DSP56001

24-BIT
DIGITAL SIGNAL PROCESSOR
USER'S MANUAL

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

MOTOROLA

SECTION 1
INTRODUCTION

The DSP56001 and DSP56000 are user-programmable, CMOS digital signal processors (DSPs) which are optimized to execute DSP algorithms in as few operations as possible, while maintaining a high degree of accuracy. The architecture has been designed to maximize throughput in data-intensive DSP applications. This design provides a dual-natured, expandable architecture with sophisticated on-chip peripherals and general-purpose I/O. The architecture, on-chip peripherals, and the low power consumption of the DSP56000/DSP56001 have minimized the complexity, cost, and design time needed to add the power of DSP to any design.

The DSP56000 is read-only memory (ROM) based, and is factory programmed with user software for minimum cost in high-volume applications. The DSP56001 is an off-the-shelf random-access memory (RAM) based processor designed to load its program from an external source. The difference between the two processors is their respective on-chip memory resources. A secure version of the DSP56000, which prevents unauthorized access to the internal program memory, is also available.

This manual is written for both the DSP56000 and DSP56001. Normally, the reference will be to the DSP56000/DSP56001. However, when the two processors differ, they will be cited individually.

1.1 ORIGIN OF THE DSP56000 ARCHITECTURE

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

- Filtering of signals
- Convolution, which is the mixing of two signals
- Correlation, which is a comparison of two signals
- Rectification of a signal
- Amplification of a signal
- Transformation of a signal

All of these functions have traditionally been performed using analog circuits. Only recently has technology provided the processing power necessary to digitally perform these and other functions using DSPs.

Figure 1-1 shows a description 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

