator Registers,” on page 3-7). The blocks and registers within the data ALU are explained in the following sections.

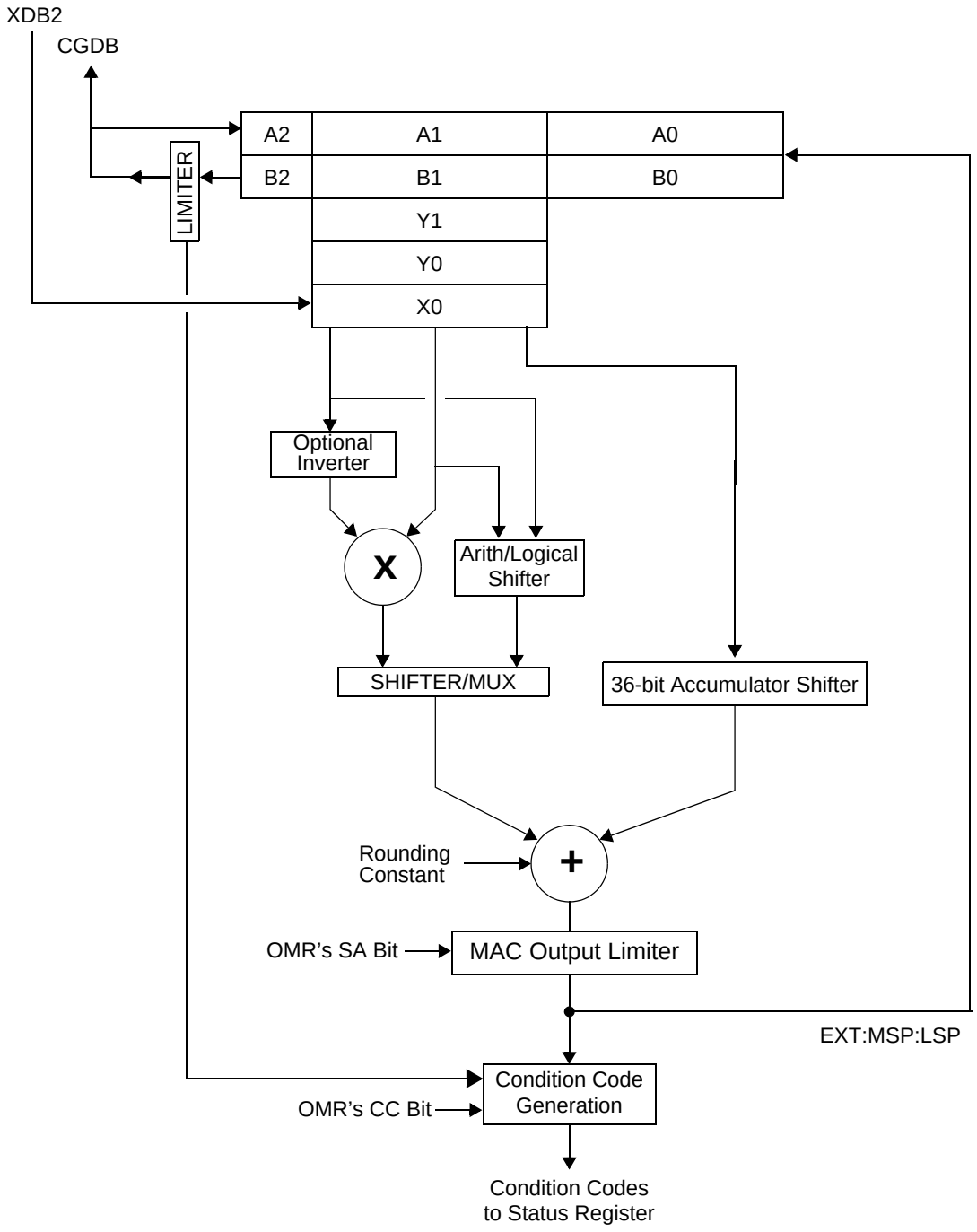


Figure 3-1. Data ALU Block Diagram

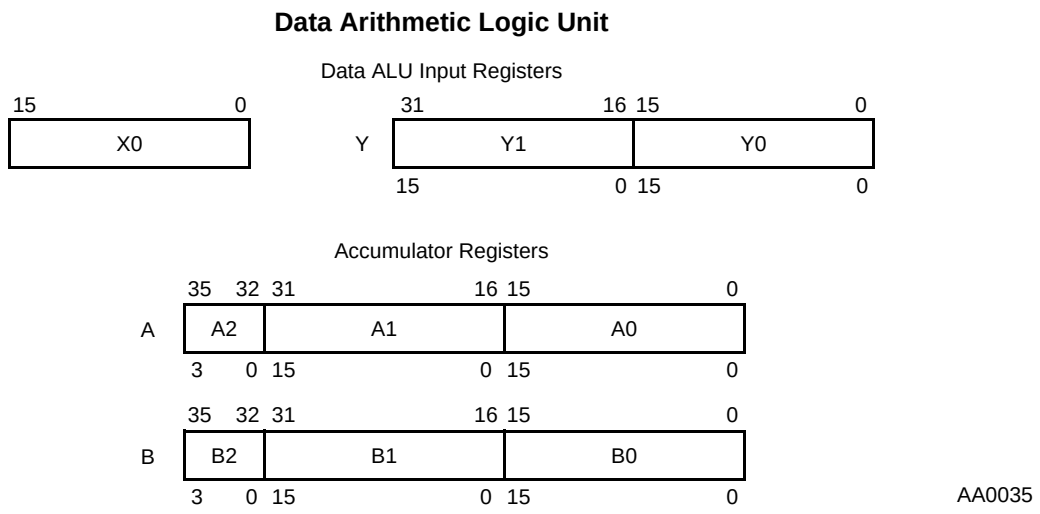


Figure 3-2. Data ALU Programming Model

3.1.1 Data ALU Input Registers (X0, Y1, and Y0)

The data ALU registers (X0, Y1, and Y0) are 16-bit registers that serve as inputs 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, or as one 16-bit register and one 32-bit register. Y1 and Y0 can be concatenated to form the 32-bit register Y, with Y1 being the most significant word and Y0 being the least significant word. Figure 3-2 shows this arrangement.

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.

3.1.2 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 the following, individual portions of the register:

- 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 DSC algorithms and general-purpose computing tasks. Accessing these registers as entire accumulators (A or B) is particularly useful for DSC tasks, because this preserves the full precision of multiplication and other ALU operations. Data limiting and saturation are also possible using the full registers, in cases where the final result of a computation that has overflowed is moved (see Section 3.4.1, “Data Limiter,” on page 3-26).

Accessing an accumulator through its individual portions (A2, A1, A0, B2, B1, or B0) is useful for systems and control programming. When accumulators are manipulated using their constituent components, saturation and limiting are disabled. This allows for microcontroller-like 16-bit integer processing for non-DSC purposes.

Section 3.2, “Accessing the Accumulator Registers,” provides a complete discussion of the ways in which the accumulators can be employed. A description of the data limiting and saturation features of the data ALU is provided in Section 3.4, “Saturation and Data Limiting.”

3.1.3 Multiply-Accumulator (MAC) and Logic Unit

The multiply-accumulator (MAC) and logic unit is the main arithmetic processing unit of the DSC. This is the block that performs all multiplication, addition, subtraction, logical, and other arithmetic operations except shifting. It accepts up to three input operands and outputs one 36-bit result of the form EXT:MSP:LSP (extension : most significant product : least significant product). 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 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-bit x 16-bit parallel signed/unsigned fractional and 16-bit 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 to 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; the extension register is filled with sign extension and the LSP of the accumulator remains unchanged.

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 Section 3.5, “Rounding,” 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.

3.1.4 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 3-3 on page 3-6. It is also possible to use this unit to right shift 32-bit values using the ASRAC and LSRAC instructions, as demonstrated in Section 8.2, “16- and 32-Bit Shift Operations,” on page 8-8.

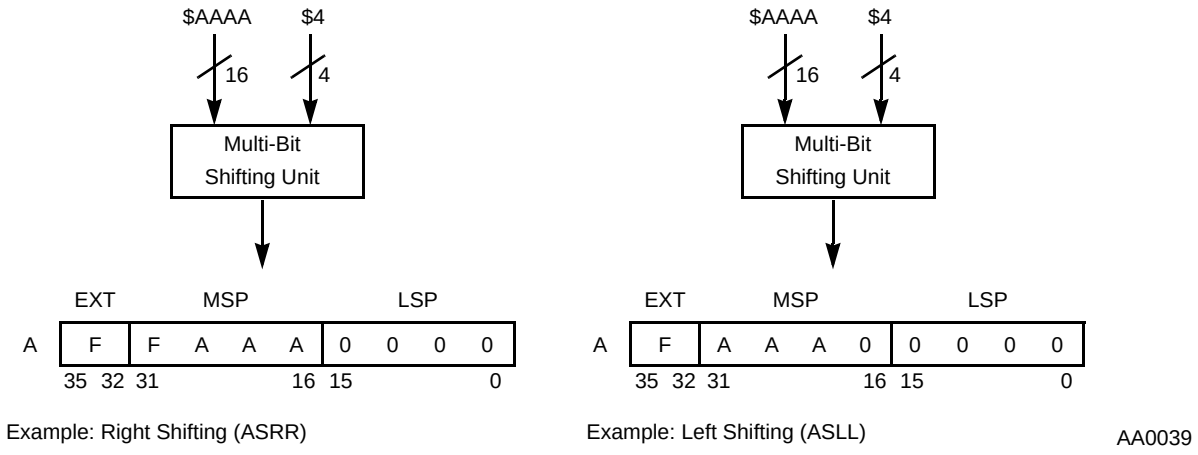


Figure 3-3. Right and Left Shifts Through the Multi-Bit Shifting Unit

The barrel shifter performs all multi-bit shifts operations: arithmetic shifts (ASLL, ASRR), and logical shift (LSRR). When the destination is a 36-bit accumulator, the extension register is always loaded with sign extension from bit 31 for arithmetic shifts (and zero extended for logical shift). The LSP is always set to zero for these operations. Note that the LSLL is implemented as an ASLL instruction but only accepts 16-bit registers as destinations. For information on LSLL, refer to Section 6.5.2, “LSLL Alias,” on page 6-12 and Appendix A.

3.1.5 Accumulator Shifter

The accumulator shifter 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, and so on
- 1-bit left shift — ASL, LSL, ROL
- 1-bit right shift — ASR, LSR, ROR
- Force to zero — MPY, IMPY16

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

3.1.6 Data Limiter and MAC Output Limiter

The data ALU contains two units that implement optional saturation of mathematical results, the Data Limiter and the MAC Output Limiter. The Data Limiter saturates values when data is moved out of an accumulator with a move instruction or parallel move. The MAC Output Limiter saturates the output of the data ALU’s MAC unit.

Section 3.4, “Saturation and Data Limiting,” provides an in-depth discussion of saturation and limiting, as well as a description of the operation of the two limiter units.

3.2 Accessing the Accumulator Registers

An accumulator register can be accessed in two different ways:

- as an entire register, F (representing accumulator A or B)
- by the individual register portion: F2, F1, or F0 (representing A2 or B2, A1 or B1 and A0 or B0)

The ability to access the accumulator registers in both ways provides important flexibility, allowing for powerful DSC algorithms as well as general-purpose computing tasks.

Accessing an entire accumulator register (A or B) is particularly useful for DSC tasks, since it preserves the complete 36-bit register—and thus the entire precision of a multiplication or other ALU operation. It also provides limiting (or saturation) capability in cases when storing a result of a computation that would overflow the destination size. See Section 3.4, “Saturation and Data Limiting.”

Accessing an accumulator through its individual portions (F2, F1, or F0) is useful for systems and control programming. For example, if a DSC algorithm is in progress and an interrupt is received, it is usually necessary to save every accumulator used by the interrupt service routine. Since an interrupt can occur at any step of the DSC task (that is, right in the middle of a DSC algorithm), it is important that no saturation takes place. Thus, an interrupt service routine can store the individual accumulator portions on the stack, effectively saving the entire 36-bit value without any limiting. Upon completion of the interrupt routine, the contents of the accumulator can be exactly restored from the stack.

The DSP56800 instruction set transparently supports both methods of access. An entire accumulator may be accessed simply through the specification of the full-register name (A or B), while portions are accessed through the use of their respective names (A0, B1, and so on).

Table 3-1 provides a summary of the various access methods. These are described in more detail in Section 3.2.1, “Accessing an Accumulator by Its Individual Portions,” and Section 3.2.2, “Accessing an Entire Accumulator.”

Table 3-1. Accessing the Accumulator Registers

Register	Read of an Accumulator Register	Write to an Accumulator Register
A B	*For a MOVE instruction:* If the extension bits are not in use for the accumulator to be read, then the 16-bit contents of the F1 portion 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 Section 3.4.1, “Data Limiter.” *When used in an arithmetic operation:* All 36 bits are sent to the MAC unit without limiting.	*For a MOVE instruction:* The 16 bits of the CGDB bus are written into the 16-bit F1 portion of the register. The extension portion of the same accumulator, F2, is filled with sign extension. The F0 portion is set to zero.
A2 B2	*For a MOVE instruction:* The 4-bit register is read onto the 4 LSBs of the CGDB bus. The upper 12 bits of the bus are sign extended. See Figure 3-5 on page 3-9.	*For a MOVE instruction:* The 4 LSBs of the CGDB are written into the 4-bit register; the upper 12 bits are ignored. The corresponding F1 and F0 portions are not modified. See Figure 3-4 on page 3-8.

Table 3-1. Accessing the Accumulator Registers (Continued)

Register	Read of an Accumulator Register	Write to an Accumulator Register
A1 B1	*For a MOVE instruction:* The 16-bit F1 portion is read onto the CGDB bus. *When used in an arithmetic operation:* The F1 register is used as a 16-bit source operand for an arithmetic operation. F1 can be used in the following: MOVE Parallel Move Several different arithmetic	*For a MOVE instruction:* The contents of the CGDB bus are written into the 16-bit F1 register. The corresponding F2 and F0 portions are not modified.
A0 B0	*For a MOVE instruction:* The 16-bit F0 register is read onto the CGDB bus.	*For a MOVE instruction:* The contents of the CGDB bus are written into the 16-bit F0 register. The corresponding F2 and F1 portions are not modified.

In all cases in Table 3-1 where a MOVE operation is specified, it is understood that the function is identical for parallel moves and bit-field operations.

3.2.1 Accessing an Accumulator by Its Individual Portions

The instruction set provides instructions for loading and storing one of the portions of an accumulator register without affecting the other two portions. When an instructions uses the F1 or F0 notation instead of F, the instruction only operates on the 16-bit portion specified without modifying the other two portions. When an instruction specifies F2, then the instruction operates only on the 4-bit accumulator extension register without modifying the F1 or F0 portions of the accumulator. Refer to Table 3-1 for a summary of accessing the accumulator registers.

Data limiting, as outlined in Section 3.4, “Saturation and Data Limiting,” is enabled only when an entire accumulator is being stored to memory. When only a portion of an accumulator is being stored (by using an instruction which specifies F2, F1, or F0), limiting through the data limiter does not occur.

When F2 is written, the register receives the low-order portion of the word; the high-order portion is not used. See Figure 3-4.

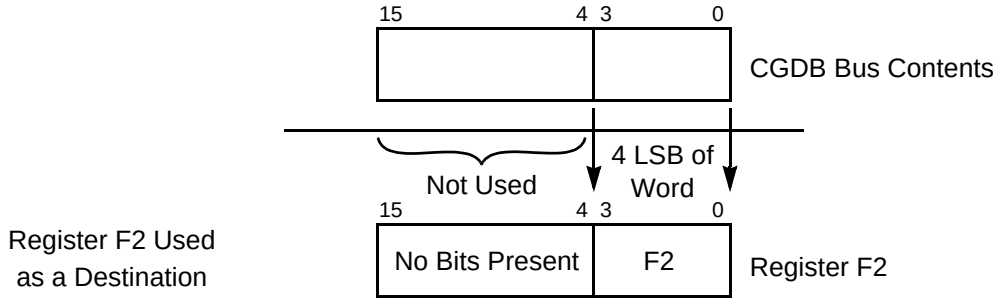


Figure 3-4. Writing the Accumulator Extension Registers (F2)

When F2 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. See Figure 3-5.

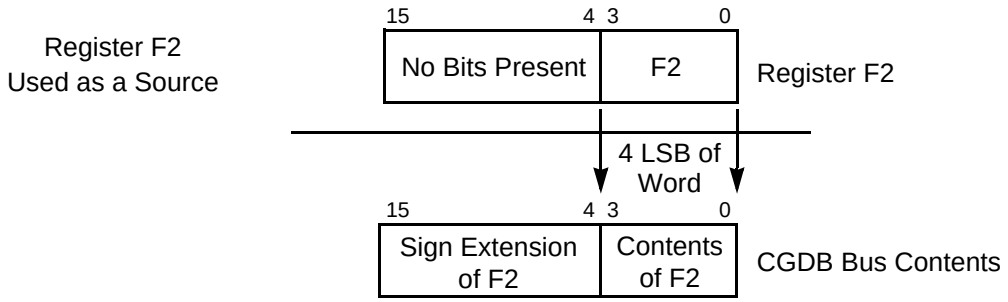
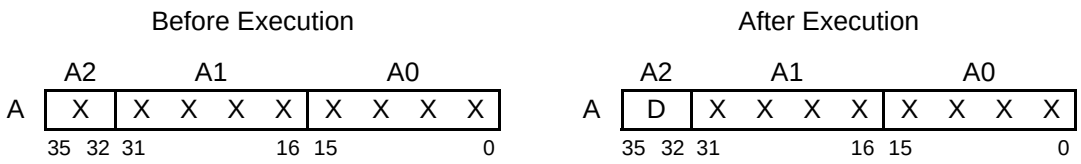


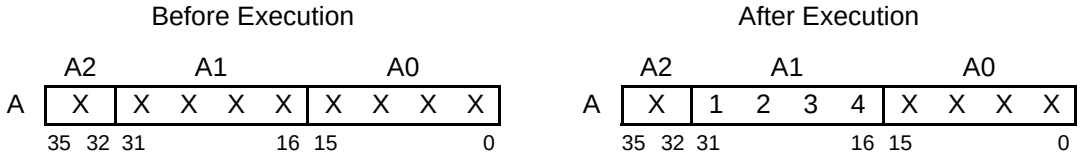
Figure 3-5. Reading the Accumulator Extension Registers (F2)

Figure 3-6 shows the result of writing values to each portion of the accumulator. Note that only the portion specified in the instruction is modified; the other two portions remain unchanged.

Writing the F2 Portion Example : `MOVE #$ABCD,A2`



Writing the F1 Portion Example : `MOVE #$1234,A1`



Writing the F0 Portion Example : `MOVE #$A987,A0`

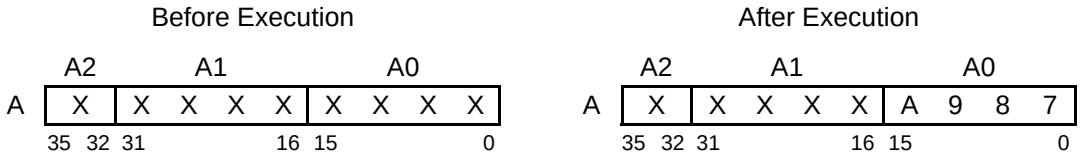


Figure 3-6. Writing the Accumulator by Portions

See Section 3.2, “Accessing the Accumulator Registers,” for a discussion of when it is appropriate to access an accumulator by its individual portions and when it is appropriate to access it as an entire accumulator.

3.2.2 Accessing an Entire Accumulator

3.2.2.1 Accessing for Data ALU Operations

The complete accumulator is accessed to provide a source, a destination, or both for an ALU or multiplication operation in the data ALU. In this case, the accumulator is written as an entire 36-bit accumulator (F), not as an individual register (F2, F1, or F0). The accumulator registers receive the EXT:MSP:LSP of the multiply-accumulator unit output when used as a destination and supply a source accumulator of the same form. Most data ALU operations specify the 36-bit accumulator registers as source operands, destination operands, or both.

3.2.2.2 Writing an Accumulator with a Small Operand

Automatic sign extension of the 36-bit accumulators is provided when the accumulator is written with a smaller size operand. This can occur when writing F from the CGDB (MOVE instruction) or with the results of certain data ALU operations (for example, ADD, SUB, or TFR from a 16-bit register to a 36-bit accumulator). If a word operand is to be written to an accumulator register (F), the F1 portion of the accumulator is written with the word operand, the LSP is zeroed, and the EXT portion receives sign extension. 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 entire accumulator to another. No sign extension is performed if an individual 16-bit register is written (F1 or F0).

NOTE:

A read of the F1 register in a MOVE instruction is identical to a read of the F accumulator for the case where 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 F accumulator produces the same result as reading the F1 register.

3.2.2.3 Extension Registers as Protection Against Overflow

The F2 extension registers offer protection against 32-bit overflow. When the result of an accumulation crosses the MSB of MSP (bit 31 of F), 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 important information. This is particularly useful during execution of DSC algorithms, when intermediate calculations (but not the final result that is written to memory or to a peripheral) may sometimes overflow.

The logic detection of “extension register in use” is also used to determine when to saturate the value of an accumulator when it is being read onto the CGDB or transferred to any data ALU register. If saturation occurs, the content of the original accumulator is not affected (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 Section 3.4.1, “Data Limiter.”

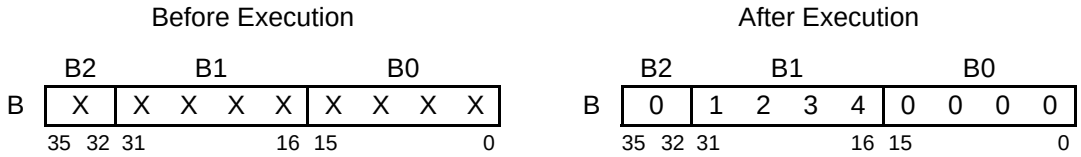
NOTE:

Limiting will be performed only when the entire 36-bit accumulator register (F) is specified as the source for a parallel data move or a register transfer. It is not performed when F2, F1 or F0 is specified.

3.2.2.4 Examples of Writing the Entire Accumulator

Figure 3-7 shows the result of writing a 16-bit signed value to an entire accumulator. Note that all three portions of the accumulator are modified. The LSP (B0) is set to zero, and the extension portion (B2) is appropriately sign extended.

Writing a Positive Value into 36-Bit Accumulator Example : `MOVE #$1234,B`



Writing a Negative Value into 36-Bit Accumulator Example : `MOVE #$A987,B`

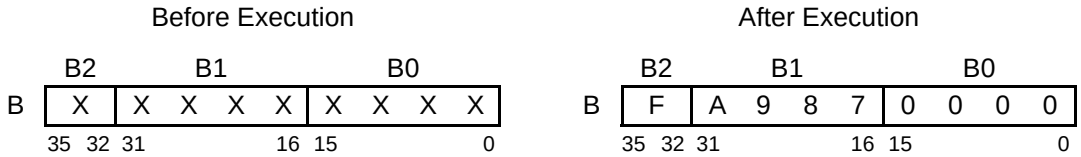


Figure 3-7. Writing the Accumulator as a Whole

Successfully using the DSP56800 Family requires a full understanding of the methods and implications of the various accumulator-register access methods. The architecture of the accumulator registers offers a great deal of flexibility and power, but it is necessary to completely understand the access mechanisms involved to fully exploit this power.

3.2.3 General Integer Processing

General integer and control processing typically involves manipulating 16- and 32-bit integer quantities. Rarely will such code use a full 36-bit accumulator such as that implemented by the DSP56800 Family. The architecture of the DSP56800 supports the manipulation of 16-bit integer quantities using the accumulators, but care must be taken when performing such manipulation.

3.2.3.1 Writing Integer Data to an Accumulator

When loading an accumulator, it is most desirable for the 36 bits of the accumulator to correctly reflect the 16-bit data. To this end, it is recommended that all accumulator loads of 16-bit data clear the least significant portion of the accumulator and also sign extend the extension portion. This can be accomplished through specifying the full accumulator register as the destination of the move, as shown in Example 3-1.

Example 3-1. Loading an Accumulator with a Word for Integer Processing

```
MOVE   X:(R0),A                ; A2 receives sign extension
                               ; A1 receives the 16-bit data
                               ; A0 receives the value $0000
```

Loading a 16-bit integer value into the A1 portion of the register is generally discouraged. In almost all cases, it is preferable to follow Example 3-1 on page 3-11. One notable exception is when 36-bit accumulator values must be stored temporarily. See Section 3.2.5, “Saving and Restoring Accumulators,” for more details.

3.2.3.2 Reading Integer Data from an Accumulator

Integer and control processing algorithms typically involve the manipulation of 16-bit quantities that would be adversely affected by saturation or limiting. When such integer calculations are performed, it is often desirable not to have overflow protection when results are stored to memory. To ensure that the data ALU’s data limiter is not active when an accumulator is being read, it is necessary to store not the full accumulator, but just the MSP (A1 portion). See Example 3-2.

Example 3-2. Reading a Word from an Accumulator for Integer Processing

```
MOVE   A1,X:Variable_1                  ; Saturation is disabled
```

Note that with the use of the A1 register instead of the A register, saturation is disabled. The value in A1 is written “as is” to memory.

3.2.4 Using 16-Bit Results of DSC Algorithms

A DSC algorithm may use the full 36-bit precision of an accumulator while performing DSC calculations such as digital filtering or matrix multiplications. Upon completion of the algorithm, however, sometimes the result of the calculation must be saved in a 16-bit memory location or must be written to a 16-bit D/A converter. Since DSC algorithms process digital signals, it is important that when the 36-bit accumulator value is converted to a 16-bit value, saturation is enabled so signals that overflow 16 bits are appropriately clipped to the maximum positive or negative value. See Example 3-3.

Example 3-3. Correctly Reading a Word from an Accumulator to a D/A

```
MOVE   A,X:D_to_A_data                  ; Saturation is enabled
```

Note the use of the A accumulator instead of the A1 register. Using the A accumulator enables saturation.

3.2.5 Saving and Restoring Accumulators

Interrupt service routines offer one example of a time when it is critical that an accumulator be saved and restored without being altered in any way. Since an interrupt can occur at any time, the exact usage of an accumulator at that instant is unknown, so it cannot be altered by the interrupt service routine without adversely affecting any calculation that may have been in progress. In order for an accumulator to be saved and restored correctly, it must be done with limiting disabled. This is accomplished through sequentially saving and restoring the individual parts of the register, and not the whole register at once. See Example 3-4 on page 3-13.

Example 3-4. Correct Saving and Restoring of an Accumulator — Word Accesses

```
; Saving the A Accumulator to the Stack
LEA    (SP)+         ; Point to first empty location
MOVE   A2,X:(SP)+    ; Save extension register
MOVE   A1,X:(SP)+    ; Save MSP register
MOVE   A0,X:(SP)     ; Save LSP register

; Restoring the A Accumulator from the Stack
MOVE   X:(SP)-,A0    ; Restore LSP register
MOVE   X:(SP)-,A1    ; Restore MSP register
MOVE   X:(SP)-,A2    ; Restore extension register
```

It is important that interrupt service routines do *not* use the MOVE A,X:(SP)+ instruction when saving to the stack. This instruction operates with saturation enabled, and may inadvertently store the value $7FFF or $8000 onto the stack, according to the rules employed by the Data Limiter. This could have catastrophic effects on any DSC calculation that was in progress.

3.2.6 Bit-Field Operations on Integers in Accumulators

When bit-manipulation operations on accumulator registers are performed, as is done for integer processing, care must be taken. The bit-manipulation instructions operate as a “Read-Modify-Write” sequence, and thus may be affected by limiting during the “Read” portion of this sequence. In order for bit-manipulation operations to generate the expected results, limiting must be disabled. To ensure that this is the case, the MSP (A1 portion) of an accumulator should be used as the target operand for the ANDC, EORC, ORC, NOTC, BFCLR, BFCHG, and BFSET instructions, not the full accumulator. See Example 3-5.

Example 3-5. Bit Manipulation on an Accumulator

```
; BFSET using the A1 register
      BFSET  #$0F00,A1     ; Reads A1 with saturation disabled
                           ; Sets bits 11 through 8 and stores back to A1
                           ; Note: A2 and A0 unmodified

; BFSET using the A register
      BFSET  #$0F00,A      ; Reads A1 with saturation enabled - may limit
                           ; Sets bits 11 through 8 and stores back to A1
                           ; A2 is sign extended and A0 is cleared
```

Since the BFTSTH, BFTSTL, BRCLR, and BRSET instructions only test the accumulator value and do not modify it, it is recommended to do these operations on the A1 register where no limiting can occur when integer processing is performed.

3.2.7 Converting from 36-Bit Accumulator to 16-Bit Portion

There are two types of 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 that an accumulator contains correct sign extension and that the least significant 16 bits are all zeros. The two techniques are shown in Example 3-6 on page 3-14.

Example 3-6. 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   A1,B                   ;Sign Extend B2, B0 set to $0000

                        ;Converting with Limiting Enabled
MOVE   A,A                    ;Sign Extend A2, Limit if Required
MOVE   A,B                    ;Sign Extend B2, Limit if Required
```

Where limiting is enabled, as in the second example in Example 3-6, limiting only occurs when the extension register is in use. You can determine if the extension register is in use by examining the extension bit (E) of the status register. Refer to Section 5.1.8, “Status Register,” on page 5-6.

3.3 Fractional and Integer Data ALU Arithmetic

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, and 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 3-14 on page 3-28). 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 3-7).

Example 3-7. 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 3-8.)

Example 3-8. 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 3-8 on page 3-15 shows the location of the decimal point (binary point), bit weights and operands alignment for different fractional and integer representations supported on the DSP56800 architecture.

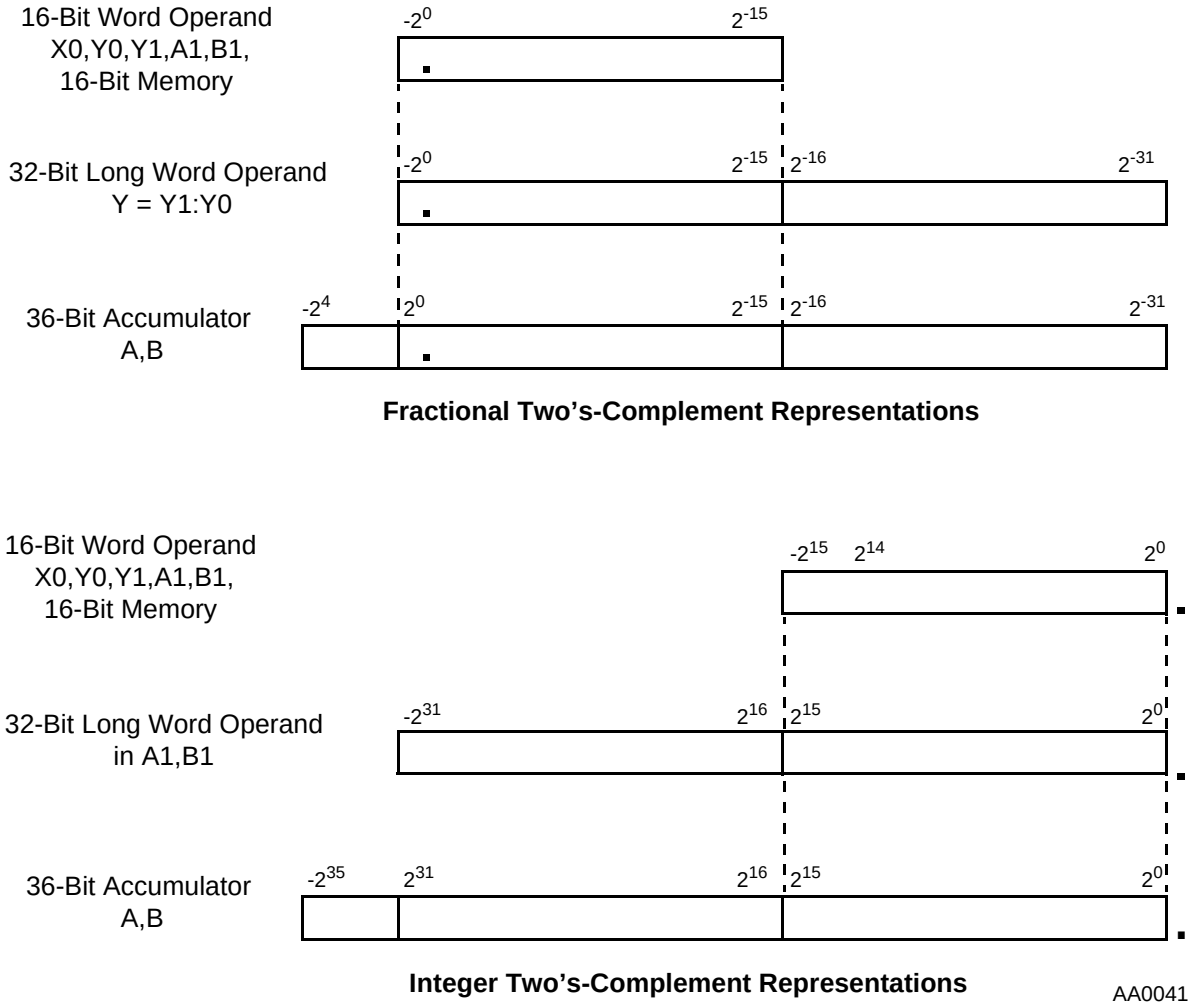


Figure 3-8. Bit Weightings and Operand Alignments

The representation of numbers allowed on the DSP56800 architecture are as follows:

- 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 also the condition code generation. 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 DSC digital-filter design software packages. The results from the DSC design tools can be used without the extensive data conversions that other formats require.
- A significant bit is not lost through sign extension.

3.3.1 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 3-2 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 3-2. Interpretation of 16-Bit Data Values

Hexadecimal Representation	Integer		Fractional	
	Binary	Decimal	Binary	Decimal
$7FFF	0111 1111 1111 1111.	32767	0.111 1111 1111 1111	0.99997
$7000	0111 0000 0000 0000.	28672	0.111 0000 0000 0000	0.875
$4000	0100 0000 0000 0000.	16384	0.100 0000 0000 0000	0.5
$2000	0010 0000 0000 0000.	8,192	0.010 0000 0000 0000	0.25
$1000	0001 0000 0000 0000.	4,096	0.001 0000 0000 0000	0.125
$0000	0000 0000 0000 0000.	0	0.000 0000 0000 0000	0.0
$F000	1111 0000 0000 0000.	- 4096	1.111 0000 0000 0000	- 0.125
$E000	1110 0000 0000 0000.	- 8192	1.110 0000 0000 0000	- 0.25
$C000	1100 0000 0000 0000.	- 16384	1.100 0000 0000 0000	- 0.5
$9000	1001 0000 0000 0000.	- 28672	1.001 0000 0000 0000	- 0.875
$8000	1000 0000 0000 0000.	- 32768	1.000 0000 0000 0000	- 1.0

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

Table 3-3. Interpretation of 36-bit Data Values

Hexadecimal Representation[1]	36-Bit Integer in Entire Accumulator (decimal)	16-Bit Integer in MSP (decimal)	Fractional Value (decimal)
$7 FFFF FFFF	34,359,738,367	(Overflows)	~ 16.0
$1 4000 0000	5,368,709,120	(Overflows)	2.5
$0 4000 0000	1,073,741,824	16,384	0.5
$0 2000 0000	536,870,912	8,192	0.25
$0 0000 0000	0	0	0.0
$F C000 0000	- 1,073,741,824	- 16,384	- 0.5

Table 3-3. Interpretation of 36-bit Data Values (Continued)

Hexadecimal Representation[1]	36-Bit Integer in Entire Accumulator (decimal)	16-Bit Integer in MSP (decimal)	Fractional Value (decimal)
$F E000 0000	- 536,870,912	- 8,192	- 0.25
$E C000 0000	- 5,368,709,120	(Overflows)	-2.5
$8 0000 0001	-34,359,738,367	(Overflows)	-16.0

1. 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.

3.3.2 Data Formats

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

- Signed fractional
- Unsigned fractional
- Signed integer
- Unsigned integer

The ranges for each of these formats, discussed in the following subsections, 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 signed fractional number that can be represented in an accumulator is approximately 16.0 and the most negative signed fractional number is -16.0 as shown in Table 3-3. An important factor to consider is that 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.

3.3.2.1 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:

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

For words and long-word signed fractions, the most negative number that can be represented is -1.0, whose internal representation is $8000 and $8000_0000, respectively. The most positive word is $7FFF or $1.0 - 2^{-15}$, and the most positive long word is $7FFF_FFFF or $1.0 - 2^{-31}$.

3.3.2.2 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 formulated as $\{1.0 + (1.0 - 2^{-[N-1]})\} = 1.99997$. The smallest unsigned number is zero ($0000).

3.3.2.3 Signed Integer

This format is used when data is being processed 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:

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

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 ($8000_0000). The most positive word is 32767 ($7FFF), and the most positive long word is 2147483647 ($7FFF_FFFF).

3.3.2.4 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:

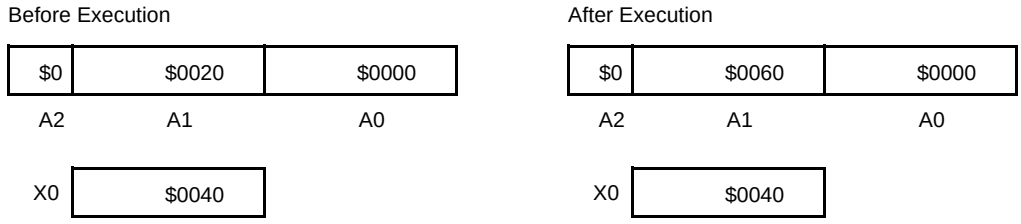
$$0 \leq UI \leq [2^{N}-1]$$

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 65535 ($FFFF). The smallest unsigned number is zero ($0000).

3.3.3 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 3-9.




```
MOVE   #64,X0        ; Load integer value 64 ($40) into X0
MOVE   #32,A         ; Load integer value 32 ($20) into 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 3-9. 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.

3.3.4 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 1 bit corresponds to a multiplication by two. An arithmetic right shift of 1 bit corresponds to a division of a signed value by two, and a logical right shift of 1 bit corresponds to a division of an unsigned value by two.

3.3.5 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 3-10. 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.

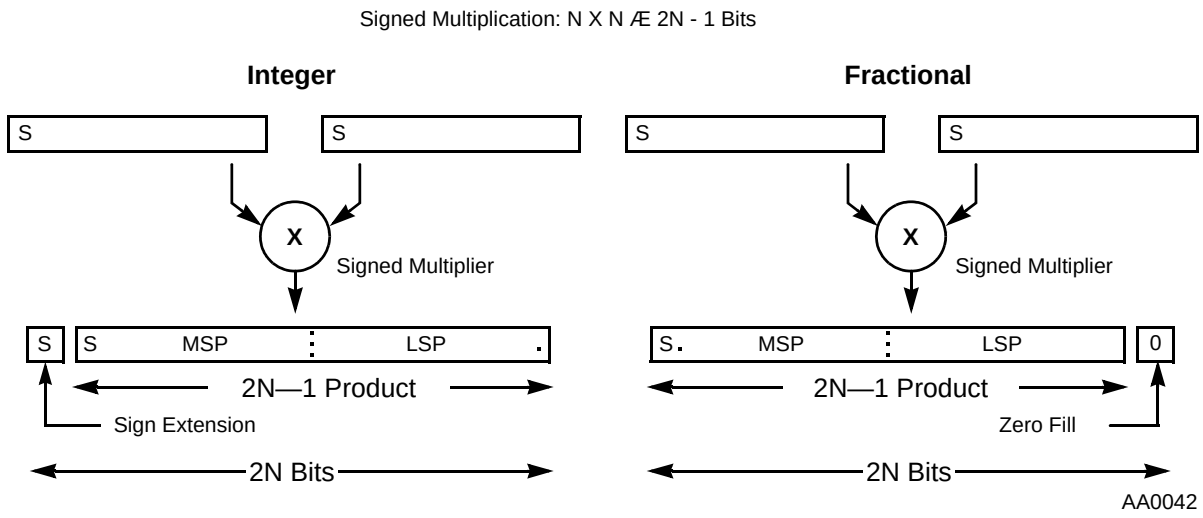


Figure 3-10. Comparison of Integer and Fractional Multiplication

The MPY, MAC, MPYR, and MACR instructions perform fractional multiplication and fractional multiply-accumulation. The IMPY16 instruction performs integer multiplication. Section 3.3.5.2, “Integer Multiplication,” explains how to perform integer multiplication.

3.3.5.1 Fractional Multiplication

Figure 3-11 on page 3-20 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.

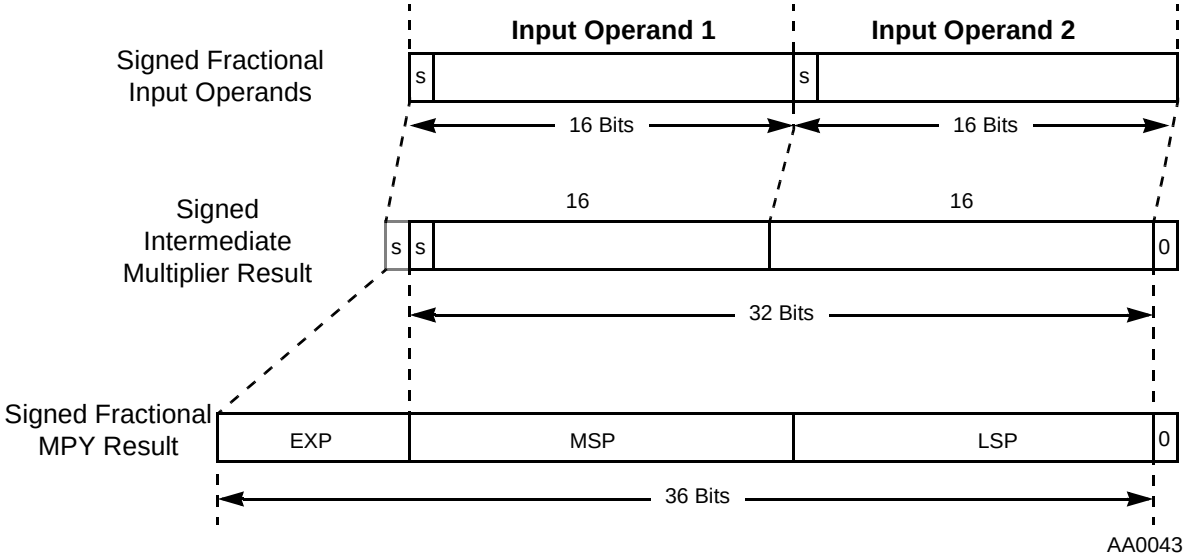


Figure 3-11. MPY Operation — Fractional Arithmetic

3.3.5.2 Integer Multiplication

Two techniques for performing integer multiplication on the DSC core are as follows:

- 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, and DECW instructions and others. 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 3-12 on page 3-21 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.

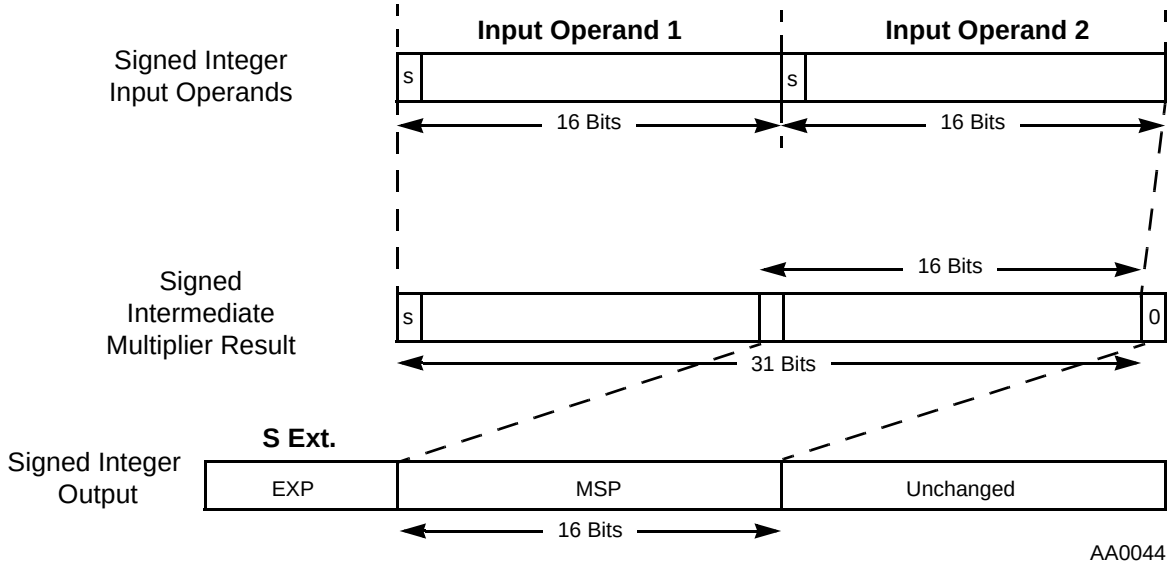


Figure 3-12. Integer Multiplication (IMPY)

At other times it is necessary to maintain the full 32-bit precision of an integer multiplication. To obtain integer results, an 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 3-9).

Example 3-9. Multiplying Two Signed Integer Values with Full Precision

```
MPY    X0,Y0,A      ; Generates correct answer shifted
                    ; 1 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 a multiply-accumulate is performed 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. See Example 3-10.

Example 3-10. Fast Integer MACs using Fractional Arithmetic

```
       MOVE                 X:(R0)+,Y0    X:(R3)+,X0
       DO     #Count,LABEL        ; Count defined as number of repetitions
       MAC    X0,Y0,A      X:(R0)+,Y0    X:(R3)+,X0
LABEL:
       ASR    A                   ; Convert to Integer only after MACs are
                                  ;   completed
```

3.3.6 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 Section 8.4, “Division,” on page 8-13 for a complete discussion of division.

3.3.7 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.

3.3.7.1 Conditional Branch Instructions for Unsigned Operations

Unsigned arithmetic is supported on operations such as addition, subtraction, comparison, and logical operations using the same ADD, SUB, CMP, and other instructions used for signed computations. The operations are performed the same for both representations. The difference lies both in which status bits are used in comparing signed and unsigned numbers and in how the data is interpreted, for which see Section 3.3.2, “Data Formats.”

Four additional Bcc instruction variants are provided for branching based on the comparison of two unsigned numbers. These variants are:

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

Unsigned comparisons are enabled 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 branch condition variants (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 Section 8.1.6, “Unsigned Load of an Accumulator,” on page 8-7. 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 DSC code.

3.3.7.2 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-times-unsigned multiplication is performed using the technique demonstrated in Example 3-11 on page 3-23.

Example 3-11. 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 one
        BGE    OVER          ; If first operand is less that one, jump to OVER
        MOVE   #$0,B
        MOVE   Y0,B1         ; Move Y0 to B without sign extension
        ADD    B,A
OVER:
;       (ASR   A)            ; Optionally convert to integer result
```

3.3.8 Multi-Precision Operations

The DSP56800 instruction set contains several instructions which simplify extended- and multi-precision mathematical operations. By using these instructions, 64-bit and 96-bit calculations can be performed, and calculations involving different-sized operands are greatly simplified.

3.3.8.1 Multi-Precision Addition and Subtraction

Two instructions, ADC and SBC, assist in performing multi-precision addition (Example 3-12) and subtraction (Example 3-13), such as 64-bit or 96-bit operations.

Example 3-12. 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;
that is, 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 3-13. 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;
that is, 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
```

3.3.8.2 Multi-Precision Multiplication

Two instructions are provided to assist with multi-precision multiplication. When these instructions are used, the multiplier accepts one signed 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 multiplication is demonstrated in Figure 3-13, with corresponding examples shown in Example 3-14, Example 3-15 on page 3-24, and Example 3-16 on page 3-25.

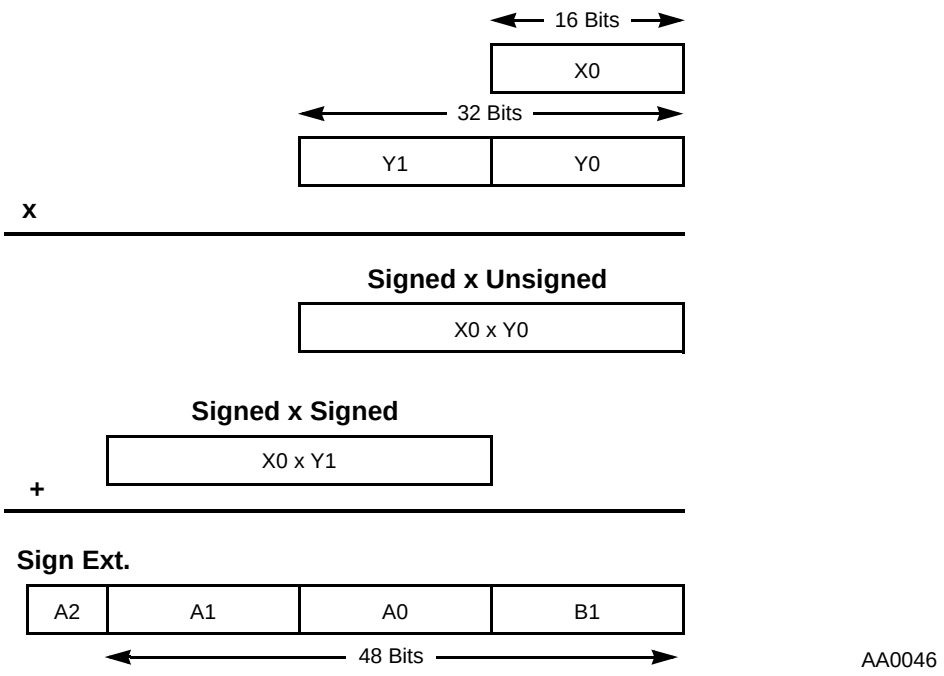


Figure 3-13. Single-Precision Times Double-Precision Signed Multiplication

Example 3-14. Fractional Single-Precision 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 3-15. Integer Single-Precision Times 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 x Upper Portion and add to Previous
ASR    A            ; Convert result to integer, A2 contains sign extension
ROR    B            ; (52-bit shift of A2:A1:A0:B1)
```

Example 3-16. Multiplying Two Fractional Double-Precision Values

```
;
;  Signed 32x32 => 64 Multiplication Subroutine
;
;  Parameters:
;     R1 = ptr to lowest word of one operand
;     R2 = ptr to lowest word of one operand
;     R3 = ptr to where results are stored

MULT_S32_X_S32:
        CLR     B             ; clears B2 portion

; Multiply lwr1 * lwr2 and save lowest 16-bits of result

        ; Operation           ;  X0     Y1     Y0     A
        ; ---------           ; -----  -----  -----  -------------------
        MOVE    X:(R1),Y0     ;  ---    ---    lwr1   -----
        ANDC    #CLRMSB,Y0    ;  ---    ---    lwr1'  -----
        MOVE    X:(R2)+,Y1    ;  ---    lwr2   lwr1'  -----
        MPYSU   Y0,Y1,A       ;  ---    lwr2   lwr1'  lwr1'.s * lwr2.u
        TSTW    X:(R1)+       ;  check if MSB set in original lwr1 value
        BGE     CORRECT_RES1  ;  perform correction if this was true
        MOVE    Y1,B1         ;  ---    lwr2   lwr1'  -----
        ADD     B,A           ;  ---    lwr2   lwr1'  lwr1.u * lwr2.u
CORRECT_RES1:
        MOVE    A0,X:(R3)+    ;  ---    lwr2   lwr1'  lwr1.u * lwr2.u

; Multiply two cross products and save next lowest 16-bits of result
        ; Operation           ;  X0     Y1     Y0     A
        ; ---------           ; -----  -----  -----  -------------------
        MOVE    A1,X:TMP      ;  (arithmetic 16-bit right shift of 36-bit accum)
        MOVE    A2,A          ;  ----   ----   ----   -----
        MOVE    X:TMP,A0      ;  ----   ----   ----   A = product1 >> 16

        MOVE    X:(R1)-,X0    ;  upr1   lwr2   lwr1'  A = product1 >> 16
        MACSU   X0,Y1,A       ;  upr1   lwr2   lwr1'  A+upr1.s*lwr2.u

        MOVE    X:(R1),Y1     ;  upr1   lwr1   lwr1'  A+upr1.s*lwr2.u
        MOVE    X:(R2),Y0     ;  upr1   lwr1   upr2   A+upr1.s*lwr2.u
        MACSU   Y0,Y1,A       ;  upr1   lwr1   upr2   A+upr1.s*lwr2.u+upr2.s*lwr1.u
        MOVE    A0,X:(R3)+    ;  upr1   lwr1   upr2   A = result w/ cross prods

; Multiply upr1 * upr2 and save highest 32-bits of result
        ; Operation           ;  X0     Y1     Y0     A
        ; ---------           ; -----  -----  -----  -------------------
        MOVE    A1,X:TMP      ;  (arithmetic 16-bit right shift of 36-bit accum)
        MOVE    A2,A          ;  upr1   lwr1   upr2   -----
        MOVE    X:TMP,A0      ;  upr1   lwr1   upr2   A = result >> 16

        MAC     X0,Y0,A       ;  upr1   lwr1   upr2   A + upr1.s * upr2.s
        MOVE    A0,X:(R3)+    ;  ---    ---    ---    -----
        MOVE    A1,X:(R3)+    ;  ---    ---    ---    -----

        RTS


; 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 Saturation and Data Limiting

DSC algorithms are sometimes capable of calculating values larger than the data precision of the machine when processing real data streams. Normally, a processor would allow the value to overflow when this occurred, but this creates problems when processing real-time signals. The solution is saturation, a technique whereby values that exceed the machine data precision are “clipped,” or converted to the maximum value of the same sign that fits within the given data precision.

Saturation is especially important when data is running through a digital filter whose output goes to a digital-to-analog converter (DAC), since it “clips” the output data instead of allowing arithmetic overflow. Without saturation, the output data may incorrectly switch from a large positive number to a large negative value, which can cause problems for DAC outputs in embedded applications.

The DSP56800 architecture supports optional saturation of results through two limiters found within the data ALU:

- the Data Limiter
- the MAC Output Limiter

The Data Limiter saturates values when data is moved out of an accumulator with a MOVE instruction or parallel move. The MAC Output Limiter saturates the output of the data ALU’s MAC unit.

3.4.1 Data Limiter

The data limiter protects against overflow by selectively limiting when reading an accumulator register as a source operand in a MOVE instruction. When a MOVE instruction specifies an accumulator (F) as a source, and if the contents of the selected source accumulator can be represented in the destination operand size without overflow (that is, the accumulator extension register is not in use), the data limiter is enabled but does not saturate, and the register contents are placed onto the CGDB unmodified. If a MOVE instruction is used and 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 3-4 on page 3-27.

The F0 portion of an accumulator is ignored by the data limiter.

Consider a simple example, shown in Example 3-17.

Example 3-17. Demonstrating the Data Limiter — Positive Saturation

```
MOVE   #$1000,R0     ; Store results starting in address $1000
MOVE   #$7FFC,A      ; Initialize A = $0_7FFC_0000

INC    A             ; A = $0_7FFD_0000
MOVE   A,X:(R0)+     ; Write $7FFD to memory (limiter enabled)
INC    A             ; A = $0_7FFE_0000
MOVE   A,X:(R0)+     ; Write $7FFE to memory (limiter enabled)
INC    A             ; A = $0_7FFF_0000
MOVE   A,X:(R0)+     ; Write $7FFF to memory (limiter enabled)

INC    A             ; A = $0_8000_0000 <=== Overflows 16-bits
MOVE   A,X:(R0)+     ; Write $7FFF to memory (limiter saturates)
INC    A             ; A = $0_8001_0000
MOVE   A,X:(R0)+     ; Write $7FFF to memory (limiter saturates)
INC    A             ; A = $0_8002_0000
MOVE   A,X:(R0)+     ; Write $7FFF to memory (limiter saturates)

MOVE   A1,X:(R0)+    ; Write $8002 to memory (limiter disabled)
```

Once the accumulator increments to $8000 in Example 3-17, the positive result can no longer be written to a 16-bit memory location without overflow. So, instead of writing an overflowed value to memory, the value of the most positive 16-bit number, $7FFF, is written instead by the data limiter block. Note that the data limiter block does not affect the accumulator; it only affects the value written to memory. In the last instruction, the limiter is disabled because the register is specified as A1.

Consider a second example, shown in Example 3-18 on page 3-27.

Example 3-18. Demonstrating the Data Limiter — Negative Saturation

```
MOVE   #$1008,R0     ; Store results starting in address $1008
MOVE   #$8003,A      ; Initialize A = $F_8003_0000

DEC    A             ; A = $F_8002_0000
MOVE   A,X:(R0)+     ; Write $8002 to memory (limiter enabled)
DEC    A             ; A = $F_8001_0000
MOVE   A,X:(R0)+     ; Write $8001 to memory (limiter enabled)
DEC    A             ; A = $F_8000_0000
MOVE   A,X:(R0)+     ; Write $8000 to memory (limiter enabled)

DEC    A             ; A = $F_7FFF_0000 <=== Overflows 16-bits
MOVE   A,X:(R0)+     ; Write $8000 to memory (limiter saturates)
DEC    A             ; A = $F_7FFE_0000
MOVE   A,X:(R0)+     ; Write $8000 to memory (limiter saturates)
DEC    A             ; A = $F_7FFD_0000
MOVE   A,X:(R0)+     ; Write $8000 to memory (limiter saturates)

MOVE   A1,X:(R0)+    ; Write $7FFD to memory (limiter disabled)
```

Once the accumulator decrements to $7FFF in Example 3-18, the negative result can no longer fit into a 16-bit memory location without overflow. So, instead of writing an overflowed value to memory, the value of the most negative 16-bit number, $8000, is written instead by the data limiter block.

Test logic exists in the extension portion of 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 3-4. Saturation by the Limiter Using the MOVE Instruction

Extension bits in use in selected accumulator?	MSB of F2	Output of Limiter onto the CGDB Bus
No	n/a	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.

Figure 3-14 on page 3-28 demonstrates the importance of limiting. Consider the A accumulator with the following 36-bit value to be read to a 16-bit destination:

0000 1.000 0000 0000 0000 0000 0000 0000 0000 (in binary)
(+ 1.0 in fractional decimal, $0_8000_0000 in hexadecimal)

If this accumulator is read without the limiting enabled by a MOVE A1,X0 instruction, the 16-bit X0 register after the MOVE instruction would contain the following, assuming signed fractional arithmetic:

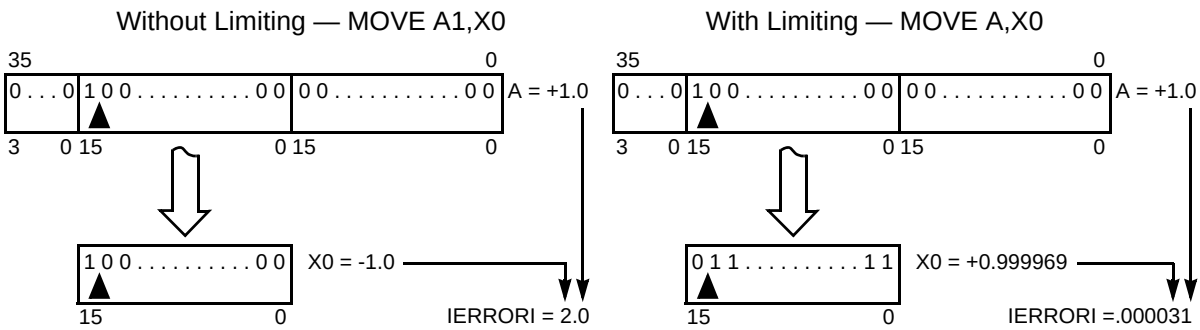
1.000 0000 0000 0000 (- 1.0 fractional decimal, $8000 in hexadecimal)

This is clearly in error because the value -1.0 in the X0 register greatly differs from the value of +1.0 in the source accumulator. In this case, overflow has occurred. To minimize the error due to overflow, it is preferable to write the maximum ("limited") value the destination can assume. In this example, the limited value would be:

0.111 1111 1111 1111 (+ 0.999969 fractional decimal, $7FFF in hexadecimal)

This is clearly closer to the original value, +1.0, than -1.0 is, and thus introduces less error. Saturation is equally applicable to both integer and fractional arithmetic.

Thus, saturation arithmetic can have a large effect in moving 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 enabled. The magnitude of the error without limiting is 2.0; with limiting it is 0.000031.



*Limiting automatically occurs when the 36-bit operands A and B are read with a MOVE instruction. Note that the contents of the original accumulator are ***not*** changed.

Figure 3-14. Example of Saturation Arithmetic

3.4.2 MAC Output Limiter

The MAC output limiter optionally saturates or limits results calculated by data ALU arithmetic operations such as multiply, add, increment, round, and so on.

The MAC Output Limiter can be enabled by setting the SA bit in the OMR register. See Section 5.1.9.3, "Saturation (SA) — Bit 4," on page 5-11.

Consider a simple example, shown in Example 3-19.

Example 3-19. Demonstrating the MAC Output Limiter

```
BFSET  #$0010,OMR    ; Set SA bit —- enables MAC Output Limiter
MOVE   #$7FFC,A      ; Initialize A = $0_7FFC_0000
NOP

INC    A             ; A = $0_7FFD_0000
INC    A             ; A = $0_7FFE_0000
INC    A             ; A = $0_7FFF_0000

INC    A             ; A = $0_7FFF_FFFF <=== Saturates to 16-bits!
INC    A             ; A = $0_7FFF_FFFF <=== Saturates to 16-bits!
ADD    #9,A          ; A = $0_7FFF_FFFF <=== Saturates to 16-bits!
```

Once the accumulator increments to $7FFF in Example 3-19, the saturation logic in the MAC Output limiter prevents it from growing larger because it can no longer fit into a 16-bit memory location without overflow. So instead of writing an overflowed value to back to the A accumulator, the value of the most positive 32-bit number, $7FFF_FFFF, is written instead as the arithmetic result.

The saturation logic operates by checking 3 bits of the 36-bit result out of the MAC unit: EXT[3], EXT[0], and MSP[15]. When the SA bit is set, these 3 bits determine if saturation is performed on the MAC unit's output and whether to saturate to the maximum positive value ($7FFF_FFFF) or the maximum negative value ($8000_0000), as shown in Table 3-5.

Table 3-5. MAC Unit Outputs with Saturation Enabled

EXT[3]	EXT[0]	MSP[15]	Result Stored in Accumulator
0	0	0	Result out of MAC Array with no limiting occurring
0	0	1	$0_7FFF_FFFF
0	1	0	$0_7FFF_FFFF
0	1	1	$0_7FFF_FFFF
1	0	0	$F_8000_0000
1	0	1	$F_8000_0000
1	1	0	$F_8000_0000
1	1	1	Result out of MAC Array with no limiting occurring

The MAC Output Limiter not only affects the results calculated by the instruction, but can also affect condition code computation as well. See Appendix A.4.2, "Effects of the Operating Mode Register's SA Bit," on page A-11 for more information.

3.4.3 Instructions Not Affected by the MAC Output Limiter

The MAC Output Limiter is always disabled (even if the SA bit is set) when the following instructions are being executed:

- ASLL, ASRR, LSRR
- ASRAC, LSRAC
- IMPY
- MPYSU, MACSU
- AND, OR, EOR
- LSL, LSR, ROL, ROR, NOT
- TST

The CMP is not affected by the OMR's SA bit except for the case when the first operand is not a register (that is, it is a memory location or an immediate value) and the second operand is the X0, Y0, or Y1 register. In this particular case, the U bit calculation is affected by the SA bit. No other bits are affected by the SA bit for the CMP instruction.

Also, the MAC Output Limiter only affects operations performed in the data ALU. It has no effect on instructions executed in other blocks of the core, such as the following:

- Bit Manipulation Instructions (Table 6-30 and Table 6-31 on page 6-26)
- Move instructions (Table 6-18 through Table 6-21)
- Looping instructions (Table 6-33 on page 6-27)
- Change of flow instructions (Table 6-32 on page 6-27)
- Control instructions (Table 6-34 on page 6-28)

NOTE:

The SA bit affects the TFR instruction when it is set, optionally limiting data as it is transferred from one accumulator to another.

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 DSC core implements two types of rounding: convergent rounding and 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 (that is, LSP > $8000) and rounds down any value below one-half (that is, LSP < $8000). The question arises as to which way the number one-half (LSP = $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 zero) and rounding up if the number is odd (bit 16 equals one), 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 identically 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 zero) and rounding up if the rounding is odd (bit 16 equals one). Figure 3-15 on page 3-31 shows the four possible cases for rounding a number in the A or B accumulator.

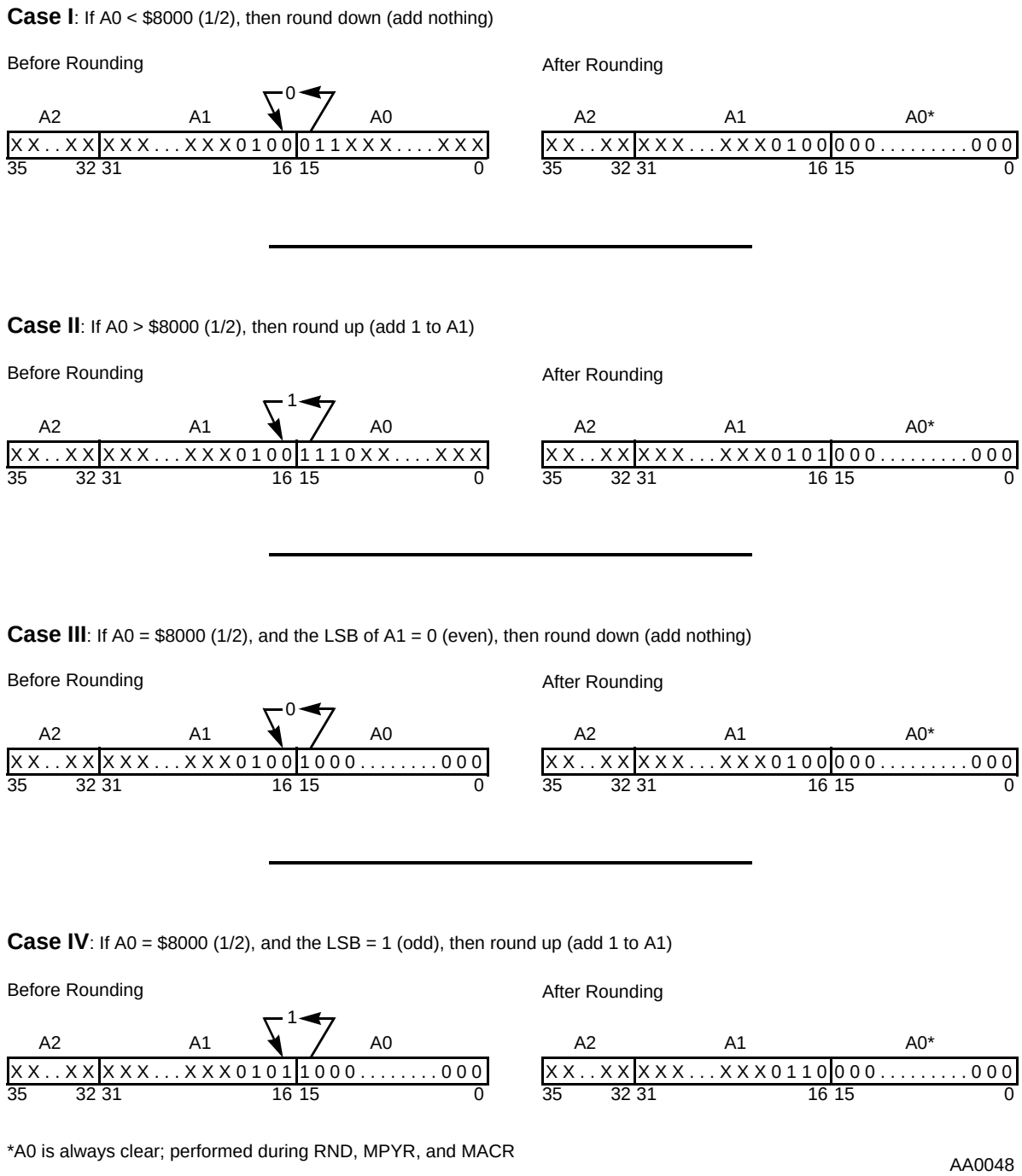


Figure 3-15. Convergent Rounding

3.5.2 Two's-Complement Rounding

When this type of rounding is selected by setting the rounding bit in the OMR, one 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 3-16 shows the two possible cases.

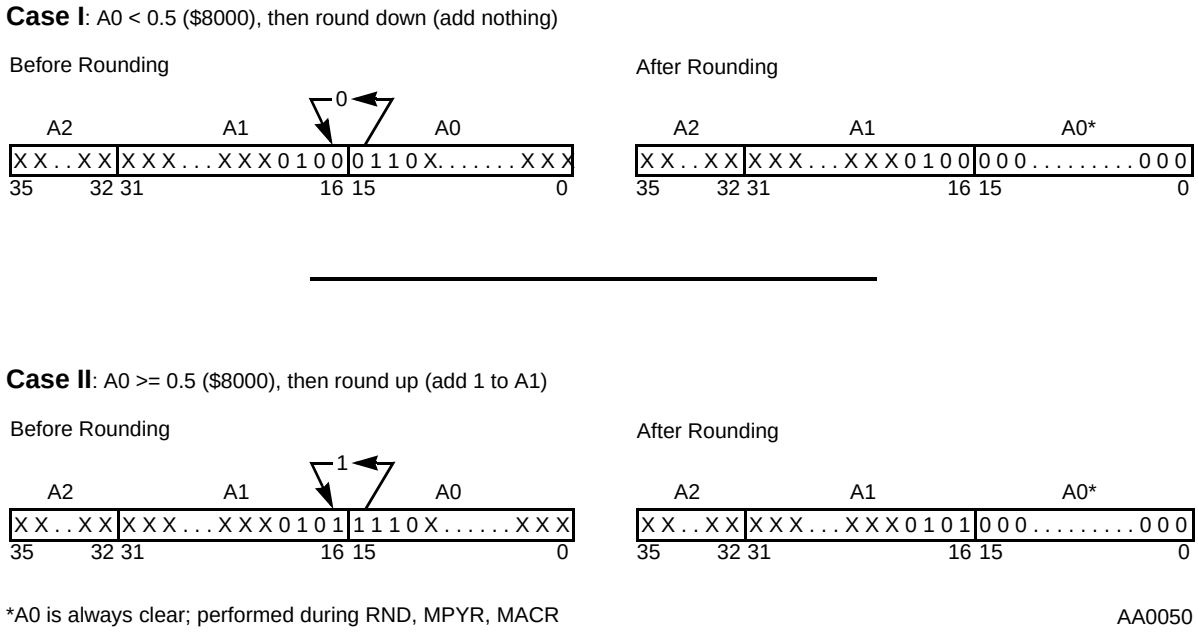


Figure 3-16. Two's-Complement Rounding

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.

3.6 Condition Code Generation

The DSC 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 an examination of 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 DSC operations. For the case of general-purpose computing, the CC bit is often set when the program is not performing DSC tasks. However, 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, or LO).

This section covers different aspects of condition code generation for the different instructions and configurations on the DSC 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 (that is, if bits 35–31 are not all the same).
- U is set according to the standard definition of the U bit.

3.6.1 36-Bit Destinations — CC Bit Cleared

Most arithmetic instructions generate a result for a 36-bit accumulator. When condition codes are being generated 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, and RND.

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

- N is set if bit 35 of the corresponding accumulator is set except during saturation. During a saturation condition, the V (overflow) bit is set and the N bit is not 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.

3.6.2 36-Bit Destinations — CC Bit Set

Most arithmetic instructions generate a result for a 36-bit accumulator. When condition codes are being generated 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. 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, and RND.

The condition codes for 32-bit destinations (CC equals one) 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.

3.6.3 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 condition codes are being generated 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 zero) are computed as follows:

- N is set if bit 35 of the corresponding accumulator is set except during saturation. During a saturation condition, the V (overflow) bit is set and the N bit is not 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.

3.6.4 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 condition codes are being generated 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 one) 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.

3.6.5 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 condition codes for a 16-bit destination are being generated, 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 condition codes are being generated 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.

3.6.6 Special Instruction Types

Some instructions do not follow the preceding rules for condition code generation, and 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 bits. 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 Appendix A.4, “Condition Code Computation,” on page A-6 for details on the condition code computation for each of these instructions.

3.6.7 TST and TSTW Instructions

There are two instructions, TST and TSTW, that 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 zero are computed as follows:

- L is set if limiting occurs in a parallel move.
- E is set if the extension register is in use — that is, 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 except during saturation.
- 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 one are computed as follows:

- L is set if limiting occurs in a parallel move.
- E is set if the extension register is in use, that is, 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:

- L is set if limiting occurs while reading an accumulator.
- 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.

3.6.8 Unsigned Arithmetic

When arithmetic on unsigned operands is being performed, the condition codes used to compare two values differ from those used for signed arithmetic. See Section 3.3.7, “Unsigned Arithmetic,” for a discussion of condition code usage for unsigned arithmetic.

Chapter 4
Address Generation Unit

This chapter 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
- Post-update an address after providing the original address value on XAB1 bus
- Calculate an effective address which is then provided on the XAB1 bus
- 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 addressing with no update
- Indirect addressing with post-increment
- Indirect addressing with post-decrement
- Indirect addressing with post-update by a register
- Indirect addressing with index by a 16-bit offset
- Indirect addressing with index by a 6-bit offset
- Indirect addressing with index by a register
- Immediate data
- Immediate short data
- Absolute addressing
- Absolute short addressing
- Peripheral short addressing
- Register direct
- Implicit

This chapter 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 pipeline dependencies related to the address generation unit.

4.1 Architecture and Programming Model

The major components of the address generation unit are as follows:

- 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 65,536 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 on page 4-3). See Section 6.1, “Introduction to Moves and Parallel Moves,” on page 6-1 for more discussion on parallel memory moves.

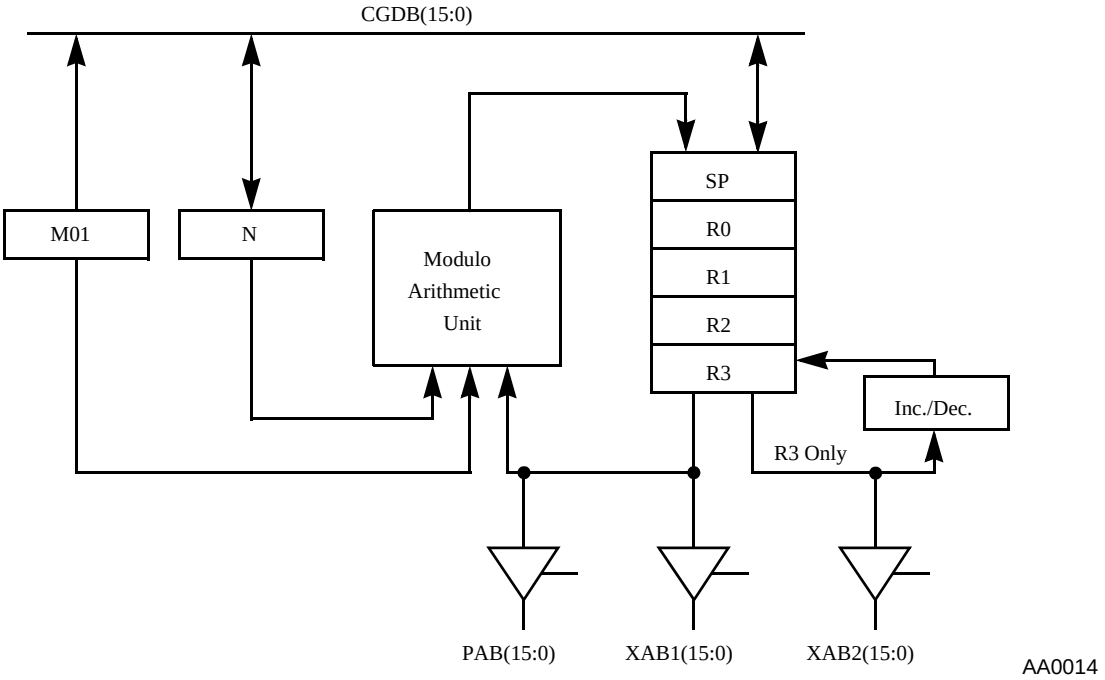


Figure 4-1. Address Generation Unit Block Diagram

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 in Table 4-5 on page 4-9.

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 in the following subsections.

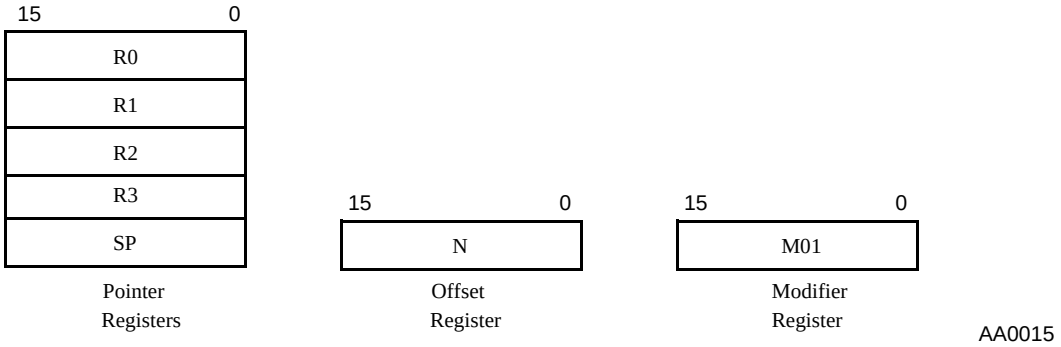


Figure 4-2. Address Generation Unit Programming Model

4.1.1 Address Registers (R0-R3)

The address register file consists of four 16-bit registers R0-R3 (denoted as Rj) which 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 minimum 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 (Rj, 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.

4.1.2 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.

Due to pipelining, if an address register (Rj, 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.

4.1.3 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 Section 8.7, “Array Indexes,” on page 8-26.

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.

4.1.4 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.

4.1.5 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 Section 4.3, "AGU Address Arithmetic." 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 (for example, 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 "wraparound" for modulo arithmetic, the summed output of the modulo adder will give the correct, updated address register value; otherwise, if wraparound is not necessary, the linear arithmetic calculation gives the correct result.

4.1.6 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; it performs no modulo arithmetic.

4.2 Addressing Modes

The DSP56800 instruction set contains a full set of operand addressing modes, optimized for high-performance signal processing as well as efficient controller code. All address calculations are performed in the address generation unit to minimize execution time.

Addressing modes specify where the operand or operands for an instruction can be found — whether an immediate value, located in a register, or in memory — and provide the exact address of the operand(s).

The addressing modes are grouped into four categories:

- Register direct — directly references the processor registers as operands
- Address register indirect — uses an address register as a pointer to reference a location in memory as an operand
- Immediate — the operand is contained as a value within the instruction itself
- Absolute — uses an address contained within the instruction to reference a location in memory as an operand

An effective address in an instruction will specify an addressing mode (that is, 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.

These addressing modes are referred to extensively in Section 6.6.4, “Instruction Summary Tables,” on page 6-17.

Several of the examples in the following sections demonstrate the use of assembler forcing operators. These can be used in an instruction to force a desired addressing mode, as shown in Table 4-1.

Table 4-1. Addressing Mode Forcing Operators

Desired Action	Forcing Operator Syntax	Example
Force immediate short data	#<xx	#<$07
Force 16-bit immediate data	#>xxxx	#>$07
Force absolute short address	X:<xx	X:<$02
Force I/O short address	X:<<xx	X:<<$FFE3
Force 16-bit absolute address	X:>xxxx	X:>$02
Force short offset	X:(SP-<xx)	X:(SP-<$02)
Force 16-bit offset	X:(Rn+>xxxx)	X:(R0+>$03)

Other assembler forcing operators are available for jump and branch instructions, as shown in Table 4-2.

Table 4-2. Jump and Branch Forcing Operators

Desired Action	Forcing Operator Syntax	Example
Force 7-bit relative branch offset	<xx	<LABEL1
Force 16-bit absolute jump address	>xxxx	>LABEL5
Force 16-bit absolute loop address	>xxxx	>LABEL4

4.2.1 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. The various options are shown in Table 4-3 on page 4-7.

Table 4-3. Addressing Mode — Register Direct

Addressing Mode: Register Direct	Notation for Register Direct in the Instruction Set Summary[1]	Examples
Any register	DD DDDDD HHH HHHH F F1 F1DD FDD Rj Rn	A, A2, A1, A0 B, B2, B1, B0 Y, Y1, Y0 X0 R0, R1, R2, R3 SP N M01 PC OMR, SR LA, LC HWS

1. The register field notations found in the middle column are explained in more detail in Table 6-16 on page 6-15 and Table 6-15 on page 6-14.

4.2.1.1 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.

4.2.1.2 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 any address register 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 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 in 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 Rj or SP and the address calculation to be performed. These addressing

modes specify that the operand is (or operands 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.

Table 4-4. Addressing Mode — Address Register Indirect

Addressing Mode: Address Register Indirect	Notation in the Instruction Set Summary[1]	Examples
Accessing Program (P) Memory		
Post-increment	P:(Rj)+	P:(R0)+
Post-update by offset N	P:(Rj)+N	P:(R3)+N
Instructions that access P memory are not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).		
Accessing Data (X) Memory		
No update	X:(Rn)	X:(R3) X:(SP)
Post-increment	X:(Rn)+	X:(R1)+ X:(SP)+
Post-decrement	X:(Rn)-	X:(R3)- X:(SP)-
Post-update by offset N available for word accesses only	X:(Rn)+N	X:(R1)+N
Indexed by offset N	X:(Rn+N)	X:(R2+N) X:(SP+N)
Indexed by 6-bit displacement R2 and SP registers only	X:(R2+xx) X:(SP-xx)	X:(R2+15) X:(SP-$1E)
Indexed by 16-bit displacement	X:(Rn+xxxx)	X:(R0-97) X:(SP+$03F7)

1. Rj represents one of the four pointer registers R0-R3; Rn is any of the AGU address registers R0-R3 or SP.

Address-register-indirect modes may require an offset and a modifier register for use in address calculations. The address register (Rj 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 4-5.

Table 4-5. Address-Register-Indirect Addressing Modes Available

Register Set	Arithmetic Types	Addressing Modes Allowed	Notes
R0/M01/N	Linear or 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 or 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, which is useful for accessing local variables and passed parameters. No modulo arithmetic is allowed.

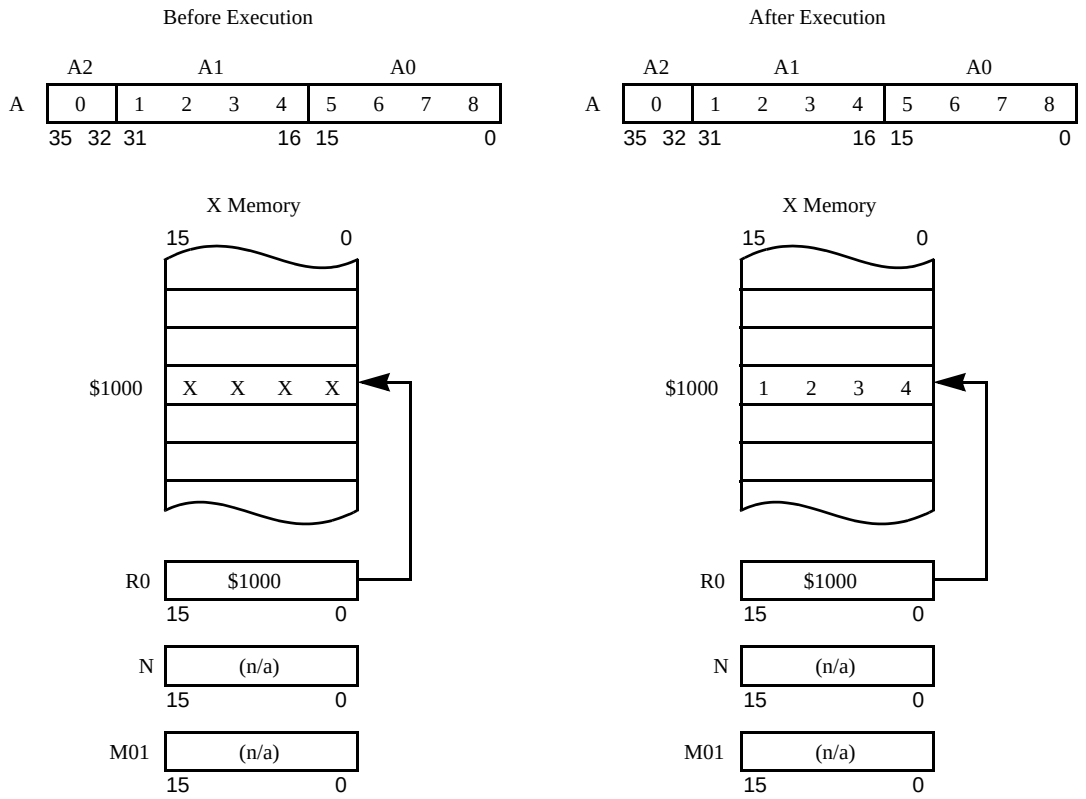
The type of arithmetic to be performed is not encoded in the instruction, but it is specified by the address modifier register (M01 for the DSP56800 core). It indicates whether linear or modulo arithmetic is performed when doing address calculations. In the case where there is not a modifier register 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 in the following subsections.

4.2.2.1 No Update: (Rj), (SP)

The address of the operand is in the address register Rj 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)`



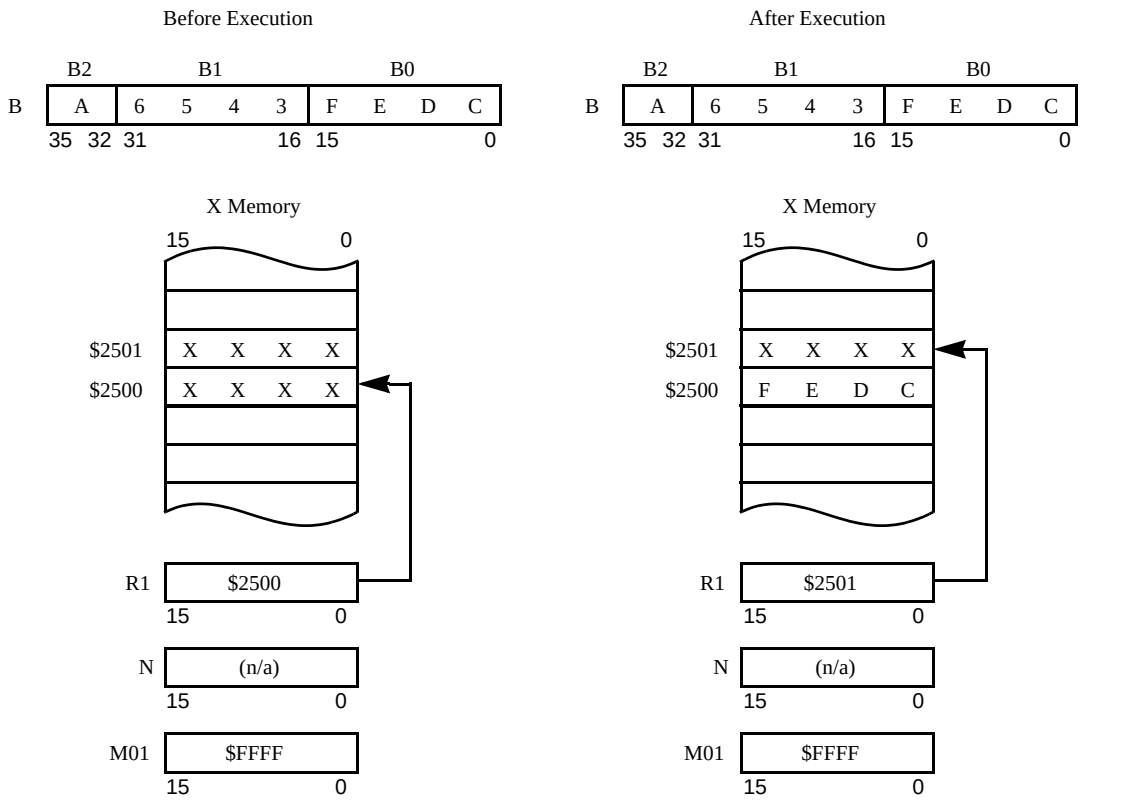
Assembler syntax: X:(Rj), X:(SP)
Additional instruction execution cycles: 0
Additional effective address program words: 0

Figure 4-3. Address Register Indirect: No Update

4.2.2.2 Post-Increment by 1: (Rj)+, (SP)+

The address of the operand is in the address register Rj 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.

Post-Increment Example: `MOVE B0,X:(R1)+`



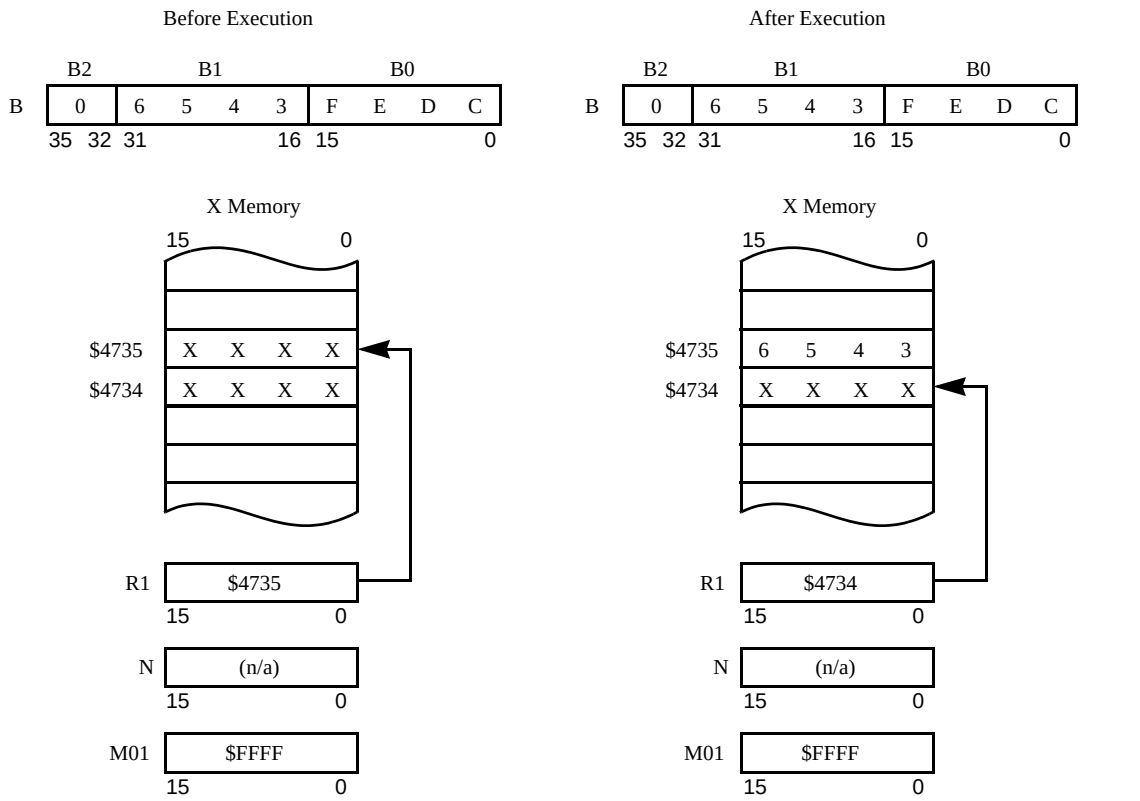
Assembler syntax: X:(Rj)+, X:(SP)+, P:(Rj)+
Additional instruction execution cycles: 0
Additional effective address program words: 0

Figure 4-4. Address Register Indirect: Post-Increment

4.2.2.3 Post-Decrement by 1: (Rn)-, (SP)-

The address of the operand is in the address register Rj 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 decrement 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)-`



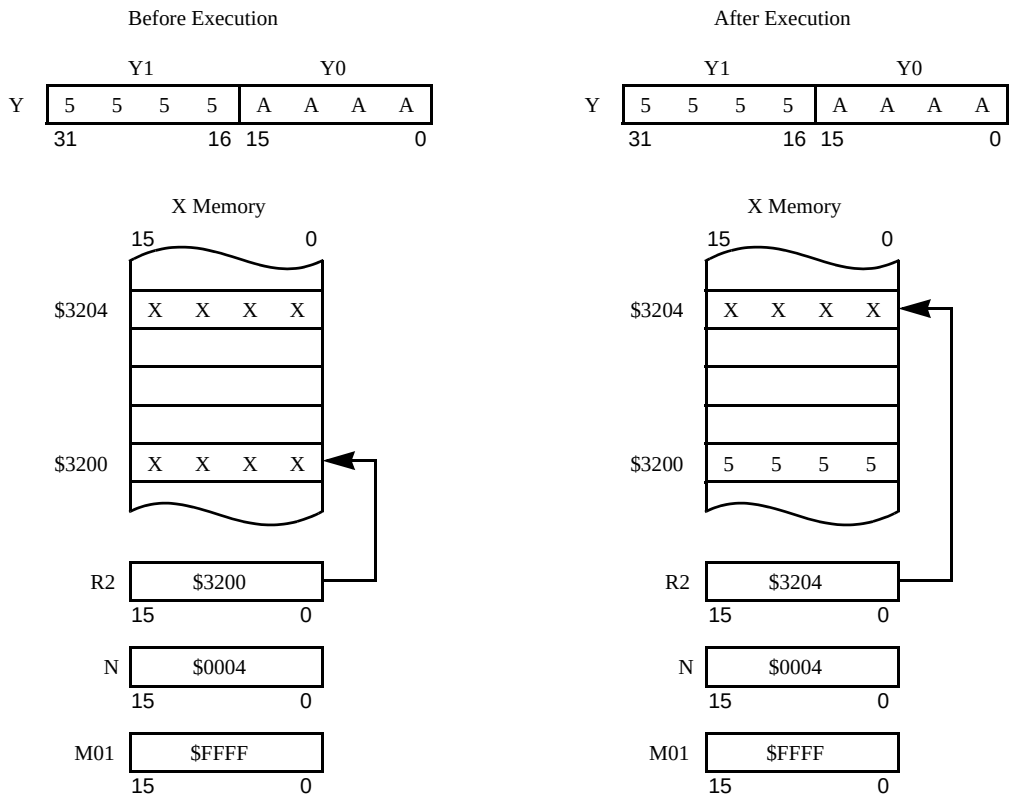
Assembler syntax: X:(Rj)-, X:(SP)-
Additional instruction execution cycles: 0
Additional effective address program words: 0

Figure 4-5. Address Register Indirect: Post-Decrement

4.2.2.4 Post-Update by Offset N: (Rj)+N, (SP)+N

The address of the operand is in the address register Rj 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.

Post-Update by Offset N Example: `MOVE Y1,X:(R2)+N`



Assembler syntax: X:(Rj)+N, X:(SP)+N, P:(Rj)+N
Additional instruction execution cycles: 0
Additional effective address program words: 0

AA0019

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

4.2.2.5 Index by Offset N: (Rj+N), (SP+N)

The address of the operand is the sum of the contents of the address register Rj 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)`

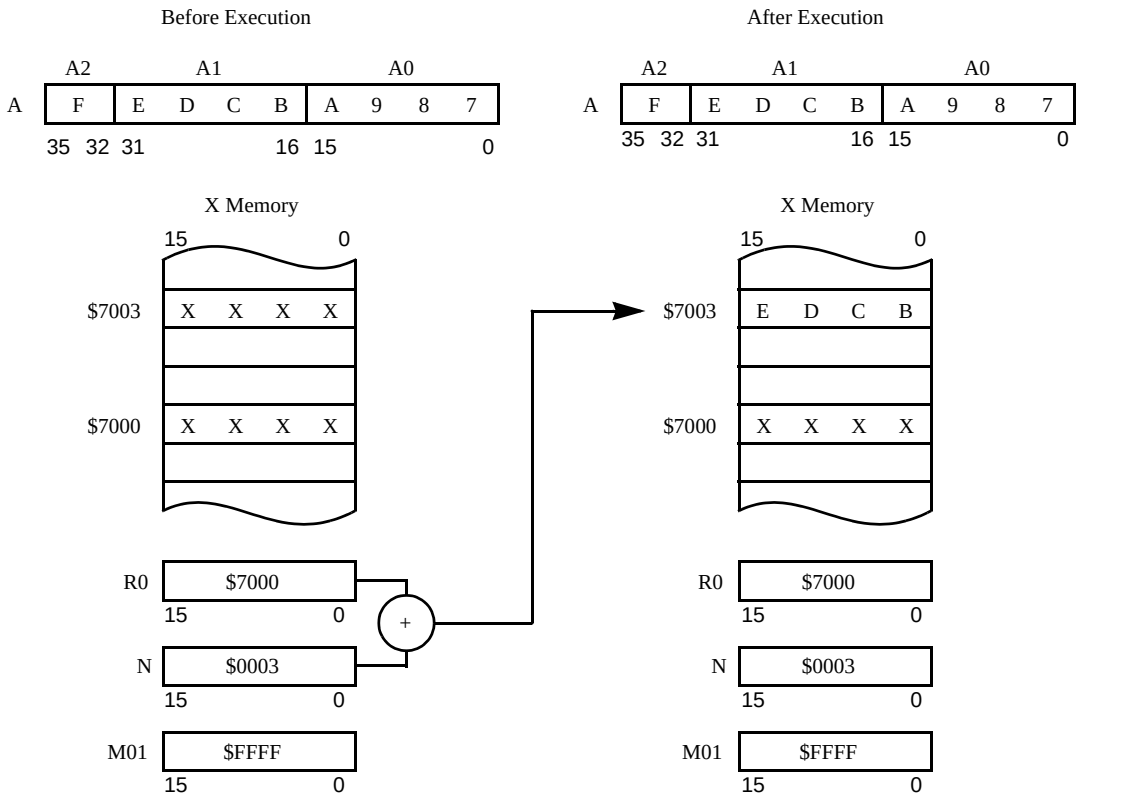
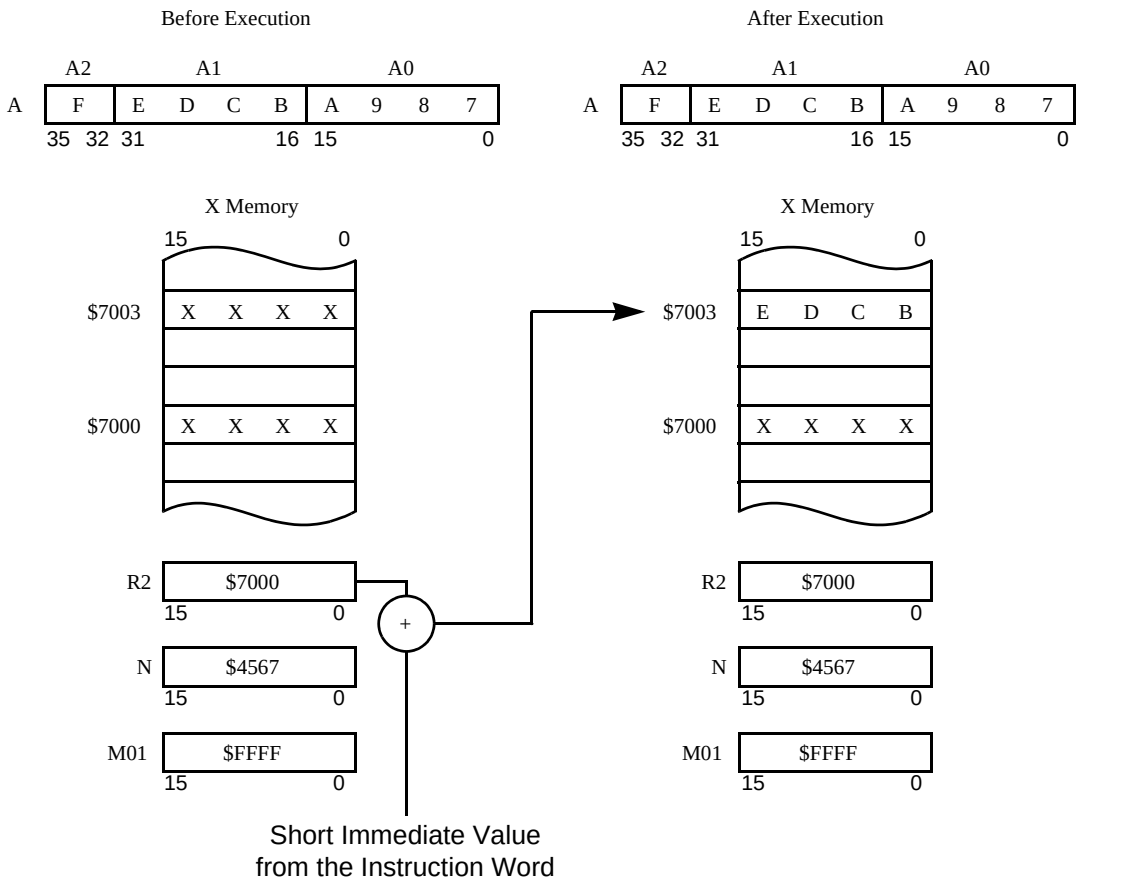


Figure 4-7. Address Register Indirect: Indexed by Offset N

4.2.2.6 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.

Indexed by Short Displacement Example: `MOVE A1,X:(R2+3)`



Assembler syntax: X:(R2+xx), X:(SP-xx)
Additional instruction execution cycles: 1
Additional effective address program words: 0

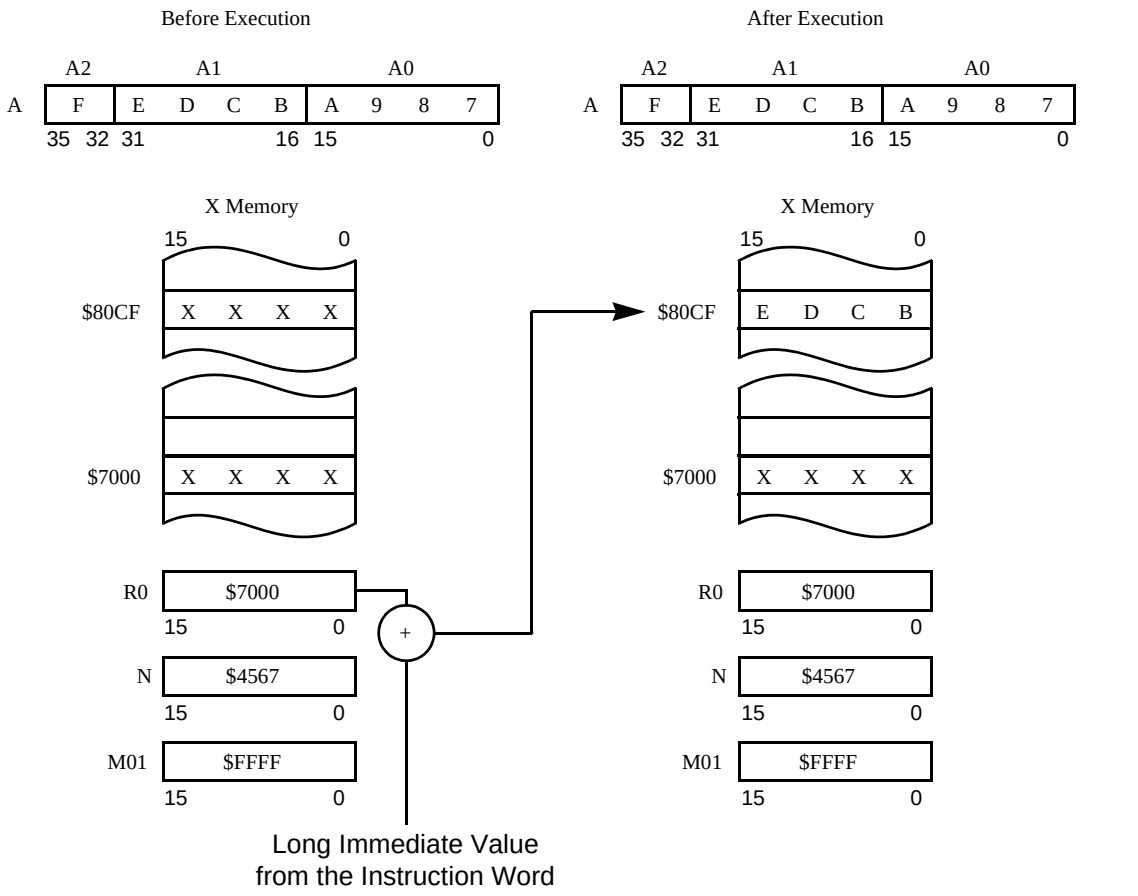
AA0021

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

4.2.2.7 Index by Long Displacement: (Rj+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)`



Assembler syntax: X:(Rj+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

4.2.3 Immediate Data Modes

The immediate data modes specify the operand directly in a field of the instruction. That is, the operand value to be used is contained within the instruction word itself (or words themselves). There are two types of immediate data modes: immediate data, which uses an extension word to contain the operand, and immediate short data, where the operand is contained within the instruction word. Table 4-6 summarizes these two modes.

Table 4-6. Addressing Mode — Immediate

Addressing Mode: Immediate	Notation in the Instruction Set Summary	Examples
Immediate short data — 5, 6, 7-bit (unsigned and signed)	#xx	#14 #<3
Immediate data — 16-bit (unsigned and signed)	#xxxx	#$369C #>1234

4.2.3.1 Immediate Data: #xxxx

This addressing mode requires one word of instruction extension. This additional word contains the 16-bit immediate data used by the instruction. This reference is classified as a program reference. Examples of the use and effects of immediate-data mode are shown in Figure 4-10 on page 4-18.

Immediate into 16-Bit Register Example: `MOVE #$A987,B1`

Before Execution

	B2	B1	B0
B	X	X X X X	X X X X
	35 32	31 16	15 0

After Execution

	B2	B1	B0
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

	B2	B1	B0
B	X	X X X X	X X X X
	35 32	31 16	15 0

After Execution

	B2	B1	B0
B	0	1 2 3 4	0 0 0 0
	35 32	31 16	15 0

Negative Immediate into 36-Bit Accumulator Example: `MOVE #$A987,B`

Before Execution

	B2	B1	B0
B	X	X X X X	X X X X
	35 32	31 16	15 0

After Execution

	B2	B1	B0
B	F	A 9 8 7	0 0 0 0
	35 32	31 16	15 0

Assembler syntax: #xxxx
Additional instruction execution cycles: 1
Additional effective address program words: 1

AA0023

Figure 4-10. Special Addressing: Immediate Data

Immediate Short into 16-Bit Address Register Example: `MOVE #$0027,N`

	Before Execution	After Execution
N	XXXX (15 … 0)	$0027 (15 … 0)

Immediate Short into 16-Bit Data Register Example: `MOVE #$FFC6,X0`

	Before Execution	After Execution
X0	XXXX (15 … 0)	$FFC6 (15 … 0)

Immediate Short into 16-Bit Accumulator Register Example: `MOVE #$001C,B1`

	Before Execution	After Execution
B	B2: X; B1: X X X X; B0: X X X X (35 32 31 … 16 15 … 0)	B2: X; B1: 0 0 1 C; B0: X X X X (35 32 31 … 16 15 … 0)

Positive Immediate Short into 36-Bit Accumulator Example: `MOVE #$001C,B`

	Before Execution	After Execution
B	B2: X; B1: X X X X; B0: X X X X (35 32 31 … 16 15 … 0)	B2: 0; B1: 0 0 1 C; B0: 0 0 0 0 (35 32 31 … 16 15 … 0)

Negative Immediate Short into 36-Bit Accumulator Example: `MOVE #$FFC6,B`

	Before Execution	After Execution
B	B2: X; B1: X X X X; B0: X X X X (35 32 31 … 16 15 … 0)	B2: F; B1: F F C 6; B0: 0 0 0 0 (35 32 31 … 16 15 … 0)

Assembler syntax: #xx
Additional instruction execution cycles: 0
Additional effective address program words: 0

AA0024

Figure 4-11. Special Addressing: Immediate Short Data

4.2.3.2 Immediate Short Data: #xx

The immediate-short-data operand is located within the instruction operation word. A 6-bit unsigned positive operand is used for DO and REP instructions, and a 7-bit signed operand is used for an immediate move to an on-core register instruction. This reference is classified as a program reference. See Figure 4-11 on page 4-19.

4.2.4 Absolute Addressing Modes

Similar to the direct addressing modes, the absolute addressing modes specify the operand value within the instruction or instruction-extension words. Unlike the direct modes, these values are not used as the operands themselves, but are interpreted as absolute data memory addresses for the operand values. The different absolute addressing modes are shown in Table 4-7.

Table 4-7. Addressing Mode — Absolute

Addressing Mode: Absolute	Notation in the Instruction Set Summary	Examples
Absolute short address — 6 bit (direct addressing)	X:aa	X:$0002 X:<$02
I/O short address[1] — 6 bit (direct addressing)	X:<<pp	X:<<$FFE3
Absolute address — 16-bit (extended addressing)	X:xxxx	X:$C002

1. I/O short addressing mode is used when the peripheral registers are mapped to the last 64 locations in X memory. When IP-BUS (or PGDB) interface maps these registers outside the X:$FFC0-X:$FFFF range, they are then accessed with other suitable standard addressing mode.

4.2.4.1 Absolute Address (Extended Addressing): xxxx

This addressing mode requires one word of instruction extension, which contains the 16-bit absolute address of the operand. 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.

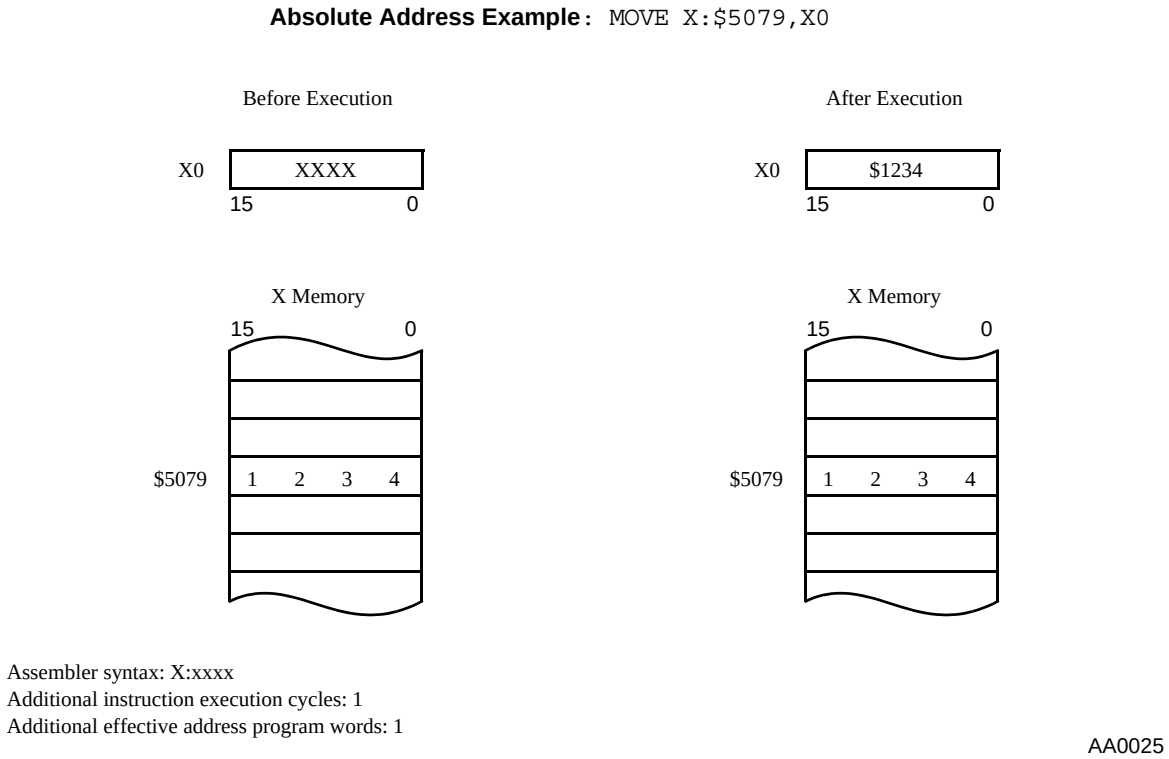
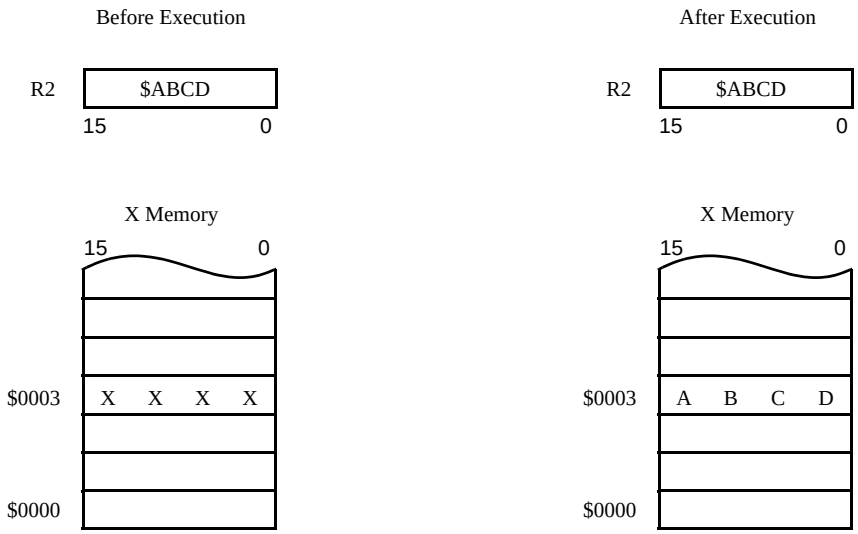


Figure 4-12. Special Addressing: Absolute Address

4.2.4.2 Absolute Short Address (Direct Addressing): <aa>

For the absolute short addressing mode, the address of the operand occupies 6 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

AA0026

Figure 4-13. Special Addressing: Absolute Short Address

4.2.4.3 I/O Short Address (Direct Addressing): <pp>

When the peripheral registers are mapped to the last 64 locations in X memory, these can be accessed with short addressing mode. For the I/O short addressing mode, the address of the operand occupies 6 bits in the instruction operation word and is one-extended. This allows direct access to the last 64 locations in X memory, which may contain the on-chip peripheral registers. No registers are used to form the address of the operand. See Figure 4-14 for examples of using the I/O short direct addressing mode. Note that when peripherals are connected to the DSP56800 core using the Freescale-standard IP-BUS (or PGDB) interface, peripheral registers may be mapped into any other data (X) memory range. Note that if the peripheral registers are mapped to an area of memory outside the range X:$FFC0-X:$FFFF, this address mode will not be available and the registers are then accessed with other suitable standard addressing mode.

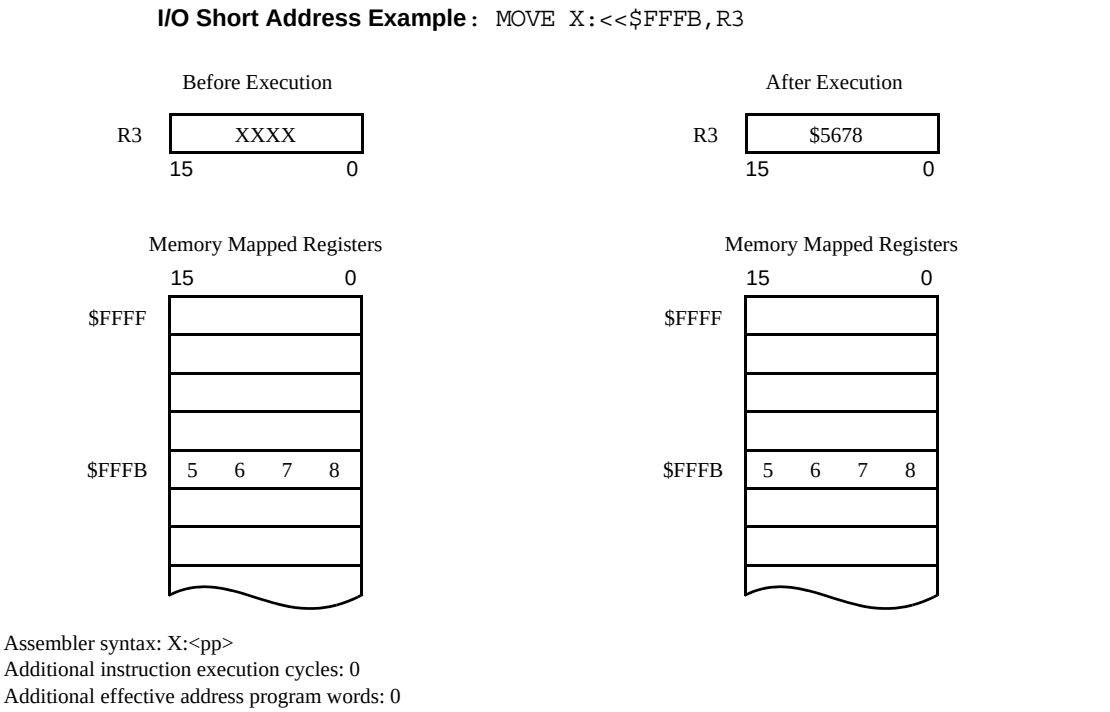


Figure 4-14. Special Addressing: I/O Short Address

4.2.5 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 implied registers and their use are defined by the individual instruction descriptions. See Appendix A, “Instruction Set Details,“ for more information.

4.2.6 Addressing Modes Summary

Table 4-8 on page 4-24 contains a summary of the addressing modes discussed in the preceding subsections of Section 4.2.

Table 4-8. Addressing Mode Summary

Addressing Mode	Uses M01[1]	Operand Reference							Assembler Syntax
		S[2]	C[3]	D[4]	A[5]	P[6]	X[7]	XX[8]	
Register Direct									
Data or control register	No		X	X					
Address register (Rj, SP)	No				X				Rn
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)
Post-increment by 1	Yes					X	X	X	(Rn)+
Post-decrement by 1	Yes						X		(Rn)-
Post-update by offset N	Yes					X	X	X	(Rn)+N
Index by offset N	Yes						X		(Rn+N)
Index by short displacement	No						X		(R2+xx) or (SP-xx)
Index by long displacement	Yes						X		(Rn+xxxx)
Immediate, Absolute, and Implicit									
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		

1. The M01 modifier can only be used on the R0/N/M01 or R1/N/M01 register sets
2. Hardware stack reference
3. Program controller register reference
4. Data ALU register reference
5. Address Generation Unit register reference
6. Program memory reference
7. X memory reference
8. Dual X memory read

4.3 AGU Address Arithmetic

When an arithmetic operation is performed in the address generation unit, it can be performed using either linear or modulo arithmetic. Linear arithmetic is used for general-purpose address computation, as found in all microprocessors. Modulo arithmetic is used to create data structures in memory such as circular buffers, first-in-first-out queues (FIFOs), delay lines, and fixed-size stacks. Using these structures allows data to be manipulated simply by updating address register pointers, rather than by moving large blocks of data.

Linear versus modulo arithmetic is selected using the modifier register, M01. Arithmetic on the R0 and R1 AGU registers may be performed using either linear or modulo arithmetic. The R2, R3, and SP registers can be modified using linear arithmetic only.

4.3.1 Linear Arithmetic

Linear arithmetic is “normal” address arithmetic, as found on general-purpose microprocessors. It is performed using 16-bit two’s-complement addition and subtraction. The 16-bit offset register N, or immediate data (+1, -1, or a displacement value), is used in the address calculations. Addresses are normally considered unsigned; offsets are considered signed.

Linear arithmetic is enabled for the R0 and R1 registers by setting the modifier register (M01) to $FFFF. The M01 register is set to $FFFF on reset.

NOTE:

To ensure compatibility with future generations of DSP56800-compatible DSC devices, care should be taken to avoid address arithmetic operations that can cause address register values to overflow. On DSP56800 Family chips, register values can be expected to “wrap” appropriately. Future generations may support address ranges > 64K, however, causing potential address-calculation errors.

4.3.2 Modulo Arithmetic

Many DSC and standard control algorithms require the use of specialized data structures, such as circular buffers, FIFOs, and stacks. The DSP56800 architecture provides support for these algorithms by implementing modulo arithmetic in the address generation unit.

4.3.2.1 Modulo Arithmetic Overview

To understand modulo address arithmetic, consider the example of a circular buffer. A circular buffer is a block of sequential memory locations with a special property: a pointer into the buffer is limited to the buffer’s address range. When a buffer pointer is incremented such that it would point past the end of the buffer, the pointer is “wrapped” back to the beginning of the buffer. Similarly, decrementing a pointer that is located at the beginning of the buffer will wrap the pointer to the end. This behavior is achieved by performing modulo arithmetic when incrementing or decrementing the buffer pointers. See Figure 4-15 on page 4-26.

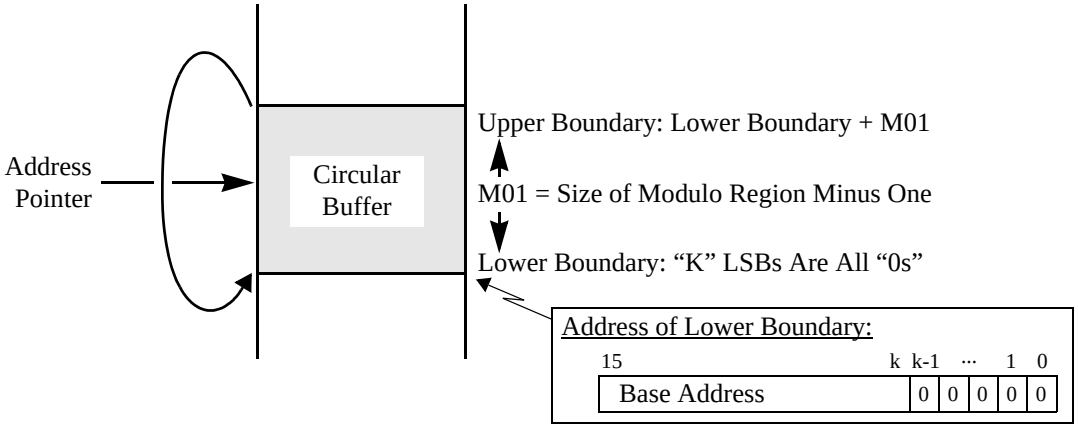


Figure 4-15. Circular Buffer

The modulo arithmetic unit in the AGU simplifies the use of a circular buffer by handling the address pointer wrapping for you. After establishing a buffer in memory, the R0 and R1 address pointers can be made to wrap in the buffer area by programming the M01 register.

Modulo arithmetic is enabled by programming the M01 register with a value that is one less than the size of the circular buffer. See Section 4.3.2.2, “Configuring Modulo Arithmetic,” for exact details on programming the M01 register. Once enabled, updates to the R0 or R1 registers using one of the post-increment or post-decrement addressing modes are performed with modulo arithmetic, and will wrap correctly in the circular buffer.

The address range within which the address pointers will wrap is determined by the value placed in the M01 register and the address contained within one of the pointer registers. Due to the design of the modulo arithmetic unit, the address range is not arbitrary, but limited based on the value placed in M01. The lower bound of the range is calculated by taking the size of the buffer, rounding it up to the next highest power of two, and then rounding the address contained in the R0 or R1 pointers down to the nearest multiple of that value.

For example: for a buffer size of M, a value 2^k is calculated such that $2^k \geq M$. This is the buffer size rounded up to the next highest power of two. For a value M of 37, 2^k would be 64. The lower boundary of the range in which the pointer registers will wrap is the value in the R0 or R1 register with the low-order k bits all set to zero, effectively rounding the value down to the nearest multiple of 2^k (64 in this case). This is shown in Figure 4-16 on page 4-27.

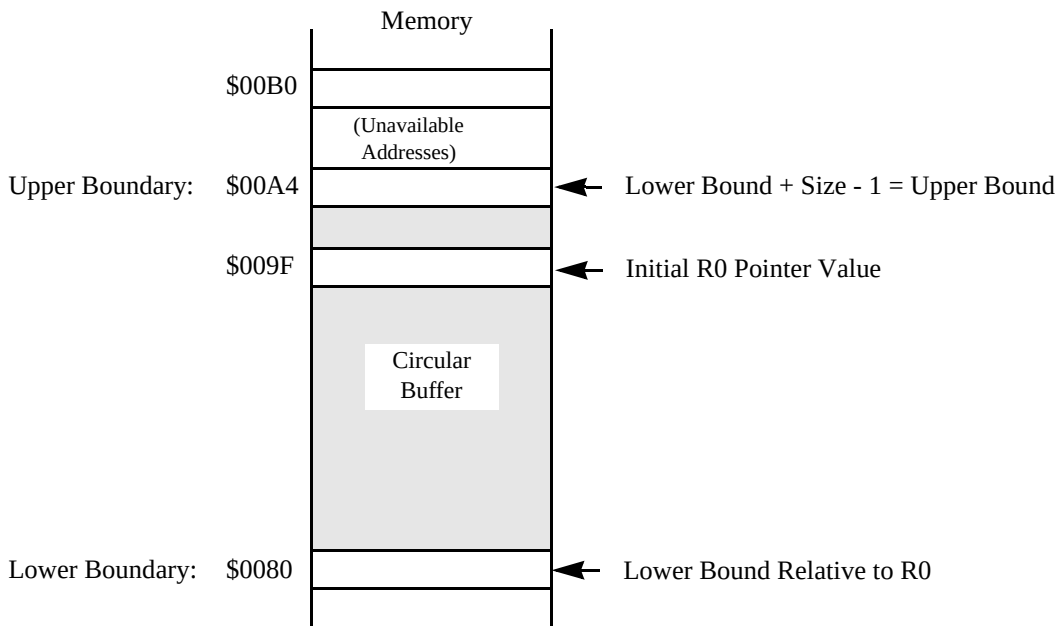


Figure 4-16. Circular Buffer with Size M=37

When modulo arithmetic is performed on the buffer pointer register, only the low-order k bits are modified; the upper 16 - k bits are held constant, fixing the address range of the buffer. The algorithm used to update the pointer register (R0 in this case) is as follows:

R0[15:k] = R0[15:k]
R0[k-1:0] = (R0[k-1:0] + offset) MOD (M01 + 1)

Note that this algorithm can result in some memory addresses being unavailable. If the size of the buffer is not an even power of two, there will be a range of addresses between M and 2^k-1 (37 and 63 in our example) that are not addressable. Section 4.3.2.7.3, “Memory Locations Not Available for Modulo Buffers,” addresses this issue in greater detail.

4.3.2.2 Configuring Modulo Arithmetic

As noted in Section 4.3.2.1, “Modulo Arithmetic Overview,” modulo arithmetic is enabled by programming the address modifier register, M01. This single register enables modulo arithmetic for both the R0 and R1 registers, although in order for modulo arithmetic to be enabled for the R1 register it must be enabled for the R0 register as well. When both pointers use modulo arithmetic, the sizes of both buffers are the same. They can refer to the same or different buffers as desired.

The possible configurations of the M01 register are given in Table 4-9.

Table 4-9. Programming M01 for Modulo Arithmetic

16-Bit M01 Register Contents	Address Arithmetic Performed	Pointer Registers Affected
$0000	(Reserved)	—
$0001	Modulo 2	R0 pointer only
$0002	Modulo 3	R0 pointer only

Table 4-9. Programming M01 for Modulo Arithmetic (Continued)

16-Bit M01 Register Contents	Address Arithmetic Performed	Pointer Registers Affected
...	...	...
$3FFE	Modulo 16383	R0 pointer only
$3FFF	Modulo 16384	R0 pointer only
$4000	(Reserved)	—
...	...	...
$7FFF	(Reserved)	—
$8000	(Reserved)	—
$8001	Modulo 2	R0 and R1 pointers
$8002	Modulo 3	R0 and R1 pointers
...	...	...
$BFFE	Modulo 16383	R0 and R1 pointers
$BFFF	Modulo 16384	R0 and R1 pointers
$C000	(Reserved)	—
...	...	...
$FFFE	(Reserved)	—
$FFFF	Linear Arithmetic	R0 and R1 pointers both set up for linear arithmetic

The high-order two bits of the M01 register determine the arithmetic mode for R0 and R1. A value of 00 for M01[15:14] selects modulo arithmetic for R0. A value of 10 for M01[15:14] selects modulo arithmetic for both R0 and R1. A value of 11 disables modulo arithmetic. The remaining 14 bits of M01 hold the size of the buffer minus one.

NOTE:

The reserved values ($0000, $4000-$8000, and $C000-$FFFE) should not be used. The behavior of the modulo arithmetic unit is undefined for these values, and may result in erratic program execution.

4.3.2.3 Supported Memory Access Instructions

The address generation unit supports modulo arithmetic for the following address-register-indirect modes:

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

As noted in the preceding discussion, modulo arithmetic is only supported for the R0 and R1 address registers.

4.3.2.4 Simple Circular Buffer Example

Suppose a five-location circular buffer is needed for an application. The application locates this buffer at X:$800 in memory. (This location is arbitrary — any location in an allowable data memory would suffice.) In order to configure the AGU correctly to manage this circular buffer, the following two pieces of information are needed:

The size of the buffer: five words
The location of the buffer: X:$0800 – X:$0804 (assume allowable memory locations)

Modulo addressing is enabled for the R0 pointer by writing the size minus one ($0004) to M01[13:0], and 00 to M01[15:14]. See Figure 4-17.

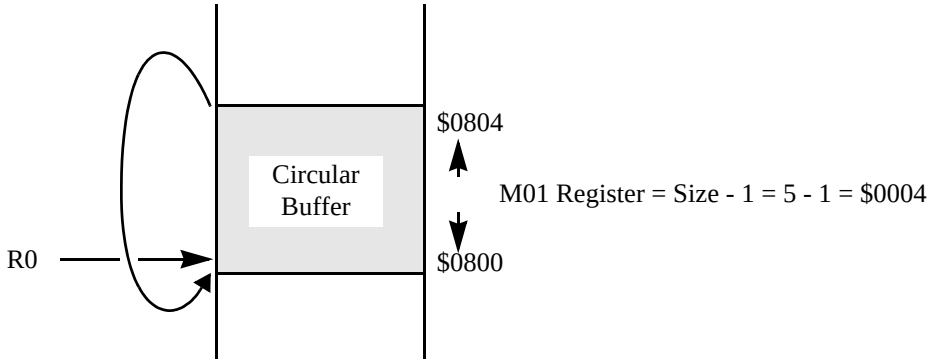


Figure 4-17. Simple Five-Location Circular Buffer

The location of the buffer in memory is determined by the value of the R0 pointer when it is used to access memory. The size of the memory buffer (five in this case) is rounded *up* to the nearest power of two (eight in this case). The value in R0 is then rounded *down* to the nearest multiple of eight. For the base address to be X:$0800, the initial value of R0 must be in the range X:$0800 – X:$0804. Note that the initial value of R0 does not have to be X:$0800 to establish this address as the lower bound of the buffer. However, it is often convenient to set R0 to the beginning of the buffer. The source code in Example 4-1 shows the initialization of the example buffer.

Example 4-1. Initializing the Circular Buffer