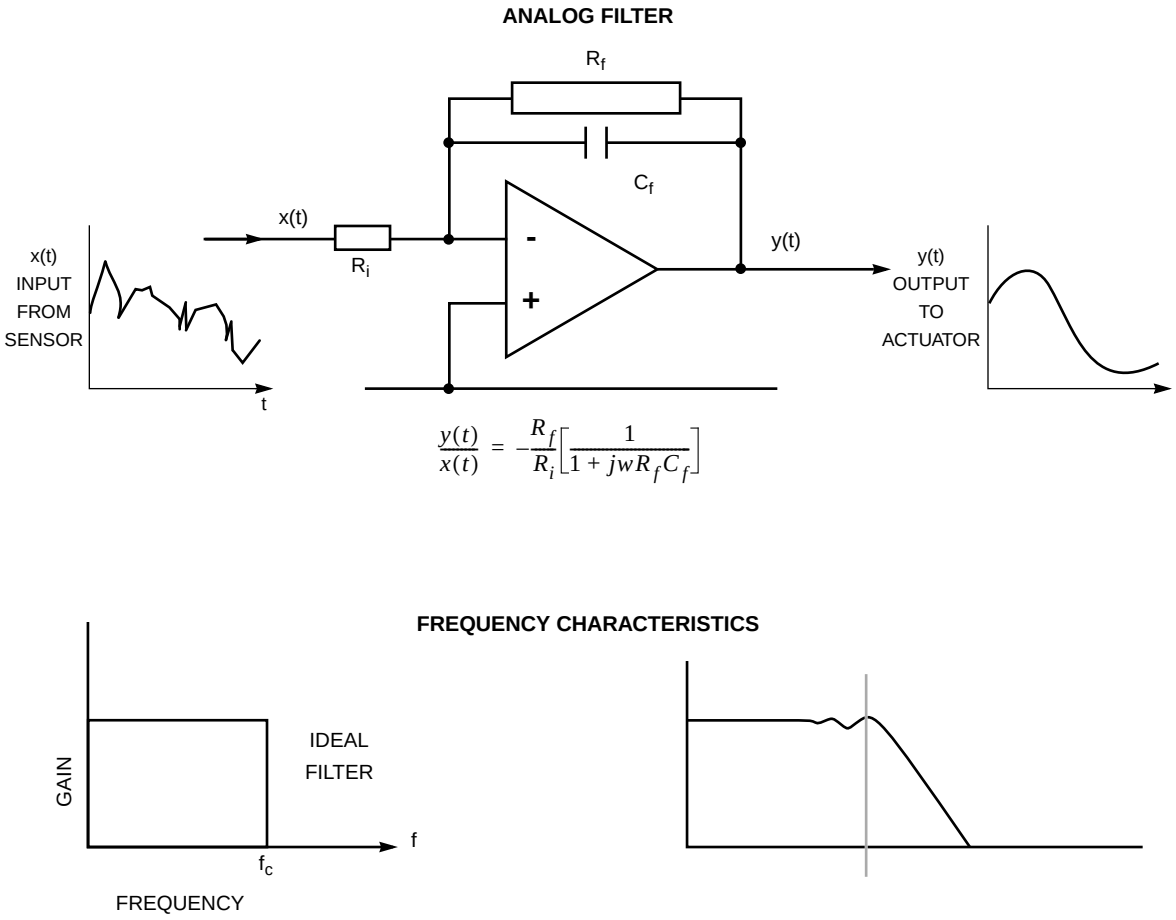


Figure 1-1 Analog Signal Processing

for acceptable response, considering variations in temperature, component aging, power-supply variation, and component accuracy. The resulting circuit typically has low noise immunity, requires adjustments, and is difficult to modify.

The equivalent circuit using a DSP is shown in Figure 1-2. This application requires an analog-to-digital (A/D) converter and digital-to-analog (D/A) converter in addition to the DSP. Even with these additional parts, the component count can be lower using a DSP due to the high integration available with current components.

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

.The filter implemented by the DSP is strictly a matter of software. The DSP can directly implement any filter that can also be implemented using analog techniques. Also, adaptive filters can be easily implemented using DSP, whereas these filters are extremely difficult to implement using analog techniques.

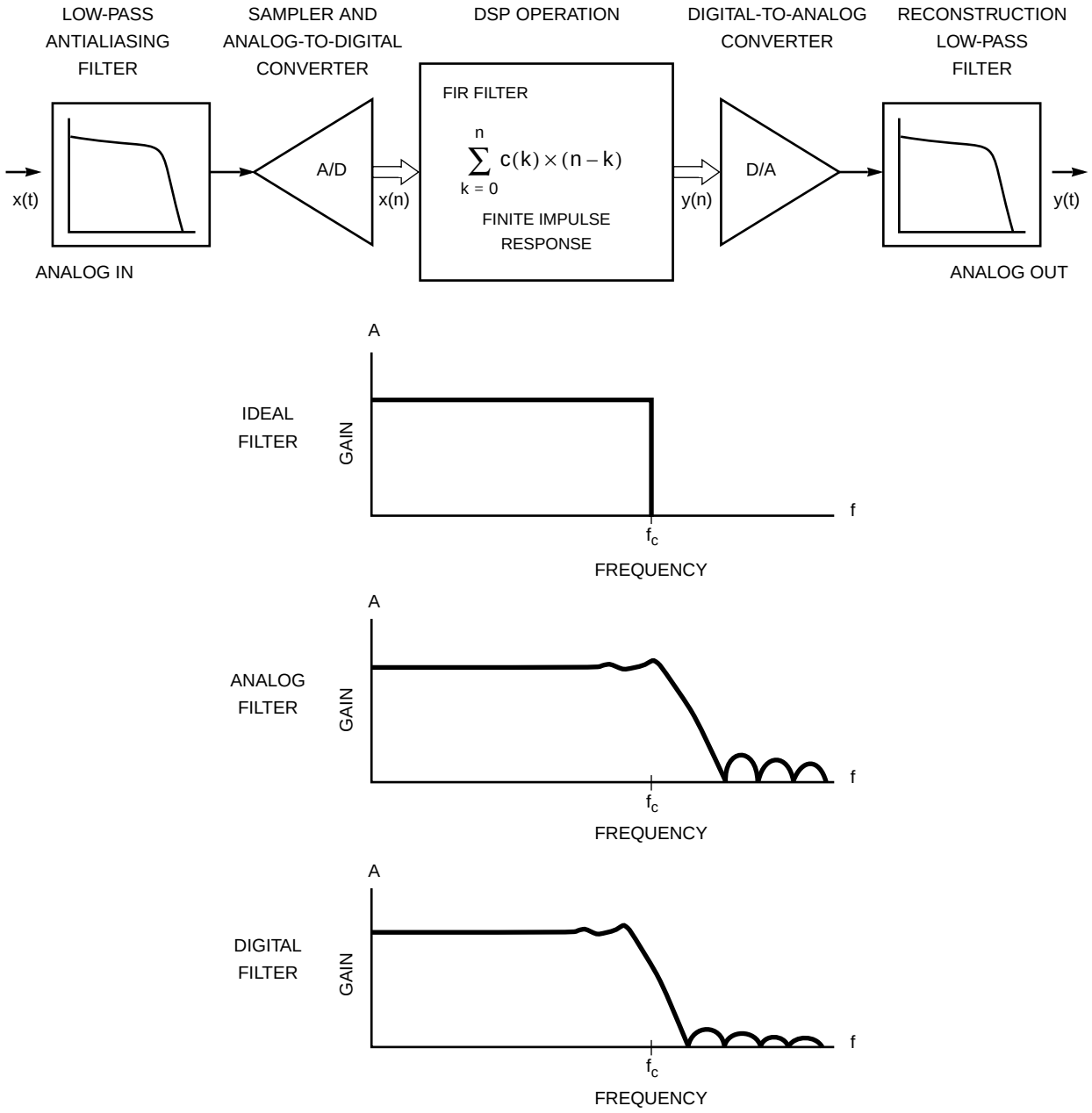


Figure 1-2 Digital Signal Processing

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

Fewer components
Stable, deterministic performance
Wide range of applications
High noise immunity and power-supply rejection
Self-test can be built in

- No filter adjustments
- Filters with much closer tolerances
- Adaptive filters easily implemented

The DSP56000/DSP5001 was not designed for a particular application but was designed to execute commonly used DSP benchmarks in a minimum time for a single-multiplier architecture. For example, a cascaded, 2nd-order, four-coefficient infinite impulse response (IIR) biquad section has four multiplies for each section. For that algorithm, the theoretical minimum number of operations for a single-multiplier architecture is four per section. Table 1-1 shows a list of benchmarks with the number of instruction cycles the DSP56000/DSP56001 uses compared to the number of multiplies in the algorithm.

Table 1-1 Benchmark Summary in Instruction Cycles

Benchmark	DSP56000/DSP56001 Number of Cycles	Number of Algorithm Multiplies
Real Multiply	3	1
N Real Multiplies	2N	N
Real Update	4	1
N Real Updates	2N	N
N Term Real Convolution (FIR)	N	N
N Term Real * Complex Convolution	2N	N
Complex Multiply	6	4
N Complex Multiplies	4N	N
Complex Update	7	4
N Complex Updates	4N	4N
N Term Complex Convolution (FIR)	4N	4N
N^{th} - Order Power Series	2N	2N
2^{nd} - Order Real Biquad Filter	7	4
N Cascaded 2^{nd} - Order Biquads	4N	4N
N Radix Two FFT Butterflies	6N	4N

These benchmarks and others are used independently or in combination to implement functions. The characteristics of these functions are controlled by the coefficients of the benchmarks being executed. Useful functions using these and other benchmarks include the following:

Digital Filtering
- Finite Impulse Response (FIR)
- Infinite Impulse Response (IIR)
- Matched Filters (Correlators)
- Hilbert Transforms
- Windowing
- Adaptive Filters/Equalizers

Signal Processing
- Compression (e.g., Linear Predictive Coding of Speech Signals)
- Expansion
- Averaging
- Energy Calculations
- Homomorphic Processing
- Mu-law/A-law to/from Linear Data Conversion

Data Processing
- Encryption/Scrambling
- Encoding (e.g., Trellis Coding)
- Decoding (e.g., Viterbi Decoding)

Numeric Processing
- Scaler, Vector, and Matrix Arithmetic
- Transcendental Function Computation (e.g., Sin(X), Exp(X))
- Other Nonlinear Functions
- Pseudo-Random-Number Generation

Modulation
- Amplitude
- Frequency
- Phase

Spectral Analysis
- Fast Fourier Transform (FFT)
- Discrete Fourier Transform (DFT)
- Sine/Cosine Transforms
- Moving Average (MA) Modeling
- Autoregressive (AR) Modeling
- ARMA Modeling

Useful applications are based on combining these and other functions. DSP applications affect almost every area in electronics because any application for analog electronic circuitry can be duplicated using DSP. The advantages in doing so are becoming more compelling as DSPs become faster and more cost effective.