```
MOVE   #(5-1),M01    ; Initialize the buffer for five locations
MOVE   #$0800,R0     ; R0 can be initialized to any location
                     ; within the buffer. For simplicity, R0
                     ; is initialized to the value of the lower
                     ; boundary
```

The buffer is used simply by accessing it with MOVE instructions. The effect of modulo address arithmetic becomes apparent when the buffer is accessed multiple times, as in Example 4-2 on page 4-30.

Example 4-2. Accessing the Circular Buffer

```
MOVE   X:(R0)+,X0   ; First time accesses location $0800
                    ; and bumps the pointer to location $0801
MOVE   X:(R0)+,X0   ; Second accesses at location $0801
MOVE   X:(R0)+,X0   ; Third accesses at location $0802
MOVE   X:(R0)+,X0   ; Fourth accesses at location $0803
MOVE   X:(R0)+,X0   ; Fifth accesses at location $0804
                    ; and bumps the pointer to location $0800

MOVE   X:(R0)+,X0   ; Sixth accesses at location $0800 <=== NOTE
MOVE   X:(R0)+,X0   ; Seventh accesses at location $0801
MOVE   X:(R0)+,X0   ; and so forth...
```

For the first several memory accesses, the buffer pointer is incremented as expected, from $0800 to $0801, $0802, and so forth. When the pointer reaches the top of the buffer, rather than incrementing from $0804 to $0805, the pointer value “wraps” back to $0800.

The behavior is similar when the buffer pointer register is incremented by a value greater than one. Consider the source code in Example 4-3, where R0 is post-incremented by three rather than one. The pointer register correctly “wraps” from $0803 to $0801 — the pointer does not have to land exactly on the upper and lower bound of the buffer for the modulo arithmetic to wrap the value properly.

Example 4-3. Accessing the Circular Buffer with Post-Update by Three

```
MOVE   #(5-1),M01   ; Initialize the buffer for five locations
MOVE   #$0800,R0    ; Initialize the pointer to $0800
MOVE   #3,N         ; Initialize “bump value” to 3
NOP
NOP
MOVE   X:(R0)+N,X0  ; First time accesses location $0800
                    ; and bumps the pointer to location $0803
MOVE   X:(R0)+N,X0  ; Second accesses at location $0803
                    ; and wraps the pointer around to $0801

MOVE   X:(R0)+N,X0  ; Third accesses at location $0801
                    ; and bumps the pointer to location $0804
MOVE   X:(R0)+N,X0  ; Fourth accesses at ...
```

In addition, the pointer register does not need to be incremented; it could be decremented instead. Instructions that post-decrement the buffer pointer also work correctly. Executing the instruction `MOVE X:(R0)-,X0` when the value of R0 is $0800 will correctly set R0 to $0804.

4.3.2.5 Setting Up a Modulo Buffer

The following steps detail the process of setting up and using the 37-location circular buffer shown in Figure 4-16 on page 4-27.

1. Determine the value for the M01 register.
   - Select the size of the desired buffer; it can be no larger than 16,384 locations. If modulo arithmetic is to be enabled only for the R0 address register, this gives the following:
     M01 = # locations - 1 = 37 - 1 = 36 = $0024
   - If modulo arithmetic is to be enabled for both the R0 and R1 address registers, be sure to set the high-order bit of M01:
     M01 = # locations - 1 + $8000 = 37 - 1 + 32768 = 32804 = $8024

2. Find the nearest power of two greater than or equal to the circular buffer size. In this example, the value would be $2^k \geq 37$, which gives us a value of k = 6.
3. From k, derive the characteristics of the lower boundary of the circular buffer. Since the "k" least-significant bits of the address of the lower boundary must all be 0s, then the buffer base address must be some multiple of 2^k. In this case, k = 6, so the base address is some multiple of $2^6 = 64$.
4. Locate 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 J = 2:
   Lower boundary = (J x 64) = (2 x 64) = 128 = $0080
   Upper boundary = (J 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 inclusively between the desired lower and upper boundaries of the circular buffer. 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 10 (16 - k) 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 to the address pointer register (R0 or R1) that will be used in a modulo arithmetic calculation.
5. Determine the upper boundary of the circular buffer, which is the lower boundary + # locations - 1.
6. Select a value for the offset register if it is 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 discussed in Section 4.3.2.6, "Wrapping to a Different Bank."
7. Perform the modulo arithmetic calculation.
   — Once the appropriate registers are set up, the modulo arithmetic operation occurs when an instruction with any of the following addressing modes using the R0 (or R1, if enabled) register is executed:
   (Rn)
   (Rn)+
   (Rn)-
   (Rn)+N
   (Rn+N)
   (Rn+xxxx)
   — If the result of the arithmetic calculation would exceed the upper or lower bound, then wrapping around is correctly performed.

4.3.2.6 Wrapping to a Different Bank

For the normal case where |N| is less than or equal to M01, the primary address 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 M01, 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 that is L blocks forward in memory (see Figure 4-18). Note that this case requires that the offset N must be a positive two's-complement integer.

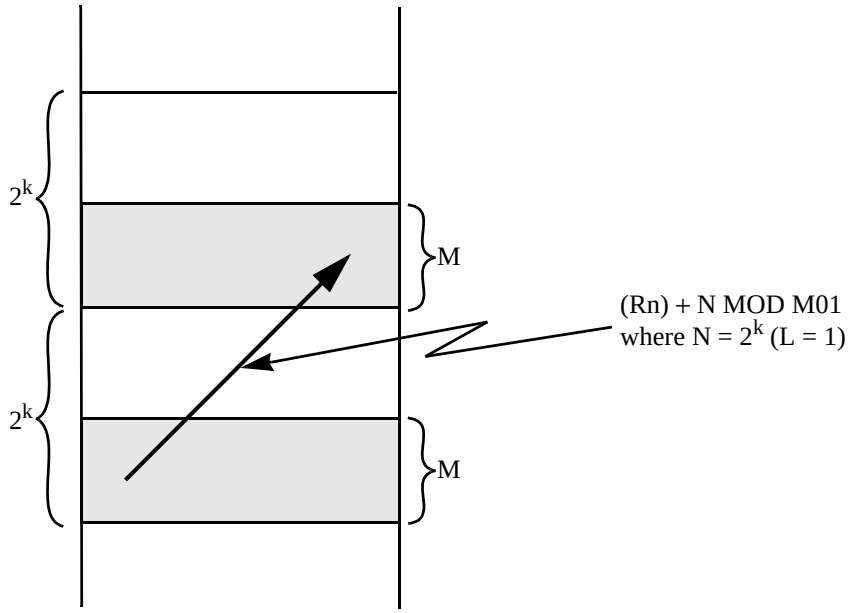


Figure 4-18. Linear Addressing with a Modulo Modifier

This technique is useful in sequentially processing multiple tables or N-dimensional arrays. 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).

4.3.2.7 Side Effects of Modulo Arithmetic

Due to the way modulo arithmetic is implemented by the DSP56800 Family, there are some side effects of using modulo arithmetic that must be kept in mind. Specifically, since the base address of a buffer must be a power of two, and since the modulo arithmetic unit can only detect a single wraparound, there are some restrictions and limitations that must be considered.

4.3.2.7.1 When a Pointer Lies Outside a Modulo Buffer

If a pointer is outside the valid modulo buffer range and an operation occurs that causes R0 or R1 to be updated, the contents of the pointer will be updated according to modulo arithmetic rules. For example, a `MOVE B,X:(R0)+N` instruction, where R0 = 6, M01 = 5, and N = 0, would apparently leave R0 unchanged since N = 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.

4.3.2.7.2 Restrictions on the Offset Register

The modulo arithmetic unit in the AGU is only capable of detecting a single wraparound of an address pointer. As a result, if the post-update addressing mode, (Rn)+N, is used, care must be taken in selecting the value of N. The 16-bit absolute value |N| must be less than or equal to M01 + 1 for proper modulo addressing. Values of |N| larger than the size of the buffer may result in the Rn address value wrapping twice, which the AGU cannot detect.

4.3.2.7.3 Memory Locations Not Available for Modulo Buffers

For cases where the size of a buffer is not a power of two, there will be a range of memory locations immediately after the buffer that are not accessible with modulo addressing. Lower boundaries for modulo buffers always begin on an address where the lowest k bits are zeros — that is, a power of two. This means that for buffers that are not an exact power of two, there are locations above the upper boundary that are not accessible through modulo addressing.

In Figure 4-16 on page 4-27, for example, the buffer size is 37, which is not a power of two. The smallest power of two greater than 37 is 64. Thus, there are 64 - 37 = 27 memory locations which are not accessible with modulo addressing. These 27 locations are between the upper boundary + 1 = $00A5 and the next power of two boundary address - 1 = $00C0 - 1 = $00BF.

These locations are still accessible when no modulo arithmetic is performed. Using linear addressing (with the R2 or R3 pointers), absolute addresses, or the no-update addressing mode makes these locations available.

4.4 Pipeline Dependencies

There are some cases within the address generation unit where the pipelined nature of the DSC 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 DSC 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 in the following examples to examine cases where their pipeline dependency occurs, how this affects the machine, and how to correctly program to avoid these dependencies.

In Example 4-4 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 4-4. 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 4-5 there is no pipeline dependency since the R2 and N registers, used in the address calculation, are not written in the previous instruction. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-5. 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 4-6 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 4-6. No Dependency with No Address Arithmetic Calculation

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

Example 4-7 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 4-7. 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 4-8 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 DSC 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 4-8. 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 4-9 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 4-9. 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 4-10 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 4-10. 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 4-11 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 4-11. 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
```

In Example 4-12 there is a pipeline dependency since the SP register written in the first instruction is used by the immediately following JSR instruction to store the subroutine return address. The stack pointer will not be updated with the immediate value in this case. This sequence may be fixed by inserting a NOP between the two instructions.

Example 4-12. Dependency with a Write to the Stack Pointer Register

```
      MOVE   #$3800,SP          ; Write to the SP register
      JSR    LABEL              ; SP implicitly used to save the return address
                                ;   of the subroutine call
```

In Example 4-13 there is a pipeline dependency due to contention in the LF bit of the SR register. During the first execution cycle of the BFSET instruction, the SR, whose LF bit is zero, is read. At the same time, the first operand of the DO instruction is fetched. During the second execution cycle of the BFSET instruction, the SR's content is modified and written back to the SR. This is also the DO instruction decode cycle, when the LF bit is set. In this case, the LF bit is first set by the DO decode, then cleared by the BFSET SR modification. A cleared LF bit signals the end of a DO loop, so the DO loop is executed only once. This sequence can be fixed by inserting a NOP instruction between these two instructions.

Example 4-13. Dependency with a Bit-Field Operation and DO Loop

```
      BFSET  #$0200,SR          ; Write to the SR register
      DO     #8,ENDLOOP         ; Repeat 8 times body of loop
;     (instructions)
ENDLOOP:
```

Chapter 5
Program Controller

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

Details of the instruction pipeline and the different processing states of the DSC chip, including reset and interrupt processing, are covered in Chapter 7, “Interrupts and the Processing States.”

5.1 Architecture and Programming Model

A block diagram of the program controller is shown in Figure 5-1 on page 5-2, and its corresponding programming model is shown in Figure 5-2 on page 5-3. 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 DSC operating modes.

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

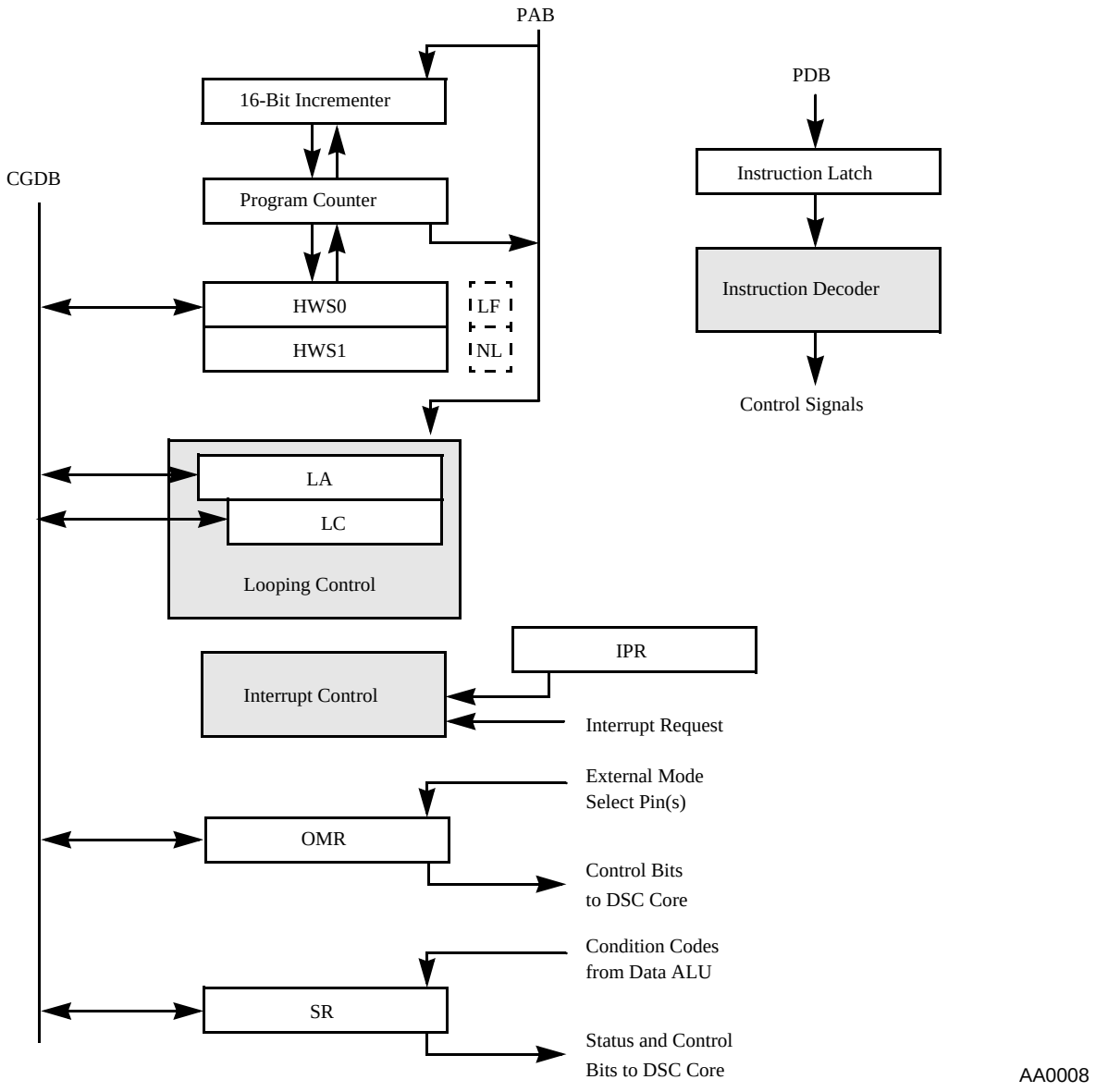


Figure 5-1. Program Controller Block Diagram

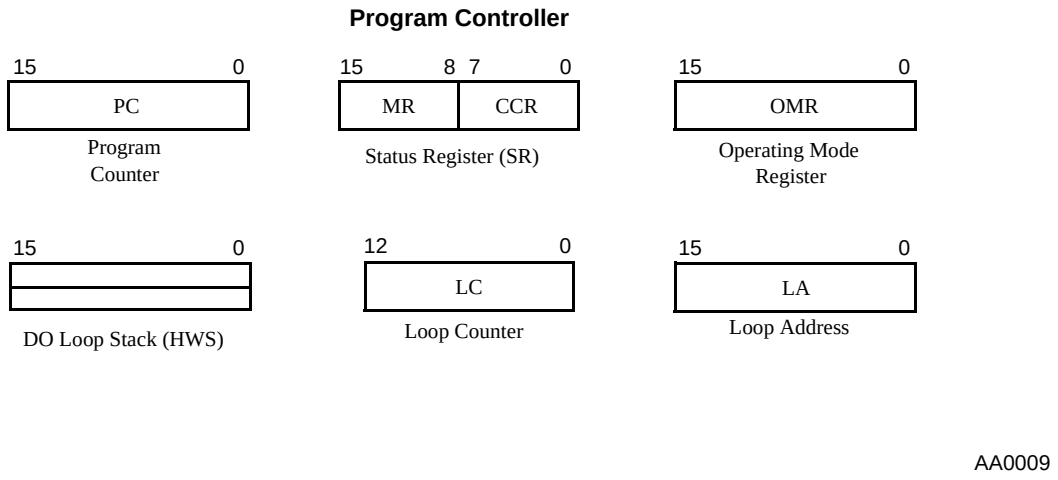


Figure 5-2. Program Controller Programming Model

5.1.1 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. The PC may point to instructions, data operands, or addresses of operands. Reference to this register is always implicit and is 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).

5.1.2 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.

5.1.3 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 DSC core (I1 and I0 in the SR). If the requesting interrupt has higher priority than the current priority level of the DSC 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 DSC 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 DSC core and indicate the level needed for an interrupt source to interrupt the processor.

The DSP56800 core provides support for internal (on-chip) peripheral interrupts and two external interrupt sources, $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$. The interrupt control unit arbitrates between interrupt requests generated externally and by the on-chip peripherals.

Asserting the reset pin causes the DSC 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 Section 7.3, “Exception Processing State,” on page 7-5. The reset processing state is discussed in Section 7.1, “Reset Processing State,” on page 7-1.

5.1.4 Looping Control Unit

The looping control unit provides hardware dedicated to support loops, which are frequent constructs in DSC 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 Section 8.6.4, “Nested Loops,” on page 8-22.

Looping is described in more detail in Section 5.3, “Program Looping,” and Section 8.6, “Loops,” on page 8-20.

5.1.5 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. It also allows for saving and restoring the LC to and from the software stack when nesting DO loops in software. Note that since the LC is only a 13-bit counter, it is zero-extended when read; when written, the top three bits of the source word are ignored. This is shown in Figure 5-3 on page 5-5.

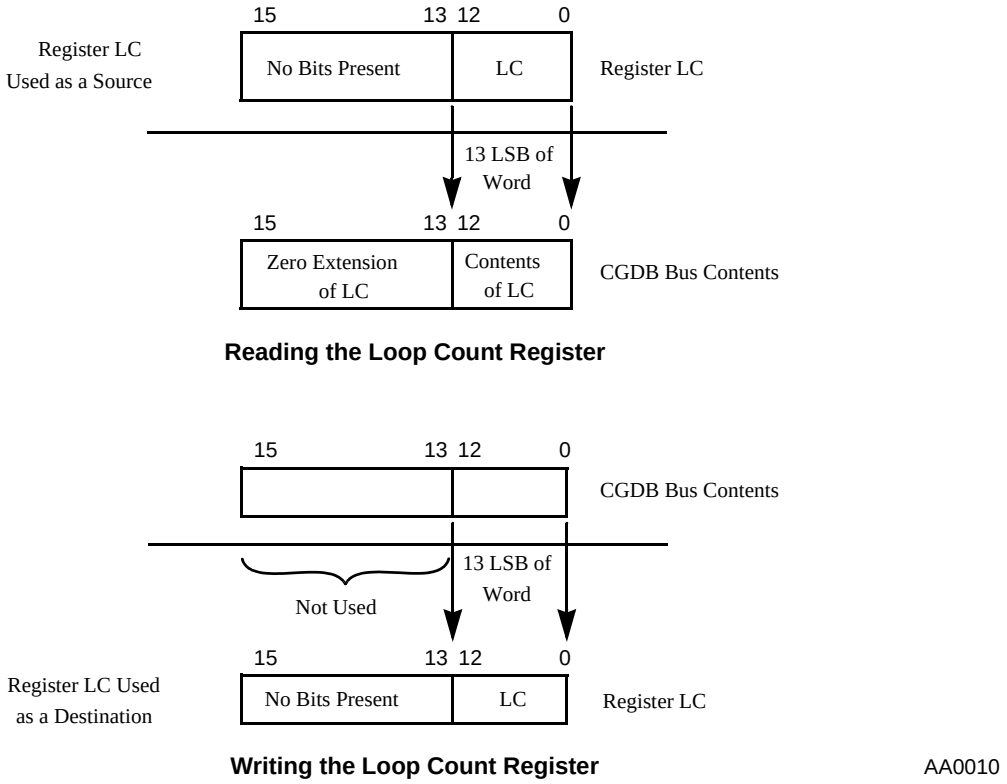


Figure 5-3. Accessing the Loop Count Register (LC)

This register is not stacked by a DO instruction and not unstacked by end-of-loop processing, as is done on other Freescale DSCs. Section 5.3, “Program Looping,” discusses what occurs when the loop count is zero. See Section 8.6.4, “Nested Loops,” on page 8-22 for a discussion of nesting loops in software.

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

5.1.6 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 one, the LC is decremented, and the next instruction is taken from the address at the top of the system stack; otherwise the PC is incremented, the 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 the nesting of loops. This register is not stacked by a DO instruction and is not unstacked by end-of-loop processing. See Section 8.6.4, “Nested Loops,” on page 8-22 for a discussion of nesting loops in software.

5.1.7 Hardware Stack

The hardware stack (HWS) is a 2-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 a write to the hardware stack would cause the stack limit to be exceeded, the write does not take place, and a non-maskable hardware-stack-overflow interrupt occurs. There is no interrupt on hardware stack underflow.

5.1.8 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 DSC 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 DSC 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, and instructions that directly reference the MR register. During processor reset, the interrupt mask bits of the mode register will be set, and the LF bit 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 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 the CGDB unless data limiting occurs when reading the A or B accumulators. During processor reset, all CCR bits are cleared. The standard definitions of the CCR bits are given in the following subsections, and more information about condition code bits is found in Section 3.6, "Condition Code Generation," on page 3-33. Refer to Appendix A, "Instruction Set Details," for 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 on page 5-7 and is also described in the following subsections.

SR
Status Register
Reset = $0300
Read/Write

Mode Register (MR)								Condition Code Register (CCR)							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	L	E	U	N	Z	V	C

LF — Loop Flag
I1,I0 — Interrupt Mask
SZ — Size
L — Limit
E — Extension
U — Unnormalized
N — Negative
Z — Zero
V — Overflow
C — Carry

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

AA0011

Figure 5-4. Status Register Format

5.1.8.1 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 zero, the carry or borrow is generated out of bit 35 of the result. If the CC bit in the OMR register is one, 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.

5.1.8.2 Overflow (V) — Bit 1

If the CC bit in the OMR register is zero 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 one 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.

5.1.8.3 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 Section 3.6, "Condition Code Generation," on page 3-33.

5.1.8.4 Negative (N) — Bit 3

If the CC bit in the OMR register is zero 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 one and if bit 31 of the result is set, the negative bit is set. Otherwise, this bit is cleared.

5.1.8.5 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 is cleared otherwise. The U bit is computed as follows: U = $\overline{\text{(Bit 31 XOR Bit 30)}}$.

If the U bit is cleared, then a positive fractional number, p, satisfies the following relation: $0.5 \leq p < 1.0$. A negative fractional number, n, it satisfies the following equation: $-1.0 \leq n < -0.5$.

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

5.1.8.6 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 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.

5.1.8.7 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; 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.

5.1.8.8 Size (SZ) — Bit 7

The size (SZ) bit (SR bit 7) is set when moving a 36-bit accumulator to data memory if bits 30 and 29 of the source accumulator are not the same — that is, if they are not both ones or zeros. This bit is latched, so it will remain set until the processor is reset or an instruction explicitly clears it.

By monitoring the SZ bit, it is possible to determine whether a value is growing to the point where it will be saturated or limited when moved to data memory. It is designed for use in the fast Fourier transform (FFT) algorithm, indicating that the next pass in the algorithm should scale its results before computation. This allows FFT data to be scaled only on passes where it is necessary instead of on each pass, which in turn helps guarantee maximum accuracy in an FFT calculation.

5.1.8.9 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 DSC 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 one to ensure future compatibility and compatibility with other family members. The interrupt mask bits are set during processor reset. See Table 5-1 on page 5-9 for interrupt mask bit definitions.

When disabling interrupts, I1 in the SR register is set to '1'. Interrupts will be disabled on the second cycle after update as shown in Example 5-1.

Example 5-1. Disabling Maskable Interrupts

```
; Disabling Maskable Interrupts
      BFSET        #$0200,SR          ; request to disable, 16-bit mask to set I1
                                      ; interrupts can still occur here
      NOP                             ; 1 cycle is required to disable interrupts
      NOP                             ; interrupts will not occur here
```

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

5.1.8.10 Reserved SR Bits — Bits 10–14

The reserved SR bits 10–14 are reserved for future expansion and will read as zero during DSC read operations. These bits should be written with zero for future compatibility.

5.1.8.11 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 Section 5.1.9.7, “Nested Looping Bit (NL) — Bit 15.” 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.

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.

5.1.9 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 instructions that directly reference the OMR. A DO loop will also affect the OMR, specifically 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 on page 5-10 and is described in the subsequent discussion.

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

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

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

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

AA0013

Figure 5-5. Operating Mode Register (OMR) Format

5.1.9.1 Operating Mode Bits (MB and MA) — Bits 1–0

The chip operating mode (MB and MA) bits (OMR bits 1 and 0) indicate the operating mode and memory maps of a DSC chip that has an external bus. Their initial values are typically established after reset by external mode select pins. After the chip leaves the reset state, MB and MA can be changed under program control. Consult the specific DSP56800 Family device manual for more detailed information about how these bits are established on reset and about their specific effect on operation.

Possible operating modes for a program RAM part are shown in Table 5-2.

Table 5-2. Program RAM Operating Modes

MB	MA	Chip Operating Mode	Reset Vector	Program Memory Configuration (consult specific 56800 Family device manual)
0	0	Bootstrap 0	BOOTROM P:$0000 (Boot from External Bus)	Internal P-RAM is write only
0	1	Bootstrap 1	BOOTROM P:$0000 (Boot from Peripheral)	Internal P-RAM is write only

Table 5-2. Program RAM Operating Modes (Continued)

MB	MA	Chip Operating Mode	Reset Vector	Program Memory Configuration (consult specific 56800 Family device manual)
1	0	Normal Expanded	External Pmem P:$E000	Internal Pmem enabled
1	1	Development	External Pmem P:$0000	Internal Pmem disabled

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 FLASH part because no bootstrapping operation is required for a FLASH part. An example of possible operating modes for a program FLASH part are shown in Table 5-3 on page 5-11.

Table 5-3. Program FLASH 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

The MB and MA bit values are typically established on reset from an external input. Once the chip leaves reset, they can be changed under software control. For more information about how they are configured on reset, consult the appropriate device's user's manual.

5.1.9.2 External X Memory Bit (EX) — Bit 3

The external X memory (EX) bit (OMR bit 3), when set, forces all primary data memory accesses 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. 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 Section 6.1, "Introduction to Moves and Parallel Moves," on page 6-1 for a description of the dual-read instructions.

5.1.9.3 Saturation (SA) — Bit 4

The Saturation (SA) bit enables automatic saturation on 32-bit arithmetic results, providing a user-enabled Saturation mode for DSC algorithms that do not recognize or cannot take advantage of the extension accumulator. When the SA bit is set, automatic saturation occurs at the output of the MAC unit for basic arithmetic operations such as multiplication, addition, and so on. The SA bit is cleared by processor reset. Automatic limiting as outlined in Section 3.4.1, "Data Limiter," on page 3-26 is not affected by the state of the SA bit.

Saturation is performed by a dedicated circuit inside the MAC unit. The saturation logic operates by checking 3 bits of the 36-bit result out of the MAC unit — EXT[3], EXT[0], and MSP[15]. When the SA bit is set, these 3 bits determine if saturation is performed on the MAC unit's output and whether to saturate to the maximum positive or negative value, as shown in Table 5-4.

Table 5-4. MAC Unit Outputs With Saturation Mode Enabled (SA = 1)

EXT[3]	EXT[0]	MSP[15]	Result Stored in Accumulator
0	0	0	(Unchanged)
0	0	1	$0 7FFF FFFF
0	1	0	$0 7FFF FFFF
0	1	1	$0 7FFF FFFF
1	0	0	$F 8000 0000
1	0	1	$F 8000 0000
1	1	0	$F 8000 0000
1	1	1	(Unchanged)

NOTE:

Saturation mode is *always* disabled during the execution of the following instructions: ASLL, ASRR, LSLL, LSRR, ASRAC, LSRAC, IMPY16, MPYSU, MACSU, AND, OR, EOR, NOT, LSL, LSR, ROL, and ROR. For these instructions, no saturation is performed at the output of the MAC unit.

5.1.9.4 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 Section 3.5, "Rounding," on page 3-30. This bit is cleared by processor reset.

5.1.9.5 Stop Delay Bit (SD) — Bit 6

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

5.1.9.6 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 this bit is set, the C, N, V, and Z condition codes are generated based on bit 31 of the data ALU result. When this bit is cleared, the C, N, V, and Z condition codes are generated based on bit 35 of the data ALU 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, and LS) can only be used when the condition codes are generated with

this bit set to one. 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 are discussed in more detail in Section 3.6, “Condition Code Generation,” on page 3-33.

5.1.9.7 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 (that is, 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, which is described further in Section 8.13, “Multitasking and the Hardware Stack,” on page 8-34. REP looping does not affect this bit.

It is important that the user never put the processor in the illegal combination specified in Table 5-5. 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 Section 5.1.8.11, “Loop Flag (LF) — Bit 15,” which discusses the LF bit in the SR.

Table 5-5. 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 (that is, 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.

5.1.9.8 Reserved OMR Bits — Bits 2, 7 and 9–14

The OMR bits 2, 7, and 9–14 are reserved. They will read as zero during DSC read operations and should be written as zero to ensure future compatibility.

5.2 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 Section 8.5, “Multiple Value Pushes,” on page 8-19) 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 (which pulls and discards the original SR). For returns from interrupt service routines that use the RTI instruction (the 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 push and pop 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 Section 8.6.4, “Nested Loops,” on page 8-22. 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 Section 8.5, “Multiple Value Pushes,” on page 8-19 and Section 8.8, “Parameters and Local Variables,” on page 8-28 for recommended techniques for using the software stack.

5.3 Program Looping

The DSC 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 for 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 Section 8.6, “Loops,” on page 8-20.

5.3.1 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 for the amount of times specified either with a 6-bit unsigned value or with the 13 least significant bits of a DSC 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 accesses 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.

For the case of REP looping with a register value, when the register contains the value zero, 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 zero is specified.

5.3.2 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 for the amount of times specified either with a 6-bit unsigned value or using the 13 least significant bits of a DSC 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 Section 8.6.1, “Large Loops (Count Greater Than 63),” on page 8-20.

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 (the last address of program loop). The contents of the loop counter are then tested for one. If the loop counter is not equal to one, 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 one, 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 zero, 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 Section 8.6.2, “Variable Count Loops,” on page 8-21 should be used. A DO loop with an immediate value of zero is not allowed.

5.3.3 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 not be nested within code. Instead, a software loop should be used for an outer loop instead of a second DO loop (see Section 8.6.4, “Nested Loops,” on page 8-22).

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

5.3.4 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 one). Two techniques for early termination of the DO loops are presented in Section 8.6.6, “Early Termination of a DO Loop,” on page 8-25.

Chapter 6
Instruction Set Introduction

As indicated by the programming model in Figure 6-3 on page 6-5, the DSC 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 Section 4.2, “Addressing Modes,” on page 4-6.) The instruction set has also been designed to allow for the efficient coding of DSC 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 DSC core, the concept of parallel moves, the DSC instruction formats, the DSC 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 it shows not only every instruction but also the operands and addressing modes allowed for each instruction.

6.1 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 DSC, 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_DSCcore_register>,<any_DSCcore_register>

MOVE   <any_DSCcore_register>,<X_Data_Memory>
MOVE   <any_DSCcore_register>,<On_chip_peripheral_register>
MOVE   <X_Data_Memory>,<any_DSCcore_register>
MOVE   <On_chip_peripheral_register>,<any_DSCcore_register>

MOVE   <immediate_value>,<any_DSCcore_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 the subset of DSC 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 DSC 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 add two numbers while reading or writing a value from memory in the same instruction.

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

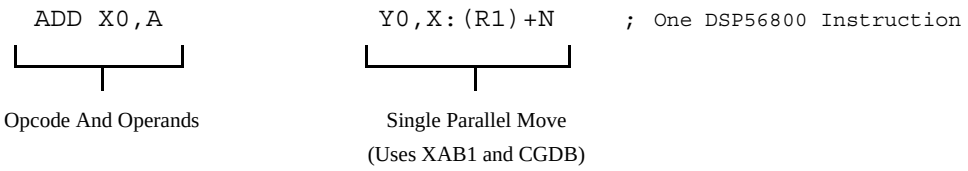


Figure 6-1. Single Parallel Move

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 uses one program word and executes in one instruction cycle.

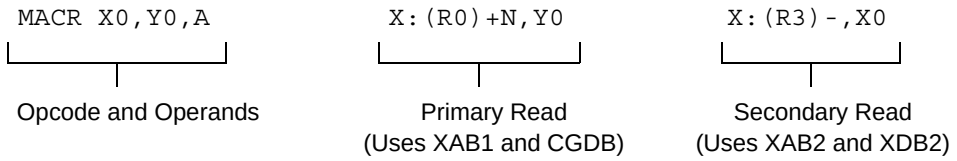


Figure 6-2. Dual Parallel Move

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 A accumulator, and the final result is stored in the A accumulator.
2. The contents of the X data memory location pointed to with the R0 register are moved into the Y0 register.
3. The contents of the 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 by one.

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 DSC core registers. These subsets include the registers and addressing modes most frequently found in high-performance numeric computation and DSC 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.

6.2 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 DSC offers parallel processing using the data ALU, AGU, program controller, and bit-manipulation unit. In the parallel move example, the DSC can perform a designated ALU operation (data ALU) and 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 (that is, operands are in data ALU registers) and allow the programmer to keep each parallel processing unit busy. Instructions that are memory oriented (for example, 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[4]	PDB Transfer[5]	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	$3C10				16-bit jump address

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 CGDB bus.
4. Specifies optional data transfers over the XDB2 bus.
5. Specifies optional data transfers over the PDB bus.

6.3 Programming Model

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

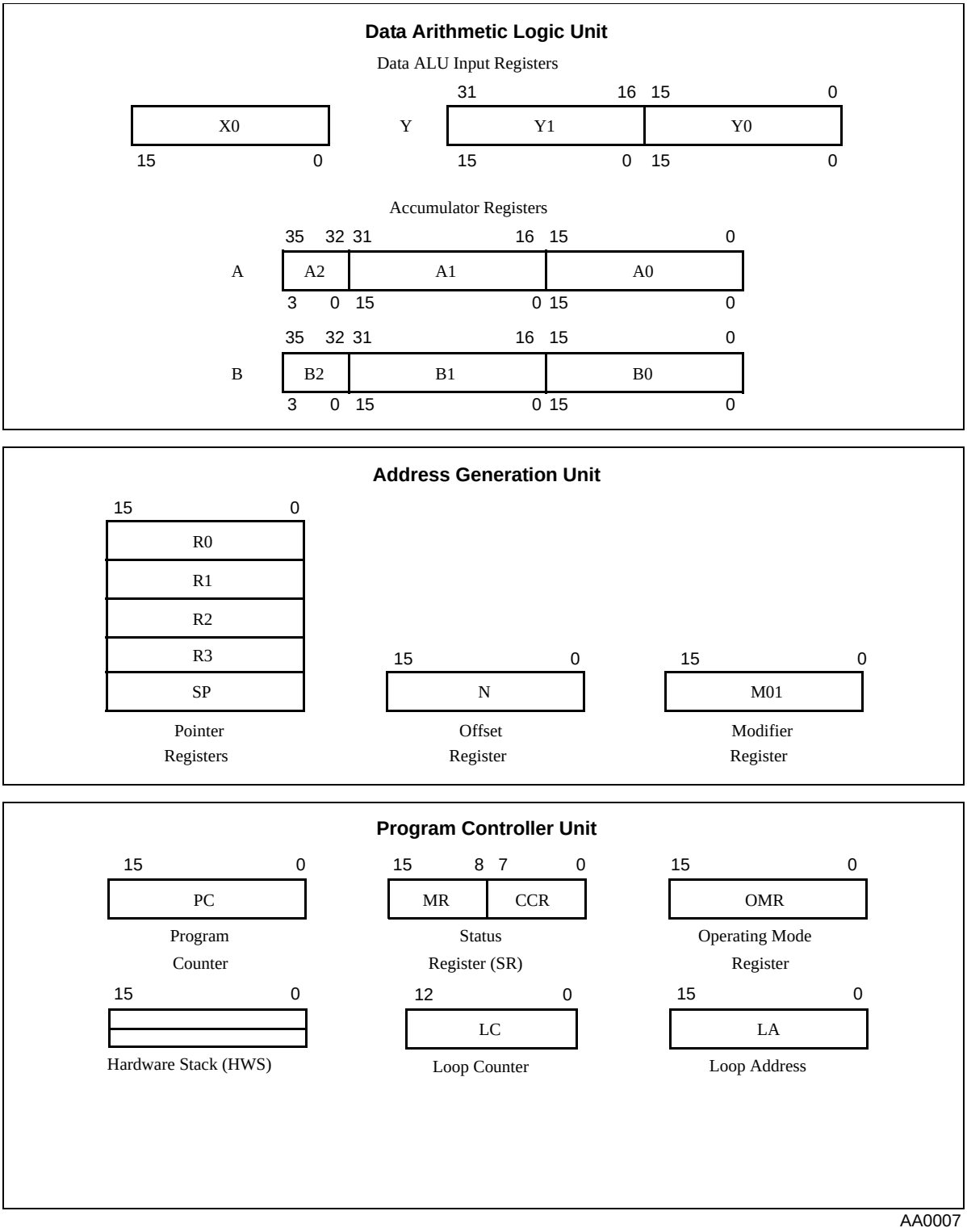


Figure 6-3. DSP56800 Core Programming Model

6.4 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 subsections. In addition, Section 6.6.4, “Instruction Summary Tables,” 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.”

6.4.1 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 are 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. See Section 6.4.2, “Logical Instructions.” 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]

Table 6-3. Arithmetic Instructions List (Continued)

Instruction	Description
CLR	Clear
CMP	Compare
DEC (or DECW)	Decrement upper word of accumulator
DIV	Divide iteration[1]
IMPY (or IMPY16)	Integer multiply[1]
INC (or INCW)	Increment upper word of accumulator
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
TST	Test a 36-bit accumulator
TSTW	Test a 16-bit register or memory location[1]

1. These instructions do not allow parallel data moves.

6.4.2 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. So 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
LSLL	Multi-bit logical shift left
LSRAC	Logical right shift with accumulate
LSR	Logical shift right
LSRR	Multi-bit logical shift right
NOT	Logical complement
OR	Logical inclusive OR
ROL	Rotate left
ROR	Rotate right

6.4.3 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 bits in a word
- Conditionally branching based on a test of bits within the upper or lower byte of a word

Bit-field instructions can operate on any X memory location, peripheral, or DSC 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 operations of the read-modify-write type. 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

Table 6-5. Bit-Field Instruction List (Continued)

Instruction	Description
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 (Rj, 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 Section 4.4, “Pipeline Dependencies,” on page 4-33).

See Section 8.1.1, “Jumps and Branches,” on page 8-2 for other instructions that can be synthesized.

6.4.4 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 Section 8.6.6, “Early Termination of a DO Loop,” on page 8-25.

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

6.4.5 Move Instructions

The move instructions move data over the various data buses: CGDB, IP-BUS (or PGDB), XDB2, and PDB. Move instructions do not affect the condition code register, except for 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 parallel moves that can be used simultaneously with many of the arithmetic instructions. The parallel moves are shown in Table 6-35 on

page 6-29 and Table 6-36 on page 6-30 and are discussed in detail in Section 6.1, “Introduction to Moves and Parallel Moves,” 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 Section 8.5, “Multiple Value Pushes,” on page 8-19, that can be used with the POP instruction alias presented in Section 6.5.5, “POP Alias,” on page 6-13.

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 (or MOVEC)	Move control register
MOVE (or MOVEI)	Move immediate data
MOVE (or MOVEM)	Move data to/from program memory
MOVE (or MOVEP)	Move data using peripheral short addressing
MOVE (or MOVES)	Move data using absolute short addressing

NOTE:

Due to instruction pipelining, if an AGU register (Rj, 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 Section 4.4, “Pipeline Dependencies,” on page 4-33.

6.4.6 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 DSC chip in a low-power state. See Section 8.1.1, “Jumps and Branches,” on page 8-2 and Section 8.11, “Jumps and JSRs Using a Register Value,” on page 8-33 for additional jump and branch instructions that 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

Table 6-8. Program Control Instruction List (Continued)

Instruction	Description
BRA	Branch
DEBUG	Enter debug mode
Jcc	Jump conditionally where cc represents condition mnemonic
JMP	Jump
JSR	Jump to subroutine
NOP	No operation
RTI	Return from interrupt
RTS	Return from subroutine
STOP	Stop processing (lowest power standby)
SWI	Software interrupt
WAIT	Wait for interrupt (low power standby)

6.5 Instruction Aliases

The DSP56800 assembler provides a number of additional useful instruction mnemonics that are actually aliases to other instructions. Each of these instructions is mapped to one of the core instructions and disassembles as such.

6.5.1 ANDC, EORC, ORC, and NOTC Aliases

The DSP56800 instruction set does not support logical operations using 16-bit immediate data. It is possible to achieve the same result, however, using the bit-manipulation instructions. To simplify implementing these operations, the DSP56800 assembler provides the following operations:

- ANDC — logically AND a 16-bit immediate value with a destination
- EORC — logically exclusive OR a 16-bit immediate value with a destination
- ORC — logically OR a 16-bit immediate value with a destination
- NOTC — logical one's-complement of a 16-bit destination

These operations are not new instructions, but aliases to existing bit-manipulation instructions. They are mapped as shown in Table 6-9.

Table 6-9. Aliases for Logical Instructions with Immediate Data

Desired Instruction	Operands	Remapped Instruction	Operands
ANDC	#xxxx,DST	BFCLR	#$\overline{\text{xxxx}}$,DST
ORC	#xxxx,DST	BFSET	#xxxx,DST

Table 6-9. Aliases for Logical Instructions with Immediate Data

Desired Instruction	Operands	Remapped Instruction	Operands
EORC	#xxxx,DST	BFCHG	#xxxx,DST
NOTC	DST	BFCHG	#$FFFF,DST

Note that for the ANDC instruction, a one's-complement of the mask value is used when remapping to the BFCLR instruction. For the NOTC instruction, all bits in the 16-bit mask are set to one.

In Example 6-2, an immediate value is logically ORed with a location in memory.

Example 6-2. Logical OR with a Data Memory Location

```
ORC    #$00FF,X:$0400      ; Set all bits of lower byte in X:$0400
```

The assembler translates this instruction into `BFSET #$00FF,X:$400`, which performs the same operation. If the assembled code is later disassembled, it will appear as a BFSET instruction.

6.5.2 LSLL Alias

Because the LSLL instruction operates identically to an arithmetic left shift, this instruction is actually assembled as an ASLL instruction. When the assembler encounters the LSLL mnemonic, an ASLL instruction is assembled. See Table 6-10.

Table 6-10. LSLL Instruction Alias

Operation	Operands	Comments
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). Use ASLL when left shifting is desired on one of the two accumulators.

6.5.3 ASL Alias

Because the ASL instruction operates similarly to a logical left shift when executed on the Y1, Y0, and X0 registers, this instruction is actually assembled as an LSL instruction. Note that while the result in the destination register will be the same as if an arithmetic shift had been performed, condition codes are calculated based on a logic shift and might differ from the expected result. See Table 6-11.

The ASL instruction is not aliased to LSL when the register specified is one of the accumulator registers.

Table 6-11. ASL Instruction Remapping

Operation	Operands	Comments
ASL	DD	Arithmetic left shift (assembled as LSL DD)

6.5.4 CLR Alias

Because CLR operates identically to a MOVE instruction with an immediate value of zero, a MOVE instruction is used to implement CLR when the specified register is a 16-bit register. When the assembler encounters the CLR mnemonic in a program, it assembles a `MOVE #0,<register>` instruction in its place. See Table 6-12.

NOTE:

This operation does not apply to the CLR instruction when it is performed on the A or B accumulators.

Table 6-12. Clear Instruction Alias

Operation	Destination	Comments
CLR	X0, Y1, Y0, A1, B1, R0–R3, N	Identical to `MOVE #0,<register>`; does *not* set condition codes

6.5.5 POP Alias

The POP instruction operates identically to a move from the stack with post-decrement. When the assembler encounters the POP instruction in a program, it assembles a `MOVE X:(SP)-,<register>` instruction in its place. If POP does not specify a destination register, it is assembled as `LEA (SP)-`.

Table 6-13. Move Word Instruction Alias — Data Memory

Operation	Source	Destination	Comments
POP		DDDDD	Pop a single stack location
		(None specified)	Simply decrements the SP; LEA (SP)-

6.6 DSP56800 Instruction Set Summary

This section presents the entire DSP56800 instruction set in tabular form. The tables provide a quick reference to the entire instruction set because they show not only the instructions themselves, but also the registers, addressing modes, cycle counts, and program words required for 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.

The summary, found in Section 6.6.4, “Instruction Summary Tables,” is based on logical groupings of instructions, listing the instructions alphabetically within each grouping. This summary also contains the number of program words required by the instruction as well as the number of cycles required for execution.

This section contains the following information:

- Usage of the instruction summary tables
- Addressing mode notation
- Register field notation
- The instruction summary tables

6.6.1 Register Field Notation

There are many different register fields used within the instruction summary tables. These will be grouped into sets that are more easily understood.

Table 6-14 shows the register set available for the most important move instructions. Sometimes the register field is broken into two different fields — one where the register is used as a source (src), and the other where it is used as a destination (dst). This is important because a different notation is used when an accumulator is being stored without saturation. Also see the register fields in Table 6-15, which are also used in move instructions as sources and destinations within the AGU.

In some cases, the notation used when specifying an accumulator determines whether or not saturation is enabled when the accumulator is being used as a source in a move or parallel move instruction. Refer to Section 3.4.1, “Data Limiter,” on page 3-26 and Section 3.2, “Accessing the Accumulator Registers,” on page 3-7 for information.

Table 6-14. Register Fields for General-Purpose Writes and Reads

Register Field	Registers in This Field	Comments
HHH	A, B, A1, B1 X0, Y0, Y1	Seven data ALU registers — two accumulators, two 16-bit MSP portions of the accumulators, and three 16-bit data registers
HHHH	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	Seven data ALU and five AGU registers
DDDDD	A, A2, A1, A0 B, B2, B1, B0 Y1, Y0, X0 R0, R1, R2, R3 N, SP M01 OMR, SR LA, LC HWS	All CPU registers

Table 6-15 shows the register set available for use as pointers in address-register-indirect addressing modes. This table also shows the notation used for AGU registers in AGU arithmetic operations.

Table 6-15. Address Generation Unit (AGU) Registers

Register Field	Registers in This Field	Comments
Rn	R0–R3 SP	Five AGU registers available as pointers for addressing and as sources and destinations for move instructions
Rj	R0, R1, R2, R3	Four pointer registers available as pointers for addressing
N	N	One index register available only for indexed addressing modes
M01	M01	One modifier register

Table 6-16 shows the register set available for use in data ALU arithmetic operations. The most common field used in this table is FDD.

Table 6-16. Data ALU Registers

Register Field	Registers in This Field	Comments
FDD	A, B X0, Y0, Y1	Five data ALU registers — two 36-bit accumulators and three 16-bit data registers accessible during data ALU operations Contains the contents of the F and DD register fields
F1DD	A1, B1 X0, Y0, Y1	Five data ALU registers — two 16-bit MSP portions of the accumulators and three 16-bit data registers accessible during data ALU operations
DD	X0, Y0, Y1	Three 16-bit data registers
F	A, B	Two 36-bit accumulators accessible during parallel move instructions and some data ALU operations
~F,F		~F,F refers to any of two valid accumulator combinations: A,B or B,A
F1	A1, B1	The 16-bit MSP portion of two accumulators accessible as source operands in parallel move instructions

6.6.2 Immediate Value Notation

Immediate values, including absolute and offset addresses, are utilized in the instruction set summary using the notation defined in Table 6-17. The <MASKx> notation is used in Bit Manipulation Instructions in Table 6-30 and Table 6-31. The <OFFSET7> and <ABS16> notations are used in change of flow and loop intructions in Table 6-32 and Table 6-33.

Table 6-17. Immediate Value Notation

Immediate Value Field	Description
<MASK8>	8-bit mask value
<MASK16>	16-bit mask value
<OFFSET7>	7-bit signed PC-relative offset
<ABS16>	16-bit absolute address

6.6.3 Using the Instruction Summary Tables

This section contains helpful information on using the summary tables. It contains some notation used within the tables.

The register field notation is found in Section 6.6.1, “Register Field Notation.”