DSPs are also being used as high-speed math processors in many purely digital computer applications. Some typical applications for DSPs are presented in the following list:

Telecommunication
- Tone Generation
- Dual-Tone Multifrequency (DTMF)
- Subscriber Line Interface
- Full-Duplex Speakerphone
- Teleconferencing
- Voice Mail
- Adaptive Differential Pulse Code Modulation (ADPCM) Transcoder
- Medium-Rate Vocoders
- Noise Cancelation
- Repeaters
- Integrated Services Digital Network (ISDN) Transceivers
- Secure Telephones

Data Communication
- High-Speed Modems
- Multiple Bit-Rate Modems
- High-Speed Facsimile

Radio Communication
- Secure Communications
- Point-to-Point Communications
- Broadcast Communications
- Cellular Mobile Telephone

Computer
- Array Processors
- Work Stations
- Personal Computers
- Graphics Accelerators

Image Processing
- Pattern Recognition
- Optical Character Recognition
- Image Restoration
- Image Compression
- Image Enhancement
- Robot Vision

Graphics
- 3-D Rendering
- Computer-Aided Engineering (CAE)
- Desktop Publishing
- Animation

Instrumentation
- Spectral Analysis
- Waveform Generation
- Transient Analysis
- Data Acquisition

Speech Processing
- Speech Synthesizer
- Speech Recognizer
- Voice Mail
- Vocoder
- Speaker Authentication
- Speaker Verification

Audio Signal Processing
- Digital AM/FM Radio
- Digital Hi-Fi Preamplifier
- Noise Cancelation
- Music Synthesis
- Music Processing
- Acoustic Equalizer

High-Speed Control
- Laser-Printer Servo
- Hard-Disk Servo
- Robotics
- Motor Controller
- Position and Rate Controller

Vibration Analysis
- Electric Motors
- Jet Engines
- Turbines

Medical Electronics
- Cat Scanners
- Sonographs
- X-Ray Analysis
- Electrocardiogram
- Electroencephalogram
- Nuclear Magnetic Resonance Analysis

Digital Video
- Digital Television
- High-Resolution Monitors

Radar and Sonar Processing
- Navigation
- Oceanography
- Automatic Vehicle Location
- Search and Tracking

Seismic Processing
- Oil Exploration
- Geological Exploration

As shown in Figure 1-3, the keys to DSP are as follows:

- The Multiply/Accumulate (MAC) operation
- Fetching operands for the MAC
- Program control to provide versatile operation
- Input/Output to move data in and out of the DSP

MAC is the basic operation used in DSP. Figure 1-3 shows how the architecture of the DSP56000/DSP56001 was designed to match 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 DSP56000/DSP56001 by using two separate memories (X and Y) 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 accumulator as needed. Since the two memories and the MAC are independent, it is possible to perform two moves (a multiply and an accumulate) in a single operation. As a result, many of the benchmarks shown in Table 1-1 can be executed at or near the theoretical maximum speed for a single-multiplier architecture.

Figure 1-3 shows how the MAC, memories, and program control unit in Figure 1-3 are configured in the DSP56000/DSP56001. Three independent memories and memory buses move two operands to the MAC while concurrently fetching a program instruction. The address generation unit (AGU) is divided into two arithmetic units which independently control the X and Y memories and feed operands to the MAC. Figure 1-3 also features an additional block labeled "I/O". Many other DSPs need external communications circuitry to interface with peripheral circuits (such as A/D converters, D/A converters, or host processors). The DSP56000/DSP56001 provides on-chip serial and parallel interfaces, represented by the I/O block, to simplify this connection problem. Figure 1-4 is a block diagram of the DSP56000 showing all the major

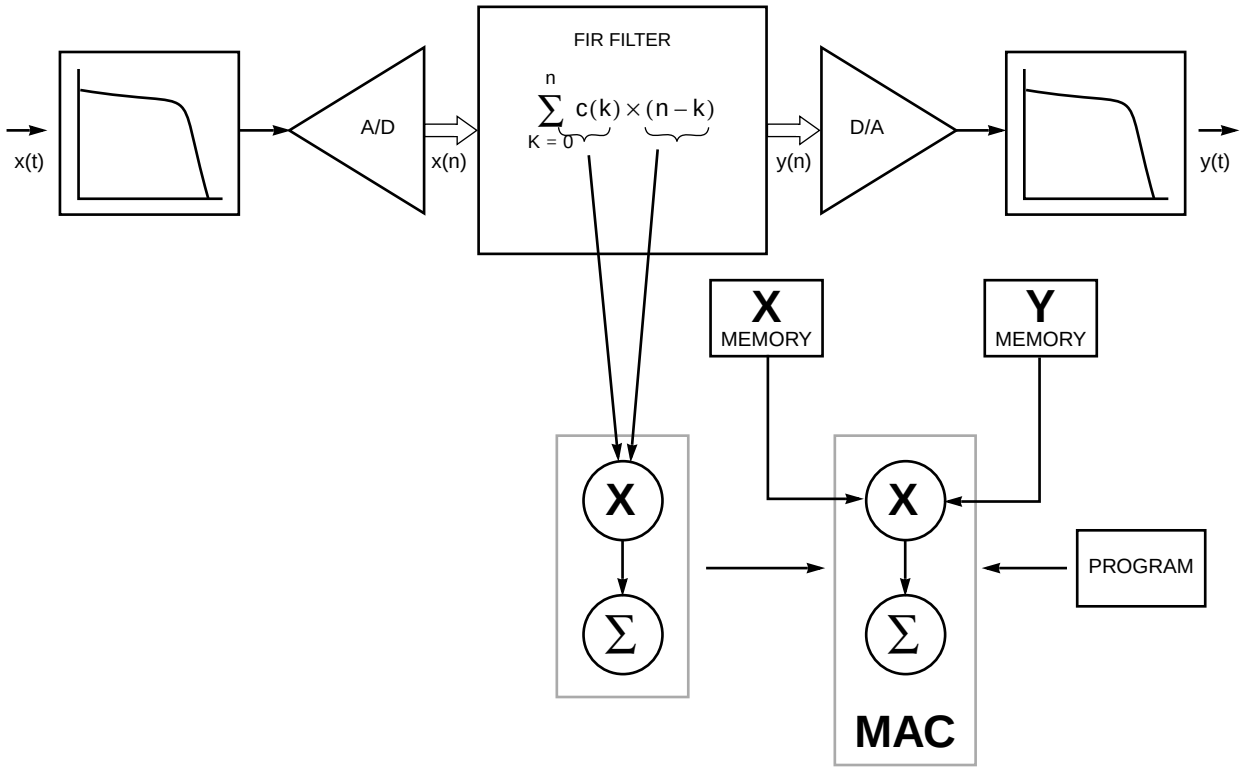


Figure 1-3 DSP Hardware Origins

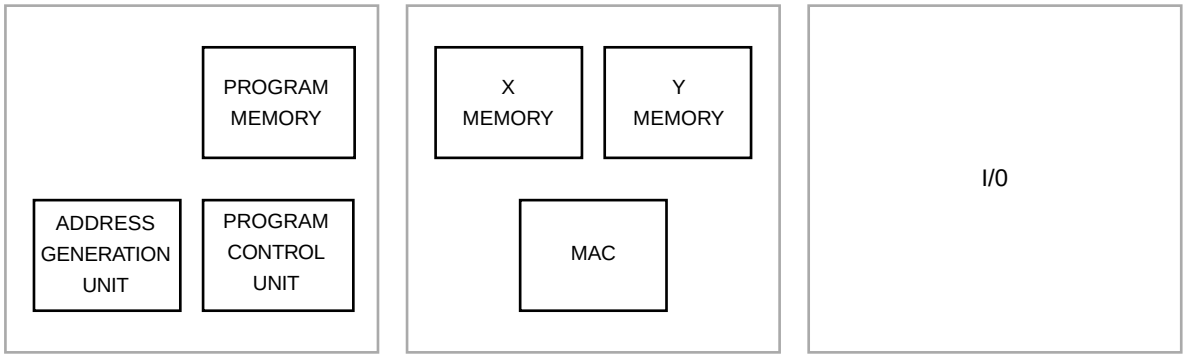


Figure 1-3 DSP Block Diagram

blocks with their interconnecting buses. The DSP56000 Family of processors has a dual Harvard architecture optimized for MAC operations

1.2 SUMMARY OF DSP56000 FAMILY FEATURES

The DSP56000 and DSP56001 are the first two members of Motorola's Family of HCMOS, low-power, general-purpose DSPs. The DSP56001 features 512 words of full-speed, on-chip, program RAM, two preprogrammed data ROMs, and special on-chip bootstrap

hardware to permit convenient loading of user programs into the program RAM. The DSP56001 is an off-the-shelf part, since it has no user-programmable, on-chip ROMs. The DSP56000 features 3.75K words of full-speed, on-chip, program ROM instead of 512 words of program RAM.