Some additional notation to be considered is found in the instruction summary tables when allowed registers for multiplications are specified (Table 6-23 on page 6-20). In these tables, the following entry is found:

(±)Y0,X0,FDD

The notation (±) in this entry indicates that an optional + or - sign can be specified before the input register combination. If a - is specified, the multiplication result is negated. This allows each of the following examples to be valid DSP56800 instructions:

```
MAC    X0,Y0,A            ; A + X0*Y0 -> A
MAC    +X0,Y0,A           ; A + X0*Y0 -> A
MAC    -X0,Y0,A           ; A - (X0*Y0) -> A
```

As an example, Table 6-36 on page 6-30 shows all registers and addressing modes that are allowed when performing a dual read instruction, one of the DSP56800's parallel move instructions. The instructions shown in Example 6-3 are allowed.

Example 6-3. 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 6-4 is *not* allowed:

Example 6-4. Invalid Instruction

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

Consulting the information in Table 6-36 on page 6-30 shows that this instruction is *not* valid for each of the following reasons:

- The only operands accepted for ADD or SUB are X0,F, Y1,F, Y0,F, A,B, or B,A, 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 the first memory read.
- The post-decrement addressing mode is not available for the first memory read.
- The X0 register may not be a destination for the first memory read because it is not listed in the Destination 1 column.
- The post-update by N addressing mode is not allowed for the second memory read. The second memory read is always identified as the memory move that uses R3 in instructions with two memory moves. For the second memory read, only the post-increment and post-decrement addressing modes are allowed.
- The Y0 register may not be a destination for the second memory read because it is not listed in the Destination 2 column.

6.6.4 Instruction Summary Tables

A summary of the entire DSP56800 instruction set is presented in this section in tabular form. In these tables, Table 6-18 on page 6-18 through Table 6-36 on page 6-30, the instructions are broken into several different categories and then listed alphabetically.

The tables specify the operation, operands, and any relevant comments. There are separate fields for sources and destinations of move instructions. There are also two additional fields:

- C — Time required to execute the instruction
- W — Number of program words occupied by the instruction

Instruction execution times are measured in oscillator clock cycles. This should not be confused with instruction cycles, which comprise the timing granularity of the DSP56800 execution units. Each instruction cycle is equivalent to two oscillator clock cycles. The numbers given for instruction times assume that internal memory — or external memory that requires no wait states — is used.

All parallel move instructions are located in the last two tables in this section:

- Table 6-35 on page 6-29
- Table 6-36 on page 6-30

Table 6-18. Move Word Instructions

Operation	Source	Destination	C	W	Comments
MOVE or MOVEC	DDDDD	X:(Rn) X:(Rn)+ X:(Rn)-	2	1	Move signed 16-bit integer word from memory with optional post-update
	DDDDD	X:(Rn+N)	4	1	Address = Rn + N. Rn does not change.
	DDDDD	X:(Rn)+N	2	1	Post-update of Rn register
	HHHH	X:(R2+xx)	4	1	xx: offset ranging from 0 to 63
	DDDDD	X:(Rn+xxxx)	6	2	Signed 16-bit offset
	HHHH	X:(SP-xx)	4	1	Unsigned 6-bit offset
	DDDDD	X:xxxx	4	2	Unsigned 16-bit address
	X:(Rn) X:(Rn)+ X:(Rn)-	DDDDD	2	1	Move signed 16-bit integer word to memory with optional post-update
	X:(Rn+N)	DDDDD	4	1	Address = Rn + N. Rn does not change.
	X:(Rn)+N	DDDDD	2	1	Post-update of Rn register
	X:(R2+xx)	HHHH	4	1	xx: offset ranging from 0 to 63
	X:(Rn+xxxx)	DDDDD	6	2	Signed 16-bit offset
	X:(SP-xx)	HHHH	4	1	Unsigned 6-bit offset
	X:xxxx	DDDDD	4	2	Unsigned 16-bit address
POP		DDDDD	2	1	**ALIAS**, refer to Section 6.5.5, “POP Alias.” Implemented as: MOVE X:(SP)-,<register>
		(None specified)			**ALIAS**, refer to Section 6.5.5, “POP Alias.” Implemented as: LEA (SP)-
MOVE or MOVEP	X:pp or X:<<pp	HHHH	2	1	X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23
	HHHH	X:pp or X:<<pp			
MOVE or MOVES	X:aa or X:<aa	HHHH	2	1	X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22
	HHHH	X:aa or X:<aa			
MOVE	(parallel)		2	1	Refer to Table 6-36 on page 6-30.

Table 6-19. Immediate Move Instructions

Operation	Source	Destination	C	W	Comments
MOVE or MOVEI	#<-64,63>	HHHH	2	1	Signed 7-bit integer data (data is put in the lowest 7 bits of the word portion of any accumulator, upper 8 bits and extension reg are sign extended, LSP portion is set to "0")
	#xxxx	DDDDD	4	2	Signed 16-bit immediate data. When LC is the destination, use 13-bit values only.
		X:(R2+xx)	6	2	Signed 16-bit immediate data move.
		X:(SP-xx)	6	2	
		X:xxxx	6	3	
MOVE or MOVEP	#xxxx	X:pp or X:<<pp	4	2	Move 16-bit immediate data to the last 64 locations of X data memory-peripheral registers. X:<<pp represents a 6-bit absolute I/O address.
MOVE or MOVES	#xxxx	X:aa or X:<aa	4	2	Move 16-bit immediate date to a location within the first 64 words of X data memory. X:aa represents a 6-bit absolute address.

Table 6-20. Register-to-Register Move Instructions

Operation	Source	Destination	C	W	Comments
MOVE or MOVEC	DDDDD	DDDDD	2	1	Move signed word to register

Table 6-21. Move Word Instructions — Program Memory

Operation[1]	Source	Destination	C	W	Comments
MOVE or MOVEM	P:(Rj)+ P:(Rj)+N	HHHH	8	1	Read signed word from program memory
	HHHH	P:(Rj)+ P:(Rj)+N			Write word to program memory

1. These instructions are not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

Table 6-22. Conditional Register Transfer Instructions

Operation	Data ALU Transfer		AGU Transfer		C	W	Comments
	Source	Destination	Source	Destination			
Tcc	DD	F	(No transfer)		2	1	Conditionally transfer one register
	A	B	(No transfer)				
	B	A	(No transfer)				
	DD	F	R0	R1			Conditionally transfer one data ALU register and one AGU register
	A	B	R0	R1			
	B	A	R0	R1			
Note: The Tcc instruction does not allow the following condition codes: HI, LS, NN, and NR.							

Table 6-23. Data ALU Multiply Instructions

Operation	Operands	C	W	Comments
IMPY or IMPY16	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	Integer 16x16 multiply with 16-bit result When the destination is an accumulator F, the F0 portion is unchanged by the instruction. **Note**: Assembler also accepts first two operands when they are specified in opposite order.
MAC	(±)Y1,X0,FDD (±)Y0,X0,FDD (±)Y1,Y0,FDD (±)Y0,Y0,FDD (±)A1,Y0,FDD (±)B1,Y1,FDD	2	1	Fractional multiply accumulate; multiplication result optionally negated before accumulation. **Note**: Assembler also accepts first two operands when they are specified in opposite order.
	(parallel)			Refer to Table 6-35 & Table 6-36.
MACR	(±)Y1,X0,FDD (±)Y0,X0,FDD (±)Y1,Y0,FDD (±)Y0,Y0,FDD (±)A1,Y0,FDD (±)B1,Y1,FDD	2	1	Fractional MAC with round, multiplication result optionally negated before addition. **Note**: Assembler also accepts first two operands when they are specified in opposite order.
	(parallel)			Refer to Table 6-35 & Table 6-36.
MPY	(±)Y1,X0,FDD (±)Y0,X0,FDD (±)Y1,Y0,FDD (±)Y0,Y0,FDD (±)A1,Y0,FDD (±)B1,Y1,FDD	2	1	Fractional multiply where one operand is optionally negated before multiplication. **Note**: Assembler also accepts first two operands when they are specified in opposite order.
	(parallel)			Refer to Table 6-35 & Table 6-36.

Table 6-23. Data ALU Multiply Instructions (Continued)

Operation	Operands	C	W	Comments
MPYR	(±)Y1,X0,FDD (±)Y0,X0,FDD (±)Y1,Y0,FDD (±)Y0,Y0,FDD (±)A1,Y0,FDD (±)B1,Y1,FDD	2	1	Fractional multiply where one operand is optionally negated before multiplication. Result is rounded. **Note**: Assembler also accepts first two operands when they are specified in opposite order.
	(parallel)			Refer to Table 6-35 & Table 6-36.

Table 6-24. Data ALU Extended Precision Multiplication Instructions

Operation	Operands	C	W	Comments
MACSU	X0,Y1,FDD X0,Y0,FDD Y0,Y1,FDD Y0,Y0,FDD Y0,A1,FDD Y1,B1,FDD	2	1	Signed or unsigned 16x16 fractional MAC with 32-bit result. The first operand is treated as signed and the second as unsigned.
MPYSU	X0,Y1,FDD X0,Y0,FDD Y0,Y1,FDD Y0,Y0,FDD Y0,A1,FDD Y1,B1,FDD	2	1	Signed or unsigned 16x16 fractional multiply with 32-bit result. The first operand is treated as signed and the second as unsigned.

Table 6-25. Data ALU Arithmetic Instructions

Operation	Operands	C	W	Comments
ABS	F	2	1	Absolute value.
	(parallel)			Refer to Table 6-35 on page 6-29.
ADC	Y,F	2	1	Add with carry (sets C bit also).
ADD	DD,FDD	2	1	36-bit addition of two registers.
	F1,DD			
	~F,F			~F,F refers to any of two valid combinations: A,B or B,A
	Y,F			
	X:(SP-xx),FDD	6	1	Add memory word to register.
	X:aa,FDD	4	1	X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22
	X:xxxx,FDD	6	2	
	FDD,X:(SP-xx)	8	2	Add register to memory word, storing the result back to memory.
	FDD,X:xxxx	8	2	
	FDD,X:aa	6	2	
	#<0-31>,FDD	4	1	Add an immediate integer 0–31.
	#xxxx	6	2	Add a signed 16-bit immediate integer.
	(parallel)	2	1	Refer to Table 6-35 & Table 6-36.

Table 6-25. Data ALU Arithmetic Instructions (Continued)

Operation	Operands	C	W	Comments
CLR	F	2	1	Clear 36-bit accumulator and set condition codes.
	F1DD Rj N			**ALIAS**, refer to Section 6.5.4, "CLR Alias." Implemented as: MOVE #0,<register> (does *not* set condition codes)
	(parallel)			Refer to Table 6-35 on page 6-29.
CMP	DD,FDD	2	1	36-bit compare of two accumulators or data registers.
	F1,DD			
	~F,F			~F,F refers to any of two valid combinations: A,B or B,A
	X:(SP-xx),FDD	6	1	Compare memory word with 36-bit accumulator.
	X:aa,FDD	4	1	X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22
	X:xxxx,FDD	6	2	**Note**: Condition codes set based on 36-bit result
	#<0-31>,FDD	4	1	Compare accumulator with an immediate integer 0–31.
	#xxxx,FDD	6	2	Compare accumulator with signed 16-bit immediate integer.
	(parallel)	2	1	Refer to Table 6-35 on page 6-29,
DEC or DECW	FDD	2	1	Decrement word.
	X:(SP-xx)	8	1	Decrement word in memory using appropriate addressing mode.
	X:aa	6	1	
	X:xxxx	8	2	
	(parallel)	2	1	Refer to Table 6-35 on page 6-29.
DIV	DD,F	2	1	Divide iteration.
INC or INCW	FDD	2	1	Increment word.
	X:(SP-xx)	8	1	Increment word in memory using appropriate addressing mode.
	X:aa	6	1	
	X:xxxx	8	2	
	(parallel)	2	1	Refer to Table 6-35 on page 6-29.
NEG	F	2	1	Two's-complement negation.
	(parallel)			Refer to Table 6-35 on page 6-29.
RND	F	2	1	Round.
	(parallel)			Refer to Table 6-35 on page 6-29.
SBC	Y,F	2	1	Subtract with carry (set C bit also).
SUB	DD,FDD	2	1	36-bit subtract of two registers. 16-bit source registers are first sign extended internally and concatenated with 16 zero bits to form a 36-bit operand.
	F1,DD			
	~F,F			
	Y,F			~F,F refers to any of two valid combinations: A,B or B,A
	X:(SP-xx),FDD	6	1	Subtract memory word from register. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22
	X:aa,FDD	4	1	
	X:xxxx,FDD	6	2	
	#<0-31>,FDD	4	1	Subtract an immediate value 0–31.
	#xxxx,FDD	6	2	Subtract a signed 16-bit immediate integer.
	(parallel)	2	1	Refer to Table 6-35 & Table 6-36.

Table 6-25. Data ALU Arithmetic Instructions (Continued)

Operation	Operands	C	W	Comments
TFR	DD,F	2	1	Transfer register to register.
	~F,F			Transfer one accumulator to another (36-bits). ~F,F refers to any of two valid combinations: A,B or B,A
	(parallel)			Refer to Table 6-35 on page 6-29.
TST	F	2	1	Test 36-bit accumulator.
	(parallel)			Refer to Table 6-35 on page 6-29.
TSTW	DDDDD (except HWS)	2	1	Test 16-bit word in register. All registers allowed except HWS. Limiting can occur if an accumulator specified and the extension register is in use.
	X:(Rn)	2	1	Test a word in memory using appropriate addressing mode. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. Refer to Table 6-29 for another form of TSTW that tests and decrements an AGU register; (executed in the AGU unit).
	X:(Rn)+	2	1	
	X:(Rn)-	2	1	
	X:(Rn+N)	4	1	
	X:(Rn)+N	2	1	
	X:(Rn+xxxx)	6	2	
	X:(R2+xx)	4	1	
	X:(SP-xx)	4	1	
	X:aa	2	1	
	X:<<pp	2	1	
	X:xxxx	4	2	

Table 6-26. Data ALU Miscellaneous Instructions

Operation	Operands	C	W	Comments
NORM	R0,F	2	1	Normalization iteration instruction for normalizing the F accumulator

Table 6-27. Data ALU Logical Instructions

Operation	Operands	C	W	Comments
AND	DD,FDD	2	1	16-bit logical AND
	F1,DD			
EOR	DD,FDD	2	1	16-bit exclusive OR (XOR)
	F1,DD			
NOT	FDD	2	1	One's-complement (bit-wise negation)
OR	DD,FDD	2	1	16-bit logical OR
	F1,DD			

ALIAS: the ANDC, EORC, ORC, and NOTC can also be used to perform logical operations on registers and data memory locations. ANDC, EORC, and ORC allow logical operations with 16-bit immediate data. See Section 6.5.1, "ANDC, EORC, ORC, and NOTC Aliases," for additional information.

Table 6-28. Data ALU Shifting Instructions

Operation	Operands	C	W	Comments
ASL	F	2	1	Arithmetic shift left entire register by 1 bit
	DD			**ALIAS**, refer to Section 6.5.3, “ASL Alias.” Implemented as: LSL DD
	(parallel)			Refer to Table 6-35.
ASLL	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	Arithmetic shift left of the first operand by value specified in four LSBs of the second operand; places result in FDD
ASR	FDD	2	1	Arithmetic shift right entire register by 1 bit
	(parallel)			Refer to Table 6-35.
ASRR	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	Arithmetic shift right of the first operand by value specified in four LSBs of the second operand; places result in FDD
ASRAC	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	2	1	Arithmetic word shifting with accumulation
LSL	FDD	2	1	1-bit logical shift left of word
LSLL	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	**ALIAS**, refer to Section 6.5.2, “LSLL Alias.” Implemented as: ASLL <operands>
LSR	FDD	2	1	1-bit logical shift right of word
LSRR	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	Logical shift right of the first operand by value specified in four LSBs of the second operand; places result in FDD (when result is to an accumulator F, zero extends into F2)
LSRAC	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	2	1	Logical word shifting with accumulation
ROL	FDD	2	1	Rotate 16-bit register left by 1 bit through the carry bit
ROR	FDD	2	1	Rotate 16-bit register right by 1 bit through the carry bit

Table 6-29. AGU Arithmetic Instructions

Operation	Operands	C	W	Comments
LEA	(Rn)+	2	1	Increment the Rn pointer register
	(Rn)-	2	1	Decrement the Rn pointer register
	(Rn)+N	2	1	Add N index register to the Rn register and store the result in the Rn register
	(R2+xx)	2	1	Add a 6-bit unsigned immediate value to R2 and store in the R2 pointer
	(SP-xx)	2	1	Subtract a 6-bit unsigned immediate value from SP and store in the SP register
	(Rn+xxxx)	4	2	Add a 16-bit signed immediate value to the specified source register
TSTW	(Rn)-	2	1	Test and decrement AGU register. Refer to Table 6-25 for other forms of TSTW that are executed in the Data ALU.

Table 6-30. Bit-Manipulation Instructions

Operation	Operands	C	W	Comments
BFTSTH	#<MASK16>,DDDDD	4	2	BFTSTH tests all bits selected by the 16-bit immediate mask. If all selected bits are set, then the C bit is set. Otherwise it is cleared. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22 X:<<pp represents a 6-bit absolute I/O address.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	
BFTSTL	#<MASK16>,DDDDD	4	2	BFTSTL tests all bits selected by the 16-bit immediate mask. If all selected bits are clear, then the C bit is set. Otherwise it is cleared. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22 X:<<pp represents a 6-bit absolute I/O address.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Table 6-30. Bit-Manipulation Instructions (Continued)

Operation	Operands	C	W	Comments
BFCHG	#<MASK16>,DDDDD	4	2	BFCHG tests all bits selected by the 16-bit immediate mask. If all selected bits are set, then the C bit is set. Otherwise it is cleared. Then it inverts all selected bits. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22 X:<<pp represents a 6-bit absolute I/O address.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	
BFCLR	#<MASK16>,DDDDD	4	2	BFCLR tests all bits selected by the 16-bit immediate mask. If all selected bits are set, then the C bit is set. Otherwise it is cleared. Then it clears all selected bits. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22 X:<<pp represents a 6-bit absolute I/O address.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	
BFSET	#<MASK16>,DDDDD	4	2	BFSET tests all bits selected by the 16-bit immediate mask. If all selected bits are clear, then the C bit is set. Otherwise it is cleared. Then it sets all selected bits. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22 X:<<pp represents a 6-bit absolute I/O address.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Table 6-31. Branch on Bit-Manipulation Instructions

Operation	Operands	C[1]	W	Comments
BRCLR	#<MASK8>,DDDDD,<OFFSET7>	10/8	2	BRCLR tests all bits selected by the immediate mask. If all selected bits are clear, then the carry bit is set and a PC relative branch occurs. Otherwise it is cleared and no branch occurs. All registers in DDDDD are permitted except HWS. <MASK8> specifies a 16-bit immediate value where either the upper or lower 8 bits contains all zeros. <OFFSET7> specifies a 7-bit PC relative offset. X:aa represents a 6-bit absolute address. X:<<pp represents a 6-bit absolute I/O address.
	#<MASK8>,X:(R2+xx),<OFFSET7>	12/10	2	
	#<MASK8>,X:(SP-xx),<OFFSET7>	12/10	2	
	#<MASK8>,X:aa,<OFFSET7>	10/8	2	
	#<MASK8>,X:<<pp,<OFFSET7>	10/8	2	
	#<MASK8>,X:xxxx,<OFFSET7>	12/10	3	

Table 6-31. Branch on Bit-Manipulation Instructions (Continued)

Operation	Operands	C[1]	W	Comments
BRSET	#<MASK8>,DDDDD,<OFFSET7>	10/8	2	BRSET tests all bits selected by the immediate mask. If all selected bits are set, then the carry bit is set and a PC relative branch occurs. Otherwise it is cleared and no branch occurs. All registers in DDDDD are permitted except HWS. <MASK8> specifies a 16-bit immediate value where either the upper or lower 8 bits contains all zeros. <OFFSET7> specifies a 7-bit PC relative offset. X:aa represents a 6-bit absolute address. X:<<pp represents a 6-bit absolute I/O address.
	#<MASK8>,X:(R2+xx),<OFFSET7>	12/10	2	
	#<MASK8>,X:(SP-xx),<OFFSET7>	12/10	2	
	#<MASK8>,X:aa,<OFFSET7>	10/8	2	
	#<MASK8>,X:<<pp,<OFFSET7>	10/8	2	
	#<MASK8>,X:xxxx,<OFFSET7>	12/10	3	

1. First cycle count is if branch is taken (condition is true); second is if branch is not taken.

Table 6-32. Change of Flow Instructions

Operation	Operands	C[1]	W	Comments
Bcc	<OFFSET7>	6/4	1	7-bit signed PC relative offset
BRA	<OFFSET7>	6	1	7-bit signed PC relative offset
Jcc	<ABS16>	6/4	2	16-bit absolute address
JMP	<ABS16>	6	2	16-bit absolute address
JSR	<ABS16>	8	2	Push 16-bit return address and jump to 16-bit target address
RTI		10	1	Return from interrupt, restoring 16-bit PC and SR from the stack
RTS		10	1	Return from subroutine, restoring 16-bit PC from the stack

1. First cycle count is if branch is taken (condition is true); second is if branch is not taken.

Table 6-33. Looping Instructions

Operation	Operands	C	W	Comments
DO	#<1-63>,<ABS16>	6	2	Load LC register with unsigned value and start hardware DO loop with 6-bit immediate loop count. The last address is 16-bit absolute. Loop count = 0 not allowed by assembler.
	DDDDD,<ABS16>			Load LC register with unsigned value. If LC is not equal to zero, start hardware DO loop with 16-bit loop count in register. Otherwise, skip body of loop (adds three additional cycles). The last address is 16-bit absolute. Any register allowed except: SP, M01, SR, OMR, and HWS.
ENDDO		2	1	Remove one value from the hardware stack and update the NL and LF bits appropriately. **Note**: Does not branch to the end of the loop.

Table 6-33. Looping Instructions (Continued)

Operation	Operands	C	W	Comments
REP	#<0-63>	6	1	Hardware repeat of a one-word instruction with immediate loop count.
	DDDDD			Hardware repeat of a one-word instruction with loop count specified in register. Any register allowed except: SP, M01, SR, OMR, and HWS.

Table 6-34. Control Instructions

Operation	Operands	C	W	Comments
DEBUG		4	1	Generate a debug event.
ILLEGAL		4	1	Execute the illegal instruction exception. This instruction is made available so that code may be written to test and verify interrupt handlers for illegal instructions.
NOP		2	1	No operation.
STOP		n/a	1	Enter STOP low-power mode.
SWI		8	1	Execute the trap exception at the highest interrupt priority level, level 1 (non-maskable).
WAIT		n/a	1	Enter WAIT low-power mode.

Table 6-35. Data ALU Instructions — Single Parallel Move

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
MAC MPY MACR MPYR	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	X:(Rj)+ X:(Rj)+N	X0 Y1 Y0 A B A1 B1
ADD SUB CMP TFR	X0,F Y1,F Y0,F A,B B,A	X0 Y1 Y0 A B A1 B1	X:(Rj)+ X:(Rj)+N
ABS ASL ASR CLR RND TST INC or INCW DEC or DECW NEG	F (F = A or B)	(Rj = R0-R3)	

1. These instructions occupy only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Each instruction in Table 6-35 requires one program word and executes in one instruction cycle. The data type accessed by the single memory move in all single parallel move instructions is signed word.

The solid double line running down the center of the table indicates that the data ALU operation is independent from the parallel memory move. As a result, any valid operation can be combined with any valid memory move. Example 6-5 lists examples of valid single parallel move instructions.

Example 6-5. Examples of Single Parallel Moves

```
MAC    Y1,X0,A      X:(R0)+,X0
MAC    Y1,X0,A      X0,X:(R0)+
ASL    B            X:(R0)+,Y1
ASL    B            Y1,X:(R0)+
```

It is not permitted to perform `MAC A,B X:(R0)+,X0` because the MAC instruction requires three operands, as shown in Table 6-35. The operands are not independent of the operation performed. This is why a single line is used to separate the operation from the operands instead of a double line.

For the MAC, MPY, MACR, and MPYR instructions, the assembler accepts the two source operands in any order.

Table 6-36. Data ALU Instructions — Dual Parallel Read

Data ALU Operation[1]		First Memory Read		Second Memory Read	
Operation[2]	**Operands**	**Source 1**	**Destination 1**	**Source 2**	**Destination 2**
MAC MPY MACR MPYR	Y1,X0,F Y1,Y0,F Y0,X0,F (F = A or B)	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0
ADD SUB	X0,F Y1,F Y0,F (F = A or B)				
MOVE					

1. These parallel instructions are not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).
2. These instructions occupy only 1 program word and executes in 1 instruction cycle for every addressing mode.

NOTE:

The data types accessed by the two memory moves in all dual parallel read instructions are signed words.

6.7 The Instruction Pipeline

Instruction execution is pipelined to allow most instructions to execute at a rate of one instruction every two clock cycles. However, certain instructions 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 to be 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 DSC 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 DSC, one or more of the two memory sources may be accessed in one instruction cycle (that is, 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 Section 7.2.2, “Instruction Pipeline with Off-Chip Memory Accesses,” on page 7-3 for a discussion of off-chip memory accesses.

Chapter 7
Interrupts and the Processing States

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 in the following sections except the debug processing state, which is discussed in Section 9.3, “OnCE Port,” on page 9-4. In addition, special cases of interrupt pipelines are discussed at the end of the section. Section 8.10, “Interrupts,” on page 8-30 discusses software techniques for interrupt processing.

Table 7-1. Processing States

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

7.1 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 DSC is typically held in reset during the power-up process through assertion of the $\overline{\text{RESET}}$ pin, making this the first processing state entered by the DSC. 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, stop delay, rounding, and external X memory

The DSC remains in the reset state until the $\overline{\text{RESET}}$ pin is deasserted. When hardware deasserts the $\overline{\text{RESET}}$ pin, the following occur:

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

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

7.2 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, bit-field instructions, instructions with parallel moves, and so on. 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.

7.2.1 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 DSC 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 (six machine cycles) for the pipeline to fill and the first instruction to execute, an instruction usually executes on each instruction cycle thereafter (two machine cycles).

7.2.2 Instruction Pipeline with Off-Chip Memory Accesses

The three sets of internal on-chip address and data buses (XAB1/CGDB, XAB2/XDB2, PAB/PDB) allow for fast memory access when memories are being accessed on-chip. The DSC 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 allows 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 DSC 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 First Access**	**X Memory Second 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

Note: The 'mv' and 'mvm' cycle time values reflect the additional time required for all MOVE instructions and for MOVEM instructions, respectively.

7.2.3 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 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 this code segment.

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 ORC's execution time slot. The following code produces the expected results of reading the external FLASH:

```
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.

After an address register has been written by a MOVE instruction, one instruction cycle should be allowed before the new contents are available for use as an address register by another MOVE instruction. The proper instruction sequence follows:

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

Section 4.4, “Pipeline Dependencies,” on page 4-33 contains more details on interlocks caused during address generation.

7.3 Exception Processing State

The exception processing state is the state where the DSC core recognizes and processes interrupts that can be generated by conditions inside the DSC 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 DSC memory and a peripheral device or to begin execution of a DSC algorithm upon reception of a new sample. An interrupt can also be used to exit the DSC’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 7-7). During this fetch the PC is not updated. The instruction located at these two addresses in the interrupt vector table must always be 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 DSC core, from the DSC peripherals, or from external pins. The DSC core features a prioritized interrupt vector scheme with up to 64 vectors to provide faster interrupt servicing. The interrupt priority structure is discussed in Section 7.3.3, “Interrupt Priority Structure.”

7.3.1 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 DSC core, from a peripheral on the DSC chip, or from an instruction executed by the DSC core. Any hardware interrupt request from a pin is first synchronized with the DSC 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).
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 16-bit return address that will later be used to return to the interrupted code, as well as the condition code state. 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.

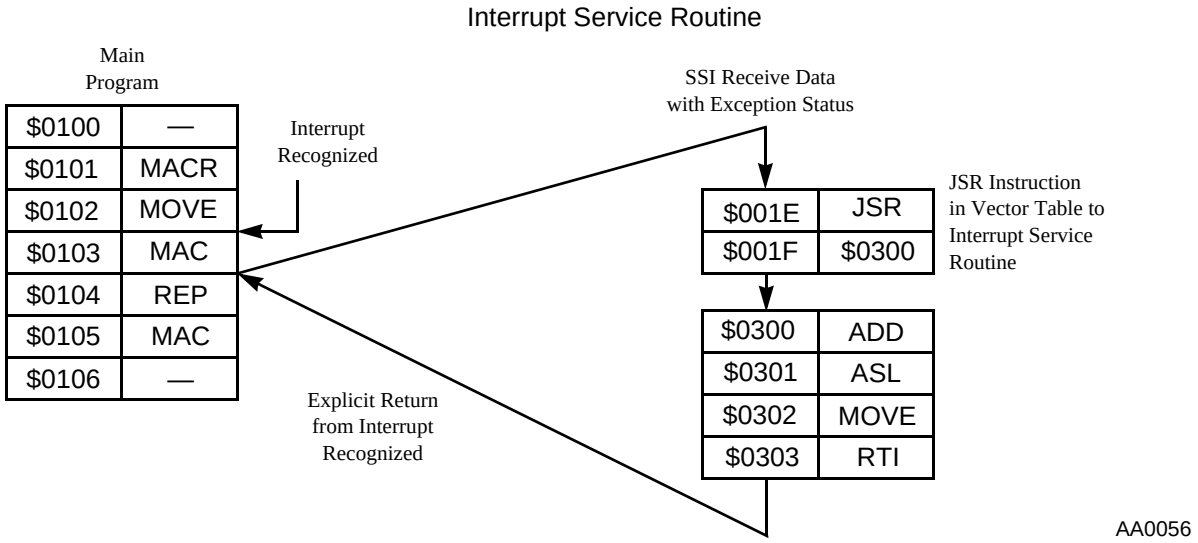


Figure 7-1. Interrupt Processing

Steps 1 through 3 listed on page page 7-5 require two additional instruction cycles, effectively making the interrupt pipeline five levels deep.

7.3.2 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 DSC core as well as all peripherals that can generate an interrupt. Table 7-4 lists the reset and interrupt vectors available on DSP56800-based DSC chips. The interrupt vectors used by on-chip peripherals, or by additional device-specific interrupts 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	$\overline{\text{IRQA}}$
$0012	0	$\overline{\text{IRQB}}$
$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)
...		
$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 different operating modes are discussed in Section 5.1.9.1, “Operating Mode Bits (MB and MA) — Bits 1–0,” on page 5-10.

7.3.3 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

7.3.4 Configuring Interrupt Sources

The interrupt unit in 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 DSC core. Each maskable interrupt source can individually be enabled or disabled as required by the application. The exact method for doing so is dependent on the particular DSP56800-based device, as some of the interrupt handling logic is implemented as an on-chip peripheral.

One example of how interrupts can be enabled and disabled, and their priority level established, is with an interrupt priority register (IPR).

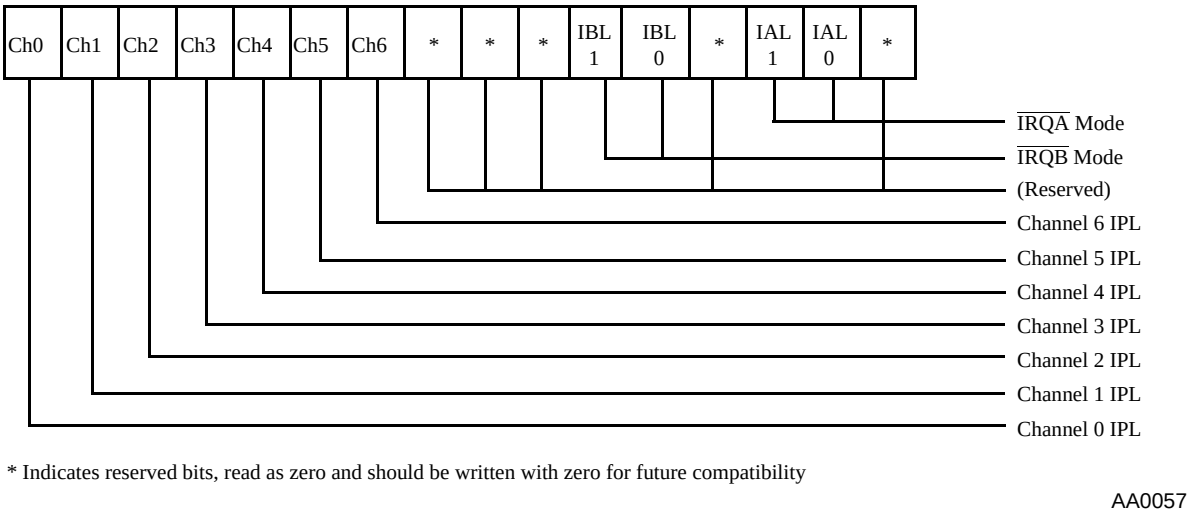


Figure 7-2. Example Interrupt Priority Register

In the example interrupt priority register (IPR), shown in Figure 7-2, the interrupt 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. The IPR also specifies the trigger mode of the external interrupt sources. Figure 7-3 shows how it might be programmed for different interrupts.

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. Example On-Chip Peripheral and IRQ Interrupt Programming

7.3.5 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 DSC 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 DSC 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.

7.3.5.1 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 DSC 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 DSC core. Edge-sensitive interrupts are automatically cleared when the interrupt is recognized and serviced by the DSC 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 $\overline{\text{IRQ}}$ 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 $\overline{\text{IRQA}}$ or $\overline{\text{IRQB}}$ 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 $\overline{\text{IRQ}}$ pin is available. In this case the interrupts are wire ORed onto a single $\overline{\text{IRQ}}$ 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.

7.3.5.2 DSC 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 Section 5.1.7, “Hardware Stack,” on page 5-6). 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 set up 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 Section 9.3, “OnCE Port,” on page 9-4 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.

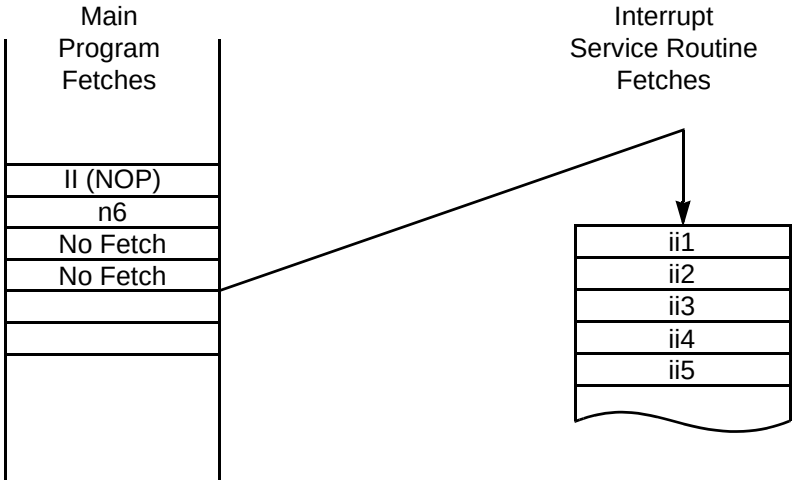
7.3.5.3 DSC Core Software Interrupt Sources

The two software interrupt sources are listed below:

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

An SWI is a non-maskable interrupt that is serviced immediately following the SWI instruction execution (that is, 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

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.

7.3.6 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 DSC 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	$\overline{\text{IRQA}}$ (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 DSC 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 requests will remain pending until their registers are read/written. Further, if the interrupt comes from an $\overline{\text{IRQ}}$ 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 first service the interrupt with the highest priority level. 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. The information in 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 discussed in Section 8.10.1, "Setting Interrupt Priorities in Software," on page 8-30.

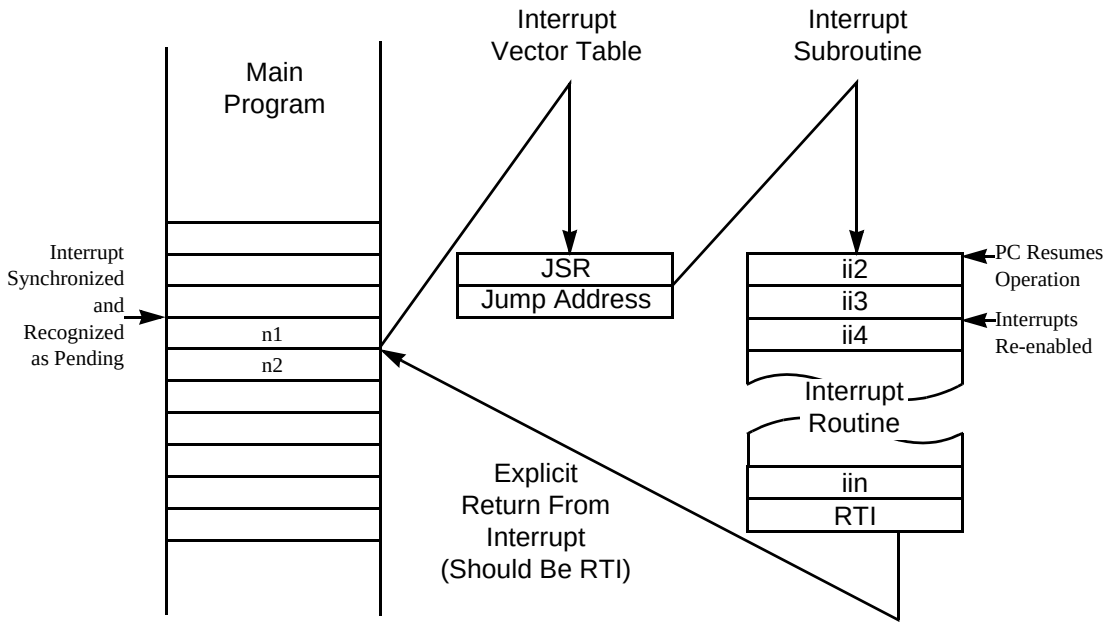
7.3.7 The Interrupt Pipeline

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 shows the interrupt service routine. The interrupt service routine must be told to return to the main program by executing an RTI instruction.

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 (that is, 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.



(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

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.

There 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. In this illustration, the first instruction (n7 in Figure 7-6) following an illegal instruction (n6) is lost as a consequence of the illegal opcode. The second instruction following an illegal instruction will be the next instruction that will be fetched, decoded, and executed normally (n8).

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

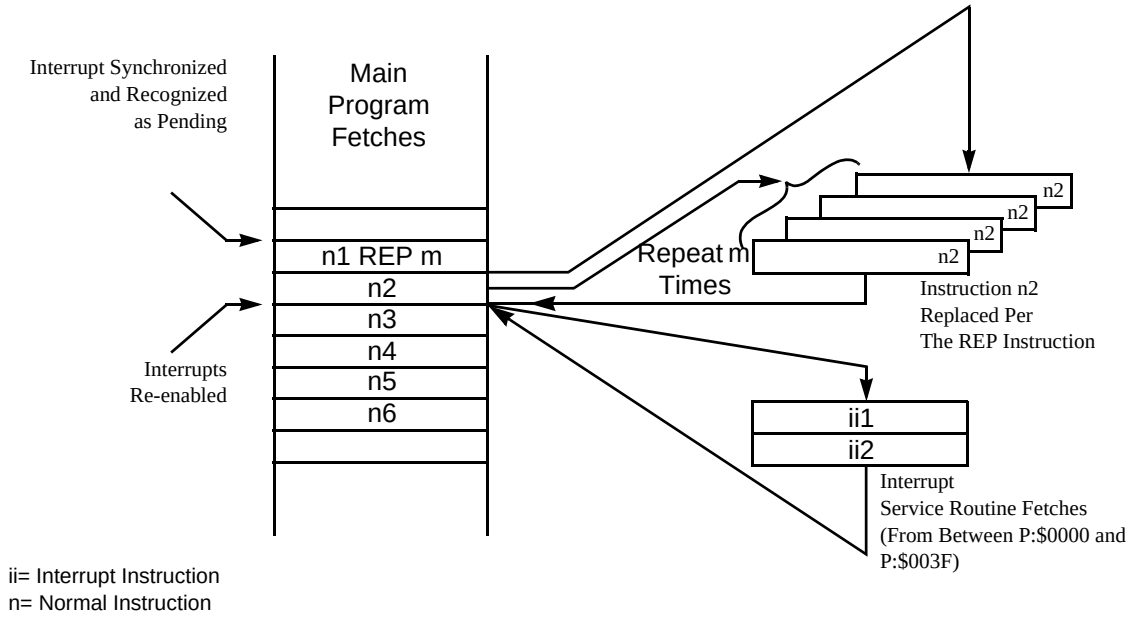
In DO loops, if the illegal instruction is in the loop address (LA) location and the instruction preceding it (that is, at LA-1) is being interrupted, the loop counter (LC) will be decremented as if the loop had reached the LA instruction. When the interrupt service ends and the instruction flow returns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

7.3.8 Interrupt Latency

Interrupt latency represents the time between when 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 of an instruction affects interrupt latency.

There are some special cases to consider. The SWI, STOP, and WAIT instructions are not interruptible. 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.



(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	n4	n5
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

7.4 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 (as defined by the I1 and I0 bits in the status register) releases the DSC 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 until the DSC 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.

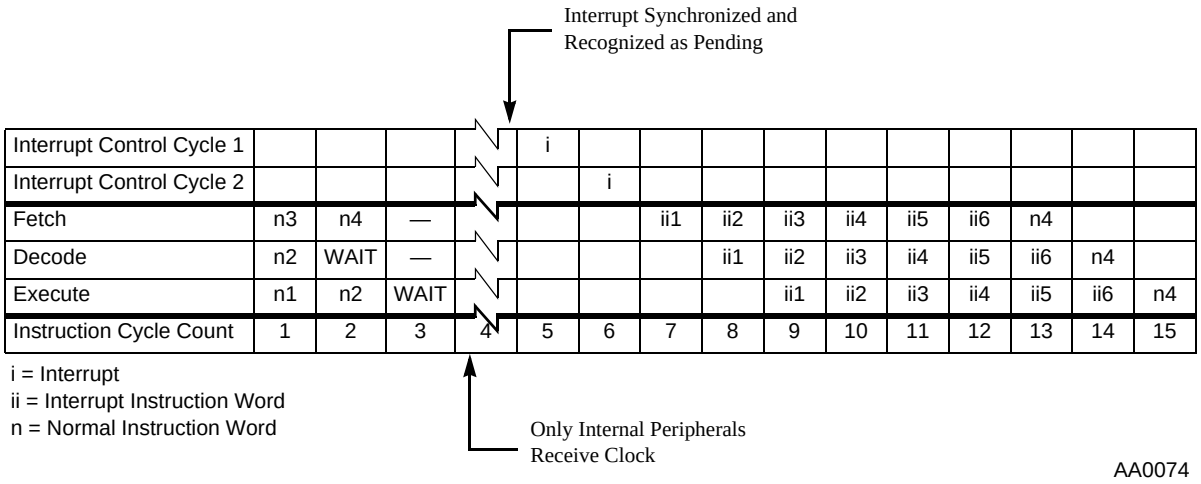


Figure 7-8. Wait Instruction Timing

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 in the preceding example. 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.

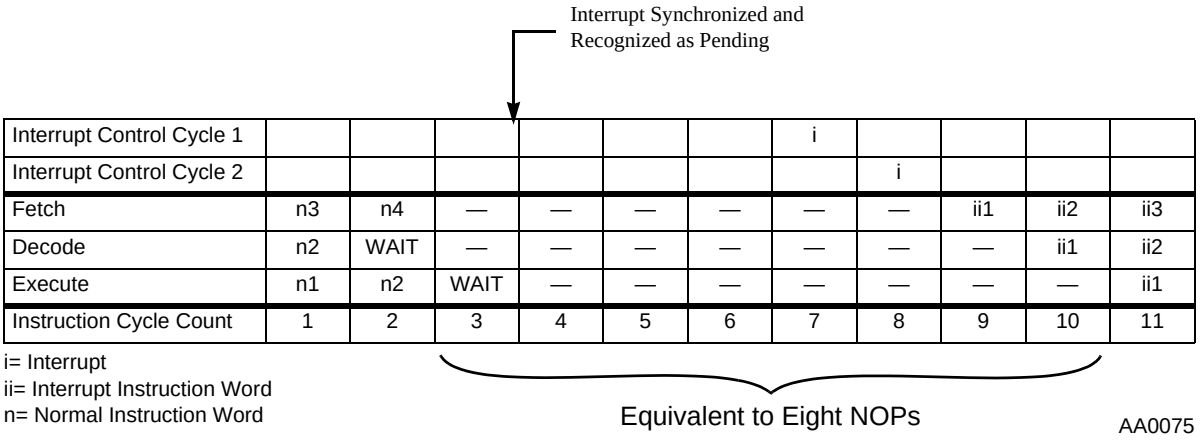


Figure 7-9. Simultaneous Wait Instruction and Interrupt

7.5 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 the processor is 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

Any 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 equal to either 16 (T) cycles or 131,072 T cycles as determined by the stop delay (SD) bit in the OMR. One T cycle is equal to one half of a clock cycle. For example, according to Table 6-34 on page 6-28, one NOP instruction executes in 2 clock cycles; therefore, one NOP instruction executes in 4T cycles, i.e., 1 instruction cycle equals 2 clock cycles and is equal to 4T cycles.

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.

As shown in Figure 7-10, the STOP instruction is fetched in stop cycle 1, 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.

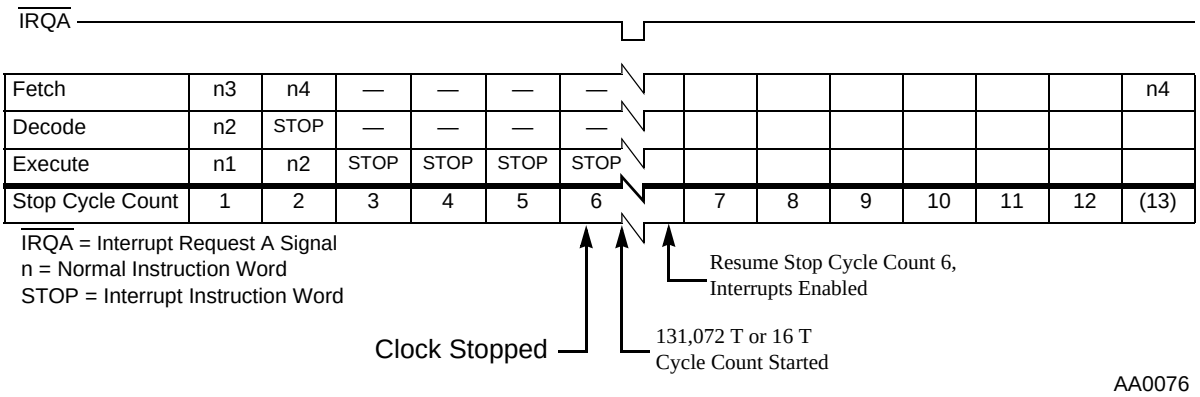


Figure 7-10. STOP Instruction Sequence

Figure 7-11 shows the system being restarted through asserting the $\overline{\text{IRQA}}$ signal. If the exit from the 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.

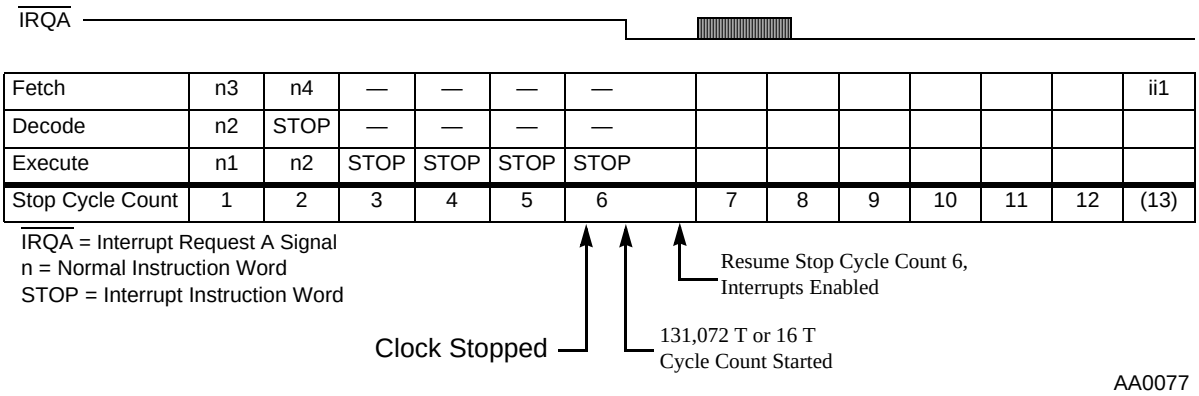


Figure 7-11. STOP Instruction Sequence

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 startup 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 7, 8, 9, and 10. (It takes 17 T 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 after fewer 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 refetch 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. Ensure that no stack error is pending.
3. Execute the STOP instruction and enter the stop state.
4. 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 $\overline{\text{IRQA}}$ vector will be fetched.

5. 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 a 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 or may not be disabled 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 3 T, totaling 35 T.

A stack error interrupt that is pending before the processor enters the stop state is not cleared and will remain pending. During the clock-stabilization delay in stop mode, any edge-triggered $\overline{\text{IRQ}}$ 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 Section 7.1, “Reset Processing State.” For example, the stabilization time recommended in *DSP56824 Technical Data* 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 DSC 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).

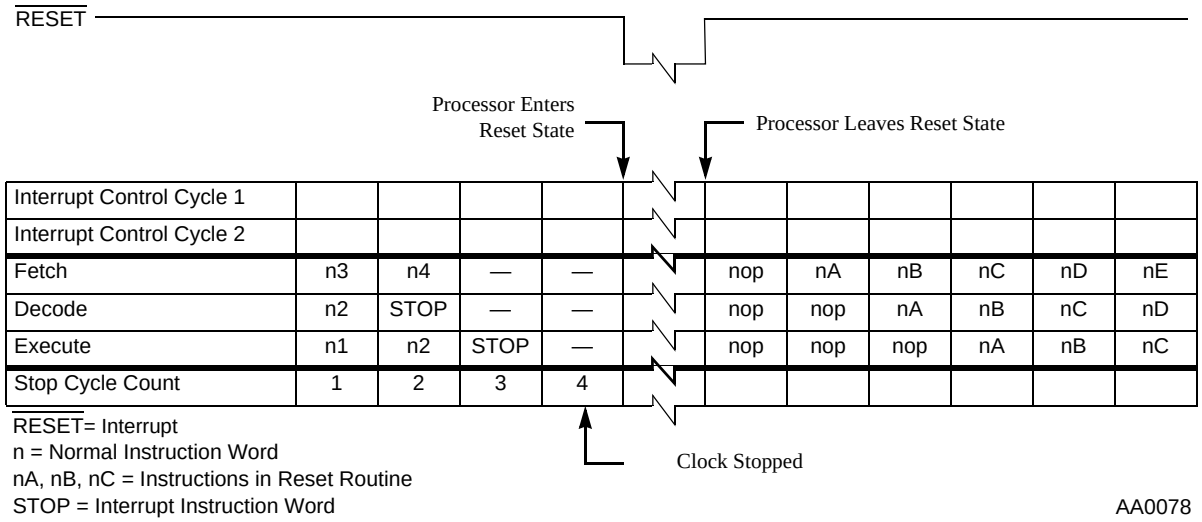


Figure 7-12. STOP Instruction Sequence Recovering with $\overline{\text{RESET}}$

7.6 Debug Processing State

The debug processing state is a state where the DSC 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 and the operation of the OnCE port is covered in more detail in Chapter 9, “JTAG and On-Chip Emulation (OnCE™).”

Chapter 8
Software Techniques

Different software techniques can be used to fully exploit the DSP56800 architecture's resources and enhance its features. For example, small sequences of DSP56800 instructions can emulate more powerful instructions. This chapter discusses how better 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

8.1 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 is set or clear
BR1SET, BR1CLR	Branches if at least one selected bit in bit field is set or clear
JR1SET, JR1CLR	Jumps if at least one selected bit in bit field is set or clear

Table 8-1. Operations Synthesized Using DSP56800 Instructions (Continued)

Operation	Description
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 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

8.1.1 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.

8.1.1.1 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 field; however, JRSET and JRCLR operations allow jumps to anywhere in the 64K-word program address space, and can specify a 16-bit mask. The following code shows that these two operations allow the same addressing modes as the BFTSTH and BFTSTL instructions.

Example 8-1. JRSET and JRCLR

```
; JRSET Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTH       #xxxx,X:<ea>        ; 16-bit mask allowed
      JCS          label               ; 16-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               ; 16-bit jump address allowed
```

8.1.1.2 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 following code shows that these two operations allow the same addressing modes as the BFTSTH and BFTSTL instructions.

Example 8-2. BR1SET and BR1CLR

```
; 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
```

8.1.1.3 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 64K-word program address space, and can specify a 16-bit mask. The following code shows that these two operations allow the same addressing modes as the BFTSTH and BFTSTL instructions.

Example 8-3. JR1SET and JR1CLR

```
; JR1SET Operation
; Emulated in 5 Icyc (4 Icyc if false), 4 Instruction Words
      BFTSTL       #xxxx,X:<ea>        ; 16-bit mask allowed
      JCC          label               ; 16-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               ; 16-bit jump address allowed
```

8.1.1.4 JVS, JVC, BVS, and BVC Operations

Although there is no instruction for jumping or branching on overflow, such an operation can be emulated as shown in the following code. Note that the carry bit will be destroyed by this operation since it receives the result of the BFTSTH instruction. The following code shows JVS and BVC.

Example 8-4. JVS, JVC, BVS 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              ; 16-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
```

8.1.1.5 Other Jumps and Branches on Condition Codes

Jumping and branching using some of the other condition codes (PL, MI, EC, ES, LMC, LMS) can be accomplished in the same manner as for overflow; see Section 8.1.1.4, “JVS, JVC, BVS, and BVC Operations.” Remember that this technique destroys the value in the carry bit. The following code shows JPL and BES.

Example 8-5. 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              ; 16-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.

8.1.2 Negation Operations

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.

8.1.2.1 NEGW Operation

The NEGW operation can be emulated as shown in the following code:

```
; 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 one 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 following code:

```
; 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 following code can be used for cases when 16-bit data is being processed and when 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            ; One's-complement of A1, A2 unchanged
      INCW         A            ; Increment to get two's-complement,
                                ; A2 may be incorrect
```

8.1.2.2 Negating the X0, Y0, or Y1 Data ALU registers

Although the NEG instruction is supported on accumulators only, NEG can be emulated to perform a negation of the data ALU's X0, Y0, or Y1 registers, as shown in the following code:

```
; NEG Operation
; Emulated at 2 Icyc
      NOT          Y0
      INCW         Y0
```

8.1.2.3 Negating an AGU register

It is possible to negate one of the AGU registers (Rn) without destroying any other register, as shown in the following code:

```
; NEG Operation
; Emulated at 3 Icyc
      NOTC         R0
      LEA          (R0)+
```

8.1.2.4 Negating a Memory Location

It is possible to negate a memory location, as shown in the following code:

```
; 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.

8.1.3 Register Exchanges

The XCHG operation can be emulated as shown in the following code:

```
; XCHG Operation
; Emulated at 4 Icyc
      PUSH         X0
      MOVE         A,X0
      POP          A
```

The macro instruction PUSH is described in Section 8.5, “Multiple Value Pushes.”

If a register is available, the exchange of any two registers can be emulated as shown in the following code:

```
; XCHG Operation
; Emulated at 3 Icyc
      MOVE         X0,N
      MOVE         A,X0
      MOVE         N,A
```

A faster exchange of any two registers can be emulated using one address register when N equals 0, as shown in the following code:

```
; XCHG Operation
; N register is 0, Emulated at 2 Icyc
      MOVE         A,X:(R0)
      TFR          X0,A   X:(R0)+N,X0
```

8.1.4 Minimum and Maximum Values

The MAX operation returns the maximum of two values; the MIN operation returns the minimum.

8.1.4.1 MAX Operation

The MAX operation can be emulated as shown in the following code:

```
; MAX Operation
      MAX          X0,A

;     ------ becomes ------

; MAX operation
; Emulated at 4 Icyc
      CMP          X0,A
      TLT          X0,A         ; (can also use TLE if desired)
```

8.1.4.2 MIN Operation

The MIN operation can be emulated as shown in the following code:

```
; MIN Operation
      MIN          Y0,A

;     ------ becomes ------

; MIN Operation
; Emulated at 4 Icyc
      CMP          Y0,A
      TGT          Y0,A         ; (can also use TGE if desired)
```

8.1.5 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 following code shows both versions:

```
; Sign-Extension Operation of 16-bit Accumulator Data
; Emulated in 1 Icyc, 1 Instruction Word
      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
```

8.1.6 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 following code:

```
; 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.

8.2 16- and 32-Bit Shift Operations

This technique presents many different methods for performing shift operations on the DSP56800 architecture. Different techniques offer different advantages. Some techniques require several registers, while 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.

8.2.1 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 following code may be adequate. These techniques may also be appropriate if there are no registers available for use in the shifting operation, since more than one register are 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.

8.2.2 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 following code:

```
; 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 following code:

```
; 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
```

8.2.3 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 following example 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 following code 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 following code should be preceded by a shift of 16 bits, as documented later in this section.

Similar code that follows 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 eight 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
```

8.2.4 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 an LSRAC instruction. This is shown in the following code:

```
; 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
```

8.2.5 Arithmetic Shifts by a Fixed Amount

Arithmetic shifts (left or right) by a fixed amount can be emulated with the ASRxx operations.

8.2.5.1 Right Shifts (ASR12–ASR20)

For arithmetic right shifts there is a faster way to shift an accumulator for large shift counts. The following code 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. If desired, it is possible to use this technique for bit shifts greater than 20, but it is not possible to use this technique for shifts of 11 or fewer bits 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 2-word, 2 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 2-word, 2 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 2-word, 2 Icyc macro)
      MOVE         A2,A
      POP          A0

; ASR15 Operation
; Emulated in 5 Icyc, 5 Instruction Words
      ASL          A
      PUSH         A1            ; (PUSH is a 2-word, 2 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
```

8.2.5.2 Left Shifts (ASL16–ASL19)

For arithmetic left shifts there is a faster way to shift an accumulator for large shift counts. The following code 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. If desired, it is possible to use this technique for bit shifts greater than 19, but it is not possible for shifts of 15 or fewer bits without losing information.

```
; ASL16 Operation
; Emulated in 4 Icyc, 4 Instruction Words
      PUSH          A1            ; (PUSH is a 2-word, 2 Icyc macro)
      MOVE          A0,A
      POP           A2

; ASL17 Operation
; Emulated in 5 Icyc, 5 Instruction Words
      ASL           A
      PUSH          A1            ; (PUSH is a 2-word, 2 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 2-word, 2 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 2-word, 2 Icyc macro)
      MOVE          A0,A
      POP           A2
```

8.3 Incrementing and Decrementing Operations

Almost any piece of data can be incremented or decremented. 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 (see Section 4.4, “Pipeline Dependencies,” on page 4-33). The TSTW instruction is not available for incrementing an AGU pointer or for decrementing the SP register.

```
; Different ways to increment on the DSC56800 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 entire 64K locations
                               ; of X data memory
      INCW        X:(SP-37)    ; on a value located on the stack

; Different ways to decrement on the DSC56800 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 or SP)
      DECW        X:$0         ; on anywhere within the first 64 locations
                               ; of X data memory
      DECW        X:$C200      ; on anywhere within the entire 64K locations
                               ; of X data memory
      DECW        X:(SP-37)    ; on a value located on the stack
```

The many different techniques available help 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.

8.4 Division

It is possible to perform fractional or integer division on the DSP56800 core. There are several questions to consider when implementing division on the DSC core:

- Are both operands always guaranteed to be positive?
- Are operands fractional or integer?
- Is only the quotient 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.

Once all these questions have been answered, it is possible to select the appropriate division algorithm. 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 four-quadrant division that generate both a quotient and a remainder. These take the largest 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, consult the following references (or others as required):

Theory and Application of Digital Signal Processing, Lawrence R. Rabiner and Bernard Gold (Prentice-Hall: 1975), pages 524–530.

Computer Architecture and Organization, John Hayes (McGraw-Hill: 1978), pages 190–199.

8.4.1 Positive Dividend and Divisor with Remainder

The algorithms in the following code 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 the 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)
;   Results: B1 = Remainder, B0 = Quotient, X0 (not changed)

      TSTW        B           ; TSTW always clears carry bit and more efficient
                              ;   than using BFCLR  #$0001,SR
      REP         #16         ; Carry bit must be clear for first DIV
      DIV         X0,B        ; Form positive quotient in B0

      TST         B           ; Verify if remainder needs correction
      BGE         Skip_Corr   ;   skip correction if not required
      ADD         X0,B        ; Correct the remainder stored in B1
Skip_Corr:                    ; At this point, positive quotient in B0
                              ;  and positive remainder in B1. End of Algorithm.


; Division of Integer, Positive Data (B1:B0 / X0). Registers used: Y1
;   Results: B1 = Remainder, B0 = Quotient, X0 (not changed)

      ASL         B           ; Shift of dividend required for integer division
      TSTW        B           ; TSTW always clears carry bit and more efficient
                              ;   than using BFCLR  #$0001,SR
      REP         #16         ; Carry bit must be clear for first DIV
      DIV         X0,B        ; Form positive quotient in B0

      MOVE        B0,Y1       ; Save quotient in Y1, (at this point, remainder
                              ;   is not yet correct).
      ADD         X0,B        ; Correct remainder in B1
      ASR         B           ; Required for correct integer remainder
      MOVE        Y1,B0       ; At this point, positive quotient in B0
                              ;  and positive remainder in B1. End of Algorithm.
```

NOTE:

The REP instruction is not interruptible; therefore, if user requires a interruptible sequence on the division, it is advisable to use the DO instruction or perform loop unrolling on the REP sequences.

8.4.2 Signed Dividend and Divisor with No Remainder

The algorithms in the following code provide fast ways to divide two 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 four-quadrant division because they allow any combination of positive or negative operands for the dividend and divisor. One algorithm is presented for the division of fractional numbers, and a second is presented for the division of integer numbers.

```
; 4-Quadrant Signed Fractional Division with no Remainder: (B1:B0 / X0)
; Generates signed quotient only, no remainder. Registers used: Y1
;   Results: B1 = unknown, B0 = Quotient, X0 (not modified), B (modified)

; Setup
      MOVE          B,Y1          ; Copy dividend to Y1
      ABS           B             ; Force the dividend positive
      TSTW          B             ; TSTW always clears carry bit and more efficient
                                  ;   than using BFCLR  #$0001,SR
; Division Operation
      REP           #16           ; Carry bit must be clear for first DIV
      DIV           X0,B          ; Form positive quotient in B0

; Compute Correct Quotient
      EOR           X0,Y1         ; If dividend and divisor both neg or both pos
      BGE           QDONE         ;   quotient already has correct sign,
      NEG           B             ; Else quotient is negative of computed result
QDONE:                            ; At this point, the correctly signed quotient
                                  ;   is at B0 but the remainder is not correct.
                                  ; End of Algorithm.
```

```
; 4-Quadrant Signed Integer Division with no Remainder: (B1:B0 / X0)
; Generates signed quotient only, no remainder. Registers used: Y1
;   Results: B1 = unknown, B0 = Quotient, X0 (not modified), B (modified)

; 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 the dividend positive
      TSTW          B             ; TSTW always clears carry bit and more efficient
                                  ;   than using BFCLR  #$0001,SR
; Division Operation
      REP           #16           ; Carry bit must be clear for first DIV
      DIV           X0,B          ; Form positive quotient in B0

; Compute Correct Quotient
      EOR           X0,Y1         ; If dividend and divisor both neg or both pos
      BGE           QDONE         ;   quotient already has correct sign,
      NEG           B             ; Else quotient is negative of computed result
QDONE:
                                  ; At this point, the correctly signed quotient
                                  ;   is at B0 but the remainder is not correct.
                                  ; End of Algorithm.
```

8.4.3 Signed Dividend and Divisor with Remainder

The algorithms in the following code 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.

```
; Four-Quadrant Signed Fractional Division with Remainder: (B1:B0 / X0)
; Generates signed quotient and remainder Registers used: Y1, Y0, A, N
;   Results: Y1 = Remainder, Y0 = Quotient, X0 (not modified), B (modified)

; Setup
      MOVE         B1,Y1        ; Save dividend sign to identify quadrant
      MOVE         B1,N         ; Save dividend sign - remainder must have same
      ABS          B            ; Force dividend positive
      TSTW         B            ; TSTW always clears carry bit and more efficient
                                ;   than using BFCLR  #$0001,SR
; Division Operation
      REP          #16          ; Carry bit must be clear for first DIV
      DIV          X0,B         ; Form positive quotient in B0

; Compute Correct Quotient
      TFR          B,A          ; Save to compute true remainder
      EOR          X0,Y1        ; If dividend and divisor both neg or both pos
      BGE          QDONE        ;   quotient already has correct sign
      NEG          B            ; Else quotient is negative of computed result
QDONE:
      MOVE         B0,Y0        ; Store true quotient
      TFR          A,B          ; Restore original remainder for final check
      MOVE         X0,A         ; Copy the original divisor
      ABS          A            ; Only absolute value of divisor needed
      ADD          B,A          ; Compute correct amplitude of remainder
      BRCLR        #$8000,N,RDONE ; Remainder must be same sign as the dividend
      MOVE         #0,A0        ; Prevent any unwanted carry
      NEG          A            ; Assign the same sign as the dividend
RDONE:
      MOVE         A1,Y1        ; At this point, signed quotient in Y0 and
                                ;   correct remainder in Y1. End of Algorithm.


; Verify Results: dividend = quotient * divisor + remainder
;   Remainder is fractional and corresponds to the lower 16 bits

; Setup
      MOVE         A2,A         ; Set up accumulator with sign of remainder
      MOVE         Y1,A0        ; Move fractional remainder to LSP
      MAC          Y0,X0,A      ; Multiply quotient with divisor and add remainder
                                ; Accumulator contains original dividend value
```

```
; Four-Quadrant Signed Integer Division with Remainder: (B1:B0 / X0)
; Generates signed quotient and remainder. Registers used: Y1, Y0, A, N
;   Results: Y1 = Remainder, Y0 = Quotient, X0 (not modified), B (modified)

; Setup
      ASL          B             ; Shift of dividend required for integer division
      MOVE         B1,Y1         ; Save dividend sign to identify quadrant
      MOVE         B1,N          ; Save dividend sign - remainder must have same
      ABS          B             ; Force dividend positive
      TSTW         B             ; TSTW always clears carry bit and more efficient
                                 ;   than using BFCLR  #$0001,SR
; Division Operation
      REP          #16           ; Carry bit must be clear for first DIV
      DIV          X0,B          ; Form positive quotient in B0

; Compute Correct Quotient
      TFR          B,A           ; Save to compute true remainder
      EOR          X0,Y1         ; If dividend and divisor both neg or both pos
      BGE          QDONE         ;   quotient already has correct sign
      NEG          B             ; Else quotient is negative of computed result
QDONE:
      MOVE         B0,Y0         ; Store true quotient
      TFR          A,B           ; Restore original remainder for final check
      MOVE         X0,A          ; Copy original divisor
      ABS          A             ; Only absolute value of divisor needed
      ADD          B,A           ; Compute correct amplitude of remainder
      BRCLR        #$8000,N,RDONE ; Remainder must be same sign as the dividend
      MOVE         #0,A0         ; Prevent any unwanted carry
      NEG          A             ; Assign the same sign as the dividend
RDONE:
      ASR          A             ; Shift required for correct integer remainder
      MOVE         A1,Y1         ; At this point, signed quotient in Y0 and
                                 ;   correct remainder in Y1. End of Algorithm.


; Verify Results: dividend = quotient * divisor + remainder
;   Remainder is integer and corresponds to the lower 16 bits
;   When using MAC instruction, shift at the end to correct value for integer

; Setup
      MOVE         A2,A          ; Set up accumulator with sign of remainder
      MOVE         Y1,A0         ; Move integer remainder to LSP
      ASL          A             ; Correct remainder for fractional operation
      MAC          Y0,X0,A       ; Multiply quotient with divisor and add remainder
      ASR          A             ; Correct result to obtain integer value
                                 ; Accumulator contains original dividend value
```

8.4.4 Algorithm Examples

This subsection provides examples of values calculated with the division algorithms in this section.

Example 8-6. Simple Fractional Division

A simple example of fractional division is the following case:

0.125 / 0.5 = 0.25 (remainder = 0)

For this case a positive fractional algorithm can be selected. The hex representation of the data used is:

(B1_B0 / X0) $1000_0000 / $4000

Using algorithm: Positive Fractional Division with Remainder, the following results are generated:

B0: quotient = $2000 = 0.25
B1: remainder = $0000 = 0.0

Example 8-7. Signed Fractional Division

Another example of fractional division is the following case:

-0.262871216 / 0.39035 = -6.7340 E-01 (all numbers are truncated to 9 and 5 decimal places from hex representation)

For this case a four-quadrant fractional algorithm can be selected. The hex representation of the data used is:

(B1_B0 / X0) $DE5A_3C69 / $31F7

Using algorithm: 4-Quadrant Signed Fractional Division with no Remainder, the following results are generated:

B0: quotient = $A9CE = -0.67340 (truncated to 5 decimal places)

Example 8-8. 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. The hex representation of the data used is:

(B1_B0 / X0) $0000_0040 / $0009

Using algorithm: Positive Integer Division with Remainder, the following results are generated:

B0: quotient = $0007 = 7
B1: remainder = $0001= 1

Example 8-9. Signed Integer Division

Another example of integer division is the following case:

-492,789,125 / -15,896 = 31,000 with remainder -13,125 (sign of remainder is same as dividend)

For this case a four-quadrant integer algorithm can be selected. The hex representation of the data used is:

(B1_B0 / X0) $E2A0_A27B / $C1E8

Using algorithm: 4-Quadrant Signed Integer Division, the following results are generated:

Y0: quotient = $7918 = 31,000
Y1: remainder = $CCBB = -13,125

When remainders are computed, the results can be easily checked by multiplying the quotient to the divisor and adding the remainder to the product as shown at the end of the 4-Quadrant algorithms with remainder. The final answer should be the same as the original dividend.

8.4.5 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:

$$-1.0 \leq \text{quotient} < +1.0 - 2^{-[N-1]}$$

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:

$$|\,\text{divisor}\,| \leq |\,\text{dividend}\,|$$

If this condition can be true when dividing fractional numbers, it must be prevented from occurring by first scaling the dividend.

For the 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 integer numbers are being divided, it must be guaranteed that the final result can fit into a signed, 16-bit integer value. Otherwise, to prevent this from occurring, it is first necessary to scale the numerator.

8.5 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 following code. (This instruction macro works quite well when pushing a single variable.)

```
; Expansion of the PUSH Instruction Macro
; Emulated in 2 Icyc, 2 Instruction Words

PUSH   MACRO  REG1                    ; Push REG1 in stack
       LEA    (SP)+                   ; Increment the SP (1 Icyc, 1 Word)
       MOVE   REG1,X:(SP)             ; Place value onto the stack
       ENDM                           ; (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:

```
; Faster technique for pushing multiple values onto the stack
; Finishes in 5 Icyc, 5 Instruction Words
PUSHN  MACRO                      ; Pushing X0,Y0,R0,R1
       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
       ENDM
```

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 in swapping two registers. There are two techniques for doing this, the first using an unused register and the second using a location on the stack. The second technique uses the PUSH instruction macro and works whenever there are no other registers available. The two techniques are shown in the following code:

```
; 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 preceding example.

8.6 Loops

The DSP56800 core contains a powerful and flexible hardware DO loop mechanism. It allows for loop counts of up to 8,192 iterations, 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.

8.6.1 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. When necessary, specifying an immediate value larger than 63 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 following code 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 register 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.

8.6.2 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 specified using a register. This allows a variable number of loop iterations from 1 to 2^k times (where k is the number of bits in the LC register, or 13). It is important to consider what takes place if this variable is zero or negative. Whenever a DO loop is executed and the loop count is zero, the loop will execute 2^{13} times. 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 following code.

```
; 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, the response is the same as for the DO loop and can be solved using the preceding technique presented for DO looping.

8.6.3 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 the hardware looping mechanism is used. 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 DSC 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 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 following code.

```
; Software Looping: Case 1
; Data ALU Register Used for Loop Count
      MOVE   #3,X0        ; Load loop count to execute the loop three times
LABEL:                    ; Enters loop at least once
;      (instructions)
      DECW   X0
      BGT    LABEL        ; Back to top-of-loop if positive and not 0
```

```
; Software Looping: Case 2
; AGU Register Used for Loop Count
      MOVE   #3-1,R0       ; Load loop count to execute the loop three times
LABEL:                     ; Enters loop at least once
;      (instructions)
      TSTW   (R0)-
      BGT    LABEL         ; Back to top-of-loop if positive and not 0


; Software Looping: Case 3
; Memory Location (one of first 64 XRAM locations) Used for Loop Count
      MOVE   #3,X:$7       ; Load loop count to execute the loop three times
LABEL:                     ; Enters loop at least once
;      (instructions)
      DECW   X:$7
      BGT    LABEL         ; Back to top-of-loop if positive and not 0
```

8.6.4 Nested Loops

This section gives recommendations for and a detailed discussion of nested loops.

8.6.4.1 Recommendations

For nested looping it is recommended that the innermost loop be a hardware DO loop when appropriate and that all outer loops be 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 is comparable to 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 ISRs 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 Section 8.12, “Freeing One Hardware Stack Location.”

The following code 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 preceding 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 following code:

```
; 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.

8.6.4.2 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. Rather, it is recommended that the hardware nesting of DO loops be used to provide more efficient interrupt processing, as described in Section 8.6.4.1, “Recommendations.”

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 following code shows the maximum nesting of hardware loops allowed on the DSP56800 processor:

```
; Hardware Nested Looping Example of the Maximum Depth Allowed
;
        DO      #3,OuterLoop        ; Beginning of outer loop
        PUSH    LC
        PUSH    LA
        DO      X0,InnerLoop        ; Beginning of inner loop
;       (instructions)
        REP     Y0                  ; Skips ASL if y0 = 0
        ASL     A
;       (instructions)
InnerLoop:                          ; End of inner loop
        POP     LA
        POP     LC
        NOP                         ; three instructions required after POP
        NOP                         ; three instructions required after POP
        NOP                         ; three instructions required after POP
OuterLoop:                          ; End of outer loop
```

The HWS's current depth can be determined by the NL and LF bits, as shown in Table 5-5, "Looping Status," on page 5-13. From these bits it is possible to determine whether there are no loops currently in progress, a single loop, or two nested loops.

For nested DO loops, it is required that there be at least three instructions after the POP of the LA and LC registers and before the label of any outer loop. This requirement shows up in the preceding example as three NOPs but can be fulfilled by 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 Section 8.13, "Multitasking and the Hardware Stack."

8.6.4.3 Comparison of Outer Looping Techniques

A comparison of the execution overhead and extra code size of software and hardware outer loops shows that for loop nesting, 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	Number of Icyc to Set Up Loop	Additional Number of Icyc Executed Each Loop	Total Number of Instruction 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.

8.6.5 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 Section 8.6.4, “Nested Loops,” 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 following code:

```
; 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. Also note that this overhead is 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 Section 8.12, “Freeing One Hardware Stack Location.”

8.6.6 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 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 a 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 any more instructions in the loop. In this case it is recommended to use the technique shown in the following code:

```
;
;       ------ Technique 1 ------
;

        PUSH   LC           ; Save outer loop registers if nested loop
        PUSH   LA
        DO     #LoopCnt,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



;
;       ------ Technique 2 ------
;
        PUSH   LC           ; Save outer loop registers if nested loop
        PUSH   LA
        DO     #LoopCnt,LABEL
;       (instructions)
        Bcc    OVER         ; 3 Icyc for each iteration
        ENDDO               ; 6 Icyc if loop terminates when false
        BRA    LABEL
OVER:
        (instructions)
LABEL:
        POP    LA           ; Restore outer loop registers if nested loop
        POP    LC
```

8.7 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 nth 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.

8.7.1 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. Arrays that are small in size 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.

8.7.2 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 the first two addressing modes — X:(Rn+xxxx) and 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 fewer instruction cycle and uses one fewer 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.

8.7.3 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 Section 8.8, “Parameters and Local Variables.”

8.7.4 Local Array with a Variable

This type of array indexing is done with the X:(Rj+N) or X:(SP+N) addressing mode. It is similar to the technique described in Section 8.7.3, “Local Array with a Constant,” 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.

8.7.5 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.

8.8 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 and 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 following example (which 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      ; First data variable to be passed to subroutine
       MOVE   X0,X:(SP)+    ; Push onto stack
       MOVE   X:$47,X0      ; Second 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 second data variable
       MOVE   X:(R0),X0     ; Get data pointed to by pointer variable
       ADD    X0,B
       MOVE   B,X:(SP-8)    ; Store sum in first 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 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 Section 8.5, "Multiple Value Pushes." Then the JSR instruction is executed, which pushes the return address and the SR onto the stack. Upon being entered, the subroutine first allocates space for local variables by updating the SP. Then, both passed parameters and local variables can be accessed with the stack addressing modes.

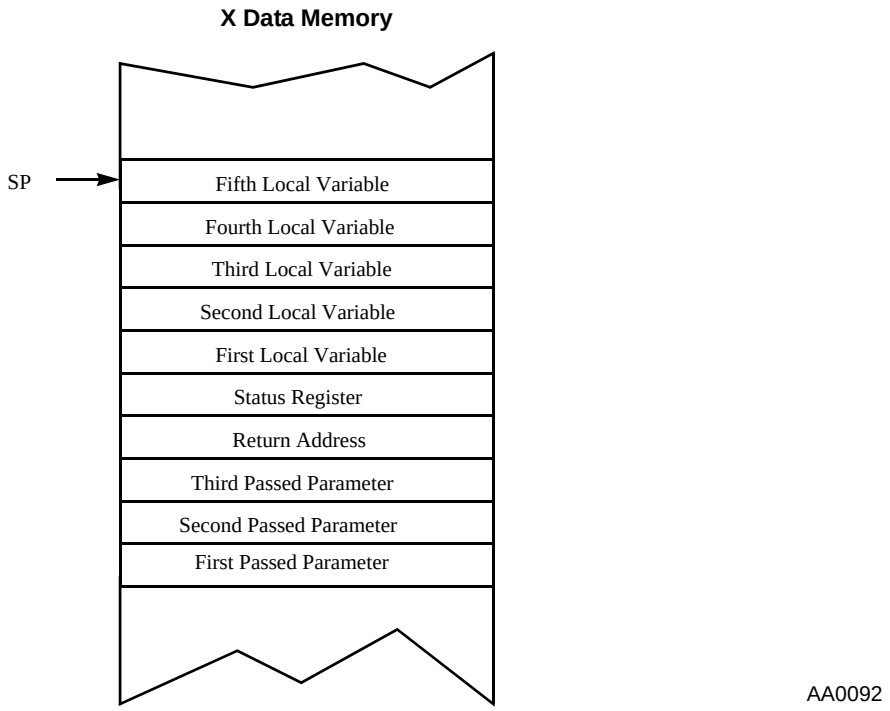


Figure 8-1. Example of a DSP56800 Stack Frame

8.9 Time-Critical DO Loops

Often, a program spends most of its time in time-critical loops. For the efficient execution of these loops, it is important to have an adequate number of registers. 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 following code.

```
        MOVE   #$1234,R3          ; Contents of this register not
                                  ; required in tight loop
        MOVE   #$05AA,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 preceding example there are four PUSH instruction macros in a row. For more efficient and compact code, use the technique outlined in Section 8.5, “Multiple Value Pushes.” 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.

8.10 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, within 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 gives the user the capability to set up interrupt priorities within software.

When programming interrupts, it is necessary to correctly set up the following tasks:

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.

8.10.1 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, 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.

8.10.1.1 High Priority or a Small Number of Instructions

During ISRs that are short, it is recommended that level 0 interrupts remain disabled. Since the routines 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.

```
; DSP56800 core (Interrupts Remain Masked, 9 Overhead Cycles)
      JSR    ISR1                ; located in interrupt vector table


; Interrupt Service Routine for Short ISR
;
ISR1:
;      (interrupt code)
      RTI
```

8.10.1.2 Many Instructions of Equal Priority

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.

```
; DSP56800 core (Interrupts Remain Masked, 11 Overhead Cycles)
      JSR    ISR2                ; located in interrupt vector table


; Interrupt Service Routine for Long ISR
;
ISR2:
      BFCLR  #$0200,SR           ; re-enable interrupts with new mask
;      (interrupt code)
      RTI
```

8.10.1.3 Many Instructions and 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.

```
; DSP56800 core
      JSR    ISR3          ; Instr located in Interrupt Vector Table


; Generic ISR - DSP56800 core (20 Overhead Cycles including JSR)
ISR3:                      ; 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   #NEW_MASK,X:IPR ; Load NEW_MASK - define which can preempt 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
```

8.10.2 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 outlined in Section 8.12, "Freeing One Hardware Stack Location." Alternatively, if it can be guaranteed that the main program will never use more than one DO loop at a time (that is, 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.

8.10.3 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. Usually, 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   #TASK_NUMBER,N      ; Task 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
```

8.11 Jumps and JSRs Using a Register Value

Sometimes it is necessary to perform a jump or a 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 DSC core.

```
; JMP <register> Operation
; 8 Icyc
      LEA    (SP)+        ; Update SP to point to unused location

; Note: Can use any core register in <register>, e.g. MOVE #LABEL,X0
      MOVE   <register>,X:(SP)+  ; Push address of target code location
      MOVE   SR,X:(SP)    ; Push SR onto stack last
      RTS                 ; Will return to address specified in <register> and
                          ;   correct the SP register to its original value



; Jcc <register> Operation
;   Examples: JEQ, JLE, JNN will use BNE, BGT and BNR respectively as 1st instr
;   (Use Bcc instruction whenever possible since it is a single word instruction)
; 10 Icyc (3 Icyc if condition false)
; To execute a BEQ <register>
      BNE    OVER         ; Use condition exactly opposite the desired cc

      LEA    (SP)+        ; Update SP to point to unused location
      MOVE   <register>,X:(SP)+ ; Push address of target code location
      MOVE   SR,X:(SP)    ; Push SR onto stack last
      RTS                 ; Will do a return to the desired target code location
                          ;   and correct the SP register to its original value

OVER:                     ; Start of code segment if BEQ fails (BNE succeeds!)
;      (instructions)



; JSR <register> Operation - destroys one register, N
; 11 Icyc
      MOVE   #NEXT_SEGMENT,N     ; P address of NEXT_SEGMENT

      LEA    (SP)+        ; Update SP to point to unused location
      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 last
      RTS                 ; Go to address in top two values on stack and
                          ;   correct the SP register to its original value

NEXT_SEGMENT:             ; Segment of code executed after returning from
                          ;    P:<register>
;      (instructions)
```

8.12 Freeing One Hardware Stack Location

There are certain cases where a section of code should use DO looping, but it is not 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 following code 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 Section 8.6.4, “Nested Loops,” to reduce the overhead needed for freeing up one HWS location. This greatly simplifies the setup code required when entering and exiting the ISR.

8.13 Multitasking and the Hardware Stack

For multitasking, 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.

8.13.1 Saving the Hardware Stack

An example of reading the entire contents of the HWS to X memory is shown in the following code:

```
; Save HWS
; 4 Icyc, 4 words
      MOVE   SR,X:(R2)+    ; Read HWS pointer's MSB (LF) and
                           ;   save to memory
      MOVE   HWS,X:(R2)+   ; Read first 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 second stack location and
                           ;   save in X memory
```

8.13.2 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 follows:

```
; Restore HWS, 10 words, 14 Icyc worst case
; Assumes R2 points to "stored" HWS
; Destroys R2 register

      MOVE   HWS,LA        ; First read of HWS ensures NL bit is cleared
      MOVE   HWS,LA        ; Second 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:
```

Chapter 9
JTAG and On-Chip Emulation (OnCE™)

The DSP56800 family includes extensive integrated test and debug support. Two modules, the On-Chip Emulation (OnCE) module and the test access port (TAP, commonly called the JTAG port) provide board- and chip-level testing and software debugging capability. Both are accessed through a common JTAG/OnCE interface. Using these modules allows the user to insert the DSC chip into a target system while retaining debug control. This capability is especially important for devices without an external bus, since 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 Freescale-designed module used to debug application software used with the chip. The port is a separate on-chip block that allows non-intrusive interaction with the DSC and is accessible through the pins of the JTAG interface. The OnCE port makes it possible to examine contents of registers, memory, or on-chip peripherals in a special debug environment. No user-accessible resources need be sacrificed to perform debugging operations.

The JTAG port conforms to the *IEEE Standard Test Access Port and Boundary-Scan Architecture* specification (IEEE 1149.1a-1993) as defined by the Joint Test Action Group (JTAG). The JTAG module uses a boundary scan technique to test the interconnections between integrated circuits after they are assembled onto a printed circuit board. Using a boundary scan allows a tester to observe and control signal levels at each component pin through a special register coupled to each pin, called a boundary scan cell. This is important for testing continuity and determining if pins are stuck at a one or zero level.

This chapter presents an overview of the capabilities of the JTAG and OnCE modules. Since their operation is highly dependent upon the architecture of a specific DSP56800 device, the exact implementation is necessarily device dependent. For more complete information on interfacing, the debug and test commands available, and other implementation details, consult the appropriate device's user's manual.

9.1 Combined JTAG and OnCE Interface

The JTAG and OnCE modules are tightly coupled. The JTAG port provides the interface for both modules and handles communications with host development and test systems. Figure 9-1 on page 9-2 shows a block diagram of the JTAG/OnCE modules and external host interface.

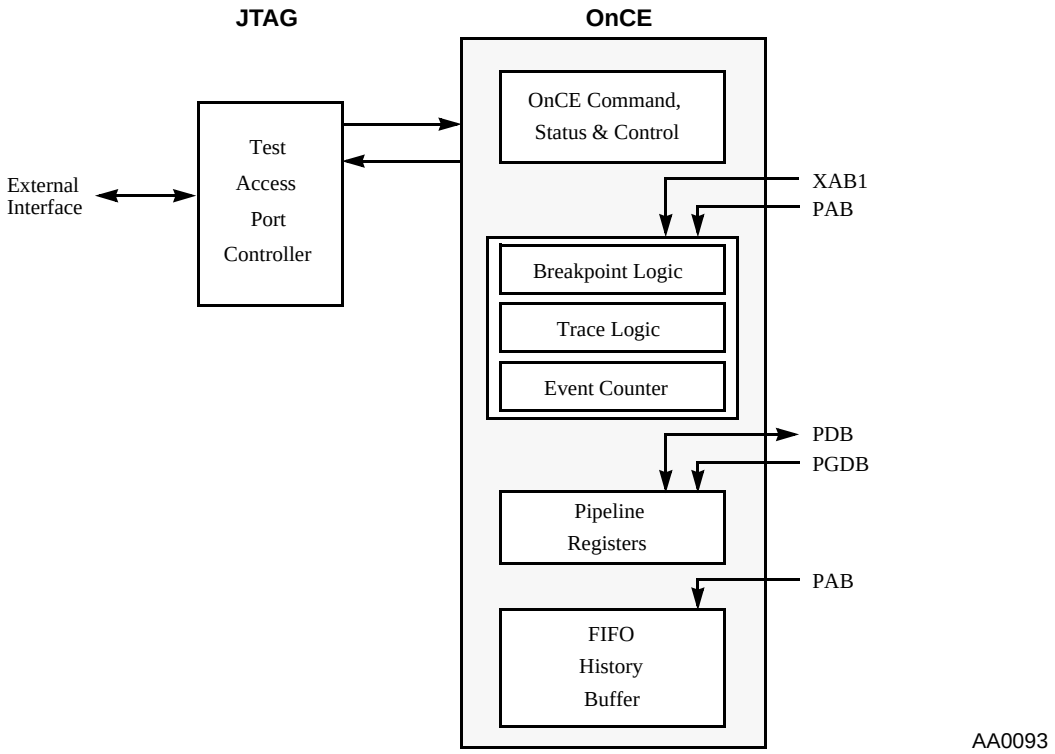


Figure 9-1. JTAG/OnCE Interface Block Diagram

As already noted, the JTAG module is the master. It enables interaction with the debug services provided by the OnCE, and its external serial interface is used by the OnCE port for sending and receiving debugging commands and data.

9.2 JTAG Port

Problems associated with testing high-density circuit boards have led to the development of a proposed standard under the sponsorship of the Test Technology Committee of IEEE and the Joint Test Action Group (JTAG). The resulting standard, called the *IEEE Standard Test Access Port and Boundary-Scan Architecture*, specifies industry-standard, in-circuit device testing and diagnosis. The DSP56800 family provides a dedicated test access port (TAP) that is fully compatible with this standard, commonly referred to as the “JTAG port.”

This section provides an overview of the capabilities of the JTAG port as implemented on the DSP56800. Information provided here is intended to supplement the supporting IEEE 1149.1a-1993 document, which outlines the internal details, applications, and overall methodology of the standard. Specific details on the implementation of the JTAG port for a given DSP56800-based device are provided in that device’s user’s manual.

9.2.1 JTAG Capabilities

The DSP56800 JTAG port has the following capabilities:

- Performing boundary scan operations to test circuit-board electrical continuity
- Sampling the DSP56800-based device system pins during operation and transparently shifting out the result in the boundary scan register; preloading values to output pins prior to performing a boundary scan operation
- Querying identification information (manufacturer, part number, and version) from a DSP56800-based device
- Adding a weak pull-up device on all input signals to cause all open inputs to report a logic 1 and to force a predictable internal state while performing external boundary scan operations
- Disabling the output drive to pins during circuit-board testing
- Forcing test data onto the outputs of a DSP56800-based device
- Providing a means of accessing the OnCE controller and circuits to control a target system
- Providing a means of entering the debug mode of operation
- Bypassing the DSP56800 core for a given circuit-board test by effectively reducing the boundary scan register to a single cell

Section 9.2.2, “JTAG Port Architecture,” provides an overview of the port’s architecture and commands. For additional information on the JTAG port’s implementation and command set, see the appropriate DSP56800-based device’s user’s manual.

9.2.2 JTAG Port Architecture

The JTAG module consists of the logic necessary to support boundary scan testing as defined in the IEEE specification. Although tightly coupled to the DSP56800’s core logic, it is an independent module, and, when disabled, it is guaranteed to have no impact on the function of the core.

The JTAG port consists of the following components:

- Serial communications interface
- Command decoder and interpreter
- Boundary scan register
- ID register

These units, and the overall OnCE port architecture, are shown in Figure 9-2 on page 9-4.

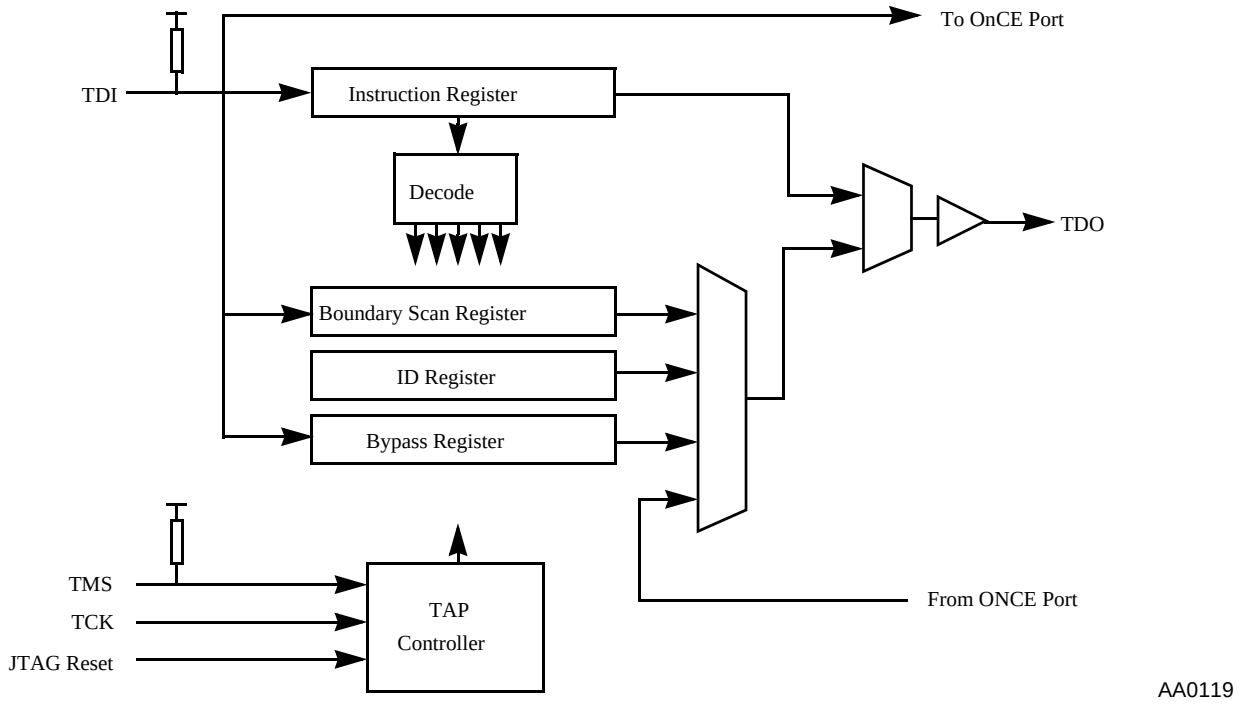


Figure 9-2. JTAG Block Diagram

The serial interface supports communications with the host development or test system. It is implemented as a serial interface to occupy as few external pins on the device as possible. Consult the device's user's manual for a full description of the interface signals. All JTAG and OnCE commands and data are sent over this interface from the host system. The JTAG interface is also used by the OnCE port when it is active. In this mode, the JTAG acts as the OnCE port's interface controller, and transparently passes all communications through to the OnCE port.

Commands sent to the JTAG module are decoded and processed by the command decoder. Commands for the JTAG port are completely independent from the DSP56800 instruction set, and are executed in parallel by the JTAG logic.

Registers in the JTAG module hold chip identification information and the information gathered by boundary scan operations. The ID register contains the industry-standard Freescale identification information, which is unique for each Freescale DSC. The boundary scan register holds a snapshot of the device's pins when sampled by the JTAG port.

9.3 OnCE Port

The OnCE port provides emulation and debug capability directly on the chip, eliminating the need for expensive and complicated stand-alone in-circuit emulators (ICEs). The OnCE port permits full-speed, non-intrusive emulation on a user's target system. This section describes the OnCE emulation environment for use in debugging real-time embedded applications.

The OnCE port has an associated interrupt vector in the DSP56800 interrupt vector table. 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.

As emulation capabilities are necessarily tied to the particular implementation of a DSP56800-based device, the appropriate device's user's manual should be consulted for complete details on implementation and supported functions.

9.3.1 OnCE Port Capabilities

The capabilities of the OnCE port include the following:

- Interrupting and breaking into debug mode on a program memory address
- Interrupting and breaking into debug mode on a data memory address (read, write, or access)
- Interrupting and breaking into debug mode on an on-chip peripheral register access
- Entering debug mode using a microprocessor instruction
- Examining or modifying the contents of any core or memory-mapped peripheral register
- Examining or modifying any desired sections of program or data memory
- Full-speed stepping on one or more instructions (up to 256)
- Tracing one or more instructions
- Saving or restoring the current state of the chip's pipeline
- Displaying the contents of the real-time instruction trace buffer
- Returning to user mode from debug mode

Depending on the implementation for a particular DSP56800-based device, additional debugging and emulation capabilities may be provided. Consult the user's manual for the device in question for more information.

9.3.2 OnCE Port Architecture

The OnCE port module is composed of four different sub-modules, each of which performs a different task:

- Command, status, and control
- Breakpoint and trace
- Pipeline save and restore
- FIFO history buffer

These units, and the overall once port architecture, are shown in Figure 9-3 on page 9-6.

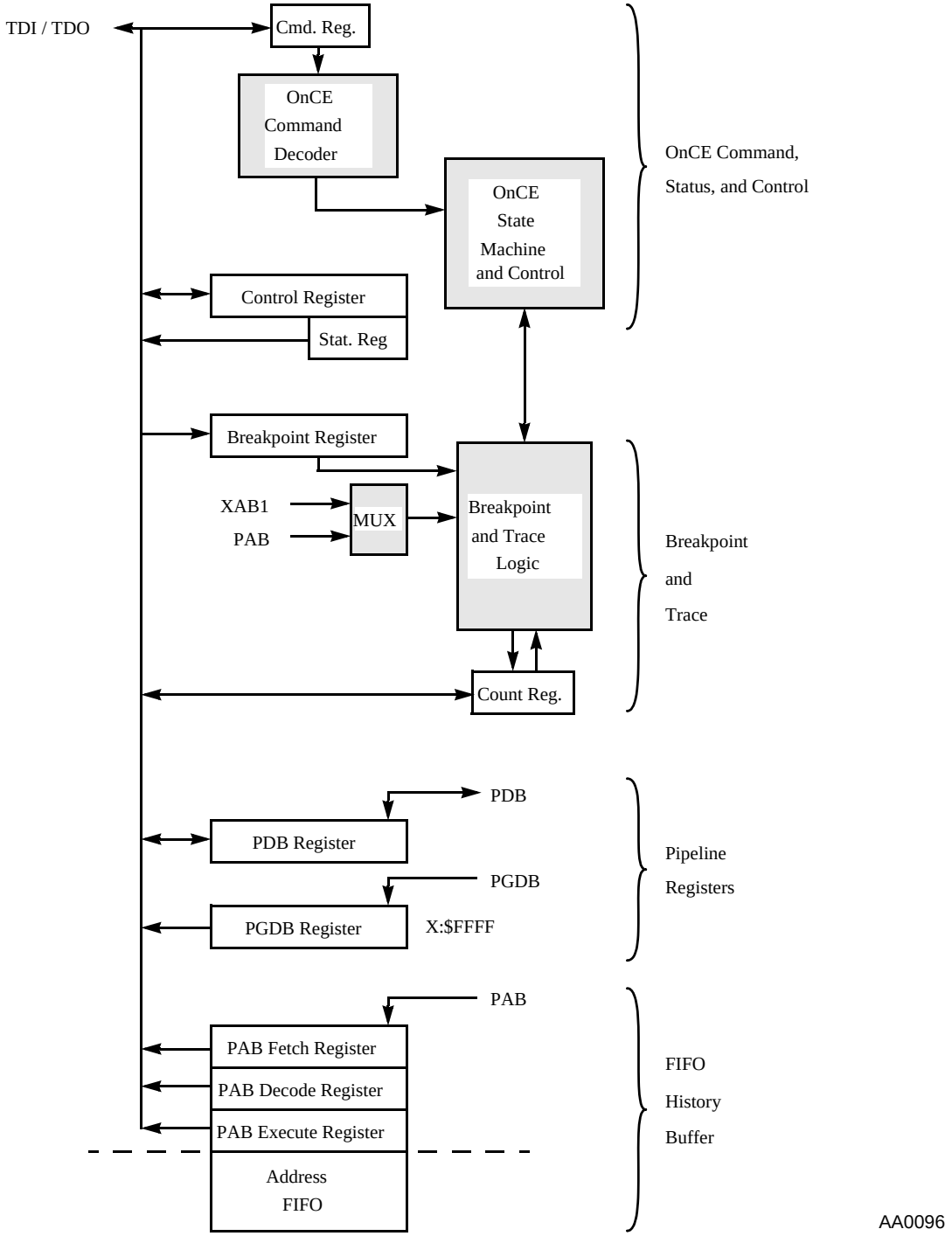


Figure 9-3. OnCE Block Diagram

Together, these sub-modules provide a full-featured emulation and debug environment. Communication with the OnCE port module is handled via the JTAG port and thus may be considered the primary communications sub-module for the OnCE port, although it operates independently. The operations of the OnCE port occur independently of the main DSP56800 core logic, and require no core resources.

9.3.2.1 Command, Status, and Control

The command, status and control portion of the OnCE port module handles the processing of emulation and debugging commands from a host development system. Communications with a host system are provided by the JTAG port module, and are passed transparently through to this logic, which is responsible for coordinating all emulation and debugging activity.

As previously noted, all emulation and debug processing takes place independently of the main DSP56800 processor core. This allows for instructions to be executed in debug mode at full speed, without any overhead introduced by the debugging logic.

9.3.2.2 Breakpoint and Trace

The OnCE port module includes address-comparison hardware for setting breakpoints on program or data memory accesses. This allows breakpoints to be set on program FLASH as well as program RAM locations. Breakpoints can be programmed for reads, writes, program fetches, or memory accesses. Breakpoints are also possible during on-chip peripheral register accesses, since these are implemented as memory-mapped registers in the X data space.

Full-speed instruction stepping capability is also provided. Up to 256 instructions can be executed at full speed before the processor core is halted and the debug processing state is re-entered. This allows the user to single step through a program or execute whole functions at a time.

9.3.2.3 Pipeline Save and Restore

To resume normal chip activity when the chip is returning from the debug mode, the previous chip pipeline state must be reconstructed. The OnCE port module provides logic to correctly save and restore the pipeline state when entering and exiting debug mode. Pipeline saves and restores operate transparently to the user, although the pipeline state may be examined while in debug mode if desired.

9.3.2.4 FIFO History Buffer

To ease debugging activity and to help keep track of program flow, a read-only FIFO buffer is provided that tracks the execution history of an application. It stores the address of the instruction currently being executed by the processor core, as well as the addresses of the last five execution flow instructions.

The FIFO history buffer is intended to provide a snapshot of the recent execution history of the processor core. To give a larger picture of instruction flow, not all instructions are recorded in the buffer. Only the addresses of the following execution flow instructions are stored:

BRA	JMP
JSR	Bcc (with condition true)
Jcc (with condition true)	

Sequential program flow can be assumed between recorded instructions, so it is possible for the user to reconstruct the program flow extending back through quite a large number of instructions. To complete the execution history, the first location of the FIFO always holds the address of the last executed instruction, regardless of whether or not it caused a change of program flow.

Appendix A
Instruction Set Details

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.1 Notation

Each instruction description contains notation used to abbreviate certain operands and operations. The symbols and their respective descriptions are listed in Table A-1 through Table A-7 on page A-4.

Table A-1 shows the register set available for the most important move instructions. Sometimes the register field is broken into two different fields — one where the register is used as a source and the other where it is used as a destination. This is important because a different notation is used when an accumulator is being stored without saturation. In addition, see the register fields in Table A-2 on page A-2, which are also used in move instructions as sources and destinations within the AGU.

Table A-1. Register Fields for General-Purpose Writes and Reads

Register Field	Registers in This Field	Comments
HHH	A, B, A1, B1 X0, Y0, Y1	Seven data ALU registers — two accumulators, two 16-bit MSP portions of the accumulators and three 16-bit data registers
HHHH	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	Seven data ALU and five AGU registers
DDDDD	A, A2, A1, A0 B, B2, B1, B0 Y1, Y0, X0 R0, R1, R2, R3 N, SP M01 OMR, SR LA, LC HWS	All CPU registers

Table A-2 shows the register set available for use as pointers in address-register-indirect addressing modes. The most common fields used in this table are Rn and Rj. This table also shows the notation used for AGU registers in AGU arithmetic operations.

Table A-2. Address Generation Unit (AGU) Registers

Register Field	Registers in This Field	Comments
Rn	R0-R3 SP	Five AGU registers available as pointers for addressing and as sources and destinations for move instructions
Rj	R0, R1, R2, R3	Four pointer registers available as pointers for addressing
N	N	One index register available only for indexed addressing modes
M01	M01	One modifier register

Table A-3 shows the register set available for use in data ALU arithmetic operations. The most common field used in this table is FDD.

Table A-3. Data ALU Registers

Register Field	Registers in This Field	Comments
FDD	A, B X0, Y0, Y1	Five data ALU registers — two 36-bit accumulators and three 16-bit data registers accessible during data ALU operations Contains the contents of the F and DD register fields
F1DD	A1, B1 X0, Y0, Y1	Five data ALU registers — two 16-bit MSP portions of the accumulators and three 16-bit data registers accessible during data ALU operations
DD	X0, Y0, Y1	Three 16-bit data registers
F	A, B	Two 36-bit accumulators accessible during parallel move instructions and some data ALU operations
~F,F		~F,F refers to any of two valid accumulator combinations: A,B or B,A
F1	A1, B1	The 16-bit MSP portion of two accumulators accessible as source operands in parallel move instructions

Address operands used in the instruction field sections of the instruction descriptions are given in Table A-4. Addressing mode operators that are accepted by the assembler for specifying a specific addressing mode are shown in Table A-5.

Table A-4. Address Operands

Symbol	Description
ea	Effective address
xxxx	Absolute address (16 bits) - X:xxxx
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. 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

Miscellaneous operand notation, including generic source and destination operands and immediate data specifiers, are summarized in Table A-6.

Table A-6. Miscellaneous Operands

Symbol	Description
S, Sn	Source operand register
D, Dn	Destination operand register
#xx	Immediate short data (7 bits for MOVEI, 6 bits for DO/REP)
#xxxx	Immediate data (16 bits)
<MASK8>	8-bit mask value #ii00 implies 8-bit immediate data mask in the upper byte #00ii implies 8-bit immediate data mask in the lower byte
<MASK16>	16-bit mask value
<OFFSET7>	7-bit signed PC-relative offset
<ABS16>	16-bit absolute address

Table A-7. Other Symbols

Symbol	Description
( )	Optional letter, operand, or operation[1]
(...)	Any arithmetic or logical instruction that 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)

1. For instruction names that contain parentheses, such as DEC(W) or IMPY(16), the portion within the parentheses is optional.

A.2 Programming Model

The registers in the DSP56800 core programming model are shown in Figure A-1.

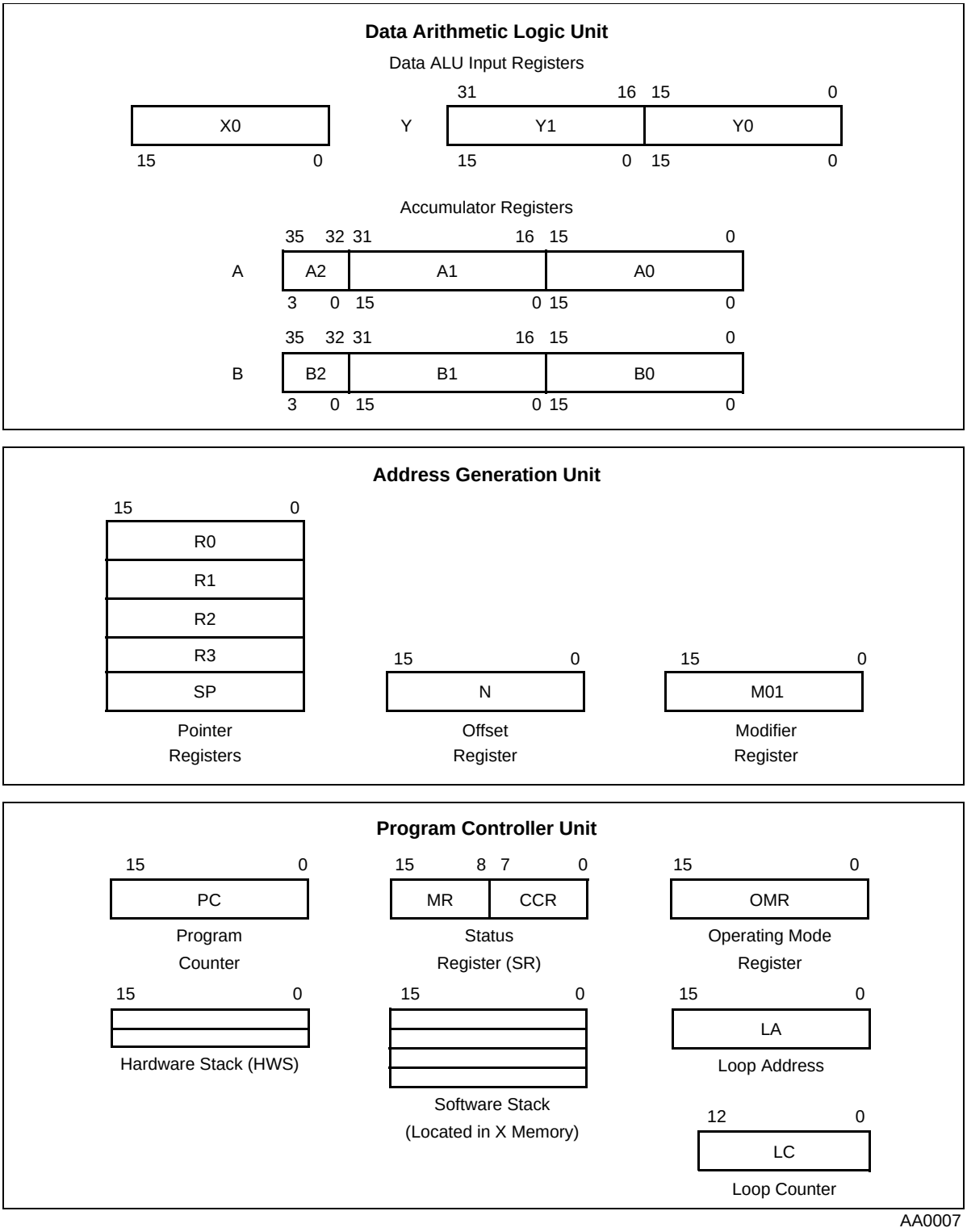


Figure A-1. DSP56800 Core Programming Model

A.3 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 in the following discussion and summarized in Table 4-5 on page 4-9.

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 Rj (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+N) 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 DSC-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 4-9 on page 4-27.

A.4 Condition Code Computation

The bits in the Condition Code Register (CCR) are set to reflect the status of the processor 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 some condition code bits is affected by the OMR’s Saturation (SA) and condition code (CC) bits. The SA bit enables the MAC Output Limiter, which can alter the results of computations and thus the condition code bits affected. The CC bit specifies whether condition codes are generated using the information in the extension register. See Section A.4.2, “Effects of the Operating Mode Register’s SA Bit,” and Section A.4.3, “Effects of the OMR’s CC Bit,” for more information.

A.4.1 The Condition Code Bits

The DSP56800 family defines eight condition code bits, which are contained in the lower-order 8 bits of the Status Register (SR) as follows:

SR
Status Register
Reset = $0300
Read/Write

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	L	E	U	N	Z	V	C

LF — Loop Flag
I1,I0 — Interrupt Mask
SZ — Size
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

Figure A-2. Status Register (SR)

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 that indicates that an overflow has occurred in the data ALU or that limiting has occurred when moving an accumulator register to memory. The SZ bit is a latching bit that indicates the size of an accumulator when it is moved to data memory.

A.4.1.1 Size (SZ) — Bit 7

The SZ bit is set only when moving one of the two accumulators (A or B) to data memory. It is set if, during this move, bits 30 and 29 of the specified accumulator are not the same — that is, not 00 or 11 — as follows:

$$SZ = SZ \mid (\text{Bit } 30 \oplus \text{Bit } 29)$$

SZ is not affected otherwise. Note that the SZ bit is latched once it is set — it is only cleared by a processor reset or an instruction that explicitly clears it.

SZ is not affected by the OMR's CC or SA bits.

A.4.1.2 Limit (L) — Bit 6

The L bit is set to indicate that one of two conditions has occurred: an overflow has occurred in a data ALU operation (see Section A.4.1.7, “Overflow (V) — Bit 1,” on page A-10), or limiting has occurred when moving one of the two accumulators (A or B) with a move or parallel move instruction. L is not affected otherwise.

The L bit is latched once it is set; it is cleared only by a processor reset or an instruction that explicitly clears it. The complete formula for calculating L is the following:

L = L | V | (limiting due to a move)

L is not affected by the OMR’s CC or SA bits. Note, however, that the V bit is affected by both the CC and SA bits. As a result, the L bit can be indirectly affected by these two control bits.

NOTE:

The TFR instruction performs a register-to-register transfer and is not considered a “move” instruction in terms of the preceding discussion. The L bit will therefore *not* be set due to the register-to-register move, even if SA is set and saturation occurs. The TFR instruction can set the L bit if it has a parallel move and if limiting occurs in that parallel move.

A.4.1.3 Extension in Use (E) — Bit 5

The E bit is updated based on the result of a data ALU operation to indicate whether the MSP and LSP of the result contain all of the significant bits, or if the extension bits are needed to express the result. If the E bit is clear, the MSP and LSP contain all the significant bits — the high-order bits represent only sign extension.

Based on the size of the result or destination, the E bit is calculated as follows:

For 20- and 36-bit results or destinations:

E is cleared if the upper 5 bits of the result are 00000 or 11111. E is set otherwise.

For 16-bit results or destinations:

If one of the operands is located in X0, Y0, or Y1, or comes from memory, the value is first sign extended. Sign extension is also performed when the source operand is located in an accumulator. If one of the operands is 5-bit immediate data, that value is first zero extended. A 20-bit arithmetic operation is then performed, where the result is located in the lowest 16 bits. E is cleared if all of the upper 5 bits of the 20-bit result are 00000 or 11111, and is set otherwise.

For 32-bit results or destinations:

If one of the operands comes from memory or the Y register, or is 16-bit immediate data, it is first sign extended. Sign extension is also performed when the source operand is located in an accumulator. If one of the operands is 5-bit immediate data, it is first zero extended. A 36-bit arithmetic operation is then performed, where the long result is located in the lowest 32 bits. E is cleared if all of the upper 5 bits of the result are 00000 or 11111 and is set otherwise.

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

NOTE:

When the SA bit in the OMR register is set to one, the E bit is set based on the result *before* passing through the MAC Output Limiter. If SA is set to one and saturation does occur in the MAC Output Limiter, this can result in the E bit being set, even though the result is saturated to a value where the extension portion is not in use.

A.4.1.4 Unnormalized (U) — Bit 4

The U bit is updated under the following conditions. If the SA bit in the OMR is set to one, this bit is cleared if saturation occurs in the MAC Output Limiter. If the SA bit is zero or no saturation occurs, U is set if the two MSBs of the MSP of the result are the same following a data ALU operation; it is cleared otherwise. The computation of U varies depending on the size of the operation's destination or result.

For 20-, 32-, and 36-bit destinations or results, U is computed according to the following formula (32-bit destinations are first extended as described for the E bit):

$$U = \sim(\text{Bit } 31 \oplus \text{Bit } 30)$$

Sixteen-bit destinations are first extended as described for the E bit. Then U is computed as follows:

$$U = \sim(\text{Bit } 15 \oplus \text{Bit } 14)$$

The U bit is not affected by the OMR's CC bit.

A.4.1.5 Negative (N) — Bit 3

The N bit is updated based on the result of a data ALU operation. In general, it reflects the sign bit (MSB) of the result, according to the following rules:

For 20- or 36-bit results:

N = bit 35 for A or B (bit 31 if the OMR's CC bit is set to one)
N = bit 15 for Y1, Y0, or X0

For 32-bit results:

N = bit 31 for A, B, or Y (the OMR's CC bit has no effect)
N = bit 15 for Y1, Y0, or X0

For 16-bit results:

N = bit 31 for A, B, or Y (the OMR's CC bit has no effect)
N = bit 15 for 16-bit destination

When the SA bit in the OMR register is set to one, the N bit is set based on the result *before* passing through the MAC Output Limiter.

For the ASRAC and LSRAC instructions, the N bit is calculated differently based on the SA bit in the OMR register. When the SA bit is zero and the destination is one of the accumulators, the N bit is obtained from bit 35. When SA is one and the destination is one of the accumulators, the N bit is set based on bit 31 of the result *before* passing through the MAC output limiter.

For the IMPY instruction, a 31-bit integer product is calculated internally to the data ALU, and the lowest 16 bits of this product are stored in the destination register. When SA is one or CC is one, the N bit is set to the value in bit 30 of this internally computed result. When SA is zero and CC is zero, the N bit is set to the value in bit 15 of this internally computed result. These two values are identical except in the case where overflow occurs (that is, the result is larger than and will not fit in 16 bits).

For the ASLL instruction, if the CC bit is set, the N bit is always cleared. If CC is 0, the N bit is set according to the standard definition outlined in the preceding discussion.

A.4.1.6 Zero (Z) — Bit 2

The Z bit is updated based on the result of a data ALU operation. Z is set if the result of an operation is zero — that is, all significant bits are set to zero. It is cleared otherwise.

The number of bits used to compute the value for Z is determined by the size of the result and whether or not the OMR's CC bit is set:

For 36-bit results:

Z is set if bits 35 to 0 of the result are all zero, or bits 31 to 0 if the OMR's CC bit is set.

For 32-bit results:

Z is set if bits 31 to 0 of the result are all zero. It is set using bits 15 to 0 of the result if Y1, Y0, or X0 is the destination.

For 20-bit results:

Z is set if bits 35 to 16 of the result are all zero, or bits 31 to 16 if the OMR's CC bit is set.

For 16-bit results:

Z is set if bits 31 to 16 of the result are all zero for A, B, Y; it is set if bits 15 to 0 of the result are all zero for 16-bit destinations.

Z is not affected by the OMR's SA bit.

A.4.1.7 Overflow (V) — Bit 1

The V bit is updated under the following conditions. If the SA bit in the OMR is set to one, V is set when saturation occurs in the MAC Output Limiter. If the SA bit is zero or no saturation occurs, it is set when an arithmetic overflow occurs as the result of a data ALU operation. Overflow occurs when the carry into the result's MSB is not equal to the carry out of the MSB, thus changing the sign of the value. The result of the ALU operation is therefore not representable in the destination — the result has overflowed. V is cleared when overflow does not occur.

In general, overflow is calculated based on the size of the result or destination of the operation. When the CC bit in the OMR is set, however, overflow is determined based on the 32-bit result for what would otherwise be 36-bit results. The same is true for 20-bit results: when the CC bit is set, overflow is determined based on the 16-bit result.

For the IMPY instruction, V is set if the computed result does not fit in 16 bits and is cleared otherwise. The SA bit has no effect in this case.

A.4.1.8 Carry (C) — Bit 0

The C bit is updated based on the result of a data ALU operation. C is set either if a carry is generated out of the most significant bit (MSB) of the result for an addition, or if a borrow is generated in a subtraction. C is cleared otherwise.

For 20- or 36-bit results, the carry or borrow is generated out of bit 35. For 32-bit results, the carry or borrow is generated out of bit 31. The carry or borrow is generated out of bit 15 for 16-bit results.

C is not affected by the OMR's CC or SA bits.

A.4.2 Effects of the Operating Mode Register's SA Bit

The SA bit in the Operating Mode Register (OMR) can affect the computation of certain condition code bits. This bit enables the MAC Output Limiter within the data ALU. When enabled, the results of many operations are limited to fit with 32 bits, the extension portion containing only sign information. This limiting operation has both direct and indirect effects on the way condition codes are computed.

The SA bit directly affects the following condition code bits:

- U — cleared if saturation occurs in the MAC Output Limiter
- V — set when saturation occurs in the MAC Output Limiter

The remaining bits in the Condition Code Register are not affected by the SA bit, with the following exceptions:

- L — may be indirectly affected through effects on the V bit
- N — affected only by the ASRAC, LSRAC, and IMPY instructions
- C — affected only by the ASL instruction

The value of the SA bit is designed not to affect condition code computation for the TSTW instruction. Only the U condition code bit is affected by the SA bit for the CMP instruction. These instructions operate independently of the CC bit and correctly generate both signed and unsigned condition codes.

The SA bit only affects operations in the data ALU, not operations performed in other blocks. These include move instructions, bit-manipulation instructions, and address calculations performed by the AGU.

NOTE:

When SA is set to one for an application, condition codes are not always set in an intuitive manner. It is best to examine the instruction details to determine the effect on condition codes when SA is one. See Section A.7, "Instruction Descriptions."

A.4.3 Effects of the OMR's CC Bit

The CC bit in the OMR may affect the computation of the condition code bits. The CC bit establishes how many of the bits of an arithmetic or logic operation result are used when calculating condition codes. Specifically:

- When CC = 0, the result is interpreted as 36 bits with a valid extension portion.
- When CC = 1, the result is interpreted as 32 bits with the extension portion ignored.

Signed values can be computed in both cases, but computation of unsigned values must be performed with the CC bit set to one. Without setting CC to one prior to executing the TST and CMP instructions, the HI, HS, LO, and LS branch/jump conditions cannot be used.