The heart of the processor consists of three execution units operating in parallel: the data arithmetic logic unit (ALU), the AGU, and the program control unit. The DSP56000/DSP56001 has MCU-style on-chip peripherals, program memory, data memory, and a memory expansion port. The MPU-style programming model and instruction set allow straightforward generation of efficient, compact code.

The high throughput of the DSP56000/DSP56001 makes it well-suited for communication, high-speed control, numeric processing, computer applications, and audio applications. The main features facilitating this throughput are as follows:

- **Speed** — At 10.25 million instructions per second (MIPS), the DSP56000/DSP56001 can execute a 1024-point complex Fast Fourier Transform (FFT) in 3.23 ms.
- **Precision** — The data paths are 24 bits wide, providing 144 dB of dynamic range; intermediate results held in the 56-bit accumulators can range over 336 dB.
- **Parallelism** — Each on-chip execution unit (AGU, program control unit, data ALU),

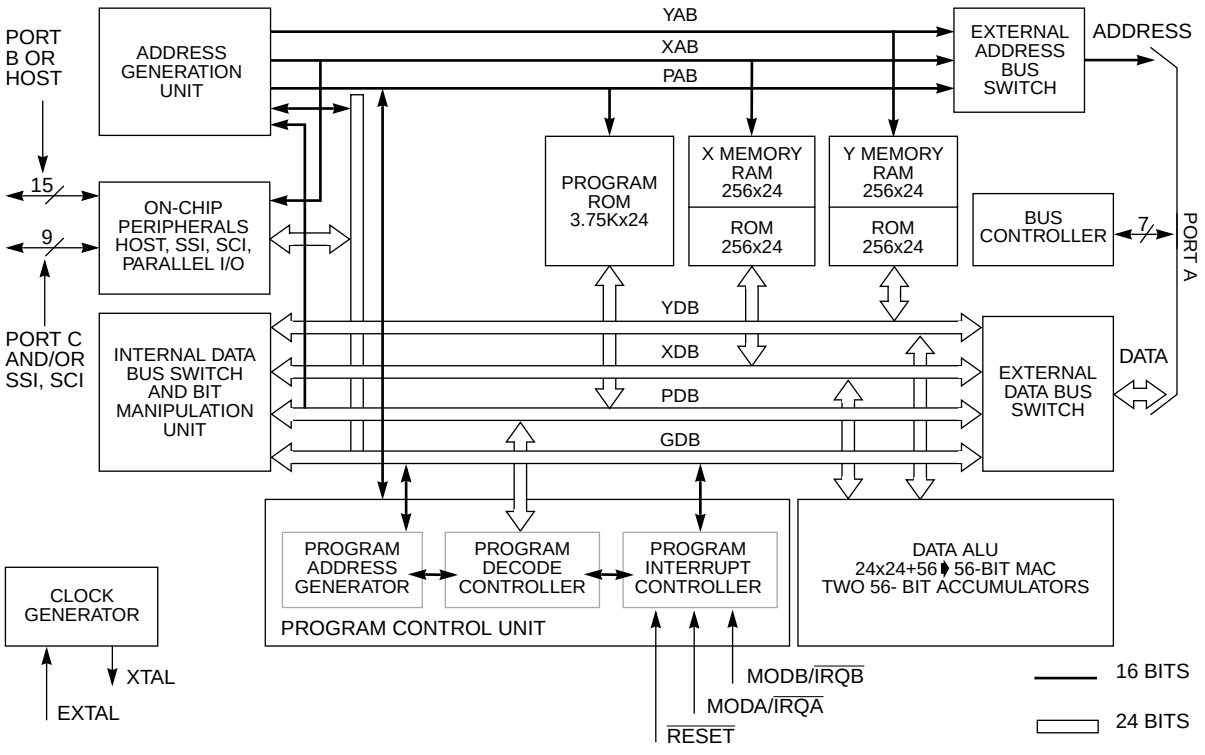


Figure 1-4 DSP56000 Block Diagram

memory, and peripheral operates independently and in parallel with the other units through a sophisticated bus system. The data ALU, AGUs, and program control unit operate in parallel so that an instruction prefetch, a 24-bit x 24-bit multiplication, a 56-bit addition, two data moves, and two address-pointer updates using one of three types of arithmetic (linear, modulo, or reverse-carry) can be executed in a single instruction cycle. This parallelism allows a four-coefficient IIR filter section to be executed in only four cycles, the theoretical minimum for single-multiplier architecture. At the same time, the two serial controllers can send and receive full-duplex data, and the host port can send/receive simplex data.