When the CC bit is set, the following condition code bits are affected:

- V — set based on the MSB of the result's MSP portion
- Z — set using only the MSP and LSP portions of the result

The remaining bits in the Condition Code Register are not affected by the CC bit, with the following exceptions:

- L — may be indirectly affected through effects on the V bit
- N — affected only by the ASRAC, LSRAC, IMPY, and ASLL instructions
- C — affected only by the ASL instruction

The value of the CC bit does not affect condition code computation for the TSTW instruction. These instruction operates independently of the CC bit and correctly generate both signed and unsigned condition codes.

The CC bit only affects operations in the data ALU, not operations performed in other blocks. These include move instructions, bit-manipulation instructions, and address calculations performed by the AGU.

A.4.4 Condition Code Summary by Instruction

Table A-9 provides a detailed view of the condition codes affected by each instruction, and the circumstances under which each condition code is set or cleared. Table A-8 describes the notation used. Items in the "Notes" column of Table A-9 are explained immediately following the table on page A-15.

Table A-8. Notation Used for the Condition Code Summary Table

Notation	Description
*	Set by the result of the operation according to the standard definition.
—	Not affected by the operation.
*16	Set according to the standard definition for 16-bit results.
*32	Set according to the standard definition for 32-bit results.
*36	Set according to the standard definition for 36-bit results.
*A	Set by the result of the operation according to the size of destination.
*B	Set by the result of the operation according to the size of destination.
=0	Cleared.
=1	Set.
?	Set according to the special computation defined for the operation.
(number)	Set according to the special computation defined by the note with the corresponding number. The notes may be found immediately after Table A-9.
C	L bit can be set if overflow has occurred in result.
T	L bit can be set if limiting occurs when reading an accumulator during a parallel move or by the instruction itself. An example of the latter case is `BFCHG #$8000,A`, which must first read the A accumulator before performing the bit-manipulation operation.
CT	L bit can be set if overflow has occurred in the result or if limiting occurs when an accumulator is being read.

The condition code computation shown in Table A-9 may differ from that defined in the opcode descriptions; see Section A.7, "Instruction 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-9 gives the chip implementation viewpoint, while the opcode descriptions give the user viewpoint.

The “Comments” column in the table is also used to report if any of the upper bits in the status register are modified. These are not status bits because they do not lie in the status portion of the status register, but rather in the control portion. Sometimes these bits are also affected by instructions. Examples include the interrupt mask bits, I1 and I0, and the looping bits, LF and NL (NL lies in the OMR register).

The following instruction mnemonics are not found in Table A-9: ANDC, EORC, NOTC and ORC. This is because each of these is an alias for another instruction and not an instruction in its own right. To determine the condition code calculation for each of these, determine the instructions to which these mnemonics are mapped (see Section 6.5.1, “ANDC, EORC, ORC, and NOTC Aliases,” on page 6-11) and look at the condition code information for the corresponding real instructions.

Table A-9. Condition Code Summary

Instruction	SZ	L	E	U	N	Z	V	C	Comments
ABS	*	CT	*36	*36	*36	*36	*36	—	
ADC	—	C	*36	*36	*36	*36	*36	*36	
ADD	*	CT	*A	*A	*A	*A	*A	*A	
AND	—	—	—	—	*16	*16	=0	—	
ASL	*	CT	*A	*A	*A	*A	(1)	(2)	
ASLL	—	—	—	—	(14)	*32	—	—	
ASR	*	T	*A	*A	*A	*A	=0	(3)	
ASRAC	—	—	—	—	(12)	*36	—	—	
ASRR	—	—	—	—	*32	*32	—	—	
Bcc	—	—	—	—	—	—	—	—	
BFCHG	—	T	—	—	—	—	—	(4)	
BFCLR	—	T	—	—	—	—	—	(4)	
BFSET	—	T	—	—	—	—	—	(4)	
BFTSTH	—	T	—	—	—	—	—	(4)	
BFTSTL	—	T	—	—	—	—	—	(5)	
BRA	—	—	—	—	—	—	—	—	
BRCLR	—	T	—	—	—	—	—	(5)	
BRSET	—	T	—	—	—	—	—	(4)	
CLR	*	CT	*36	*36	*36	*36	*36	—	Never overflows
CMP	*	CT	*A	*A	*A	*A	*A	*A	
DEBUG	—	—	—	—	—	—	—	—	
DEC(W)	*	CT	*B	*B	*B	*B	*B	*B	
DIV	—	C	—	—	—	—	(1)	(6)	

Table A-9. Condition Code Summary (Continued)

Instruction	SZ	L	E	U	N	Z	V	C	Comments
DO	—	T	—	—	—	—	—	—	Affects LF, NL bits
ENDDO	—	—	—	—	—	—	—	—	Condition code not affected
EOR	—	—	—	—	*16	*16	=0	—	
ILLEGAL	—	—	—	—	—	—	—	—	Sets I1, I0 bits in SR
IMPY16	—	C	—	—	(13)	*16	(11)	—	
INCW	*	CT	*B	*B	*B	*B	*B	*B	
Jcc	—	—	—	—	—	—	—	—	
JMP	—	—	—	—	—	—	—	—	
JSR	—	—	—	—	—	—	—	—	
LEA	—	—	—	—	—	—	—	—	
LSL	—	—	—	—	*16	*16	=0	(7)	
LSLL	—	—	—	—	*32	*32	—	—	
LSR	—	—	—	—	*16	*16	=0	(8)	
LSRAC	—	—	—	—	(12)	*36	—	—	
LSRR	—	—	—	—	*32	*32	—	—	
MAC	*	CT	*A	*A	*A	*A	*A	—	
MACR	*	CT	*A	*A	*A	*A	*A	—	
MACSU	—	C	*A	*A	*A	*A	*A	—	
MOVE	* (10)	T (10)	— (10)	— (10)	— (10)	— (10)	— (10)	— (10)	NA unless SR is the destination in the instruction
MPY	*	CT	*A	*A	*A	*A	*A	—	V cleared
MPYR	*	CT	*A	*A	*A	*A	*A	—	V cleared
MPYSU	—	C	*A	*A	*A	*A	*A	—	V cleared
NEG	*	CT	*A	*A	*A	*A	*A	*A	
NOP	—	—	—	—	—	—	—	—	
NORM	—	C	*36	*36	*36	*36	(1)	—	
NOT	—	—	—	—	*16	*16	=0	—	
OR	—	—	—	—	*16	*16	=0	—	
POP	—	—	—	—	—	—	—	—	

Table A-9. Condition Code Summary (Continued)

Instruction	SZ	L	E	U	N	Z	V	C	Comments
REP	—	T	—	—	—	—	—	—	
RND	*	CT	*36	*36	*36	*36	*36	—	
ROL	—	—	—	—	*16	*16	=0	(7)	
ROR	—	—	—	—	*16	*16	=0	(8)	
RTI	Restored — (9)								
RTS	—	—	—	—	—	—	—	—	
SBC	—	C	*36	*36	*36	*36	*36	*36	
STOP	—	—	—	—	—	—	—	—	
SUB	*	CT	*A	*A	*A	*A	*A	*A	
SWI	—	—	—	—	—	—	—	—	Affects I1, I0 bits in SR
Tcc	—	—	—	—	—	—	—	—	
TFR	*	T	—	—	—	—	—	—	
TST	*	CT	*36	*36	*36	*36	0	0	Never overflows
TSTW	—	—	—	—	*36	*36	0	0	Never overflows
WAIT	—	—	—	—	—	—	—	—	

NOTES:

1. V is set if the MSB of the destination operand (bit 35 for an accumulator or bit 31 for the Y register) is changed as a result of the left shift; V is cleared otherwise.
2. C is set if the MSB of the source operand (bit 35 for an accumulator or bit 31 for the Y register) is set and is cleared otherwise.
3. C is set if bit 0 of the 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 that 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 the MSB of the result is cleared (bit 35 for an accumulator or bit 31 for the Y register). The C bit is cleared if the MSB of the result is set.
7. For the accumulators, C is set if bit 31 of the source operand is set and is cleared otherwise. For the Y1, Y0, and X0 registers, C is set if bit 15 of the source operand is set and is cleared otherwise.
8. For the accumulators, C is set if bit 16 of the source operand is set and is cleared otherwise. For the Y1, Y0, and X0 registers, C is set if bit 0 of the source operand is set and is cleared otherwise.

9. The “?” bit is set according to value pulled from stack.
10. If the SR is specified as a destination operand (for example, `MOVE X:(R0),SR`), each bit is set according to the corresponding bit of the source operand. If SR is not specified as a destination operand, none of the status bits are affected.
11. The V bit for the IMPY instruction is set if the calculated integer product does not fit in 16 bits.
12. The setting of the N bit for the ASRAC and LSRAC instructions depends on the OMR’s SA bit. If SA is one, then the N bit is equal to bit 31 of the result. If SA is zero, then N is equal to bit 35 of the result.
13. When SA is zero and CC is zero for the IMPY instruction, the N bit is set using *16. When SA is one or CC is set to one, this bit is set as described in Section A.4.1.5, “Negative (N) — Bit 3.”
14. When CC is one for the ASLL instruction, the N bit is cleared. When CC is zero, this bit is set as described under Section A.4.1.5, “Negative (N) — Bit 3.”

See Section 3.6, “Condition Code Generation,” on page 3-33 for additional information on condition codes.

A.5 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 use 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 depends 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 oscillator clock cycles per instruction is dependent on many factors, including the number of words per instruction, the addressing mode, whether the instruction fetch pipe is full or not, the number of external bus accesses, and the number of wait states inserted in each external access. The symbols used 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-11 on page A-18 gives the number of instruction program words and the number of machine clock cycles for each instruction mnemonic.
- Table A-12 on page A-19 gives the number of additional instruction words (if any) and additional clock cycles (if any) for each type of parallel move operation.
- Table A-13 on page A-20 gives the number of additional (if any) clock cycles for each type of MOVEC operation.
- Table A-14 on page A-20 gives the number of additional (if any) clock cycles for each type of MOVEM operation.
- Table A-15 on page A-20 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-16 on page A-20 gives the number of additional clock cycles (if any) for each type of branch or jump (Bcc, Jcc, and JSR) operation.

- Table A-17 on page A-21 gives the number of additional clock cycles (if any) for the RTS or RTI instruction.
- Table A-18 on page A-21 gives the number of additional clock cycles (if any) for the TSTW instruction.
- Table A-19 on page A-21 gives the number of additional instruction words (if any) and additional clock cycles (if any) for each effective addressing mode.
- Table A-20 on page A-22 gives the number of additional clock cycles (if any) for external data, external program, and external I/O memory accesses.

The symbols used in the tables are summarized in Table A-10.

Table A-10. 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
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

The assumptions for calculating execution time are the following:

- All instruction cycles are counted in oscillator clock cycles. Two oscillator 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 that flush the pipeline, such as JMP, Jcc, RTS, and so on.

In order to better understand and use the following tables, examine the three examples for computing an instruction's execution time that are presented at the end of this section: Example A-1 on page A-22, Example A-2 on page A-23, and Example A-3 on page A-25.

Table A-11. Instruction Timing Summary

Mnemonic	Instruction Words	Clock Cycles	Mnemonic	Instruction Words	Clock Cycles
ABS	1	2+mv	LSRAC	1	2
ADC	1	2	LSRR	1	2
ADD	1+mv	2+(ea or mv)	MAC	1	2+mv
AND	1	2	MACR	1	2+mv
ANDC	2+ea	4+mvb	MACSU	1	2
ASL	1	2+mv	MOVE[1]	1	2+mv
ASLL	1	2	MOVEC	1+ea	2+mvc
ASR	1	2+mv	MOVEI	1+ea	2+ea
ASRAC	1	2	MOVEM	1	8+mvm
ASRR	1	2	MOVEP	1+ea	2+ea
Bcc	1	4+jx	MOVES	1+ea	2+ea
BFCHG	2+ea	4+mvb	MPY	1	2+mv
BFCLR	2+ea	4+mvb	MPYR	1	2+mv
BFSET	2+ea	4+mvb	MPYSU	1	2
BFTSTH	2+ea	4+mvb	NEG	1	2+mv
BFTSTL	2+ea	4+mvb	NOP	1	2
BRA	1	6+jx	NORM	1	2
BRCLR	2+ea	8+mvb+jx	NOT	1	2
BRSET	2+ea	8+mvb+jx	NOTC	2+ea	4+mvb
CLR	1	2+mv	OR	1	2
CMP	1+mv	2+(ea or mv)	ORC	2+ea	4+mvb
DEBUG	1	4	POP	1	2+ea
DECW	1+ea	2+(ea or mv)	REP	1	6
DIV	1	2	RND	1	2+mv
DO	2	6	ROL	1	2
ENDDO	1	2	ROR	1	2
EOR	1	2	RTI	1	10+rx

Table A-11. Instruction Timing Summary (Continued)

Mnemonic	Instruction Words	Clock Cycles	Mnemonic	Instruction Words	Clock Cycles
EORC	2+ea	4+mvb	RTS	1	10+rx
ILLEGAL	1	4	SBC	1	2
IMPY16	1	2	STOP[2]	1	n/a
INCW	1+ea	2+(ea or mv)	SUB	1+ea	2+(ea or mv)
Jcc	2	4+jx	SWI	1	8
JMP	2	6+jx	Tcc	1	2
JSR	2	8+jx	TFR	1	2+mv
LEA	1+ea	2+ea	TST	1	2+mv
LSL	1	2	TSTW	1	2+mv
LSLL	1	2	WAIT[3]	1	n/a
LSR	1	2			

1. This MOVE applies only to the case where two reads are performed in parallel from the X memory.

2. The STOP instruction disables the internal clock oscillator. After the clock is turned on, an internal counter counts 65,536 cycles before enabling the clock to the internal DSC circuits.

3. The WAIT instruction takes a minimum of 16 cycles to execute when an internal interrupt is pending at the time the WAIT instruction is executed.

Table A-12. 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-13. MOVEC Timing Summary

MOVEC Operation	+ mvc Cycles
16-bit immediate → register	2
Register → register	0
X memory ↔ register	ea + ax

Table A-14. MOVEM Timing Summary

MOVEM	+ mvm Cycles
Register ↔ P memory	ap
Note: 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-15. Bit-Field Manipulation Timing Summary

Bit-Field Manipulation Operation	+ mvb Cycles
BFCHG, BFCLR, or BFSET on X memory	ea + (2 * ax)
BFTSTH or BFTSTL on X memory	ea + ax
BFTSTH, BFTSTL, BFCHG, BFCLR, or BFSET on register	0
BRSET or BRCLR with condition true	2 + ea + (2 * ax)
BRSET or BRCLR with condition false	ea + (2 * ax)

Table A-16. 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, one of them being the instruction word located at the jump instruction's second-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-17. 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 and the fetch of the SR as performed by the RTI and RTS instructions.

Table A-18. TSTW Timing Summary

TSTW Operation	+mv Cycles
Register	0
X memory	ea + ax

Table A-19. 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 N	0	0
Indexed by offset N	0	2
Special		
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-20. 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:Int	—	—	—	—	—	0
X:X:	Ext:Int	—	—	—	—	—	wx
X:X:	I/O:Int	—	—	—	—	—	0

1. wx — external X memory access wait states
2. wp — external P memory access wait states

Three examples using the preceding tables follow.

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 following 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).
- External X memory accesses require zero wait state, (assume external mem requires no wait state and BCR contains the value $00).
- 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 inTable A-11 on page A-18.

   According to Table A-11 on page A-18, the MACR instruction will require one instruction program word and will execute in (2 + mv) oscillator clock cycles. The term “mv” represents the additional instruction program words (if any) and the additional oscillator clock cycles (if any) 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-12 on page A-19.

   The parallel move portion of the MACR instruction consists of an XX memory read. According to Table A-12 on page A-19, the parallel move portion of the instruction will require mv = axx additional oscillator clock cycles. The term “axx” represents the number of additional oscillator clock cycles (if any) that are required to access two operands in the X memory.

3. Evaluate the “axx” term using Table A-20 on page A-22.

   The parallel move portion of the MACR instruction consists of an XX Memory Read. According to Table A-20 on page A-22, the term “axx” depends upon where the referenced X memory locations are located in the DSP56800 memory space. External X memory accesses may require additional oscillator clock cycles depending on the memory device’s speed. Here we assume external X memory accesses require wx = 0 wait state or additional oscillator clock cycle. 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-20 on page A-22, the XX memory reference in the parallel move portion of the MACR instruction will require axx = wx = 0 additional oscillator clock cycle.

4. Compute the final results.

   Thus, based upon the assumptions given for Table A-11 on page A-18, 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 + 0) = 2 oscillator clock cycles.

NOTE:

If a similar calculation were made for a MOVEC, MOVEM, or one of the bit-field manipulation instructions (BFCHG, BFCLR, BFSET, BFTSTH or BFTSTL), using Table A-12 on page A-19 would no longer be appropriate. The user would refer to Table A-13 on page A-20, Table A-14 on page A-20, or Table A-15 on page A-20, 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 following instruction:

```
JEQ    $2000
```

Where the following conditions are true:

- OMR = $02 (normal expanded memory map).
- External P memory accesses require four wait states (assume external memory access requires 4 wait states in this example).

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-11 on page A-18.

   According to Table A-11 on page A-18, 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 oscillator clock cycles (if any) required for a jump-type instruction.

2. Evaluate the “jx” term using Table A-16 on page A-20.

   According to Table A-16 on page A-20, the Jcc instruction will require 2 + 2 * ap additional oscillator clock cycles if the “ea” condition is true; otherwise, 2 * ap if the condition is false. The term “ap” represents the number of additional oscillator clock cycles (if any) that are required to access a P memory operand. Note that the “+ (2 * ap)” term represents the two program memory instruction fetches executed at the end of a one-word jump instruction to refill the instruction pipeline.

3. Evaluate the “ap” term using Table A-20 on page A-22.

   According to Table A-20 on page A-22, the term “ap” depends upon where the referenced P memory location is located in the 16-bit DSC memory space. External memory accesses require additional oscillator clock cycles according to the number of wait states required. Here we assume that external P memory accesses require wp = 4 wait states or additional oscillator clock cycles. For this example the P memory reference is assumed to be an external reference. Thus, according to Table A-20 on page A-22, the Jcc instruction will use the value ap = wp = 4 oscillator clock cycles.

4. Compute the final results.

   Thus, based upon the assumptions given for Table A-11 on page A-18, the instruction

   ```
   JEQ    $2000
   ```

   will require 2 program words.

   If the condition is true, the instruction will execute in (4 + jx) = (4 + 2 + (2 * ap)) = (4 + 2 + (2 * wp)) = (4 + 2 + (2 * 4)) = 14 oscillator clock cycles. However, when the condition is false this instruction will execute in two fewer oscillator clock cycles, (4 + jx) = (4 + (2 * ap)) = (4 + (2 * wp)) = (4 + (2 * 4)) = 12.

Problem

Calculate the number of DSP56800 instruction program words and the number of oscillator clock cycles required for the following instruction:

```
RTS
```

Where the following conditions are true:

- OMR = $02 (normal expanded memory map).
- External P memory accesses require four wait states.
- Return Address (on the stack) = $0100 (internal P memory).

Solution

To determine the number of instruction program words and the number of oscillator clock cycles required for the given instruction, the user should perform the following 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-11 on page A-18.

   According to Table A-11 on page A-18, the RTS instruction will require one instruction program word and will execute in (10 + rx) oscillator clock cycles. The term “rx” represents the number of additional oscillator clock cycles (if any) required for an RTS instruction.

2. Evaluate the “rx” term using Table A-17 on page A-21.

   According to Table A-17 on page A-21, the RTS instruction will require rx = 2 * ap + 2 * ax additional oscillator clock cycles. In this case “ax = 0” because the instruction accesses the stack on internal memory. The term “ap” represents the number of additional oscillator clock cycles (if any) 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-20 on page A-22.

   According to Table A-20 on page A-22, the term “ap” depends upon where the referenced P memory location is located in the 16-bit DSC memory space. External memory accesses may require additional oscillator clock cycles, according to the memory device’s speed. Here we assume that external P memory accesses require wp = 4 wait states or additional oscillator clock cycles. For this example the P memory reference is assumed to be an 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-20 on page A-22, the RTS instruction will use the value ap = 0 additional oscillator clock cycles.

4. Compute the final results.

   Thus, based upon the assumptions given for Table A-11 on page A-18, the instruction

   ```
   RTS
   ```

   will require one instruction program word and will execute in (10 + rx) = (10 + (2 * ap) + (2 * ax)) = (10 + (2 * 0) + (2 * 0)) = 10 oscillator clock cycles.

A.6 Instruction Set Restrictions

These items are restrictions on the DSP56800 instruction set:

- A NORM instruction cannot be immediately followed by an instruction that 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.
- 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 (that is, at LA or at LA-1).
- A REP instruction cannot repeat on an instruction that 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 (that is, 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. In addition, it 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 (that is, 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 be 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.7 Instruction Descriptions

The following section describes each instruction in the DSP56800 Family instruction set in complete detail. Aspects of each instruction description are explained in Section A.1, “Notation,” at the beginning of this appendix.

The “Operation” and “Assembler Syntax” fields appear at the beginning of each description. For instructions that allow parallel operations, these fields include the parenthetical comment “(single parallel move)” or “(dual parallel read)”. 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 gives 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 the execution of any data ALU operation.

I/O short addressing mode is used in every example that accesses the peripheral registers. It is assumed that peripheral registers are mapped to the last 64 locations in X memory. When IP-BUS (or PGDB) interface maps these registers outside the X:$FFC0-X:$FFFF range, short addressing mode can not be used, instead peripheral registers can be accessed with any other suitable addressing mode.

Whenever a bit in the condition code register is defined according to the standard definition as given in Section A.4, “Condition Code Computation,” 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 is given in the “Condition Code” section of that instruction in bold text to alert the user to any special conditions concerning its use.

ABS Absolute Value ABS

Operation:

|D|→D
|D|→D (single parallel move)

Assembler Syntax:

ABS D
ABS D (single parallel move)

Description: Take the absolute value of the destination operand (D) and store the result in the destination accumulator or 16-bit register. Duplicate destination is not allowed when this instruction is used in conjunction with a parallel read.

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 (that is, 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	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	C

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Set if overflow has occurred in result

Instruction Fields:

Operation	Operands	C	W	Comments
ABS	F	2	1	Absolute value.

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
ABS	F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word

Add Long with Carry

ADC

Operation:

S + C + D → D

Assembler Syntax:

ADC S,D

Description: Add the source operand (S) and the carry bit (C) to the second operand, and store the result in the destination (D). The source operand (register Y) is first sign extended internally to form a 36-bit value before being added to the destination accumulator. When the saturation bit (SA) is set, the MAC Output Limiter is enabled and this instruction will saturate the result if an overflow occurred, (refer to Section 3.4, “Saturation and Data Limiting,” on page 3-26).

Usage: This instruction is typically used in multi-precision addition operations (see Section 3.3.8, “Multi-Precision Operations,” on page 3-23) 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	8000
	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 register 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).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if overflow has occurred in result
E — Set if the signed integer portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if accumulator result is zero; cleared otherwise
V — Set if overflow has occurred in accumulator result
C — Set if a carry (or borrow) occurs from accumulator result

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
ADC	Y,A Y,B	2	1	Add with carry (sets C bit also)

Timing: 2 oscillator clock cycles

Memory: 1 program word

Operation:

S + D →	D	
S + D →	D	(single parallel move)
S + D →	D	(dual parallel read)

Assembler Syntax:

ADD	S,D	
ADD	S,D	(single parallel move)
ADD	S,D	(dual parallel read)

Description: Add the source register to the destination register and store the result in the destination (D). If the destination is a 36-bit accumulator, 16-bit source registers are first sign extended internally and concatenated with 16 zero bits to form a 36-bit operand. When the destination is X0, Y0, or Y1, 16-bit addition is performed. In this case, if the source operand is one of the accumulators, the FF1 portion (properly sign extended) is used in the 16-bit addition; the FF2 and FF0 portions are ignored.

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

A2	A1	A0
0	0100	0000

X0
FFFF

After Execution

A2	A1	A0
0	00FF	0000

X0
FFFF

Explanation of Example:

Prior to execution, the 16-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). Operands of 16 bits 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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the extended portion of the accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Set if overflow has occurred in the result
C — Set if a carry (or borrow) occurs from accumulator result

Instruction Fields:

Operation	Operands	C	W	Comments
ADD	DD,FDD	2	1	36-bit addition of two registers
	F1,DD			
	A,B			
	B,A			
	Y,A			
	Y,B			
	X:(SP-xx),FDD	6	1	Add memory word to register. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22.
	X:aa,FDD	4	1	
	X:xxxx,FDD	6	2	
	FDD,X:(SP-xx)	8	2	Add register to memory word, storing the result back to memory
	FDD,X:xxxx	8	2	
	FDD,X:aa	6	2	
	#<0-31>,FDD	4	1	Add an immediate integer 0–31
	#xxxx,FDD	6	2	Add a signed 16-bit immediate integer

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
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
		X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Parallel Dual Reads:

Data ALU Operation[1]		First Memory Read		Second Memory Read	
Operation[2]	**Operands**	**Source 1**	**Destination 1**	**Source 2**	**Destination 2**
ADD	X0,F Y1,F Y0,F (F = A or B)	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0

1. This parallel instruction is not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

2. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

Timing: 2 + mv oscillator clock cycles for ADD instructions with a single or dual parallel move. Refer to previous table for ADD instructions without a parallel move.

Memory: 1 program word for ADD instructions with a single or dual parallel move. Refer to previous table for ADD instructions without a parallel move.

Operation:

S•D → D
S•D[31:16] → D[31:16]

where • denotes the logical AND operator

Assembler Syntax:

```
AND     S,D
AND     S,D
```

Description: Perform a logical AND operation on the source operand (S) and 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 operation is performed on the source and bits 31–16 of the accumulator. The remaining bits of the destination accumulator are not affected. The result is not affected by the saturation bit (SA).

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

A2	A1	A0
6	1234	5678

X0: 7F00

After Execution

A2	A1	A0
6	1200	5678

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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of accumulator result or MSB of register result is set
Z — Set if bits 31–16 of accumulator result or all bits or register result are zero
V — Always cleared

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
AND	DD,FDD	2	1	16-bit logical AND
	F1,DD			

Timing: 2 oscillator clock cycles

Memory: 1 program word

Operation:

#xxxx•X:<ea> → X:<ea>
#xxxx•D → D

where • denotes the logical AND operator

Assembler Syntax:

```
ANDC    #iiii,X:<ea>
ANDC    #iiii,D
```

Implementation Note:

This instruction is an alias to the BFCLR instruction, and assembles as BFCLR with the 16-bit immediate value inverted (one's-complement) and used as the bit mask. It will disassemble as a BFCLR instruction.

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 in the following discussion. 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

X:$A000	C3FF
SR	0301

After Execution

X:$A000	4155
SR	0300

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$A000 contains the value $C3FF. Execution of the instruction tests the state of bits: 0, 2, 4, 6, 8, 10, 12, and 14 in X:$A000. It clears C (because not all the bits tested were set in destination X:$A000). Result from logical AND is written back to the tested location.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**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 all bits specified by the mask are set
Cleared if *not* all bits specified by the mask are set

Instruction Fields:

Operation	Operands	C	W	Comments
ANDC	#<MASK16>,DDDDD	4	2	AND operation, implemented using BFCLR. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

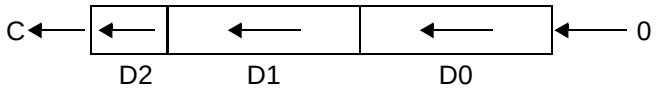
Memory: Refer to the preceding Instruction Fields table

Operation:

(see figure)

Assembler Syntax:

ASL D
ASL D (single parallel move)



(single parallel move)

Description: Arithmetically shift the destination operand (D) 1 bit to the left, and store the result in the destination. 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. A duplicate destination is not allowed when ASL is used in conjunction with a parallel read.

Implementation Note:

When a 16-bit register is specified as the operand for ASL, this instruction is actually assembled as an LSL with the same register argument.

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 1 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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Set if bit 35 of accumulator result is changed due to left shift
C — Set if bit 35 of accumulator was set prior to the execution of the instruction

See Section 3.6.5, “16-Bit Destinations,” Section 3.6.2, “36-Bit Destinations — CC Bit Set,” and Section 3.6.4, “20-Bit Destinations — CC Bit Set.”

Instruction Fields:

Operation	Operands	C	W	Comments
ASL	F	2	1	Arithmetic shift left entire register by 1 bit
	DD			**ALIAS**, refer to Section 6.5.3, “ASL Alias.” Implemented as: LSL DD

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
ASL	F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word

ASLL — Multi-Bit Arithmetic Left Shift — ASLL

Operation:

S1 << S2 → D

Assembler Syntax:

ASLL S1,S2,D

Description: Arithmetically shifts the source operand S1 to the left by the value contained in the lowest 4 bits of S2, and stores the result in the destination (D) with zeros shifted into the LSB. For 36-bit destinations, only the MSP is shifted and the LSP is cleared, with sign extension from bit 31 (the FF2 portion is ignored). The result is not affected by the state of the saturation bit (SA).

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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	V	C

N — Set if MSB of result is set
Z — Set if result equals zero

Note: If the CC bit is set, N is undefined and Z is set if the LSBs 31–0 are zero.

Instruction Fields:

Operation	Operands	C	W	Comments
ASLL	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	Arithmetic shift left of the first operand by value specified in four LSBs of the second operand; places result in FDD

Timing: 2 oscillator clock cycles

Memory: 1 program word

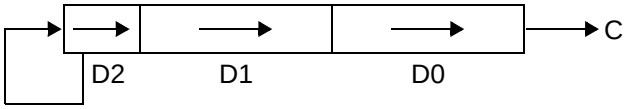
ASR — Arithmetic Shift Right — ASR

Operation:

(see figure)

Assembler Syntax:

ASR D
ASR D (single parallel move)



(single parallel move)

Description: Arithmetically shift the destination operand (D) 1 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. A duplicate destination is not allowed when ASR is used in conjunction with a parallel read.

Example:

```
ASR          B        X:(R2)+,Y0        ; divide B by 2,
                                        ;    update R2, load Y0
```

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 1 bit to the right and stores the result back in the B accumulator. C is set by the operation because bit 0 of B 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 B is set. The E bit of CCR (bit 5) is set because the signed integer portion of B is used by the result.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if data limiting has occurred during parallel move
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Always cleared
C — Set if bit 0 of source operand was set prior to the execution of the instruction

See Section 3.6.5, “16-Bit Destinations,” Section 3.6.2, “36-Bit Destinations — CC Bit Set,” and Section 3.6.4, “20-Bit Destinations — CC Bit Set.”

Instruction Fields:

Operation	Operands	C	W	Comments
ASR	FDD	2	1	Arithmetic shift right entire register by 1 bit

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
ASR	F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word

ASRAC Arithmetic Right Shift with Accumulate ASRAC

Operation:

S1 >> S2 + D → D

Assembler Syntax:

ASRAC S1,S2,D

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 (D). Operand S1 is internally sign extended and concatenated with 16 zero bits to form a 36-bit value before the shift operation. The result is not affected by the state of the saturation bit (SA).

Usage: This instruction is typically used for multi-precision arithmetic right shifts.

Example:

```
ASRAC        Y1,X0,A          ; right shift Y1 by X0 and
                              ;    accumulate in A
```

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). 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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	V	C

N — Set if bit 35 of accumulator result is set
Z — Set if result equals zero

See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
ASRAC	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	2	1	Arithmetic word shifting with accumulation

Timing: 2 oscillator clock cycles

Memory: 1 program word

ASRR Multi-Bit Arithmetic Right Shift ASRR

Operation:

S1 >> S2 → D

Assembler Syntax:

ASRR S1,S2,D

Description: Arithmetically shift the source operand S1 to the right by the value contained in the lowest 4 bits of S2, and store the result in the destination (D). For 36-bit destinations, only the MSP is shifted and the LSP is cleared, with sign extension from bit 31 (the FF2 portion is ignored). The result is not affected by the state of the saturation bit (SA).

Example:

```
ASRR        Y1,X0,A          ; right shift of 16-bit Y1 by X0
```

Before Execution

A2	A1	A0
0	1234	5678

Y1	AAAA
X0	0004

After Execution

A2	A1	A0
F	FAAA	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 ASRR instruction arithmetically shifts the value $AAAA four 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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	V	C

N — Set if MSB of result is set
Z — Set if result equals zero

Instruction Fields:

Operation	Operands	C	W	Comments
ASRR	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	Arithmetic shift right of the first operand by value specified in four LSBs of the second operand; places result in FDD

Timing: 2 oscillator clock cycles

Memory: 1 program word

Operation:

If cc, then PC + label → PC
else PC + 1 → PC

Assembler Syntax:

Bcc <OFFSET7>

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. This instruction is more compact than the Jcc instruction, but can only be used to branch within a small address range

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

$\overline{X}$ denotes the logical complement of X
+ denotes the logical OR operator
• denotes the logical AND operator
⊕ denotes the logical exclusive OR operator

Example:

```
            CMP     X0,A
            BNE     LABEL ; branch to label if Z condition clear
            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 two 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.

Condition Codes Affected:

The condition codes are tested but not modified by this instruction.

Instruction Fields:

Operation	Operands	C[1]	W	Comments
Bcc	<OFFSET7>	6 or 4	1	7-bit signed PC relative offset

1. The clock-cycle count depends on whether the branch is taken. The first value applies if the branch is taken, and the second applies if it is not.

Timing: 4 + jx oscillator clock cycles

Memory: 1 program word

BFCHG Test Bit Field and Change BFCHG

Operation: **Assembler Syntax:**

(<bit field> of destination) → (<bit field> of destination)BFCHG #iiii,X:<ea>
(<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 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, and 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 $0010. Execution of the instruction tests the state of the bits 4, 8, and 9 in X:$FFE2; does not set C (because not all of the bits specified in the immediate mask were 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	*	*	*	*	*	I1	I0	SZ	**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 all bits specified by the mask are set
Cleared if *not* all bits specified by the mask are set

Note: If all bits in the mask are set to zero, the destination is unchanged, and the C bit is set. Refer to Table A-9 on page A-13 when the destination is the SR register.

Instruction Fields:

Operation	Operands	C	W	Comments
BFCHG	#<MASK16>,DDDDD	4	2	BFCHG tests all bits selected by the 16-bit immediate mask. If all selected bits are set, then the C bit is set. Otherwise it is cleared. Then it inverts all selected bits. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

BFCLR Test Bit Field and Clear BFCLR

Operation:

0 →(<bit field> of destination)
0 →(<bit field> of destination)

Assembler Syntax:

BFCLR #iiii,X:<ea>
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 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, and 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 $7F95. Execution of the instruction tests the state of the bits 4, 8, and 9 in X:$FFE2; clears C (because not all of the bits specified in the immediate mask 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	*	*	*	*	*	I1	I0	SZ	**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 all bits specified by the mask are set
Cleared if *not* all bits specified by the mask are set

Note: If all bits in the mask are set to zero, the destination is unchanged, and the C bit is set. Refer to Table A-9 on page A-13 when the destination is the SR register.

Instruction Fields:

Operation	Operands	C	W	Comments
BFCLR	#<MASK16>,DDDDD	4	2	BFCLR tests all bits selected by the 16-bit immediate mask. If all selected bits are set, then the C bit is set. Otherwise it is cleared. Then it clears all selected bits. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

BFSET — Test Bit Field and Set — BFSET

Operation:

1 → (<bit field> of destination)
1 → (<bit field> of destination)

Assembler Syntax:

BFSET #iiii,X:<ea>
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 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 $8921. Execution of the instruction tests the state of bits 10, 12, 13, 14, and 15 in X:$FFE2; does not set C (because not all of the bits specified in the immediate mask were 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	*	*	*	*	*	I1	I0	SZ	**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 all bits specified by the mask are set
Cleared if *not* all bits specified by the mask are set

Note: If all bits in the mask are set to zero, the destination is unchanged, and the C bit is set. Refer to Table A-9 on page A-13 when the destination is the SR register.

Instruction Fields:

Operation	Operands	C	W	Comments
BFSET	#<MASK16>,DDDDD	4	2	BFSET tests all bits selected by the 16-bit immediate mask. If all selected bits are set, then the C bit is set. Otherwise it is cleared. Then it sets all selected bits. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

BFTSTH Test Bit Field High BFTSTH

Operation:

Test <bit field> of destination for ones
Test <bit field> of destination for ones

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, and 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 $0FF0. Execution of the instruction tests the state of bits 4, 8, and 9 in X:$FFE2 and sets C (because all of the bits specified in the immediate mask were set).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	N	Z	V	**C**

L — Set if data limiting occurred during 36-bit source move
C — Set if all bits specified by the mask are set
Cleared if *not* all bits specified by the mask are set

Note: If all bits in the mask are set to zero, the destination is unchanged, and the C bit is set. Refer to Table A-9 on page A-13 when the destination is the SR register.

Instruction Fields:

Operation	Operands	C	W	Comments
BFTSTH	#<MASK16>,DDDDD	4	2	BFTSTH tests all bits selected by the 16-bit immediate mask. If all selected bits are set, then the C bit is set. Otherwise it is cleared. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

BFTSTL Test Bit Field Low BFTSTL

Operation:

Test <bit field> of destination for zeros
Test <bit field> of destination for zeros

Assembler Syntax:

BFTSTL #iiii,X:<ea>
BFTSTL #iiii,D

Description: Test all selected bits of the destination operand. If all selected bits are cleared, 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, and 9
```

Before Execution		After Execution	
X:$FFE2	18EC	X:$FFE2	18EC
SR	0000	SR	0001

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE2 contains the value $18EC. Execution of the instruction tests the state of bits 4, 8, and 9 in X:$FFE2 and sets C (because all of the bits specified in the immediate mask were clear).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	N	Z	V	**C**

L — Set if data limiting occurred during 36-bit source move
C — Set if all bits specified by the mask are cleared
Cleared if *not* all bits specified by the mask are cleared

Note: If all bits in the mask are set to zero, the destination is unchanged, and the C bit is set. Refer to Table A-9 on page A-13 when the destination is the SR register.

Instruction Fields:

Operation	Operands	C	W	Comments
BFTSTL	#<MASK16>,DDDDD	4	2	BFTSTL tests all bits selected by the 16-bit immediate mask. If all selected bits are clear, then the C bit is set. Otherwise it is cleared. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

Operation:

PC+label → PC

Assembler Syntax:

BRA <OFFSET7>

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 two 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.

Instruction Fields:

Operation	Operands	C	W	Comments
BRA	<OFFSET7>	6	1	7-bit signed PC relative offset

Timing: 6+jx oscillator clock cycles

Memory: 1 program word

BRCLR Branch if Bits Cleared BRCLR

Operation:

Branch if <bit field> of destination is all zeros
Branch if <bit field> of destination is all zeros

Assembler Syntax:

BRCLR #iiii,X:<ea>,aa
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 an 8-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   #$0013,X:<<$FFE2,LABEL
            INCW    A
            INCW    A
LABEL:
            ADD     B,A
```

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 $18EC. Execution of the instruction tests the state of bits 4, 1, and 0 in X:$FFE2 and sets C (because all of the bits specified in the immediate mask were clear). Since C is set, program execution is transferred to the address offset from the current program counter by the displacement specified in the instruction (the two INCW instructions are not executed).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	N	Z	V	**C**

L — Set if data limiting occurred during 36-bit source move
C — Set if all bits specified by the mask are cleared
Cleared 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. Refer to Table A-9 on page A-13 when the destination is the SR register.

Instruction Fields:

Operation	Operands	C[1]	W	Comments
BRCLR	#<MASK8>,DDDDD,<OFFSET7>	10/8	2	BRCLR tests all bits selected by the immediate mask. If all selected bits are clear, then the carry bit is set and a PC relative branch occurs. Otherwise it is cleared and no branch occurs. All registers in DDDDD are permitted except HWS. MASK8 specifies a 16-bit immediate value where either the upper or lower 8 bits contains all zeros. AA specifies a 7-bit PC relative offset. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK8>,X:(R2+xx),<OFFSET7>	12/10	2	
	#<MASK8>,X:(SP-xx),<OFFSET7>	12/10	2	
	#<MASK8>,X:aa,<OFFSET7>	10/8	2	
	#<MASK8>,X:<<pp,<OFFSET7>	10/8	2	
	#<MASK8>,X:xxxx,<OFFSET7>	12/10	3	

1. The first cycle count refers to the case when the condition is true and the branch is taken. The second cycle count refers to the case when the condition is false and the branch is not taken.

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

BRSET — Branch if Bits Set — BRSET

Operation:

Branch if <bit field> of destination is all ones
Branch if <bit field> of destination is all ones

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 an 8-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   #$00F0,X:<<$FFE2,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 $0FF0. Execution of the instruction tests the state of bits 4, 5, 6, and 7 in X:$FFE2 and sets C (because all of the bits specified in the immediate mask were set). Since C is set, program execution is transferred to the address offset from the current program counter by the displacement specified in the instruction (the two INCW instructions are not executed)

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	N	Z	V	**C**

L — Set if data limiting occurred during 36-bit source move
C — Set if all bits specified by the mask are set
Cleared 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. Refer to Table A-9 on page A-13 when the destination is the SR register.

Instruction Fields:

Operation	Operands	C[1]	W	Comments
BRSET	#<MASK8>,DDDDD,<OFFSET7>	10/8	2	BRSET tests all bits selected by the immediate mask. If all selected bits are set, then the carry bit is set and a PC relative branch occurs. Otherwise it is cleared and no branch occurs. All registers in DDDDD are permitted except HWS. MASK8 specifies a 16-bit immediate value where either the upper or lower 8 bits contains all zeros. AA specifies a 7-bit PC relative offset. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK8>,X:(R2+xx),<OFFSET7>	12/10	2	
	#<MASK8>,X:(SP-xx),<OFFSET7>	12/10	2	
	#<MASK8>,X:aa,<OFFSET7>	10/8	2	
	#<MASK8>,X:<<pp,<OFFSET7>	10/8	2	
	#<MASK8>,X:xxxx,<OFFSET7>	12/10	3	

1. The first cycle count refers to the case when the condition is true and the branch is taken. The second cycle count refers to the case when the condition is false and the branch is not taken.

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

CLR — Clear Accumulator — CLR

Operation:

0 → D
0 → D (single parallel move)

Assembler Syntax:

CLR D
CLR D (single parallel move)

Description: Set the A or B accumulator to zero. Data limiting may occur during a parallel write. The 16-bit registers are cleared using the MOVE instruction.

Implementation Note:

When a 16-bit register is used as the operand for CLR, this instruction is actually assembled as a `MOVE #0,<register>` instruction. It will disassemble as MOVE instruction.

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	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	C

SZ — Set according to the standard definition of the SZ bit (parallel move)
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).

Instruction Fields:

Operation	Operands	C	W	Comments
CLR	F	2	1	Clear 36-bit accumulator and set condition codes.
	F1DD			**ALIAS**, refer to Section 6.5.4, “CLR Alias.” Implemented as: MOVE #0,<register> (does *not* set condition codes)
	Rj			
	N			

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
CLR	F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word

Operation:

D - S
D - S (single parallel move)

Assembler Syntax:

CMP S,D
CMP S,D (single parallel move)

Description: Subtract the first operand from the second operand and update the CCR without storing the result. If the second operand is a 36-bit accumulator, 16-bit source registers are first sign extended internally and concatenated with 16 zero bits to form a 36-bit operand. When the second operand is X0, Y0, or Y1, 16-bit subtraction is performed. In this case, if the first operand is one of the four accumulators; the FF1 portion (properly sign extended) is used in the 16-bit subtraction (the FF2 and FF0 portions are ignored).

Usage: This instruction can be used for both integer and fractional two's-complement data.

Note: 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

A2	A1	A0
0	0020	0000

Y0	0024
SR	0300

After Execution

A2	A1	A0
0	0020	0000

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).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
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 Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
CMP	DD,FDD	2	1	36-bit compare of two accumulators or data reg
	F1,DD			
	A,B			
	B,A			
	X:(SP-xx),FDD	6	1	Compare memory word with 36 bit accumulator.
	X:aa,FDD	4	1	X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. **Note**: Condition codes set based on 36-bit result
	X:xxxx,FDD	6	2	
	#<0-31>,FDD	4	1	Compare register with an immediate integer 0–31
	#xxxx,FDD	6	2	Compare register with a signed 16-bit immediate integer

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
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
		X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

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 EOnCE port's OCR register, and wait for EOnCE 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.

Instruction Fields:

Operation	Operands	C	W	Comments
DEBUG		4	1	Generate a debug event

Timing: 4 oscillator clock cycles

Memory: 1 program word

DEC(W) Decrement Word DEC(W)

Operation:

D – 1 → D
D – 1 → D (single parallel move)

Assembler Syntax:

DECW D
DECW D (single parallel move)

Description: Decrement a 16-bit destination by one. If the destination is an accumulator, only the EXT and MSP portions of the accumulator are used and the LSP remains unchanged. The condition codes are calculated based on the 16-bit result. Duplicate destination is not allowed when this instruction is used in conjunction with a parallel read.

Usage: This instruction is typically used when processing integer data.

Example:

```
DECW        A       X:(R2)+,X0   ; Decrement the 20 MSBs of A and then
                                 ;    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	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
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 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 Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
DEC or DECW	FDD	2	1	Decrement word
	X:(SP-xx)	8	1	Decrement word in memory using appropriate addressing mode.
	X:aa	6	1	X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22.
	X:xxxx	8	2	

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
DEC or DECW	F (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
		X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: Refer to the preceding Instruction Fields table

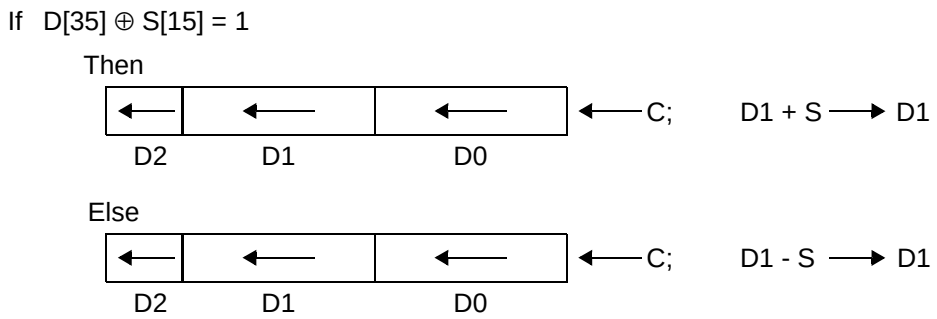
Memory: Refer to the preceding Instruction Fields table

Operation:

(see figure)

Assembler Syntax:

DIV S,D



Description: This instruction is a divide iteration that is used to calculate 1 bit of the result of a division. After the correct number of iterations, this instruction 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 that is stored in the full 36-bit destination accumulator. The 16-bit divisor is a signed value and is stored in the source operand.* (The division of signed numbers is handled using the techniques documented in Section 8.4, “Division,” on page 8-13.) This instruction can be used for both integer and fractional division. Each DIV iteration calculates 1 quotient bit using a non-restoring division algorithm (see the description that follows). After the 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 and is a signed fraction. The formed quotient occupies the low-order portion of the destination accumulator (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 (that is, it is fractional) and precludes division by zero.

The DIV instruction calculates 1 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 that is 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 1 bit to the left. C is moved into the LSB (bit 0) of the accumulator.

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 (FF1 portion), and the result is stored back into the MSP of the destination accumulator. If the result of the exclusive OR operation in the first step was one (that is, the sign bits were different), the source operand S is added to the accumulator. If the result of the exclusive OR operation was zero (that is, 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 the C bit with the next quotient bit.

Usage:

The DIV iteration instruction can be used in one of several different division algorithms, depending on the needs of an application. Section 8.4, “Division,” on page 8-13 shows the correct usage of this instruction for fractional and integer division routines, discusses in detail issues related to division, and provides several examples. The division routine is greatly simplified if both operands are positive, or if it is not necessary also to 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	*	*	*	*	*	I1	I0	SZ	**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

Example:

```
DIV     Y0,A            ; divide A by Y0
```

Before Execution

0	0702	0000
A2	A1	A0
	2000	0004
	Y1	Y0
	SR	0301

After Execution

0	0E00	0001
A2	A1	A0
	2000	0004
	Y1	Y0
	SR	0301

Explanation of Example:

This example shows only a single iteration of the division instruction. Please refer to Section 8.4 for a complete description of a division algorithm.

Instruction Fields:

Operation	Operands	C	W	Comments
DIV	DD,F	2	1	Divide iteration

Timing: 2 oscillator clock cycles

Memory: 1 program word

Start Hardware Do Loop

DO

Operation upon Executing DO Instruction:

HWS[0] → HWS[1]; #xx → LC
PC → HWS[0]; LF → NL; expr → LA
1→ LF

HWS[0] → HWS[1]; S → LC
PC → HWS[0]; LF → NL; expr → LA
1→ LF

Assembler Syntax:

```
DO      #xx,expr
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 DSC 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 (that is, 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 (that is, 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 top of HWS 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 loops can also be nested as shown in Section 8.6, "Loops," on page 8-20. 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 (that is, 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 zeros. The thirteen zeros 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 Section 8.6, "Loops," on page 8-20.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	N	Z	V	C

LF — Set when a DO loop is in progress
L — Set if data limiting occurred

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 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 (that is, 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 DSC 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

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	C	W	Comments
DO	#<1-63>,<ABS16>	6	2	Load LC register with unsigned value and start hardware DO loop with 6-bit immediate loop count. The last address is 16-bit absolute. Loop_count of 0 is not allowed by assembler.
	DDDDD,<ABS16>			Load LC register with unsigned value. If LC is not equal to zero, start hardware DO loop with 16-bit loop count in register. Otherwise, skip body of loop (adds three additional cycles). The last address is 16-bit absolute. Any register allowed except: SP, M01, SR, OMR, and 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's 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
            CMP     Y1,A         ; compare loop counter with Y1
            JNE     CONTINU      ; jump if LC not equal to Y1
            ENDDO                ; equal, restore all DO registers
            JMP     ENDLP        ; jump to ENDLP, continue after loop
CONTINU:                         ; LC not equal to Y1, continue loop
            MOVE    #1,X:$4000   ; (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 (that is, `JMP ENDLP` 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. If desired, this must be done explicitly by the programmer.

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).

Condition Codes Affected:

The condition codes are not affected by this instruction.

Instruction Fields:

Operation	Operands	C	W	Comments
ENDDO		2	1	Remove one value from the hardware stack and update the NL and LF bits appropriately **Note**: Does not branch to the end of the loop

Timing: 2 oscillator clock cycles

Memory: 1 program word

EOR — Logical Exclusive OR — EOR

Operation:

$S \oplus D \rightarrow D$
$S \oplus D[31:16] \rightarrow D[31:16]$

where $\oplus$ denotes the logical exclusive OR operator

Assembler Syntax:

```
EOR     S,D
EOR     S,D
```

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. The result is not affected by the state of the saturation bit (SA).

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

B2	B1	B0
5	5555	6789

Y1: FF00

After Execution

B2	B1	B0
5	AA55	6789

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 ORs 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 bits (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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of accumulator result or MSB of register result is set
Z — Set if bits 31–16 of accumulator result or all bits of register result are zero
V — Always cleared

Instruction Fields:

Operation	Operands	C	W	Comments
EOR	DD,FDD	2	1	16-bit exclusive OR (XOR)
	F1,DD			

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
```

Implementation Note:

This instruction is an alias to the BFCHG instruction, and assembles as BFCHG with the 16-bit immediate value as the bit mask. This instruction will disassemble as a BFCHG instruction.

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        #$0FF0,X:<<$FFE0   ; Exclusive OR with immediate data
```

Before Execution		After Execution	
X:$FFE0	5555	X:$FFE0	5AA5
SR	0301	SR	0300

Explanation of Example:

Prior to execution, the 16-bit X memory location X:$FFE0 contains the value $5555. Execution of the instruction performs a logical XOR of the 16-bit immediate data value ($0FF0) with the destination contents ($5555). In this case, it tests the 8 bits [11:4] in and writes back the result ($5AA5) in destination X:$FFE0. The C bit is cleared because not all of the tested 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	*	*	*	*	*	I1	I0	SZ	**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 all bits specified by the mask are set.

Instruction Fields:

Operation	Operands	C	W	Comments
EORC	#<MASK16>,DDDDD	4	2	Implemented using the BFCHG instruction.
	#<MASK16>,X:(R2+xx)	6	2	All registers in DDDDD are permitted except HWS.
	#<MASK16>,X:(SP-xx)	6	2	X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22.
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK16>,X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

ILLEGAL Illegal Instruction Interrupt ILLEGAL

Operation:

Begin illegal instruction exception routine

Assembler Syntax:

ILLEGAL

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 DSC 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 not interruptible, 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 re-initialized.

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.

Instruction Fields:

Operation	Operands	C	W	Comments
ILLEGAL		4	1	Execute the illegal instruction exception. This instruction is made available so that code may be written to test and verify interrupt handlers for illegal instructions.

Timing: 4 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

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 destination (D). If the destination is an accumulator, the product is stored in the MSP with sign extension while the LSP remains unchanged. The order of the first two operands is not important. The V bit is set if the calculated integer product does not fit into 16 bits.

Usage: This instruction is useful in general computing when it is necessary to multiply two 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 instructions 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 20 bits.

Example:

```
IMPY16      Y0,X0,A            ; form 16-bit product
```

Before Execution

A2	A1	A0
F	AAAA	789A

X0	0003
Y0	0004

After Execution

A2	A1	A0
0	000C	789A

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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	**N**	**Z**	**V**	C

L — Set if overflow has occurred in result
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	C	W	Comments
IMPY or IMPY16	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	Integer 16x16 multiply with 16-bit result. When the destination register is F, the F0 portion is unchanged by the instruction. **Note**: Assembler also accepts first two operands when they are specified in opposite order.

Timing: 2 oscillator clock cycles

Memory: 1 program word

INC(W) Increment Word INC(W)

Operation:

D + 1 → D
D + 1 → D (single parallel move)

Assembler Syntax:

INCW D
INCW D (single parallel move)

Description: Increment a 16-bit destination by one. If the destination is an accumulator, only the EXT and MSP portions of the accumulator are used and the LSP remain unchanged. The condition codes are calculated based on the 16-bit result. Duplicate destination is not allowed when this instruction is used in conjunction with a parallel read

Usage: This instruction is typically used when processing integer data.

Example:

```
INCW        A       X:(R0)+,X0        ; Increment the 20 MSBs of A
                                      ;    update X0 and R0
```

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	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
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 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 Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
INC or INCW	FDD	2	1	Increment word
	X:(SP-xx)	8	1	Increment word in memory using appropriate addressing mode. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22.
	X:aa	6	1	
	X:xxxx	8	2	

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
INC or INCW	F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

Operation:

If (cc), then S → PC
else PC+1→ PC

Assembler Syntax:

Jcc S {<ABS16>}

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 16-bit absolute address. The Bcc instruction, which is more compact, operates almost identically, and can be used for very short jumps.

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	$Z+(N \oplus V)=0$
LE — less than or equal	$Z+(N \oplus V)=1$
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

$\overline{X}$ denotes the logical complement of X
+ denotes the logical OR operator
• denotes the logical AND operator
⊕ denotes the logical exclusive OR operator

Example:

```
            CMP     X0,A
            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 16-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.

Condition Codes Affected:

The condition codes are tested but not modified by this instruction.

Instruction Fields:

Operation	Operands	C[1]	W	Comments
Jcc	<ABS16>	6 or 4	2	16-bit absolute address

1. The clock-cycle count depends on whether the branch is taken. The first value applies if the branch is taken, and the second applies if it is not.

Timing: 4 + jx oscillator clock cycles

Memory: 2 program words

JMP Jump JMP

Operation:

S → PC

Assembler Syntax:

JMP S {<ABS16>}

Description: Jump to program memory at the location given by the instruction's effective address. The effective address is a 16-bit absolute address.

Example:

```
JMP          LABEL
```

Explanation of Example:

In this example, program execution is transferred to the address represented by label. The DSC core supports up to 16-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.

Instruction Fields:

Operation	Operands	C	W	Comments
JMP	<ABS16>	6	2	16-bit absolute address

Timing: 6 + jx oscillator clock cycles

Memory: 2 program words

Operation:

SP+1 → SP
PC → X:(SP)
SP+1 → SP
SR → X:(SP)
S → PC

Assembler Syntax:

JSR S {<ABS16>}

Description: Jump to subroutine in program memory at the location given by the instruction's effective address. The effective address is a 16-bit absolute address.

Example:

```
JSR          LABEL                ; jump to absolute address of a
                                  ;    subroutine indicated by LABEL
```

Explanation of Example:

In this example, program execution is transferred to the subroutine at the address represented by LABEL. The DSC core supports up to 16-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.

Instruction Fields:

Operation	Operands	C	W	Comments
JSR	<ABS16>	8	2	Push return address and status register and jump to 16-bit target address

Timing: 8 + jx oscillator clock cycles

Memory: 2 program words

Load Effective Address

LEA

Operation:

ea → D

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)+N        ; update R0 using (R0)+N
```

Before Execution

R0	8001
N	0C01
M01	1000

After Execution

R0	8C02
N	0C01
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.

Instruction Fields:

Operation	Operands	C	W	Comments
LEA	(Rn)+	2	1	Increment the Rn pointer register
	(Rn)-	2	1	Decrement the Rn pointer register
	(Rn)+N	2	1	Add register N to Rn and store the result in Rn.
	(R2+xx)	2	1	Add a 6-bit unsigned immediate value to R2 and store in the R2 Pointer
	(SP-xx)	2	1	Subtract a 6-bit unsigned immediate value from SP and store in the SP register
	(Rn+xxxx)	4	2	Add a 16-bit signed immediate value to the specified source register.

Timing: 2+ea oscillator clock cycles

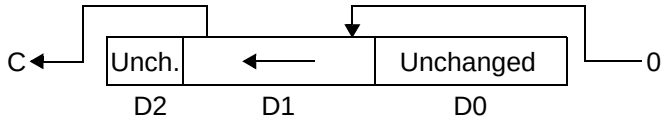
Memory: 1+ea program words

Operation:

(see figure)

Assembler Syntax:

LSL D



Description: Logically shift 16 bits of the destination operand (D) by 1 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 (FF1 portion), and the remaining portions of the accumulator 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). The result is not affected by the state of the saturation bit (SA).

Example:

```
LSL          B                    ; shift 1 bit left
```

Before Execution

B2	B1	B0
6	8000	00AA

SR 0300

After Execution

B2	B1	B0
6	0000	00AA

SR 0305

Explanation of Example:

Prior to execution, the 36-bit B accumulator contains the value $6:8000:00AA. Execution of the `LSL B` instruction shifts the 16-bit value in the B1 register 1 bit to the left and stores the result back in the B1 register. C is set by the operation because bit 31 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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	**V**	**C**

N — Set if bit 31 of accumulator result or MSB or register result is set
Z — Set if the MSP of result or all bits of 16-bit register result are zero
V — Always cleared
C — Set if bit 31 of accumulator or MSB of register was set prior to the execution of the instruction

Instruction Fields:

Operation	Operands	C	W	Comments
LSL	FDD	2	1	1-bit logical shift left of word

Timing: 2 oscillator clock cycles

Memory: 1 program word

LSLL — Multi-Bit Logical Left Shift — LSLL

Operation:

S1 << S2 → D

Assembler Syntax:

LSLL S1,S2,D

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. The LSLL instruction operates identically to an multi bit arithmetic left shift.

Implementation Note:

This instruction is actually implemented by the assembler using the ASLL instruction. It will disassemble as ASLL.

Example:

```
LSLL        Y1,X0,Y1            ; left shift of 16-bit Y1 by X0
```

Before Execution

Y1	AAAA
X0	0004

After Execution

Y1	AAA0
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 four 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	*	*	*	*	*	I1	I0	S	L	E	U	**N**	**Z**	V	C

N — Set if bit 15 of result is set
Z — Set if the result in D is zero

Instruction Fields:

Operation	Operands	C	W	Comments
LSLL	Y1,X0,DD Y0,X0,DD Y1,Y0,DD Y0,Y0,DD A1,Y0,DD B1,Y1,DD	2	1	Logical shift left of the first operand by value specified in four LSBs of the second operand; places result in DD. Use ASLL when left shifting is desired on one of the two accumulators. **ALIAS**, refer to Section 6.5.2, “LSLL Alias.” Implemented as: ASLL <operands>

Timing: 2 oscillator clock cycles

Memory: 1 program word

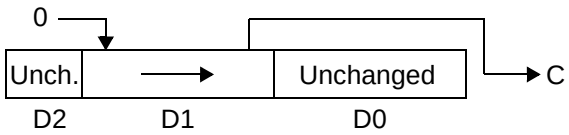
LSR — Logical Shift Right — LSR

Operation:

(see figure)

Assembler Syntax:

LSR D



Description: Logically shift 16 bits of the destination operand (D) by 1 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 (FF1 portion), and the remaining portions of the accumulator 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 31 if the destination is a 36-bit accumulator). The result is not affected by the state of the saturation bit (SA).

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 1 bit to the right and stores the result back in the B1 register. C is set by the operation because bit 16 of B 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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	**V**	**C**

N — Always cleared
Z — Set if the MSP of result or all bits of 16-bit register result are zero
V — Always cleared
C — Set if bit 16 of accumulator or bit 0 of 16-bit register was set prior to the execution of the instruction

Instruction Fields:

Operation	Operands	C	W	Comments
LSR	FDD	2	1	1-bit logical shift right of word

Timing: 2 oscillator clock cycles

Memory: 1 program word

LSRAC Logical Right Shift with Accumulate LSRAC

Operation:

S1 >> S2 + D →D

Assembler Syntax:

LSRAC S1,S2,D

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 (D). Operand S1 is internally zero extended and concatenated with 16 zero bits to form a 36-bit value before the shift operation. The result is not affected by the state of the saturation bit (SA).

Usage: This instruction is used for multi-precision logical right shifts.

Example:

```
LSRAC        Y1,X0,A                    ; 16-bit add
```

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 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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	V	C

N — Set if bit 35 of result is set
Z — Set if result equals zero

See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
LSRAC	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	2	1	Logical word shifting with accumulation

Timing: 2 oscillator clock cycles

Memory: 1 program word

LSRR Multi-Bit Logical Right Shift LSRR

Operation:

S1 >> S2 → D

Assembler Syntax:

LSRR S1,S2,D

Description: Logically shift the source operand S1 to the right by the value contained in the lowest 4 bits of S2, and store the result in the destination (D). For 36-bit destinations, only the MSP is shifted and the LSP is cleared, with zero extension from bit 31 (the FF2 portion is ignored). The result is not affected by the state of the saturation bit (SA).

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 four 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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	V	C

N — Set if MSB of result is set
Z — Set if result equals zero

Instruction Fields:

Operation	Operands	C	W	Comments
LSRR	Y1,X0,FDD Y0,X0,FDD Y1,Y0,FDD Y0,Y0,FDD A1,Y0,FDD B1,Y1,FDD	2	1	Logical shift right of the first operand by value specified in four LSBs of the second operand; places result in FDD (when result is to an accumulator F, zero extends into F2)

Timing: 2 oscillator clock cycles

Memory: 1 program word

MAC — Multiply-Accumulate — MAC

Operation:

± D + S1 * S2 → D
D + S1 * S2 → D (single parallel move)
D + S1 * S2 → D (dual parallel read)

Assembler Syntax:

MAC	(±)S1,S2,D	
MAC	S1,S2,D	(single parallel move)
MAC	S1,S2,D	(dual parallel read)

Description: Multiply the two signed 16-bit source operands, and add or subtract the 32-bit fractional product to or from the destination (D). Both source operands must be located in the FF1 portion of an accumulator or in X0, Y0, or Y1. The fractional product is first sign extended before the 36-bit addition (or subtraction) is performed. If the destination is one of the 16-bit registers, it is first sign extended internally and concatenated with 16 zero bits to form a 36-bit operand before the operation to the fractional product; the high-order 16 bits of the result are then stored.

Usage: This instruction is used for multiplication and accumulation of fractional data or integer data when a full 32-bit product is required (see Section 3.3.5.2, “Integer Multiplication,” on page 3-20). When the destination is a 16-bit register, this instruction is useful only for fractional data.

Example:

```
MAC          X0,Y1,A        X:(R1)+,Y1   X:(R3)+,X0
```

Before Execution

A2	A1	A0
0	0003	0003

X0	4000
Y1	0AA0

After Execution

A2	A1	A0
0	0553	0003

X0	2000
Y1	0450

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 data memory, and the two address registers (R1 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	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	C

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Set if overflow has occurred in result

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
MAC	(±)Y1,X0,FDD (±)Y0,X0,FDD (±)Y1,Y0,FDD (±)Y0,Y0,FDD (±)A1,Y0,FDD (±)B1,Y1,FDD	2	1	Fractional multiply accumulate; multiplication result optionally negated before accumulation

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
MAC	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
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Parallel Dual Reads:

Data ALU Operation[1]		First Memory Read		Second Memory Read	
Operation[2]	**Operands**	**Source 1**	**Destination 1**	**Source 2**	**Destination 2**
MAC	Y1,X0,F Y1,Y0,F Y0,X0,F (F = A or B)	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0

1. This parallel instruction is not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

2. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

Timing: 2 + mv oscillator clock cycles for MAC instructions with parallel move
2 oscillator clock cycles for MAC without parallel move

Memory: 1 program word

MACR Multiply-Accumulate and Round MACR

Operation:

± D + S1 * S2 + r → D
D + S1 * S2 + r → D (single parallel move)
D + S1 * S2 + r → D (dual parallel read)

Assembler Syntax:

MACR	(±)S1,S2,D	
MACR	S1,S2,D	(single parallel move)
MACR	S1,S2,D	(dual parallel read)

Description: Multiply the two signed 16-bit source operands, add or subtract the 32-bit fractional product to or from the third operand, and round and store the result in the destination (D). Both source operands must be located in the FF1 portion of an accumulator or in X0, Y0, or Y1. The fractional product is first sign extended before the 36-bit addition is performed, followed by the rounding operation. If the destination is one of the 16-bit registers, it is first sign extended internally and concatenated with 16 zero bits to form a 36-bit operand before being added to the fractional product. The addition is then followed by the rounding operation, and the high-order 16 bits of the result are then stored. This instruction uses the rounding technique that is selected by the R bit in the OMR. When the R bit is cleared (default mode), convergent rounding is selected; when the R bit is set, two's-complement rounding is selected. Refer to Section 3.5, "Rounding," on page 3-30 for more information about the rounding modes. Note that the rounding operation always zeros the LSP of the result if the destination (D) is an accumulator.

Usage: This instruction is used for the 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 from 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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	C

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of accumulator result is set
Z — Set if result equals zero
V — Set if overflow has occurred in result

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
MACR	(±)Y1,X0,FDD (±)Y0,X0,FDD (±)Y1,Y0,FDD (±)Y0,Y0,FDD (±)A1,Y0,FDD (±)B1,Y1,FDD	2	1	Fractional MAC with round, multiplication result optionally negated before addition.

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
MACR	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
		X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Parallel Dual Reads:

Data ALU Operation[1]		First Memory Read		Second Memory Read	
Operation[2]	**Operands**	**Source 1**	**Destination 1**	**Source 2**	**Destination 2**
MACR	Y1,X0,F Y1,Y0,F Y0,X0,F (F = A or B)	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0

1. This parallel instruction is not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

2. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

Timing: 2 + mv oscillator clock cycles for MACR instructions with a parallel move
2 oscillator clock cycles for MACR instructions without parallel move

Memory: 1 program word

MACSU Multiply-Accumulate Signed × Unsigned MACSU

Operation:

D + S1 * S2 → D (S1 signed, S2 unsigned)

Assembler Syntax:

MACSU S1,S2,D

Description: Multiply one signed 16-bit source operand by one unsigned 16-bit operand, and add the 32-bit fractional product to the destination (D). The order of the registers is important. The first source register (S1) must contain the signed value, and the second source (S2) must contain the unsigned value to produce correct fractional results. The fractional product is first sign extended before the 36-bit addition is performed. If the destination is one of the 16-bit registers, only the high-order 16 bits of the fractional result are stored. The result is not affected by the state of the saturation bit (SA). Note that for 16-bit destinations, the sign bit may be lost for large fractional magnitudes.

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 Section 3.3.8.2, “Multi-Precision Multiplication,” on page 3-23.

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, and 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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	**E**	**U**	**N**	**Z**	**V**	C

L — Set if overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Set if overflow has occurred in result

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

MACSU Multiply-Accumulate Signed × Unsigned MACSU

Instruction Fields:

Operation	Operands	C	W	Comments
MACSU	X0,Y1,FDD X0,Y0,FDD Y0,Y1,FDD Y0,Y0,FDD Y0,A1,FDD Y1,B1,FDD	2	1	Signed or unsigned 16x16 fractional MAC with 32-bit result. The first operand is treated as signed and the second as unsigned.

Timing: 2 oscillator clock cycles

Memory: 1 program word

Description: The DSP56800 Family instruction set contains a powerful set of moves, resulting not only in better DSC performance, but in simpler, more efficient general-purpose computing. The powerful set of controller and DSC 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 preceding move types 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

Description: 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 DSC core registers. These subsets include the registers and addressing modes most frequently found in high performance numeric computation and DSC 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 that 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 (that 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.

Operation:

<op> X:<ea> → D
<op> S → X:<ea>

Assembler Syntax:

<op> X:<ea>,D
<op> S,X:<ea>

<op> refers to any arithmetic instruction that allows parallel moves. A subset of these instructions, include: ADD, DECW, MACR, NEG, SUB and TFR.

Description: Perform a data ALU operation and, in parallel, move the specified register from or 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 DSC 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 6-35, “Data ALU Instructions — Single Parallel Move,” on page 6-29.

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 the following:

```
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
```

Description: 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 the following:

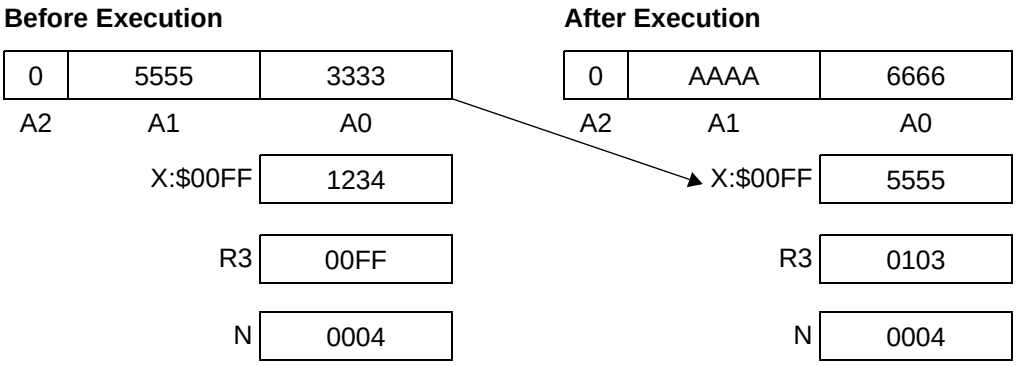
```
ADD         B,A     X:(R2)+,A    ; NOT ALLOWED--A register used twice
                                 ;    as a destination
ASL         A       X:(R2)+,A    ; NOT ALLOWED--A register used twice
                                 ;    as a destination
```

Exceptions:

Instructions TST and CMP 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.

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)+ N`, 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	*	*	*	*	*	I1	I0	**SZ**	**L**	E	U	N	Z	V	C

SZ — Set according to the standard definition of the SZ bit during parallel move
L — Set if data limiting has occurred during parallel move

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
MAC MPY MACR MPYR	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	X:(Rj)+ X:(Rj)+N	X0 Y1 Y0 A B A1 B1
ADD SUB CMP TFR	X0,F Y1,F Y0,F A,B B,A	X0 Y1 Y0 A B A1 B1	X:(Rj)+ X:(Rj)+N
ABS ASL ASR CLR RND TST INC or INCW DEC or DECW NEG	F (F = A or B)	(Rj = R0-R3)	

1. These instructions occupy only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word for all instructions of this type

Operation:

<op> X:<ea1> → D1 X:<ea2> → D2
X:<ea1> → D1 X:<ea2> → D2

Assembler Syntax:

<op> X:<ea1>,D1 X:<ea2>,D2
MOVE X:<ea1>,D1 X:<ea2>,D2

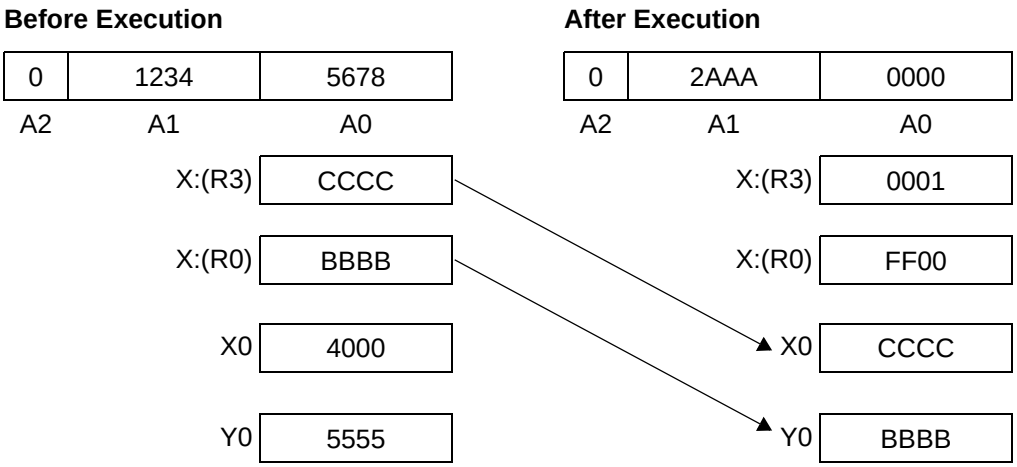
where <op> refers to a limited set of arithmetic instructions which allow double parallel reads

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 be specified. These data ALU instructions have been selected for optimal performance on the critical sections of frequently used DSC 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 6-36, “Data ALU Instructions — Dual Parallel Read,” on page 6-30. 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 rounded using convergent algorithm 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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	N	Z	V	C

L — Set if data limiting has occurred during parallel move

Parallel Dual Reads:

Data ALU Operation[1]		First Memory Read		Second Memory Read	
Operation[2]	**Operands**	**Source 1**	**Destination 1**	**Source 2**	**Destination 2**
MOVE		X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0
MAC MPY MACR MPYR	Y1,X0,F Y1,Y0,F Y0,X0,F (F = A or B)				
ADD SUB	X0,F Y1,F Y0,F (F = A or B)				

1. These parallel instructions are not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

2. These instructions occupy only 1 program word and executes in 1 instruction cycle for every addressing mode.

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:

X:<ea>→ D
S → X:<ea>
S → D

Assembler Syntax:

MOVEC	X:<ea>,D
MOVEC	S,X:<ea>
MOVEC	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 one, 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 one, 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 (that is, sticky).

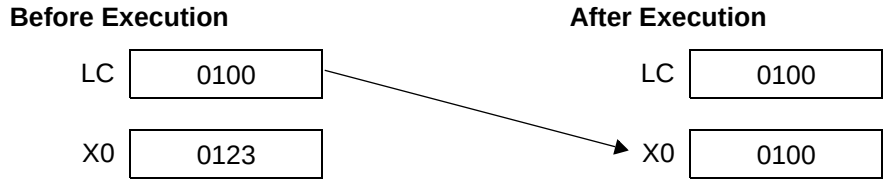
When a 36-bit accumulator (A or B) is specified as a destination operand, any 16-bit source data to be moved into that accumulator is automatically extended to 36 bits by sign extending the MSB of the source operand (bit 15) and appending the source operand with 16 LS zeros. The automatic sign extension 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 (Rj, 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 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.

Example:

```
MOVEC              LC,X0        ; move the LC register into
                                ;    the X0 register
```

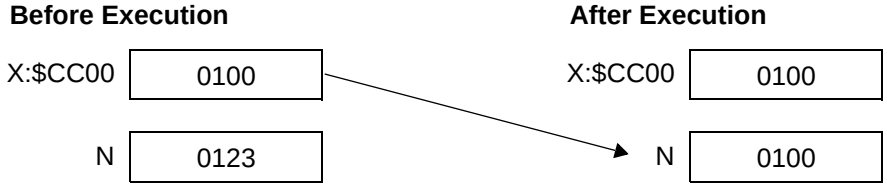


Explanation of Example:

Execution of the MOVEC instruction moves the contents of the program controller's 13-bit LC register into the data ALU's 16-bit X0 register.

Example:

```
MOVEC              X:$CC00,N    ; move X data memory value into the
                                ;    N register
```

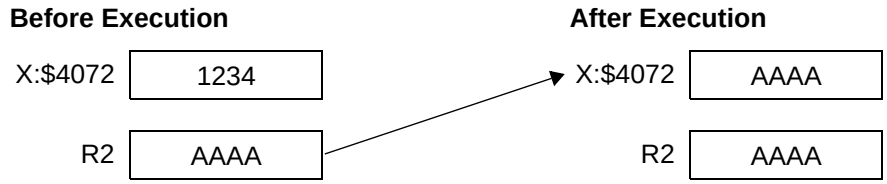


Explanation of Example:

Execution of the MOVEC instruction moves the contents of the X data memory at location $CC00 into the AGU's 16-bit N register.

Example:

```
MOVEC              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 MOVEC instruction moves the AGU's 16-bit R2 register contents into the X data memory at the location $4072.

Restrictions:

A MOVEC instruction used within a DO loop that specifies the HWS as the source or that specifies the SR or HWS as the destination cannot begin at the LA-2, LA-1, or LA within that DO loop.
A MOVEC instruction that specifies the HWS as the source or as the destination cannot be used immediately before a DO instruction.
A MOVEC instruction that specifies the HWS as the source or that specifies the SR or HWS as the destination cannot be used immediately before an ENDDO instruction.
A MOVEC instruction that specifies the SR, HWS, or SP as the destination cannot be used immediately before an RTI or RTS instruction.
A MOVEC HWS,HWS instruction is illegal and cannot be used.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

If D is the SR:

SZ — Set according to bit 7 of the source operand
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 SR is not the destination:

L — Set if data limiting has occurred during move

Instruction Fields:

Operation	Source	Destination	C	W	Comments
MOVE or MOVEC	X:(Rn) X:(Rn)+ X:(Rn)- X:(Rn)+N	DDDDD	2	1	—
	X:xxxx	DDDDD	4	2	16-bit absolute address
	X:(Rn+N)	DDDDD	4	1	—
	X:(Rn+xxxx)	DDDDD	6	2	Signed 16-bit index
	X:(R2+xx) X:(SP-xx)	X0, Y1, Y0, A, B, A1, B1 R0–R3, N	4	1	—

Instruction Fields:

Operation	Source	Destination	C	W	Comments
MOVE or MOVEC	DDDDD	X:(Rn) X:(Rn)+ X:(Rn)- X:(Rn)+N	2	1	—
	DDDDD	X:xxxx	4	2	16-bit absolute address
	DDDDD	X:(Rn+N)	4	1	
	DDDDD	X:(Rn+xxxx)	6	2	
	X0, Y1, Y0, A, B, A1, B1 R0–R3, N	X:(R2+xx) X:(SP-xx)	4	1	
	DDDDD	DDDDD	2	1	Move signed word to register

Timing: 2 + mvc oscillator clock cycles

Memory: 1 + ea program words

Operation:

#xx → D
#xxxx → D
#xxxx → X:<ea>

Assembler Syntax:

MOVEI	#xx,D
MOVEI	#xxxx,D
MOVEI	#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:

```
MOVEI        #<$FFC7,X0          ; moves negative value into X0
```

Before Execution

X0 | 1234 |

After Execution

X0 | FFC7 |

Explanation of Example:

Prior to execution, X0 contains the value $1234. Execution of the instruction moves the value $FFC7 into X0.

Example:

```
MOVEI        #$C33C,X:$A009      ; moves 16-bit value directly into a
                                 ;    memory location
```

Before Execution

X:$A009 | 1234 |

After Execution

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 MOVEP and MOVES 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.

Instruction Fields:

Operation	Source	Destination	C	W	Comments
MOVE or MOVEI	#<-64-63>	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	2	1	Signed 7-bit integer data (data is put in the lowest 7 bits of the word portion of any accumulator, upper 9 bits and extension reg are sign extended, LSP portion is set to “0”)
	#xxxx	DDDDD	4	2	Signed 16-bit immediate data. When LC is the destination, use 13-bit values only.
		X:(R2+xx)	6	2	
		X:(SP-xx)	6	2	
		X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

MOVE(M) Move Program Memory MOVE(M)

Operation:

P:<ea> → D
S→ P:<ea>

Assembler Syntax:

MOVEM	P:<ea>,D
MOVEM	S,P:<ea>

Description: Move the specified register from or 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 (that 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:

```
MOVEM                P:(R2)+N,A          ; move P:(R2) into A,
                                         ;   update R2 with N
```

Before Execution

	A2	A1	A0
A	1234	5678	
P:$0077	0116		
R2	$0077		

After Execution

	A2	A1	A0
0	0116	0000	
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 MOVEM instruction moves the 16-bit program memory location P:(R2) into the 36-bit A accumulator. R2 is then post-incremented by N.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	N	Z	V	C

L — Set if data limiting has occurred during the move

Instruction Fields:

Operation[1]	Source	Destination	C	W	Comments
MOVE or MOVEM	P:(Rj)+ P:(Rj)+N	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	8	1	Read signed word from program memory
	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	P:(Rj)+ P:(Rj)+N			Write word to program memory

1. These instructions are not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

Timing: 8 + mvm oscillator clock cycles

Memory: 1 program word

MOVE(P) Move Peripheral Data MOVE(P)

Operation:

X:<pp> → D
S → X:<pp>
#xxxx → X:<pp>

Assembler Syntax:

MOVEP	X:<pp>,D
MOVEP	S,X:<pp>
MOVEP	#xxxx,X:<pp>

Description: Move the specified operand to or from a location in the last 64 words of the X data memory map. The 6-bit short absolute address is one-extended to generate a 16-bit address.

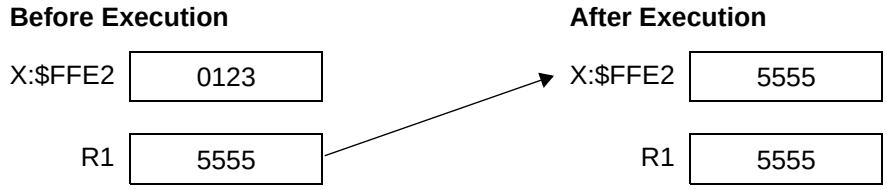
When a 36-bit accumulator (A or B) is specified as a source operand, there is a possibility that the data may be limited. If the data out of the shifter indicates that the accumulator extension register is in use, and the data is to be moved into a 16-bit destination, the value stored in the destination 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 (that 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).

Usage: This MOVEP instruction provides a more efficient way of accessing the last 64 locations in X memory, which may be allocated to memory-mapped peripheral registers. If located outside the X:$FFC0-X:$FFFF range, use other suitable addressing mode. Consult the specific DSP56800-based device's user manual for information on where in the memory map peripheral registers are located.

Example:

```
MOVEP        R1,X:<<$FFE2  ; write to location X:$FFE2
```



Explanation of Example:

Prior to execution, the peripheral location <<$FFE2 contains the value $0123. Execution of the `MOVEP R1,X:<<$FFE2` instruction moves the value $5555 contained in the R1 register into the location.

Example:

```
MOVEP        #$0342,X:$FFE4          ; moves 16-bit value into
                                     ;   peripheral location $FFE4
```

Explanation of Example:

Prior to execution, the word at X data memory location $FFE4 contains the value $AAAA. Execution of the instruction moves the value $0342 into this location. Note that $FFE4 is recognized as a peripheral mapped register.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**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 MOVEC 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[1]	Source	Destination	C	W	Comments
MOVE or MOVEP	X:pp or X:<<pp	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	2	1	Last 64 locations in data memory. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	X:pp or X:<<pp			
	#xxxx		4	2	Move 16-bit immediate data to the last 64 locations of X data memory-peripheral registers. X:<<pp represents a 6-bit absolute I/O address.

1. The MOVEP instruction provides a more efficient way of accessing the last 64 locations in X memory, which may be allocated to memory-mapped peripheral registers. If peripheral registers are located outside the X:$FFC0-X:$FFFF range, use other suitable addressing mode. Consult the specific DSP56800-based device's user manual for information on where in the memory map peripheral registers are located.

Timing: 2 + ea oscillator clock cycles

Memory: 1 + ea program word

MOVE(S) Move Absolute Short MOVE(S)

Operation:

X:<aa> → D
S → X:<aa>
#xxxx → X:<aa>

Assembler Syntax:

MOVES	X:<aa>,D
MOVES	S,X:<aa>
MOVES	#xxxx,X:<aa>

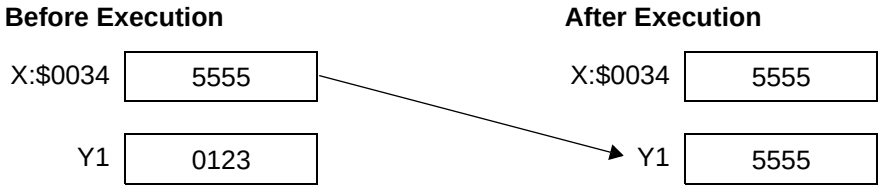
Description: Move the specified operand from or to the first 64 memory locations in X data memory. The 6-bit absolute short address is zero-extended to generate a 16-bit X data memory address.

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 (that 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
```

Explanation of Example:

Prior to execution, the contents of the X data memory location $24 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. Note that address $24 is recognized as a candidate for short addressing.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	E	U	N	Z	V	C

SZ — Set according to the standard definition of the SZ bit
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 MOVEC 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	Source	Destination	C	W	Comments
MOVE or MOVES	X:aa or X:<aa	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	2	1	First 64 locations in data memory. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22.
	A, B, A1, B1 X0, Y0, Y1 R0-R3, N	X:aa or X:<aa			
	#xxxx		4	2	Move 16-bit immediate date to a location within the first 64 words of X data memory. X:aa represents a 6-bit absolute address.

Timing: 2 + ea oscillator clock cycles

Memory: 1 + ea program word

MPY — Signed Multiply — MPY

Operation:

± S1 * S2 → D
S1 * S2 → D (single parallel move)
S1 * S2 → D (dual parallel read)

Assembler Syntax:

MPY	(±)S1,S2,D	
MPY	S1,S2,D	(single parallel move)
MPY	S1,S2,D	(dual parallel read)

Description: Multiply the two signed 16-bit source operands, and place the 32-bit fractional product in the destination (D). Both source operands must be located in the FF1 portion of an accumulator or in X0, Y0, or Y1. If an accumulator is used as the destination, the result is sign extended into the extension portion (FF2) of the accumulator. If the destination is one of the 16-bit registers, only the higher 16 bits of the fractional product are stored.

Usage: This instruction is used for multiplication of fractional data or integer data when a full 32-bit product is required (see Section 3.3.5.2, “Integer Multiplication,” on page 3-20). 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
```

Before Execution

A2	A1	A0
0	1000	0000

X0	4000
Y1	F456

After Execution

A2	A1	A0
F	FA2B	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 (-0.09112), and the 36-bit A accumulator contains the value $0:1000:0000 (0.125). Execution of the `MPY X0,Y1,A` instruction multiplies the 16-bit signed value in the X0 register by the 16-bit signed value in Y1 and stores the result ($F:FA2B:0000) into the A accumulator, X0 * Y1 = -0.04556 (truncated here to 5 decimal places).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	C

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow (result) has occurred
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Always cleared

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
MPY	(±)Y1,X0,FDD (±)Y0,X0,FDD (±)Y1,Y0,FDD (±)Y0,Y0,FDD (±)A1,Y0,FDD (±)B1,Y1,FDD	2	1	Fractional multiply where one operand is optionally negated before multiplication **Note**: Assembler also accepts first two operands when they are specified in opposite order

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
MPY	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
		X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Parallel Dual Reads:

Data ALU Operation[1]		First Memory Read		Second Memory Read	
Operation[2]	**Operands**	**Source 1**	**Destination 1**	**Source 2**	**Destination 2**
MPY	Y1,X0,F Y1,Y0,F Y0,X0,F (F = A or B)	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0

1. This parallel instruction is not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

2. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

Timing: 2 + mv oscillator clock cycles for MPY instructions with parallel move
2 oscillator clock cycles for MPY instructions without parallel move

Memory: 1 program word

MPYR Signed Multiply and Round MPYR

Operation:

± S1 * S2 + r → D
S1 * S2 + r → D (single parallel move)
S1 * S2 + r → D (dual parallel read)

Assembler Syntax:

MPYR	(±)S1,S2,D	
MPYR	S1,S2,D	(single parallel move)
MPYR	S1,S2,D	(dual parallel read)

Description: Multiply the two signed 16-bit source operands, round the 32-bit fractional product, and place the result in the destination (D). Both source operands must be located in the FF1 portion of an accumulator or in X0, Y0, or Y1. The fractional product is sign extended before the rounding operation, and the result is then stored in the destination. If the destination is one of the 16-bit registers, only the high-order 16 bits of the rounded fractional result are stored. This instruction uses the rounding technique that is selected by the R bit in the OMR. When the R bit is cleared (default mode), convergent rounding is selected; when the R bit is set, two's-complement rounding is selected. Refer to Section 3.5, "Rounding," on page 3-30 for more information about the rounding modes. Note that the rounding operation will always zero the LSP of the result if the destination (D) is an accumulator.

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
0	05D5	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 (-0.09112), and the 36-bit A accumulator contains the value $00:1000:1234 (0.12500). 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 ($0:05D5:0000) into the A accumulator, -X0 * Y1 = 0.04556 (truncated here to 5 decimal places). 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	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	C

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Always cleared

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
MPYR	(±)Y1,X0,FDD (±)Y0,X0,FDD (±)Y1,Y0,FDD (±)Y0,Y0,FDD (±)A1,Y0,FDD (±)B1,Y1,FDD	2	1	Fractional multiply where one operand is optionally negated before multiplication; result is rounded **Note**: Assembler also accepts first two operands when they are specified in opposite order

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
MPYR	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F (F = A or B)	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
		X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Parallel Dual Reads:

Data ALU Operation[1]		First Memory Read		Second Memory Read	
Operation[2]	**Operands**	**Source 1**	**Destination 1**	**Source 2**	**Destination 2**
MPYR	Y1,X0,F Y1,Y0,F Y0,X0,F (F = A or B)	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0

1. This parallel instruction is not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

2. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

Timing: 2 + mv oscillator clock cycles for MPYR instructions with parallel move
2 oscillator clock cycles for MPYR instructions without parallel move

Memory: 1 program word

MPYSU Signed Unsigned Multiply MPYSU

Operation:

S1 * S2 → D (S1 signed, S2 unsigned)

Assembler Syntax:

MPYSU S1,S2,D

Description: Multiply one signed 16-bit source operand by one unsigned 16-bit operand, and place the 32-bit fractional product in the destination (D). The order of the registers is important. The first source register (S1) must contain the signed value, and the second source (S2) must contain the unsigned value to produce correct fractional results. If the destination is one of the 16-bit registers, only the high-order 16 bits of the fractional result are stored. The result is not affected by the state of the saturation bit (SA). Note that for 16-bit destinations, the sign bit may be lost for large fractional magnitudes.

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 Section 3.3.8.2, “Multi-Precision Multiplication,” on page 3-23.

Example:

```
MPYSU        X0,Y0,A
```

Before Execution

A2	A1	A0
0	0000	0000

X0	3456
Y0	8000

After Execution

A2	A1	A0
0	3456	0000

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 `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 an MPYSU instruction, Y0 is considered unsigned and equals +1.0. This gives a multiplication result of $0:3456:0000.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	**E**	**U**	**N**	**Z**	**V**	C

L — Set if overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Always cleared

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
MPYSU	X0,Y1,FDD X0,Y0,FDD Y0,Y1,FDD Y0,Y0,FDD Y0,A1,FDD Y1,B1,FDD	2	1	Signed or unsigned 16x16 fractional multiply with 32-bit result. The first operand is treated as signed and the second as unsigned.

Timing: 2 oscillator clock cycles

Memory: 1 program word

NEG — Negate Accumulator — NEG

Operation:

0 - D → D
0 - D → D (single parallel move)

Assembler Syntax:

NEG D
NEG D (single parallel move)

Description: The destination operand (D) is subtracted from zero, and the two's-complement result is stored in the destination (D). If the destination is a 16-bit register, it is first sign extended internally and concatenated with 16 zero bits to form a 36-bit operand.

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 Section 8.1.6, "Unsigned Load of an Accumulator," on page 8-7).

Example:

```
NEG          B        X0,X:(R3)+          ; 0-B → B, save X0, update R3
```

Before Execution

B2	B1	B0
0	1234	5678

SR 0300

After Execution

B2	B1	B0
F	EDCB	A988

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	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the extension portion of accumulator is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Set if overflow has occurred in accumulator result
C — Set if a borrow is generated from the MSB of the result

See Section 3.6.2, "36-Bit Destinations — CC Bit Set," on page 3-34 and Section 3.6.4, "20-Bit Destinations — CC Bit Set," on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
NEG	F	2	1	Two's-complement negation

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
NEG	F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

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.

Instruction Fields:

Operation	Operands	C	W	Comments
NOP		2	1	No operation

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 R0 - 1 → R0
else if (E = 1)
then ASR D and R0 + 1→ R0
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 not normalized, and the accumulator is not zero, then the destination operand is arithmetically shifted 1 bit to the left, and the R0 address register is decremented by one. If the accumulator extension register is in use, the destination operand is arithmetically shifted 1 bit to the right, and the R0 address register is incremented by one. If the accumulator is normalized or zero, a NOP is executed, and the R0 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

A2	A1	A0
0	0000	8000

R0	
	0000

After Execution

A2	A1	A0
0	4000	0000

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, 15 left shifts are required for normalization.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	**E**	**U**	**N**	**Z**	**V**	C

L — Set if overflow has occurred in accumulator result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of accumulator result is set
Z — Set if result equals zero
V — Set if bit 35 is changed as a result of a left shift

See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
NORM	R0,A R0,B	2	1	Normalization iteration instruction for normalizing the F accumulator

Timing: 2 oscillator clock cycles

Memory: 1 program word

NOT — Logical Complement — NOT

Operation:

$\overline{D} \rightarrow D$
$\overline{D}[31:16] \rightarrow D[31:16]$

where the bar over the D ($\overline{D}$) denotes the logical NOT operator

Assembler Syntax:

```
NOT     D
NOT     D
```

Description: Compute 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. The result is not affected by the state of the saturation bit (SA).

Example:

```
NOT A        A,X:(R2)+          ; save A1 and take the 1's complement
                                ;     of A1
```

Before Execution

A2	A1	A0
5	1234	5678

SR: 0300

After Execution

A2	A1	A0
5	EDCB	5678

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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of accumulator result or MSB of register result is set
Z — Set if bits 31–16 of accumulator result or all 16 bits or register are zero
V — Always cleared

See Section 3.6.5, "16-Bit Destinations," on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, "36-Bit Destinations — CC Bit Set," on page 3-34 and Section 3.6.4, "20-Bit Destinations — CC Bit Set," on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
NOT	FDD	2	1	One's-complement (bit-wise negation)

Timing: 2 oscillator clock cycles

Memory: 1 program word

NOTC — Logical Complement with Carry — NOTC

Operation:

$\overline{X:<ea>} \rightarrow X:(ea)$
$\overline{D} \rightarrow D$

Assembler Syntax:

NOTC	X:<ea>
NOTC	D

Implementation Note:

This instruction is an alias to the BFCHG instruction, and assembles as BFCHG with the 16-bit immediate mask set to $FFFF. This instruction will disassemble as a BFCHG instruction.

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 in following discussion.

Example:

```
NOTC        R2
```

Before Execution

R2	CAA3
SR	3456

After Execution

R2	355C
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 in following discussion.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**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 value equals $FFFF before the complement

Instruction Fields:

Operation	Operands	C	W	Comments
NOTC	DDDDD	4	2	One's-complement (bit-wise negation). Implemented with BFCHG All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	X:(R2+xx)	6	2	
	X:(SP-xx)	6	2	
	X:aa	4	2	
	X:<<pp	4	2	
	X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

OR — Logical Inclusive OR — OR

Operation:

S + D → D
S + D[31:16] → D[31:16]

where + denotes the logical inclusive OR operator

Assembler Syntax:

```
OR      S,D
OR      S,D
```

Description: Perform a logical OR operation on 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 OR operation is performed on the source with bits 31–16 of the accumulator. The remaining bits of the destination accumulator are not affected. The result is not affected by the state of the saturation bit (SA).

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

B2	B1	B0
0	1234	5678

Y1: FF00

After Execution

B2	B1	B0
0	FF34	5678

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 ORs 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	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	**V**	C

N — Set if bit 31 of accumulator result or MSB or register result is set
Z — Set if bits 31–16 of accumulator result or all 16 bits or register are zero
V — Always cleared

Instruction Fields:

Operation	Operands	C	W	Comments
OR	DD,FDD	2	1	16-bit logical OR
	F1,DD			

Timing: 2 oscillator clock cycles

Memory: 1 program word

Logical Inclusive OR Immediate

ORC

Operation:

#xxxx + X:<ea> → X:<ea>
#xxxx + D → D

where + denotes the logical inclusive OR operator

Assembler Syntax:

ORC	#iiii,X:<ea>
ORC	#iiii,D

Implementation Note:

This instruction is an alias to the BFSET instruction, and assembles as BFSET with the 16-bit immediate value used as the bit mask. This instruction will disassemble as a BFSET instruction.

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 in following discussion. This instruction performs a read-modify-write operation on the destination and requires two destination accesses.

Example:

```
ORC          #$5050,X:<<$7C30          ; OR with immediate data
```

Before Execution

X:$7C30	00AA
SR	0300

After Execution

X:$7C30	50FA
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 14, 12, 6, and 4 in X:$7C30; 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	*	*	*	*	*	I1	I0	SZ	**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 all bits specified by the mask are set

Instruction Fields:

Operation	Operands	C	W	Comments
ORC	#<MASK16>,DDDDD	4	2	16-bit logical OR of immediate data. Implemented with BFSET. All registers in DDDDD are permitted except HWS. X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22. X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	#<MASK16>,X:(R2+xx)	6	2	
	#<MASK16>,X:(SP-xx)	6	2	
	#<MASK16>,X:aa	4	2	
	#<MASK16>,X:<<pp	4	2	
	#<MASK16>,X:xxxx	6	3	

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

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.

Implementation Note:

This instruction is implemented by the assembler using either a MOVE or LEA instruction, depending on the form. When a destination register is specified, a `MOVE X:(SP)-,<register>` instruction is assembled. When no destination register is specified, POP assembles as `LEA (SP)-`. The instruction will always disassemble as either MOVE or LEA.

Example:

```
POP         LC
```

Before Execution

X:$0100	AAAA
LC	0099
SP	0100

After Execution

X:$0100	AAAA
LC	AAAA
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.

Instruction Fields:

Operation	Operands	C	W	Comments
POP	DDDDD	2	1	Pop a single stack location. **ALIAS**, refer to Section 6.5.5, “POP Alias.” Implemented as: MOVE X:(SP)-,<register>
	(None specified)			**ALIAS**, refer to Section 6.5.5, “POP Alias.” Implemented as: LEA (SP)-

Timing: 2 oscillator clock cycles

Memory: 1 program word

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 (that is, 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 DSC 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 13 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 zeros. The 13 zeros are loaded into the LC register, specifying a loop count of zero. The A and B accumulators remain unchanged.

Note: Once in progress, the REP instruction and the REP loop may not be interrupted 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 instruction that occupies multiple words

DO	Bcc, Jcc
BRCLR, BRSET	BRA, JMP
MOVEM	JSR
REP	RTI
RTS	STOP, WAIT
SWI, DEBUG	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 preceding instructions are found immediately following a REP instruction.

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 13-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 13-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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	E	U	N	Z	V	C

L — Set if data limiting occurred using accumulator as source operand

Instruction Fields:

Operation	Operands	C	W	Comments
REP	#<0-63>	6	1	Hardware repeat of a one-word instruction with immediate loop count
	DDDDD	6	1	Hardware repeat of a one-word instruction with loop count specified in register Any register allowed except: SP, M01, SR, OMR, and HWS

Timing: 6 oscillator clock cycles

Memory: 1 program word

Operation:			Assembler Syntax:		
D + r →	D		RND	D	
D + r →	D	(single parallel move)	RND	D	(single parallel move)

Description: Round the 36-bit or 32-bit value in the specified destination operand (D). If the destination is an accumulator, store the result in the EXT:MSP portions of the accumulator and clear the LSP. This instruction uses the rounding technique that is selected by the R bit in the OMR. When the R bit is cleared (default mode), convergent rounding is selected; when the R bit is set, two's-complement rounding is selected. Refer to Section 3.5, "Rounding," on page 3-30 for more information about the rounding modes.

Example:

```
RND          A                    ; round A accumulator into
                                  ;    A2:A1, zero A0
```

Before Execution

	A2	A1	A0
I	5	1236	789A
II	0	1236	8000
III	0	1235	8000

After Execution

	A2	A1	A0
I	5	1236	0000
II	0	1236	0000
III	0	1236	0000

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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	C

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of accumulator result is set
Z — Set if result equals zero
V — Set if overflow has occurred in accumulator 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 equal zero, then Z is set. The rest of the bits are unaffected by the setting of the CC bit.

Instruction Fields:

Operation	Operands	C	W	Comments
RND	F	2	1	Round

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
RND	F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word

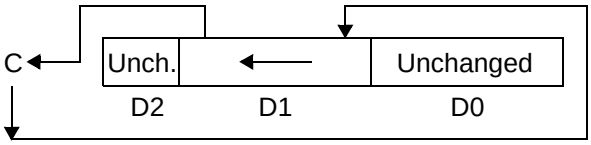
ROL — Rotate Left — ROL

Operation:

(see figure)

Assembler Syntax:

ROL D



Description: Logically shift 16 bits of the destination operand (D) 1 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 (FF1 portion), and the remaining portions of the accumulator are not modified. The MSB of the destination (bit 31 for accumulators or bit 15 for registers) 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). The result is not affected by the state of the saturation bit (SA).

Example:

```
ROL          B                          ; rotate B1 left 1 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 B accumulator contains the value $F:0001:00AA. Execution of the `ROL B` instruction shifts the 16-bit value in the B1 register 1 bit to the left, shifting bit 31 into C, rotating C into bit 16, and storing the result back in the B1 register.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	**V**	**C**

N — Set if bit 31 of accumulator result or bit 15 of register result is set
Z — Set if bits 31–16 of accumulator result or all bits in register result are zero
V — Always cleared
C — Set if bit 31 of accumulator or bit 15 or register result was set prior to the execution of the instruction

Instruction Fields:

Operation	Operands	C	W	Comments
ROL	FDD	2	1	Rotate 16-bit register left by 1 bit through the carry bit

Timing: 2 oscillator clock cycles

Memory: 1 program word

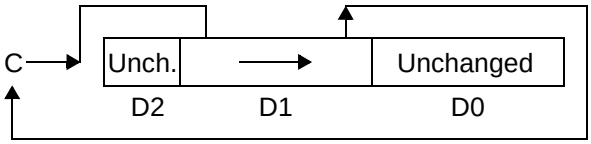
ROR — Rotate Right — ROR

Operation:

(see figure)

Assembler Syntax:

ROR D



Description: Logically shift 16 bits of the destination operand (D) 1 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 (FF1 portion), and the remaining portions of the accumulator are not modified. The LSB of the destination (bit 16 for 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 for a 36-bit accumulator). The result is not affected by the state of the saturation bit (SA).

Example:

```
ROR          B                    ; rotate B1 right 1 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 1 bit to the right, shifting bit 16 into C, rotating C into bit 31, and storing the result back in the B1 register.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	L	E	U	**N**	**Z**	**V**	**C**

N — Set if bit 31 of accumulator result or MSB of register is set
Z — Set if bits 31–16 of accumulator result or all bits of register are zero
V — Always cleared
C — Set if bit 16 of accumulator or bit 0 of register was set prior to the execution of the instruction

Instruction Fields:

Operation	Operands	C	W	Comments
ROR	FDD	2	1	Rotate 16-bit register right by 1 bit through the carry bit

Timing: 2 oscillator clock cycles

Memory: 1 program word

Return from Interrupt

RTI

Operation:

X:(SP) → SR; SP-1→ SP
X:(SP) → PC; SP-1→ SP

Assembler Syntax:

RTI

Description: Return to normal execution at the end of an interrupt service routine. The return restores the status register (SR) and program counter (PC) from the software stack. The previous PC is lost, and execution resumes at the address that is indicated by the (restored) PC.

Example:

```
RTI                                  ; pull the SR and PC registers
                                     ;    from the stack
```

Before Execution

X:$0100	1300
X:$00FF	754C
SR	0309
SP	0100

After Execution

X:$0100	1300
X:$00FF	754C
SR	1300
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.

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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	**I1**	**I0**	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

All bits — Set according to the value pulled from the stack

Instruction Fields:

Operation	Operands	C	W	Comments
RTI		10	1	Return from interrupt, restoring 16-bit PC and SR from the stack

Timing: 10 + rx oscillator clock cycles

Memory: 1 program word

RTS — Return from Subroutine — RTS

Operation:

X:(SP) → (stored SR; discarded); SP-1→ SP
X:(SP) → PC; SP-1→ SP

Assembler Syntax:

RTS

Description: Return from a call to a subroutine. To perform the return, RTS pulls and discards the previously pushed SR and pops the PC from the software stack. The previous PC is lost. The generated SR from the called function is not affected.

Example:

```
RTS                                    ; pull SR (and discard it) &
                                       ; pull PC from the stack
```

Before Execution

X:$0100	8000
X:$00FF	754C
SR	8009
SP	0100

After Execution

X:$0100	8000
X:$00FF	754C
SR	8009
SP	00FE

Explanation of Example:

The example makes the assumption that during entry of the subroutine, only the LF bit (SR bit 15) is on. During execution of the subroutine, the C and N bits were set. To perform the return, RTS pops the 16-bit PC from the software stack, and updates the SP. Program execution continues at $754C.

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 the following instruction:

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.

Manipulation of bits 10-14 in the stack location corresponding to the SR register may generate unwanted behavior. These bits will read as zero during DSC read operations and should be written as zero to ensure future compatibility.

Condition Codes Affected:

The condition codes are not affected by this instruction.

Instruction Fields:

Operation	Operands	C	W	Comments
RTS		10	1	Return from subroutine, restoring 16-bit PC from the stack

Timing: 10 + rx oscillator clock cycles

Memory: 1 program word

Subtract Long with Carry

SBC

Operation:

D - S - C → D

Assembler Syntax:

SBC S,D

Description: Subtract the source operand (S) and the carry bit (C) from the second operand, and store the result in the destination (D). The source operand (S) is always register Y, which is first sign extended internally to form a 36-bit value before being subtracted from the destination accumulator. When the saturation bit (SA) is set, the MAC Output Limiter is enabled and this instruction will saturate the result if an overflow occurred, (refer to Section 3.4, “Saturation and Data Limiting,” on page 3-26).

Usage: This instruction is typically used in multi-precision subtraction operations (see Section 3.3.8.1, “Multi-Precision Addition and Subtraction,” on page 3-23) when it is necessary to subtract 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).

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	SZ	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

L — Set if overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of accumulator result is set
Z — Set if result equals zero; cleared otherwise
V — Set if overflow has occurred in result
C — Set if a carry (or borrow) occurs from bit 35 of result

Instruction Fields:

Operation	Operands	C	W	Comments
SBC	Y,A Y,B	2	1	Subtract with carry (set C bit also)

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 DSC core, as well as the clocks to many of the on-chip peripherals such as serial ports, are gated off. 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 (that is, $[2^{19}-4]T$) or 28 clock phases (that is, $[2^{5}-4]T$) of 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 (the one 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 Section 7.5, “Stop Processing State,” on page 7-19 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 (that is, 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.

Instruction Fields:

Operation	Operands	C	W	Comments
STOP		N/A	1	Enter STOP low-power mode

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

Operation:			Assembler Syntax:		
D - S →	D		SUB	S,D	
D - S →	D	(single parallel move)	SUB	S,D	(single parallel move)
D - S →	D	(dual parallel read)	SUB	S,D	(dual parallel read)

Description: Subtract the source register from the destination register and store the result in the destination (D). If the destination is a 36-bit accumulator, 16-bit source registers are first sign extended internally and concatenated with 16 zero bits to form a 36-bit operand. When the destination is X0, Y0, or Y1, 16-bit subtraction is performed. In this case, if the source operand is one of the accumulators; the FF1 portion (properly sign extended) is used in the 16-bit subtraction (the FF2 and FF0 portions are ignored).

Usage: This instruction can be used for both integer and fractional two's-complement data.

Example:

```
SUB          X0,A    X:(R2)+N,X0        ; 16-bit subtract, load X0,
                                        ;    update R2
```

Before Execution

A2	A1	A0
0	0058	1234

X0	
	0003

After Execution

A2	A1	A0
0	0055	1234

X0	
	3456

Explanation of Example:

Prior to execution, the 16-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).

Operands of 16 bits can be subtracted from the LSP of A or B (A0 or B0). This can be achieved using the Y register. When loading the 16-bit operand into Y0 and loading Y1 with the sign extension of Y0, a 32-bit word is formed. Executing a `SUB Y,A` or `SUB Y,B` instruction generates the desired operation. Similarly, the second accumulator can also be used for the source operand.

Note: Bit 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.

Condition Codes Affected:

MR								CCR							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if MSB of result is set
Z — Set if result equals zero
V — Set if overflow has occurred in the result
C — Set if a carry (or borrow) occurs from MSB of result

See Section 3.6.5, “16-Bit Destinations,” on page 3-35 for cases with X0, Y0, or Y1 as D.
See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
SUB	DD,FDD	2	1	36-bit subtract of two registers. 16-bit source registers are first sign extended internally and concatenated with 16 zero bits to form a 36-bit operand.
	F1,DD			
	A,B			
	B,A			
	Y,A			
	Y,B			
	X:(SP-xx),FDD	6	1	Subtract memory word from register.
	X:aa,FDD	4	1	X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22.
	X:xxxx,FDD	6	2	
	#<0-31>,FDD	4	1	Subtract an immediate value 0–31
	#xxxx,FDD	6	2	Subtract a signed 16-bit immediate integer

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
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
		X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Parallel Dual Reads:

Data ALU Operation[1]		First Memory Read		Second Memory Read	
Operation[2]	**Operands**	**Source 1**	**Destination 1**	**Source 2**	**Destination 2**
SUB	X0,F Y1,F Y0,F (F = A or B)	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)-	X0

1. This parallel instruction is not allowed when the XP bit in the OMR is set (that is, when the instructions are executing from data memory).

2. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

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

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.

Instruction Fields:

Operation	Operands	C	W	Comments
SWI		8	1	Execute the trap exception at the highest interrupt priority level, level 1 (non-maskable)

Timing: 8 oscillator clock cycles

Memory: 1 program word

Tcc Transfer Conditionally Tcc

Operation:

If cc, then S → D
If cc, then S → D and R0 → R1

Assembler Syntax:

Tcc	S,D	
Tcc	S,D	R0,R1

Description: Transfer data from the specified source register (S) to the specified destination (D) if the specified condition is true. If the source is a 16-bit register, it is first sign extended and concatenated to 16 zero bits to form a 36-bit value before the transfer. When the saturation bit (SA) is set, saturation may occur if necessary—that is, the value transferred is substituted by the maximum positive (or negative) value. If a second source register R0 and a second destination register R1 are also specified, the instruction transfers the value 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

\+ 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 (that is, there is a single-instruction-cycle pipeline delay).

Example:

```
CMP         X0,A                ; compare X0 and A (sort for minimum)
TGT         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 Transfer		AGU Transfer		C	W	Comments
	Source	Destination	Source	Destination			
Tcc	DD	F	(No transfer)		2	1	Conditionally transfer one register.
	A	B	(No transfer)				
	B	A	(No transfer)				
	DD	F	R0	R1			Conditionally transfer one data ALU register and one AGU register.
	A	B	R0	R1			
	B	A	R0	R1			

Note: The Tcc instruction does not allow the following condition codes: HI, LS, NN, and NR.

Timing: 2 oscillator clock cycles

Memory: 1 program word

TFR — Transfer Data ALU Register — TFR

Operation:

S → D
S → D (single parallel move)

Assembler Syntax:

TFR S,D
TFR S,D (single parallel move)

Description: Transfer data from the specified source data ALU register (S) to the specified data ALU destination (D). The TFR instruction can be used to move the full 36-bit contents from one accumulator to another. This transfer occurs with saturation when the saturation bit, SA, is set. If the source is a 16-bit register, it is first sign extended and concatenated to 16 zero bits to form a 36-bit value before the transfer. The TFR instruction only affects the L and SZ bits in the CCR (which can be set by data movement that is associated with the instruction's parallel 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 1 instruction that executes in 1 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 $3:0123:0123 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	*	*	*	*	*	I1	I0	**SZ**	**L**	E	U	N	Z	V	C

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if data limiting has occurred during parallel move

Instruction Fields:

Operation	Operands	C	W	Comments
TFR	DD,F	2	1	Transfer register to register
	A,B			Transfer one accumulator to another (36-bits)
	B,A			

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	**Operands**	**Source**	**Destination[2]**
TFR	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
		X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word

Operation:

S - 0
S - 0 (single parallel move)

Assembler Syntax:

TST S
TST S (single 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. The result is not affected by the state of the saturation bit (SA).

Example:

```
TST          A        X:(R0)+N,B        ; set condition codes for the
                                        ;    value in A, update B & R0
```

Before Execution

A2	A1	A0
8	0203	0000

SR 0300

After Execution

A2	A1	A0
8	0203	0000

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	*	*	*	*	*	I1	I0	**SZ**	**L**	**E**	**U**	**N**	**Z**	**V**	**C**

SZ — Set according to the standard definition of the SZ bit (parallel move)
L — Set if data limiting has occurred during parallel move
E — Set if the extension portion of accumulator result is in use
U — Set according to the standard definition of the U bit
N — Set if bit 35 of accumulator result is set
Z — Set if result equals zero
V — Always cleared
C — Always cleared

See Section 3.6.2, “36-Bit Destinations — CC Bit Set,” on page 3-34 and Section 3.6.4, “20-Bit Destinations — CC Bit Set,” on page 3-34 for the case when the CC bit is set.

Instruction Fields:

Operation	Operands	C	W	Comments
TST	F	2	1	Test 36-bit accumulator

Parallel Moves:

Data ALU Operation		Parallel Memory Move	
Operation[1]	Operands	Source	Destination[2]
TST	F	X:(Rn)+ X:(Rn)+N	X0 Y1 Y0 A B A1 B1
	(F = A or B)	X0 Y1 Y0 A B A1 B1	X:(Rn)+ X:(Rn)+N

1. This instruction occupies only 1 program word and executes in 1 instruction cycle for every addressing mode.

2. The destination of the data ALU operation is not allowed to be the same register as the destination of the parallel read operation. Memory writes are allowed in this case.

Timing: 2 + mv oscillator clock cycles

Memory: 1 program word

TSTW Test Register or Memory TSTW

Operation:

S - 0

Assembler Syntax:

TSTW S

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. If the source is an accumulator, limiting can occur if the extension register (FF2) is in use.

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:$0007 contains the value $FC00 and the 16-bit SR contains the value $0300. Execution of the instruction compares the value in memory location X:$0007 with zero and updates the CCR accordingly. The value 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 and 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	*	*	*	*	*	I1	I0	SZ	**L**	E	U	**N**	**Z**	**V**	**C**

L — Set if overflow has occurred in result
V — Set if bit MSB of result is set
Z — Set if result equals zero
V — Always cleared
C — Always cleared

Instruction Fields:

Operation	Operands	C	W	Comments
TSTW	DDDDD (except HWS)	2	1	Test 16-bit word in register. All registers allowed except HWS. Limiting performed if an accumulator is specified and the extension register is in use.
	X:(Rn) X:(Rn)+ X:(Rn)- X:(Rn)+N	2	1	Test a word in memory using appropriate addressing mode.
	X:(Rn+N)	4	1	
	X:(Rn+xxxx)	6	2	
	X:(R2+xx)	4	1	
	X:(SP-xx)	4	1	
	X:aa	2	1	X:aa represents a 6-bit absolute address. Refer to **Absolute Short Address (Direct Addressing): <aa>** on page 4-22.
	X:<<pp	2	1	
	X:xxxx	4	2	X:<<pp represents a 6-bit absolute I/O address. Refer to **I/O Short Address (Direct Addressing): <pp>** on page 4-23.
	(Rn)-	2	1	Test and decrement AGU register

Timing: Refer to the preceding Instruction Fields table

Memory: Refer to the preceding Instruction Fields table

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.

Instruction Fields:

Operation	Operands	C	W	Comments
WAIT		n/a	1	Enter WAIT low-power mode

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 DSC is in the wait state equals the sum of the period before the interrupt or reset causing the DSC to exit the wait state and a minimum of 28T cycles to a maximum of 31T cycles (see the appropriate data sheet).

Memory: 1 program word

Appendix B
DSC Benchmarks

The following benchmarks illustrate source code syntax and programming techniques for the DSP56800. 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]	w_cnt[5]	cycles[6]	Comment[7]
`FIR:`	`MAC`	`Y0,X0,A`	`X:(R0)+,Y0 X:(R3)+,X0`	`; 1`	`1`	`Do each tap`

1. Used for program entry points and end-of-loop indication.
2. Indicates the data ALU, address 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. Word count specified from assembler. Sometimes a 16-bit address is used in order to create the worst case.
6. Cycle count refers to instruction cycle count required to execute the instruction.
7. Used for documentation purposes and does not affect the assembled code.

In each code example, the number of program words that each instruction occupies and the execution time (in instruction cycles) for each are listed in the comments and summed at the end.

Table B-1 shows the number of program words and instruction cycles for each benchmark. All the I/O accesses in this chapter are assumed to be implemented using I/O short addressing mode.

Table B-1. Benchmark Summary

Benchmark	Execution Time (# Icyc)	Program Length (# Words)
Real Correlation or Convolution (FIR Filter)	1N	11
N Complex Multiplies	6N	18
Complex Correlation or Convolution (Complex FIR)	5N	17
Nth Order Power Series (Real, Fractional Data)	1N	14
N Cascaded Real Biquad IIR Filters (Direct Form II)	6N	18
N Radix 2 FFT Butterflies	14N	32
LMS Adaptive Filter: Single Precision	N + 2NTaps	28

Table B-1. Benchmark Summary (Continued)

Benchmark	Execution Time (# Icyc)	Program Length (# Words)
LMS Adaptive Filter: Double Precision	2N + 5NTaps	35
LMS Adaptive Filter: Double Precision Delayed	5NTaps	30
Vector Multiply-Accumulate	2N	14
Energy in a Signal	1N	7
[3x3][3x1] Matrix Multiply	21	21
[NxN][NxN] Matrix Multiply	$N^3 + 8N^2+12N$	26
N Point 3x3 2-D FIR Convolution	$13N^2 + 14N$	39
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	14
Array Search: Index of the Highest Positive Value	2N	15
Proportional Integrator Differentiator (PID) Algorithm	9	9
Autocorrelation Algorithm	$((p + 1)^2 (N - p / 2))$	19

B.1 Benchmark Code

The following source code lists all the “defines” for the benchmarks. The addresses used in the definition section are selected arbitrarily to demonstrate the algorithm. Whenever possible, the long addressing mode will be used in order to generate the worst case scenario. The location of the peripheral space and the individual I/O assignments are dependent on chip implementation—I/O short addressing mode assumed.