- **Integration** — In addition to the three independent execution units, the DSP56000/DSP56001 has six on-chip memories, three on-chip MCU-style peripherals (serial communication interface (SCI), synchronous serial interface (SSI), and host interface), a clock generator, and seven buses (three address and four data), making the overall system low cost, low power, and compact.
- **Invisible Pipeline** — The three-stage instruction pipeline is essentially invisible to the programmer, allowing straightforward program development in either assembly language or a high-level language such as a full Kernighan and Ritchie C.
- **Instruction Set** — The 62 instruction mnemonics are MCU-like, making the transition from programming microprocessors to programming the DSP56000/DSP56001 as easy as possible. The orthogonal syntax supports controlling the parallel execution units. The hardware DO loop instruction and the repeat (REP) instruction make writing straightline code obsolete.
- **DSP56000/DSP56001 Compatibility** — The DSP56001 is identical to the DSP56000 except for the following features:

  12-word x 24-bit, on-chip program RAM instead of 3.75K program ROM

  32-word x 24-bit bootstrap ROM for loading the program RAM from either a byte-wide, memory-mapped ROM or from the host interface

  On-chip X and Y data ROMs preprogrammed as positive Mu-law and A-law to linear expansion tables and a full, four-quadrant sine-wave table, respectively

- **Low Power** — As a CMOS part, the DSP56000/DSP56001 is inherently very low power; however, the following features can reduce power consumption to exceptionally low levels:

  The WAIT instruction shuts off the clock in the central processor portion of the DSP56000/DSP56001.

  The STOP instruction halts the internal oscillator.

  Power increases linearly (approximately) with frequency; thus, reducing the clock frequency reduces power consumption.

1.3 MANUAL ORGANIZATION

This manual is intended to provide practical information to help the user:

- Understand the operation of the DSP56000 Family
- Interface the DSP56000 Family with additional memory
- Design parallel communication links
- Design serial communication links
- Code DSP algorithms
- Code communication routines
- Code data manipulation algorithms
- Locate additional support

The following list describes the contents of each section and each appendix:

Section 2. Architectural Overview and Bus Structure

This section describes each subsystem and the buses interconnecting the major components in the DSP56000/DSP56001.

Section 3. Memory

This section describes and differentiates the memory for the DSP56000 and DSP56001. It describes the program memories, data memories, and the operating mode register (OMR) bits controlling the memory maps.

Section 4. Data Arithmetic Logic Unit

This section describes in detail the data ALU (one of the three execution units comprising the central processor) and its programming model.

Section 5. Address Generation Unit

This section specifically describes the AGU (one of the three execution units comprising the central processor), its programming model, address indirect modes, and address modifiers.

Section 6. Program Control Unit

This section describes in detail the program control unit (one of the three execution units comprising the central processor) and its programming model.

Section 7. Instruction Set Introduction

This section presents a brief description of the syntax, instruction formats, operand/memory references, data organization, addressing modes, and instruction set. A detailed description of each instruction is given in **APPENDIX A INSTRUCTION SET DETAILS**.

Section 8. Processing States

This section describes the five processing states (normal, exception, reset, wait, and stop).

Section 9. Port A

The Port A section describes the external memory port, its control register, and its control signals.

Section 10. Port B

This section describes the port B parallel I/O, host interface, their registers, and the controls to enable/disable them.

Section 11. Port C

This section describes the port C parallel I/O, SCI, SSI, their registers, and the controls to enable/disable them.

Appendix A. Instruction Set Details

A detailed description of each DSP56000/DSP56001 instruction, its use, and its affect on the processor are presented.

Appendix B. Benchmarks

DSP56000/DSP56001 benchmark results are listed in this appendix.

Appendix C. Additional Support

This appendix presents a brief description of current support products and services and information on where to obtain them.