```
          page 132
          opt  cc
; Global Definition for Loop Count
N_            EQU  100                    ; loop count in various benchmarks

; Peripheral addr are dependent on device implementation (assumes short addr mode)
InputValue    EQU  $FFC8                  ; I/O peripheral address used for input
Output        EQU  $FFC9                  ; I/O peripheral address used for output

; Section B.1.1 (correlation)
A_Vec1        EQU  $0200                  ; initial address of vector A
B_Vec1        EQU  $0100                  ; initial address of vector B
```

```
; Section B.1.2 (N complex multiplication)
A_Vec2          EQU  $0200                  ; initial address of A elements
B_Vec2          EQU  $0100                  ; initial address of B elements
C_Vec2          EQU  $2000                  ; initial address of result

; Section B.1.3 (complex correlation)
A_Vec3          EQU  $0200                  ; initial address of A elements
B_Vec3          EQU  $0100                  ; initial address of B elements

; Section B.1.4 (Nth order power series)
A_Vec4          EQU  $0200                  ; initial address of A elements
B_Data4         EQU  $0100                  ; initial address of B power multiplier

; Section B.1.5 (N cascaded real biquad IIR filter)
W_Vec5          EQU  $3000                  ; initial address of W vector
C_Vec5          EQU  $2000                  ; initial address of coefficients
N_Biquads       EQU  16                     ; number of cascaded real biquads

; Section B.1.6 (N radix 2 FFT butterflies)
N_BFlies        EQU  50                     ; number of butterflies
VEC_SIZE6       EQU  2*N_BFlies             ; size of vector {re,im}* butterflies
A_Vec6          EQU  $1000                  ; initial address, size=VEC_SIZE6
DummyLoc6       EQU  A_Vec6+VEC_SIZE6       ; initial store is a dummy store
B_Vec6          EQU  DummyLoc6+1            ; initial address, size=VEC_SIZE6
Twiddle_fac     EQU  B_Vec6+VEC_SIZE6       ; address of twiddle factor
ToDummyLOC      EQU  -(2+VEC_SIZE6)         ; to advance to {B_Vec6-1}

; Section B.1.7 (LMS adaptive filter)
X_Vec7          EQU  $0100                  ; initial address of X state
Coeff           EQU  $0500                  ; initial address of coefficients
NTaps           EQU  $10                    ; number of taps for LMS filter

; Section B.1.8 (vector multiply-accumulate)
VEC_SIZE8       EQU  100                    ; size of vector
A_Vec8          EQU  $1000                  ; initial address, size=VEC_SIZE8
B_Vec8          EQU  A_Vec8+VEC_SIZE8       ; initial address, size=VEC_SIZE8
C_Vec8          EQU  B_Vec8+VEC_SIZE8       ; initial address of vector result

; Section B.1.9 (energy of a signal)
A_Vec9          EQU  $0100                  ; initial address of signal vector
```

```
; Section B.1.10 (matrix multiply [3x3][3x1])
ROW_SIZE10      EQU  3                       ; row size of C and B
MATRX_SIZE10    EQU  ROW_SIZE10*ROW_SIZE10   ; matrix A size
A_Matrx10       EQU  $1000                   ; initial address of 3x3 matrix
B_Vec10         EQU  A_Matrx10+MATRX_SIZE10  ; initial address of 3x1 vector
C_Vec10         EQU  B_Vec10+ROW_SIZE10      ; initial address of 3x1 result vector

; Section B.1.11 (matrix multiply [NxN][NxN])
ROW_SIZE11      EQU  10                      ; row size of A, B and C sq. matrices
MATRX_SIZE11    EQU  ROW_SIZE11*ROW_SIZE11   ; matrices A,B and C size
A_Matrx11       EQU  $1000                   ; initial address of A matrix
B_Matrx11       EQU  A_Matrx11+MATRX_SIZE11  ; initial address of B matrix
C_Matrx11       EQU  B_Matrx11+MATRX_SIZE11  ; initial address of C matrix (result)
RowsCnt         EQU  $0005                   ; address of count for s/w loop

; Section B.1.12 (N-point 3x3 2-D FIR convolution)
Image           EQU  $2000                   ; initial address of image
OutputImage     EQU  $7000                   ; initial address of output result image
ImageRowsCnt    EQU  $001F                   ; address of count for s/w loop
CoeffMask       EQU  $0020                   ; initial address of 3x3 coeff mask

; Section B.1.13 (sine-wave generation)
DummyLoc13      EQU  $0010                   ; dummy address for storage swap

; Section B.1.14 (array search)
A_Vec14         EQU  $1000                   ; initial address of vector, size N_

; Section B.1.15 (PID)
X_Vec15         EQU  $000A                   ; initial address of inputs X
K_Vec15         EQU  $0020                   ; initial address of gain const in PID

; Section B.1.16 (autocorrelation)
Frame_Vec16     EQU  $0200                   ; initial address of frame
Corr_Vec16      EQU  $0100                   ; address of correlation vector
LPC             EQU  8
p               EQU  10

;         org  p:$40                         ; the beginning of the program will
                                             ;   be dictated by the tool used.
```

B.1.1 Real Correlation or Convolution (FIR Filter)

```
; c(n) = SUM(I=0,...,N-1) { a(I) * b(n-I) }

      opt   cc
      MOVE  #N_,N                                ; 2   2    load size of vector
      MOVE  #A_Vec1,R0                           ; 2   2    pointer to vector
      MOVE  #B_Vec1,R3                           ; 2   2    pointer to vector
      CLR   A        X:(R0)+,Y0                  ; 1   1    clear & load 1st element in A
      MOVE          X:(R3)+,X0                  ; 1   1    load 1st element in B
      REP   N                                   ; 1   3    repeat until done
      MAC   Y0,X0,A  X:(R0)+,Y0  X:(R3)+,X0     ; 1   1    correlation and load new val
      RND   A                                   ; 1   1    rounding the result
;                                               _________
;                                Total:          11   1N+12
```

B.1.2 N Complex Multiplication

```
; 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
; NOTE: array size of complex product result must be >= 2*N_
    opt     cc
    MOVE    #N_,N                                       ; 2   2
    MOVE    #A_Vec2,R0                                  ; 2   2
    MOVE    #C_Vec2-1,R2                                ; 2   2      set R2=dest-1
    MOVE    #B_Vec2,R3                                  ; 2   2
    MOVE                X:(R2),B                        ; 1   1      dest-1 saved
    DO      N,EndDO1_2                                  ; 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
                                                        ;
    MACR    -Y0,X0,A                                    ; 1   1      ar*br-ai*bi
    MACR    Y1,X0,B     A,X:(R2)+                       ; 1   1      ar*bi+ai*br,
EndDO1_2:                                               ;            store real
    MOVE    B,X:(R2)+                                   ; 1   1
;                                                       ____________
;                                           Total:      18    6N+13
```

B.1.3 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  #N_,N                                 ; 2     2

      MOVE  #A_Vec3,R0                            ; 2     2

      MOVE  #B_Vec3,R3                            ; 2     2

      CLR   A             X:(R0)+,Y0              ; 1     1       ar and clear result

      CLR   B             X:(R3)+,Y1              ; 1     1       br and clear result

      DO    N,EndDO1_3                            ; 2     3

      MAC   Y0,Y1,A       X:(R3)+,X0              ; 1     1       ar*br, get next bi

      MAC   Y0,X0,B       X:(R0)+,Y0              ; 1     1       ar*bi, get next ai

      MAC   Y0,Y1,B       X:(R3)+,Y1              ; 1     1       ar*bi+ai*br, next br

      MAC   -Y0,X0,A                              ; 1     1       ar*br-ai*bi

      MOVE                X:(R0)+,Y0              ; 1     1       get next ar

EndDO1_3:

      RND   A                                     ; 1     1

      RND   B                                     ; 1     1

;                                                 _____________

;                                       Total:    17      5N+13
```

B.1.4 Nth Order Power Series (Real, Fractional Data)

```
; c = SUM(I=0,...,N) { a(I) * b**I }    (for N even)
;   = [....[[a(n)*b + a(n-1)]*b + a(n-2)]*b + a(n-3)]*b + ..... ]*b + a(1)]*b + a(0)
        opt    cc
        MOVE   #N_/2,N                          ; 1   1      #N_/2 = 50
        MOVE   #B_Data4,R1                      ; 2   2
        MOVE   #A_Vec4,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     N,EndDO1_4                       ; 2   3
        MAC    A1,Y0,B      X:(R0)+,A           ; 1   1      get a(n-2), next a(n-4)
        MAC    B1,Y1,A      X:(R0)+,B           ; 1   1      get a(n-3), next a(n-5)
EndDO1_4:
        RND    A                                ; 1   1
;                                               ___________
;                                     Total:    14    1N+13  Loop is N/2 times 2 inst
```

B.1.5 N Cascaded Real Biquad IIR Filters (Direct Form II)

Many digital-filter design packages generate coefficients for Direct Form II infinite impulse response (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. The general form of the IIR filter's output *y(n)* at time *n*, a linear combination of the present input, the M previous inputs and the N previous outputs, is given by:

$$y(k) = \sum_{n=1}^{N} a_n y(k-n) + \sum_{m=0}^{M} b_m x(k-m)$$

The Biquad Direct Form II realization of the IIR filter above can be described in the following two equations:

$$w(k) = \sum_{n=1}^{N} a_n \cdot w(k-n) + x(k) = a_1 \cdot w(k-1) + a_2 \cdot w(k-2) + x(k) \quad \text{for N=2}$$

$$y(k) = \sum_{m=0}^{M} b_m \cdot w(k-m) = b_0 \cdot w(k) + b_1 \cdot w(k-1) + b_2 \cdot w(k-2) \quad \text{for M=2}$$

```
; Biquad: M=N=2       (normalizing by b0). This version uses two pointers.
; w(k)/2 = x(k)/2 + (a1/2) * w(k-1) + (a2/2) * w(k-2)
; y(k)/2 = w(k)/2 + (b1/2) * w(k-1) + (b2/2) * w(k-2)
; D High Memory Order - w(k-2)1,w(k-1)1,w(k-2)2,w(k-1)2,...
; D Low  Memory Order - (a2/2)1,(a1/2)1,(b2/2)1,(b1/2)1,(a2/2)2,...
        opt   cc
        MOVE  #W_Vec5,R0                            ; 2   2     ptr to w(k-2),
        MOVE  #C_Vec5,R3                            ; 2   2     ptr to a2 coeff
        MOVE  #N_Biquads,LC                         ; 2   2     number of biquads
        MOVE  #-1,N                                 ; 1   1     allow traverse prev
        MOVE                X:InputValue,A          ; 1   1     input fr peripheral
        ASR   A                           X:(R3)+,X0 ; 1   1     X0=.5a2
        MOVE                X:(R0)+,Y0              ; 1   1     Y0=w(k-2)
        DO    LC,EndDO1_5                           ; 2   3     start casc biquads
        MAC   Y0,X0,A       X:(R0)+N,Y1  X:(R3)+,X0 ; 1   1     Y1=w(k-1) X0=.5a1
        MAC   Y1,X0,A       Y1,X:(R0)+              ; 1   1     store interm result
        ASL   A                           X:(R3)+,X0 ; 1   1     X0=.5b2
        ASR   A             A,X:(R0)+               ; 1   1
        MAC   Y0,X0,A                     X:(R3)+,X0 ; 1   1     X0=.5b1
        MAC   Y1,X0,A       X:(R0)+,Y0    X:(R3)+,X0 ; 1   1     A=.5Y(k) for biquad
```

```
EndDO1_5:
;                                                    ___________
;                                        Total:      18     6N+13   (N = #cascades)
```

B.1.6 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.

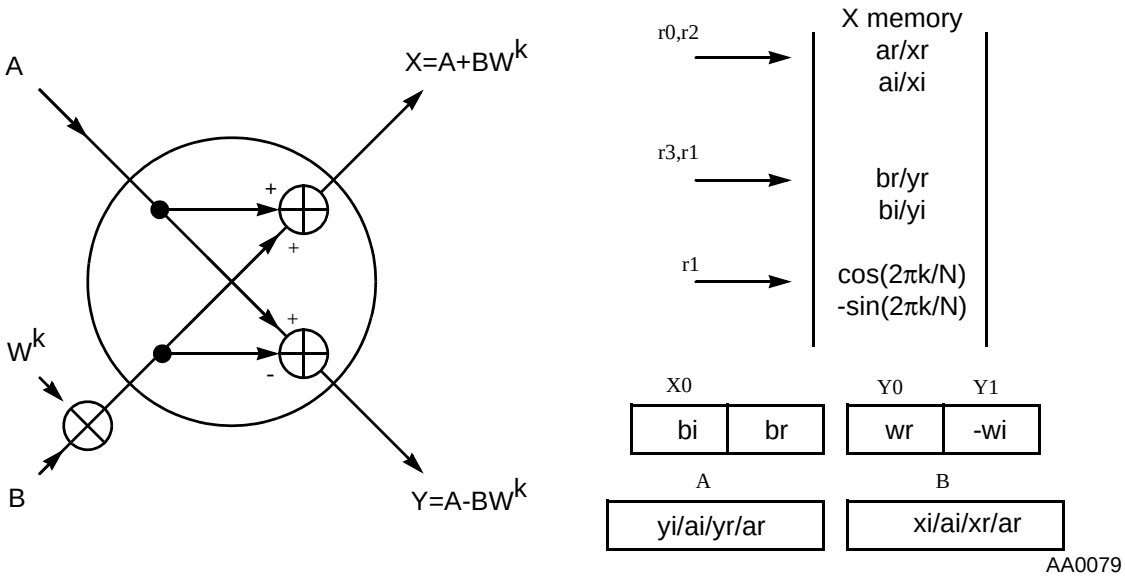


Figure B-1. N Radix 2 FFT Butterflies Memory Map

```
; 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
; R0 is pointer to Vector A {ar,ai}
; R2 points to same initial location as R0, it stores result Vector X {xr,xi}
; R3 is pointer to Vector B {br,bi}
; R1 initially points to Twiddle Factor: Wk {wr,wi}, then updated to DummyLoc6 to
;      store accum A that is meaningless at first pass. R1 then points to Vector B
;      AT end of algorithm, R1 will point again to Twiddle Factor.
; R1 points to same Vector B where it stores result Vector Y {yr,yi}
; Location of arrays for vectors A, B is conveniently selected based on butterflies
; For demonstration, B is chosent to follow A by 2*N_BFlies+1 word locations.
; After accessing Twiddle, R1 points to {Location(B)-1}. This is DummyLoc6.
;PUSH   MACRO  REG1                                          ; Macro definition
;       LEA    (SP)+                                        ; Increment SP
;       MOVE   REG1,X:(SP)                                  ; Push REG1 to stk
;       ENDM                                                ; End macro def
```

```
        opt     cc
        MOVE    #Twiddle_fac,R1                         ; 2    2    Twiddle Fac address
        MOVE    #A_Vec6,R0                              ; 2    2    A_vec start address
        MOVE    R0,R2                                   ; 1    1    X {xr,xi} strt addr
        MOVEI   #B_Vec6,R3                              ; 2    2    B_vec start address
        MOVEI   #ToDummyLOC,N                           ; 2    2    to set R1=loc{B}-1
        MOVEI   #N_BFlies,LC                            ; 2    2    for loop count
        MOVE                X:(R1)+,Y0   X:(R3)+,X0     ; 1    1    Y0=wr  X0=br
        MOVE                X:(R0),B                    ; 1    1    B=ar
        MOVE                X:(R1)+N,Y1                 ; 1    1    Y1=wi  R1=(loc)B-1
; R1 now points to {location(B_Vec6)-1}
        MOVEI               #0,N                        ; 1    1    for emulating X:(Rn)
                                                        ;              addressing mode
        DO      LC,EndDO1_6                             ; 2    3    repeat butterflies
        PUSH    X0                                      ; 2    2    save br
        MAC     Y0,X0,B     X:(R3)+,X0                  ; 1    1    B=ar+wrbr  X0=bi
        MACR    -Y1,X0,B                                ; 1    1    B=ar+wrbr+wibi=Xr
        MOVE                A,X:(R1)+                   ; 1    1    STORE Yi fr previous
                                                        ;           (1st is dummy store)
        MOVE                X:(R0)+,A                   ; 1    1    A=ar R0 points to ai
        ASL     A           B,X:(R2)+                   ; 1    1    A=2ar  STORE Xr
        SUB     B,A         X:(R0)+N,B                  ; 1    1    A=2ar-Xr=Yr  B=ai
        MOVE                A,X:(R1)+                   ; 1    1    STORE Yr
        MAC     Y0,X0,B     X:(R0)+,A                   ; 1    1    B=ai+wrbi   A=ai
        POP     X0                                      ; 1    1    restore br
        MACR    Y1,X0,B     X:(R3)+,X0                  ; 1    1    B=ai+wrbi+wibr=Xi
                                                        ;           X0=(next)br
        ASL     A           B,X:(R2)+                   ; 1    1    A=2ai  STORE Xi
        SUB     B,A         X:(R0)+N,B                  ; 1    1    A=2ai-Xi=Yi
                                                        ;           b=(next)ar
                                                        ;           R0 points (next)ar
EndDO1_6:                                               ;           end butterfly
        MOVE                A,X:(R1)+                   ; 1    1    STORE last Yi
;                                                       ___________
;                                       Total:          32     14N+19
; R1 now points to Twiddle factors (by selecting appropriate locations for vectors)
```

B.1.7 LMS Adaptive Filter

Figure B-2 gives a graphical representation of this implementation of the LMS adaptive filter.

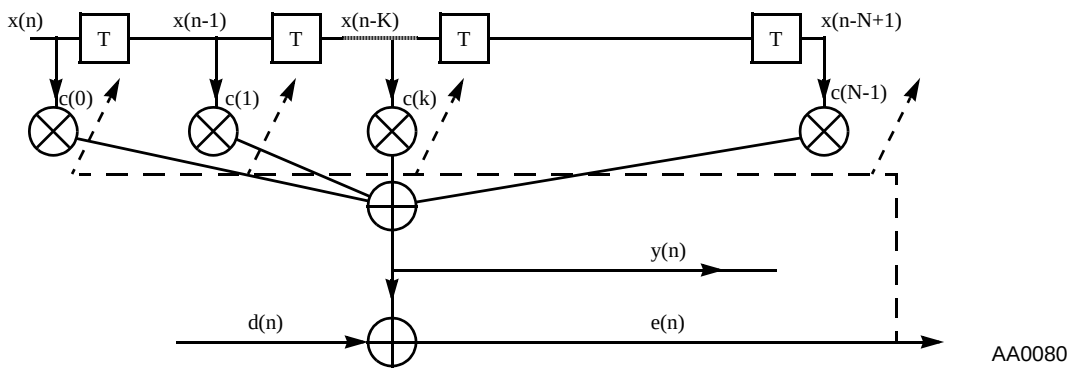


Figure B-2. LMS Adaptive Filter Graphic Representation

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 the following:

- *Adaptive Digital Filters and Signal Analysis*, Maurice G. Bellanger (Marcel Dekker: 1987)
- "The DLMS Algorithm Suitable for the Pipelined Realization of Adaptive Filters," Proc. IEEE ASSP Workshop, Academia Sinica, Beijing (IEEE: 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.1.7.1 Single Precision

Figure B-3 shows a memory map for this implementation of the single-precision LMS adaptive filter.

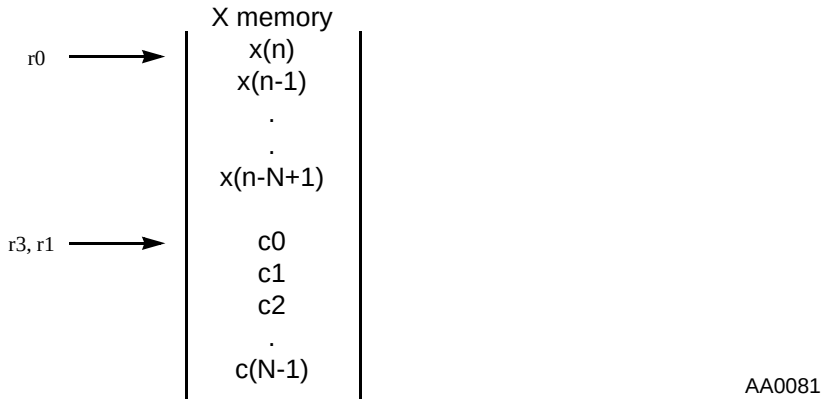


Figure B-3. LMS Adaptive Filter — Single Precision Memory Map

```
   opt     cc
   PUSH    M01                                ; 2    2    save addr mode state
   MOVE    #X_Vec7,R0                         ; 2    2    start of X
   MOVE    #N_-1,M01                          ; 2    2    modulo N_
   MOVE    M01,Y1                             ; 1    1    initialize REP loop count
   MOVE    #-2,N                              ; 1    1    adjustment for filtering
   MOVEP   X:InputValue,Y0                    ; 1    1    get input sample
   MOVE    #Coeff,R3                          ; 2    2    start of coefficients
   CLR     A         Y0,X:(R0)+               ; 1    1    save input in x(n),incr R0
   MOVE                        X:(R3)+,X0     ; 1    1    X0=c[0] and incr R3
   REP     Y1                                 ; 1    3    do fir
   MAC     Y0,X0,A   X:(R0)+,Y0  X:(R3)+,X0   ; 1    1    accum & update x[i] and c[i]
   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    #Coeff,R3                          ; 2    2    start of coefficients
   MOVE    R3,R1                              ; 1    1    start of coefficients
   MOVE              X:(R0)+,Y0               ; 1    1    Y0=x(n) and incr R0
   MOVE                        X:(R3)+,A      ; 1    1    a=c[0] and incr R3
   DO      #NTaps,EndDO1_7_1                  ; 2    3    update coefficients
   MACR    X0,Y0,A   X:(R0)+,Y0  X:(R3)+,X0   ; 1    1    A=c[i]*x[n-i]  Y0=x[n-i-1]
                                              ;           X0=c[i+1]
   TFR     X0,A      A,X:(R1)+                ; 1    1    A=c[i+1], save c[i]*x[n-i]
_COEFF_UPDATE1_7_1:
```

```
EndDO1_7_1:
   MOVE            X:(R0)+N,Y0            ; 1     1    Y0=x(n-N+1) and update R0
   POP    M01                             ; 1     1    restore previous addr mode
;                                         __________
;                              Total:     28      N+2NTaps+27  (N=size of X_VECTOR)
```

B.1.7.2 Double Precision

Figure B-4 shows a memory map for this implementation of the double-precision LMS adaptive filter.

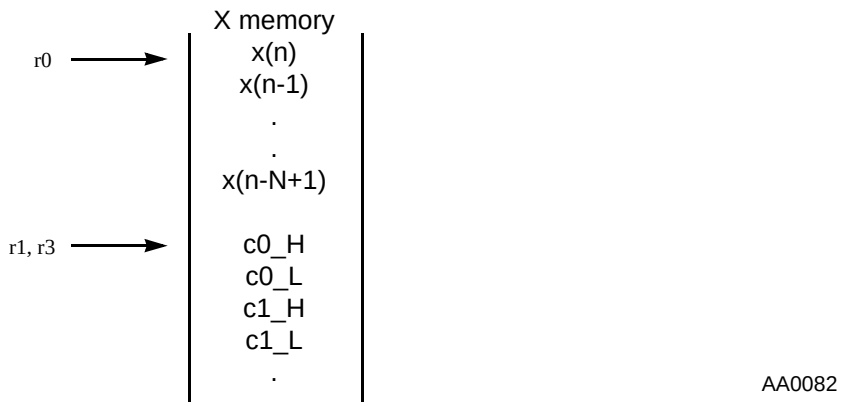


Figure B-4. LMS Adaptive Filter — Double Precision Memory Map

```
    opt    cc
    PUSH   M01                                  ; 2  2   save addr mode state
    MOVE   #X_Vec7,R0                           ; 2  2   start of X
    MOVE   #N_-1,M01                            ; 2  2   modulo N_
    MOVE   M01,Y1                               ; 1  1   initialize REP loop count
    MOVE   #2,N                                 ; 1  1   adjustment for filtering
    MOVEP  X:InputValue,Y0                      ; 1  1   get input sample
    MOVE   #Coeff,R3                            ; 2  2   start of coefficients
    CLR    A          Y0,X:(R0)+                ; 1  1   save input in x(n),incr R0
    MOVE                          X:(R3)+N,X0   ; 1  1   X0=c[0,H] and incr R3
    DO     Y1,Do_FIR                            ; 2  3   do fir
    MAC    X0,Y0,A    X:(R0)+,Y0                ; 1  1   accum & update x[i]
    MOVE              X:(R3)+N,X0               ; 1  1   update c[i,H]
Do_FIR:
    MACR   X0,Y0,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 x0.
; This section is application dependent.)
    MOVE   #Coeff,R3                            ; 2  2   start of coefficients
    MOVE   R3,R1                                ; 1  1   start of coefficients
    MOVE              X:(R0)+,y0                ; 1  1   Y0=x(n) and incr R0
    MOVE                          X:(R3)+,A     ; 1  1   a=c[0,H] and incr R3
    MOVE                          X:(R3)+,A0    ; 1  1   a0=c[0,L] and incr R3
    DO     #NTaps,EndDO1_7_2                    ; 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[i,H]
    MOVE            A0,X:(R1)+                  ; 1  1   save updated c[i,L]
    MOVE            X:(R3)+,A                   ; 1  1   fetch next c[i,H]
    MOVE            X:(R3)+,A0                  ; 1  1   fetch next c[i,L]
_COEFF_UPDATE1_7_2:
EndDO1_7_2:
    MOVE   #-2,N                                ; 1  1   adjustment for filtering
    MOVE            X:(R0)+N,Y0                 ; 1  1   update r0
    POP    M01                                  ; 1  1   restore previous addr mode
;                                               __________
;                                   Total:      35   2N+5NTaps+28 (N=size of X_Vec7)
```

B.1.7.3 Double Precision Delayed

Figure B-5 shows a memory map for this implementation of the double-precision delayed LMS adaptive filter.

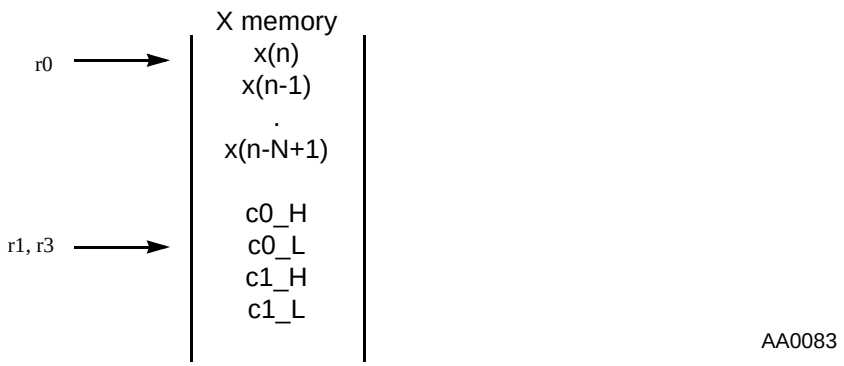


Figure B-5. LMS Adaptive Filter — Double Precision Delayed Memory Map

```
; 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
   PUSH    M01                                  ; 2  2   save addr mode state
   MOVE    #X_Vec7,R0                           ; 2  2   start of X
   MOVE    #NTaps,M01                           ; 2  2   modulo NTaps
   MOVE    #Coeff,R3                            ; 2  2   start of coefficients
   MOVE    #Coeff-2,R1                          ; 2  2   start of delayed coef
   MOVE    #0,N                                 ; 1  1   to emulate (Rn) adr mode
   CLR     B         X:(R0)+,Y0                 ; 1  1   y0 = x[n]
   MOVE              X:(R0)+,Y1  X:(R3)+,X0     ; 1  1   y1= x[n-1], x0=c[0,H]
   DO      #NTaps/2,EndDO1_7_3                  ; 2  3   do FIR and update coefficients
   MAC     Y0,X0,B   A,X:(R1)+                  ; 1  1   update coefficient
   MOVE              A0,X:(R1)+                 ; 1  1   update coefficient
   TFR     X0,A      X:(R2)+N,X0                ; 1  1   x0=loop gain * error
   MOVE                        X:(R3)+,A0       ; 1  1   a0=c[k,L]
   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   update coefficient
   MOVE              A0,X:(R1)+                 ; 1  1   update coefficient
```

```
    TFR     X0,A      X:(R2)+N,X0                ; 1   1   x0=loop gain * error
    MOVE                           X:(R3)+,A0   ; 1   1   a0=c[i,L]
    MACR    X0,Y0,A   X:(R0)+,Y1   X:(R3)+,X0   ; 1   1   y1=next x[n] x0=c[i,H]
End_LMS2:
EndDO1_7_3:
    MOVE    #-2,N                                ; 1   1   to correct R0
    MOVE              A,X:(R1)+                  ; 1   1   last coefficient update c[i,H]
    MOVE              A0,X:(R1)+                 ; 1   1   last coefficient update c[i,L]
    LEA               (R0)+N                     ; 1   1   correct R0
    POP     M01                                  ; 1   1   restore previous addr mode
;                                                ___________
;                                  Total:        30    5N+21 (N=NTaps)
```

B.1.8 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.

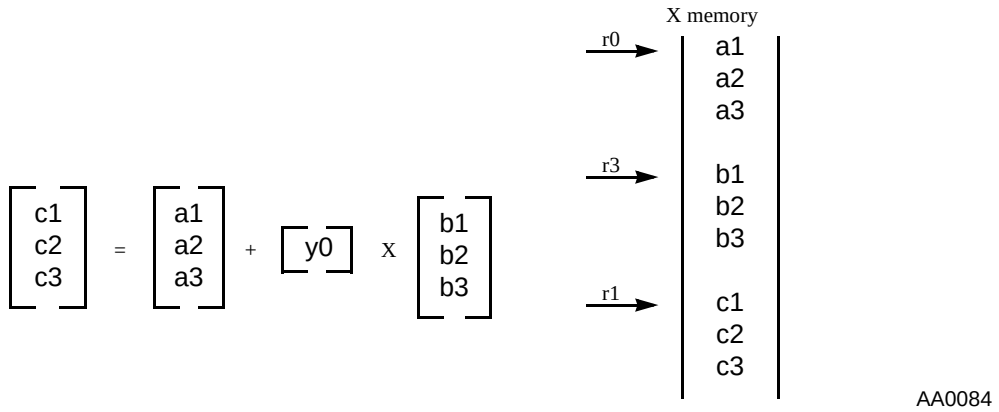


Figure B-6. Vector Multiply-Accumulate

```
   opt       cc
; Y0 is assumed to have been initialized with the multiplier scalar value
   MOVEI     #N_,N                                  ; 2    2     vector size
   MOVE      #A_Vec8,R0                             ; 2    2     point to vec a
   MOVE      #B_Vec8,R3                             ; 2    2     point to vec b
   MOVE      #C_Vec8,R1                             ; 2    2     point to vec c
   CLR       A                          X:(R3)+,X0  ; 1    1     X0=b   RESULT:A=0
   MOVE                    X:(R0)+,A                ; 1    1     A=a
   DO        N,EndDO1_8                             ; 2    3     repeat prod N times
   MAC       y0,x0,a       X:(R0)+,Y1   X:(R3)+,X0  ; 1    1     A=Y0*b+a load nxt a,b
   TFR       y1,a          A,X:(R1)+                ; 1    1     A=next(a) STORE c
EndDO1_8:
;                                                   ____________
;                                       Total:      14     2N+13
```

B.1.9 Energy in a Signal

This code calculates the energy in a signal by summing together the square of each sample.

```
      opt    cc
      MOVE   #A_Vec9,R0                          ; 2   2    point to signal a
      MOVEI  #N_,N                               ; 2   2    load vector size
      CLR    A            X:(R0)+,Y0             ; 1   1    clear and load 1st val
      DO     N,EndDO1_9                          ; 2   3    repeat size N times
      MAC    Y0,Y0,A      X:(R0)+,Y0             ; 1   1    square value & load nxt
EndDO1_9:
;                                                ___________
;                                     Total:     8     1N+8
; if vector pointer located inside 128 addresses, use MOVES X:<AA,R0 (1cyc, 1wrd)
; if vector size is less than 63, initializing N is not required.
```

Second option when the DO instruction is replaced by REP. This sequence is uninterruptible while performing the MAC instruction.

```
      opt    cc
      MOVE   #A_Vec9,R0                          ; 2   2    point to signal a
      MOVEI  #N_,N                               ; 2   2    load vector size
      CLR    A            X:(R0)+,Y0             ; 1   1    clear and load 1st val
      REP    N                                   ; 1   3    repeat size N times
      MAC    Y0,Y0,A      X:(R0)+,Y0             ; 1   1    square value & load nxt
;                                                ___________
;                                     Total:     7     1N+8
; if vector pointer located inside 128 addresses, use MOVES X:<AA,R0 (1cyc, 1wrd)
; if vector size is less than 63, initializing N is not required.
```

B.1.10 [3x3][3x1] Matrix Multiply

Figure B-7 gives a graphical overview and memory map for a [3x3][3x1] matrix multiply.

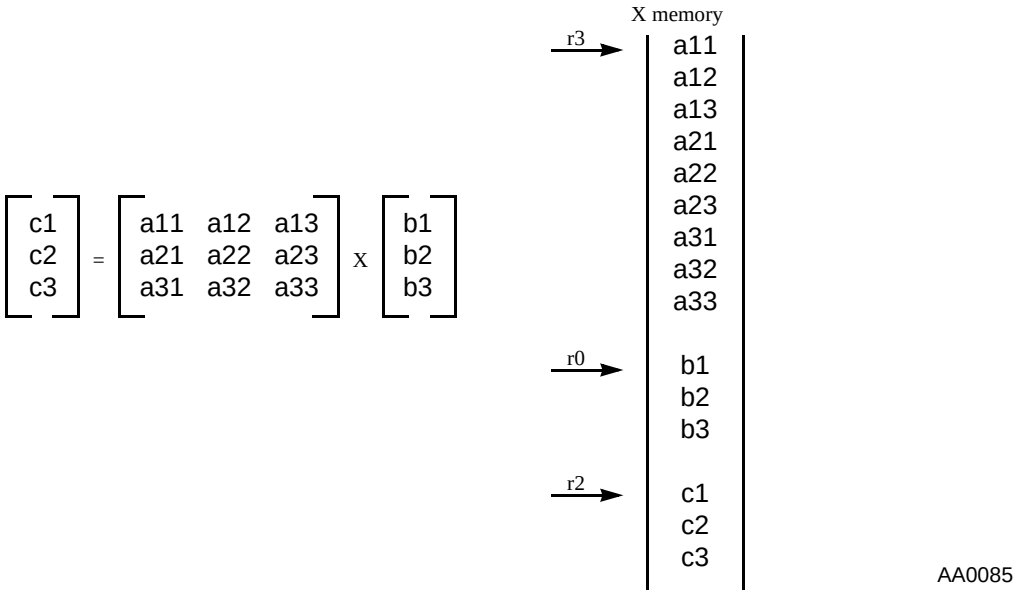


Figure B-7. [3x3][1x3] Matrix Multiply

```
        opt     cc
        MOVE    #A_Matrx10,R3                           ; 2    2      point to mat a
        MOVE    #B_Vec10,R0                             ; 2    2      point to vec b
        MOVE    #2,M01                                  ; 2    2      mod 3 addr on R0
        MOVE    #C_Vec10,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:       21     21
```

B.1.11 [NxN][NxN] Matrix Multiply (for fractional elements)

The matrix multiplications are for square NxN matrices (all fractional elements are in row-major format). Figure B-8 gives a graphical overview and memory map of an [NxN][NxN] matrix multiply.

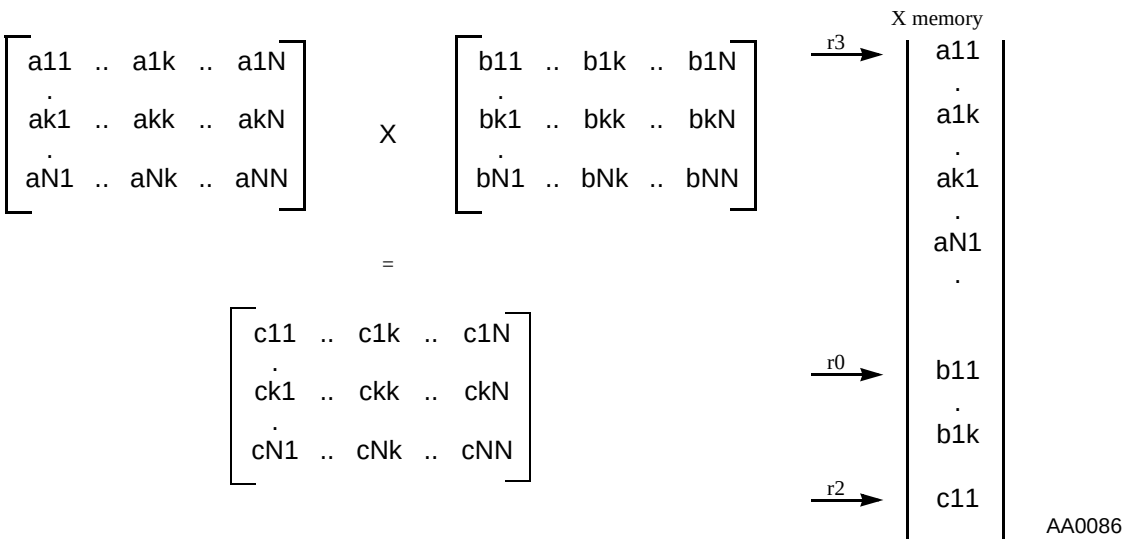


Figure B-8. [NxN][NxN] Matrix Multiply

```
        opt   cc
; This algorithm utilizes hardware nesting looping; user care necessary on next loop.
; The main assumption: no hardware loops active when this function is called.
        MOVE  #A_Matrx11,R3                          ; 2  2    point to A[1,1]
        MOVE  R3,Y1                                  ; 1  1    save pntr to A[1,1]
        MOVE  #B_Matrx11,R0                          ; 2  2    point to B[1,1]
        MOVE  R0,R1                                  ; 1  1    save pntr to B[1,1]
        MOVE  #C_Matrx11,R2                          ; 2  2    point to C[1,1] (result)
        MOVE  #ROW_SIZE11,B                          ; 1  1    number of rows (N x N)
        MOVE  B,N                                    ; 1  1    number of repetitions N
        DO    N,Traverse_A_rows_1                    ; 2  3    do all rows
        PUSH  LC                                     ; 2  2    save LC to allow nesting
        PUSH  LA                                     ; 2  2    save LA to allow nesting
        DO    N,Traverse_B_columns_1                 ; 2  3    compute a row in C
        MOVE  Y1,R3                                  ; 1  1    1st element in A row
        MOVE  R1,R0                                  ; 1  1    1st element in B column
        CLR   A                        X:(R3)+,X0    ; 1  1    clr sum, get elemnt in A
        MOVE             X:(R0)+N,Y0                 ; 1  1    element in B, next B row
        REP   #ROW_SIZE11-1                          ; 1  3    sum products except last
        MAC   Y0,X0,A    X:(R0)+N,Y0   X:(R3)+,X0    ; 1  1    traverse B rows, A col
; FOR FRACTIONAL ELEMENTS, THE FOLLOWING TWO INSTRUCTIONS ARE REQUIRED
```

```
; THE MACR DOES THE FINAL ACCUMULATION WITH ROUNDING FOR FRACTIONAL RESULTS
      MACR  Y0,X0,A                 X:(R1)+,X0    ; 1  1    last sum, next col in R1
      MOVE  A,X:(R2)+                             ; 1  1    save result in C row
; FOR INTEGER ELEMENTS, THE FOLLOWING 3 INSTRUCTIONS WILL REPLACE: MACR & MOVE ABOVE
;     MAC   Y0,X0,A                 X:(R1)+,X0    ; -  -    last sum, no rounding
;     ASR   A                                     ; -  -    convert to integer in A0
;     MOVE  A0,X:(R2)+                            ; -  -    save integer result
Traverse_B_columns_1:
      POP   LA                                    ; 1  1    restore LA, outer loop
      POP   LC                                    ; 1  1    restore LC, outer loop
      ADD   B1,Y1                                 ; 1  1    for traverse next A row
      MOVE  #B_Matrx11,R1                         ; 2  2    point to B[1,1]
Traverse_A_rows_1:
;                                                 __________
;                       Words:       Cycles:
;           Total:      31           ((N+8)N+12)N+14 = N^3+8N^2+12N+13
```

This next version makes use of software loop avoiding the hardware nested looping.

```
      opt   cc
; This algorithm utilizes software outer loop avoiding nesting and saving LC,LA regs.
; The main assumption: no hardware nesting loops active when function is called.
      MOVE  #A_Matrx11,R3                         ; 2  2    point to A[1,1]
      MOVE  R3,Y1                                 ; 1  1    save pntr to A[1,1]
      MOVE  #B_Matrx11,R0                         ; 2  2    point to B[1,1]
      MOVE  R0,R1                                 ; 1  1    save pntr to B[1,1]
      MOVE  #C_Matrx11,R2                         ; 2  2    point to C[1,1] (result)
      MOVE  #ROW_SIZE11,B                         ; 1  1    number of rows (N x N)
      MOVE  B,N                                   ; 1  1    number of repetitions N
      MOVES N,X:RowsCnt                           ; 1  1    number of A rows to do
Traverse_A_rows_2:
      DO    N,Traverse_B_columns_2                ; 2  3    compute a row in C
      MOVE  Y1,R3                                 ; 1  1    1st element in A row
      MOVE  R1,R0                                 ; 1  1    1st element in B column
      CLR   A                       X:(R3)+,X0    ; 1  1    clr sum, get elemnt in A
      MOVE              X:(R0)+N,Y0               ; 1  1    element in B, next B row
      REP   #ROW_SIZE11-1                         ; 1  3    sum products except last
      MAC   Y0,X0,A     X:(R0)+N,Y0 X:(R3)+,X0    ; 1  1    traverse B rows, A col
; FOR FRACTIONAL ELEMENTS, THE FOLLOWING TWO INSTRUCTIONS ARE REQUIRED
; THE MACR DOES THE FINAL ACCUMULATION WITH ROUNDING FOR FRACTIONAL RESULTS
      MACR  Y0,X0,A                 X:(R1)+,X0    ; 1  1    last sum, next col in R1
      MOVE  A,X:(R2)+                             ; 1  1    save result in C row
```

```
; FOR INTEGER ELEMENTS, THE FOLLOWING 3 INSTRUCTIONS WILL REPLACE: MACR & MOVE ABOVE
;       MAC    Y0,X0,A                  X:(R1)+,X0    ; -   -    last sum, no rounding
;       ASR    A                                      ; -   -    convert to integer in A0
;       MOVE   A0,X:(R2)+                             ; -   -    save integer, LSP of A
Traverse_B_columns_2:
        ADD    B1,Y1                                  ; 1   1    for traverse next A row
        MOVE   #B_Matrx11,R1                          ; 2   2    point to B[1,1]
        DECW   X:RowsCnt                              ; 1   3    decrement A rows count
        BGT    Traverse_A_rows_2                      ; 1   3    loop back if not done
;                                                     __________
;                        Words:       Cycles:
;             Total:     26           ((N+8)N+12)N+11= N^3+8N^2+12N+11
```

B.1.12 N Point 3x3 2-D FIR Convolution

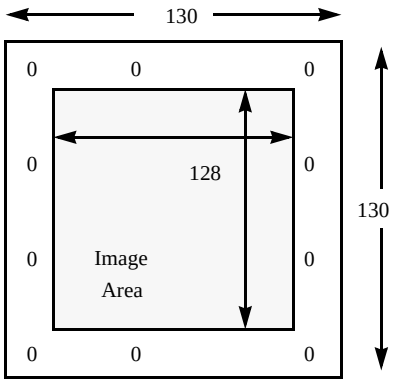
The two-dimensional FIR uses a 3x3 coefficient mask as shown in Figure B-9.

$$\begin{bmatrix} c11 & c12 & c13 \\ c21 & c22 & c23 \\ c31 & c32 & c33 \end{bmatrix}$$

AA0087

Figure B-9. 3x3 Coefficient Mask

The image is an array of 128 pixels x 128 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 130x130 array (see Figure B-10).



AA0088

Figure B-10. Image Stored as 130x130 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(1,130) followed by the beginning of the next column image(2,1). These are stored sequentially in the array Image ("im" on instruction comment) in data memory. For example:

- Image(1,1) maps to index 0.
- Image(1,130) maps to index 129.
- Image(2,1) maps to index 130.

See Table B-2 for the definitions 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 two. Other possibilities include storing a 128x128 image but computing only a 127x127 result, computing a 128x128 result without boundary conditions but throwing away the pixels on the border, and so on.

Table B-2. Variable Descriptions

Variable	Description		
R0	image(n,m)	image(n,m+1)	image(n,m+2)
	image(n+130,m)	image(n+130,m+1)	image(n+130,m+2)
	image(n+2*130,m)	image(n+2*130,m+1)	image(n+2*130,m+2)
R2	output image		
R3	FIR coefficients		

```
    opt    cc
    MOVE   #CoeffMask,R3                        ; 1   1    pointer to coef (short addr)
    MOVE   #Image,R0                            ; 2   2    top boundary
    MOVE   #128,Y1                              ; 2   2    image row,column order
    MOVE   #-261,R1                             ; 2   2    jump to next row
    MOVE   #OutputImage,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
    MOVE   Y1,X:ImageRowsCnt                    ; 1   1    number of rows to process
    PUSH   LC                                   ; 2   2    save possible incoming LC
    PUSH   LA                                   ; 2   2    save possible incoming LA
Rows_Traverse:
    DO     Y1,Cols_Traverse                     ; 2   3    process all columns
    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    for 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   #CoeffMask,R3                        ; 1   1    back to first coeff
    MOVE   Y1,N                                 ; 1   1    for 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    store output computed pixel
Cols_Traverse:
;  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
    DECW   X:ImageRowsCnt                       ; 1   1    decrement to do numb of rows
    BGT    Rows_Traverse                        ; 1   6/4  continue until all row done
    POP    LA                                   ; 1   1    restore incoming LA
    POP    LC                                   ; 1   1    restore incoming LC
                                                ; _______
;                                Total:         39    13N^2+14N+18
;                                                     Kernel: 13
```

B.1.13 Sine-Wave Generation

The following two sine-wave generation benchmarks are provided:

- Double integration technique
- Second order oscillator

B.1.13.1 Double Integration Technique

Figure B-11 gives a graphical overview of the double integration technique.

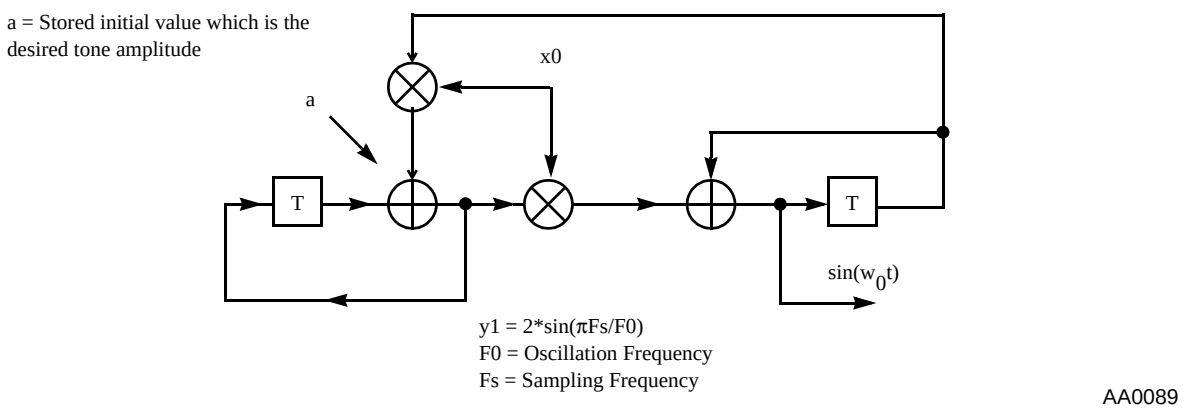


Figure B-11. Sine Wave Generator — Double Integration Technique

```
    opt     cc
    CLR     B                               ; 1   1    integration initial value
    MOVE    #$4000,A                        ; 2   2    initial value, tone amplitude
    MOVE    #0,N                            ; 1   1    set for no post-increment

    MOVE    #$4532,Y1                       ; 2   2    2*sin(pi*Fs/Fo)
    MOVE    #$1,R1                          ; 1   1    arbitrary location for store
    MOVE    Y1,Y0                           ; 1   1    copy for 2nd integ component

    DO      X0,EndDO1_13_1                  ; 2   3    repeat x0 times
    MAC     Y1,B1,A      B,X:(R1)+N         ; 1   1    accumulate 1st integration
    MAC     -Y0,A1,B                        ; 1   1    2nd integration
EndDO1_13_1:
    MOVE                 B,X:(R1)           ; 1   1    final value stored
;                                           _________
;                                   Total:  13    2N+12
```

B.1.13.2 Second Order Oscillator

Figure B-12 gives a graphical overview of a second order oscillator.

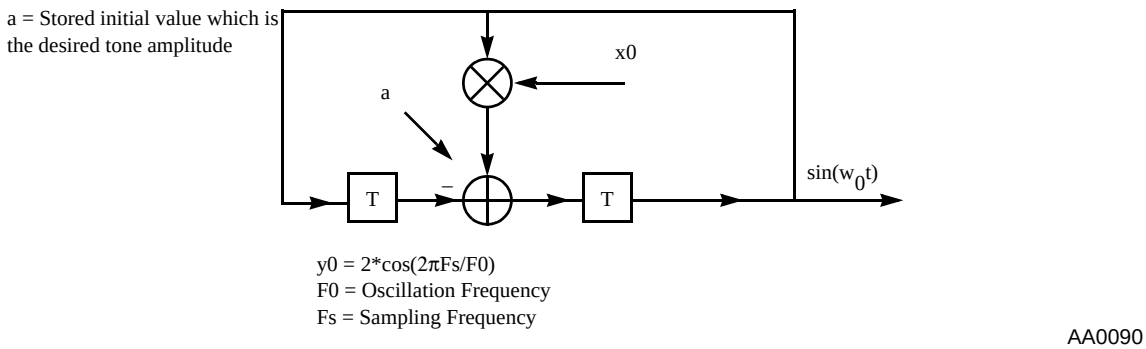


Figure B-12. Sine Wave Generator — Second Order Oscillator

```
        opt     cc
        CLR     A                       ; 1    1     integration initial value
        MOVE    #$4000,Y1               ; 2    2     initial value, tone amplitude
        MOVE    #$6D4B,Y0               ; 2    2     2*sin(pi*Fs/Fo)
        MOVE    #$1,R1                  ; 1    1     arbitrary location for store
        MOVE    #DummyLoc13,R0          ; 1    1     temporary location to swap val
        MOVE    #0,N                    ; 1    1     set for no post-increment
        DO      X0,EndDO1_13_2          ; 2    3     repeat x0 times
        MAC     -Y1,Y0,A                ; 1    1     1st integration
        NEG     A           Y1,X:(R1)+N ; 1    1     correct and store a result
        MAC     Y1,Y0,A                 ; 1    1     2nd integration
        MOVE                A,X:(R0)+N  ; 1    1     use TempLoc for swapping values
        TFR     Y1,A        X:(R0)+N,Y1 ; 1    1     prepare for next integration
EndDO1_13_2:
        MOVE                Y1,X:(R1)   ; 1    1     final approx stored
;                                       ____________
;                           Total:      16     5N+12
```

B.1.14 Array Search

The following two array search benchmarks are provided:

- Index of the highest signed value
- Index of the highest positive value

B.1.14.1 Index of the Highest Signed Value

```
        opt     cc
        MOVE    #A_Vec14,r0                     ; 2    2    vec addr, must be lwr 32k
        MOVEI   #N_,X0                          ; 2    2    load number of elements
        MOVE    #-(A_Vec14+1),N                 ; 2    2    N calc index into A_Vec14
        CLR     A           X:(R0)+,B           ; 1    1    set lowest, load 1st elmnt
        DO      X0,EndDO1_14_1                  ; 2    3    repeat # elmnts times
        ABS     B                               ; 1    1    for largest this not req
        CMP     B,A                             ; 1    1    which magnitude largest?
        TLE     B,A         R0,R1               ; 1    1    transfer if A<=B & pntrs
        MOVE    X:(R0)+,B                       ; 1    1    load next element,
                                                ;           R0 has +1 of desired pntr
EndDO1_14_1:
        LEA     (R1)+N                          ; 1    1    R1 is corrected for index
;                                               __________
;                                       Total:  14     4N+11 (worst case)
```

B.1.14.2 Index of the Highest Positive Value

```
      opt     cc
      MOVE    #A_Vec14,R0                       ; 2    2    vec addr, must be lwr 32k
      MOVEI   #N_/2,Y1                          ; 2    2    load even number of elements
      MOVE    #-(A_Vec14+2),N                   ; 2    2    N calc index into A_Vec14
      CLR     A             X:(R0)+,X0          ; 1    1    set lowest, load 1st elmnt
      DO      Y1,EndDO1_14_2                    ; 2    3    repeat # elmnts times (
      CMP     X0,A          X:(R0)+,Y0          ; 1    1    which pos element largest?
      TLE     X0,A          R0,R1               ; 1    1    transfer if A<=X0 & pntrs
      CMP     Y0,A          X:(R0)+,X0          ; 1    1    which pos element largest?
      TLE     Y0,A          R0,R1               ; 1    1    transfer if A<=Y0 & pntrs
EndDO1_14_2:
      NOP                                       ; 1    1    required to access R1
      LEA     (R1)+N                            ; 1    1    R1 is corrected for index
;                                               _________
;                                       Total:  15     2N+12 (worst case)
```

B.1.15 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.

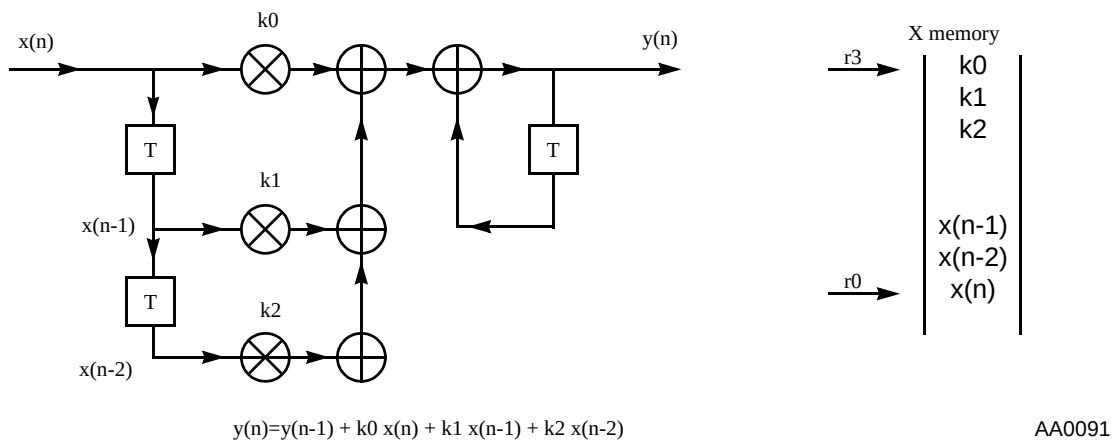


Figure B-13. Proportional Integrator Differentiator Algorithm

B.1.15.1 PID (Version 1)

```
;    y(n) = y(n-1) + k0 x(n) + k1 x(n-1) + k2 x(n-2)
; The constants and input history stored in low mem to take adv of short addressing.
     opt    cc
     MOVE   #X_Vec15+2,r0                      ; 1   1   load address of x(n-1)
     PUSH   M01                                ; 2   2   save state of M01
     MOVE   #2,M01                             ; 2   2   r0 mod 3
     MOVE   #K_Vec15,R3                        ; 1   1   load address of constants

     MOVE            X:(R0)+,B                 ; 1   1   y(n-1) in accum
     MOVE            X:(R0)+,Y0  X:(R3)+,X0    ; 1   1   x(n-2) & k2
     MAC    X0,Y0,B  X:(R0)+,Y0  X:(R3)+,X0    ; 1   1   x(n-1) & k1 [accum+k2*x(n-2)]
     MAC    Y0,X0,B              X:(R3)+,X0    ; 1   1   k0    [accum+k1*x(n-1)]
     MOVEP           X:InputValue,Y0           ; 1   1   x(n)
     MACR   Y0,X0,B                            ; 1   1   y(n)=prev + x(n)*k0
     MOVE            B,X:(R0)                  ; 1   1   save y(n)
     MOVEP           B,X:Output                ; 1   1   write y(n) to peripheral
     POP    M01                                ; 1   1   restore original M01
;                                              ________
;                                Total:        15    15
; shift out x(n-2), x(n-1) becomes x(n-2) and x(n) becomes x(n-1) for next iterate.
```

B.1.15.2 PID (Version 2)

```
;   A faster version of the PID
;   y(n) = y(n-1) + k0 x(n) + k1 x(n-1) + k2 x(n-2)
; The constants and input history stored in low mem to take adv of short addressing.
    opt    cc
    MOVE   #X_Vec15,R0                        ; 1    1   load address of x(n-2)
    MOVE   #K_Vec15,R3                        ; 1    1   load address of constants
    CLR    B                                  ; 1    1   zero the y(n) accum
; B accumulator holds y(n-1), Y1 holds the K0 coefficient
    MOVE              X:(R0)+,Y0  X:(R3)+,X0  ; 1    1   x(n-2) & k2
    MAC    X0,Y0,B    X:(R0)+,Y0  X:(R3)-,X0  ; 1    1   x(n-1) & k1 [accum+k2*x(n-2)]
    MAC    Y0,X0,B                            ; 1    1   k0   [accum+k1*x(n-1)]
    MOVEP             X:InputValue,X0         ; 1    1   x(n)
    MACR   Y1,X0,B    X0,X:(R0)+              ; 1    1   save x(n) [accum+k0*x(n)]
    MOVEP             B,X:Output              ; 1    1   write y(n) to peripheral
;                                             _________
;                                             9      9
```

B.1.16 Autocorrelation Algorithm

```
      opt    cc
      MOVE   #Corr_Vec16,R1                      ; 2   2    results
      MOVE   #Frame_Vec16,R2                     ; 2   2    start of frame elements
      DO     #LPC+1,EndDO1_16_1                  ; 2   3
      MOVE   R2,R3                               ; 1   1    start from term
      CLR    B                                   ; 1   1    clear corr for this run
      MOVE   #Frame_Vec16,R0                     ; 2   2    start of elements
      LEA    (R2)+                               ; 1   1    adjust pntr to traverse
      MOVE   LC,Y1                               ; 1   1    capture # remaining terms
      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    compute correlation term
      MOVE   B0,X:(R1)+                          ; 1   1    store lwr 16-bits of rslt
      MOVE   B1,X:(R1)+                          ; 1   1    store upr 16-bits of rslt
EndDO1_16_1:                                     ; _______
;                                                19    (p+1)^2(N-p/2) + 15(p+1) + 9
```

Glossary

See Section A.1, “Notation,” on page A-1 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 bit
CCR	condition code register
CID	chip identification register
CGDB	core global data bus
CMOS	complementary metal oxide semiconductor
COFF	common object file format
COP	computer operating properly
COPDIS	COP timer disable
CPU	central processing unit
CS	carry bit set
D/A	digital-to-analog
DAC	digital-to-analog converter
DRM	debug request mask bit

DSC digital signal controller

E extension bit

EM1–EM0 event modifier bits

EX external X memory bit

EXT extension register

FH FIFO halt bit

FIFO first-in-last-out

GE greater than or equal to

GPIO general-purpose input/output

GT greater than

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
IPL	interrupt priority level
IPR	interrupt priority register
K&R	Kernighan and Ritchie
L	limit bit
LA	loop address register
LC	loop counter register
LE	less than or equal to
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

LT less than

MA, MB operating modes

MAC multiply-accumulate

MCU microcontroller unit

MIPS million instructions per second

M01 modifier register

MR mode register

MS most significant

MSB most significant bit

MSP most significant portion

N offset register

N negative bit in condition code 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)

PAB program address bus

PC program counter

PGDB peripheral global data bus

PWD power-down mode bit

PLL phase-locked loop

R rounding bit

Rj address registers (R0–R3)

Rn address registers (R0–R3, SP)

SA saturation bit

SBO software breakpoint occurrence

SD stop delay bit

SP stack pointer

SPI serial peripheral interface

SR status register

SSI synchronous serial interface

SZ size bit

TAP test access port

TO trace occurrence

U unnormalized bit

V overflow bit

WWW World Wide Web

X external

XAB1 X memory address bus one

XAB2 X memory address bus two

XDB2 X memory data bus two

XP X/P memory bit

Z zero bit

Index

D

E

F

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

DSP56800FM
Rev. 3.1, 11/2005