SECTION 2
ARCHITECTURAL OVERVIEW AND BUS STRUCTURE

The DSP56000/DSP56001 architecture has been designed to maximize throughput in data-intensive digital signal processor (DSP) applications. This objective has resulted in a dual-natured, expandable architecture with sophisticated on-chip peripherals and general purpose I/O. The architecture is dual natured in that there are two independent, expandable data memory spaces, two address generation units (AGUs), and a data arithmetic logic unit (ALU) which has two accumulators and two shifter/limiter circuits.

The duality of the architecture facilitates writing software for DSP applications. For example, data is naturally partitioned into X and Y spaces for graphics and image-processing applications, into coefficient and data spaces for filtering applications, and into real and imaginary spaces for performing complex arithmetic.

The major components of the DSP56000/DSP56001 are as follows:

- Data Buses
- Address Buses
- Data ALU
- AGU
- X Data Memory
- Y Data Memory
- Program Control Unit
- Program Memory
- Input/Output:
  - Memory Expansion (Port A)
  - General-Purpose I/O (Ports B and C)
  - Host Interface
  - Serial Communication Interface (SCI)
  - Synchronous Serial Interface (SSI)

Figure 2-1 shows these components for the DSP56000. Figure 2-2 shows these components for the DSP56001. The processors differ only in the on-chip memory resources.The following paragraphs give a brief description for each component.

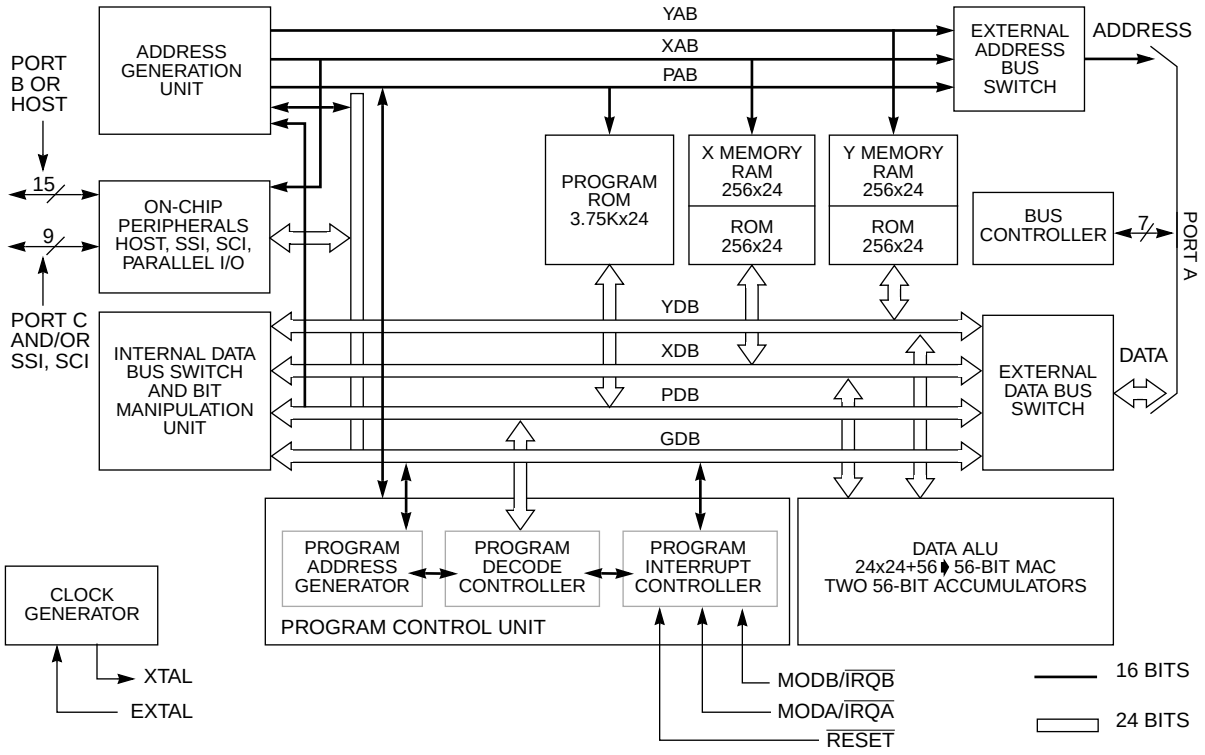


Figure 2-1 DSP56000 Block Diagram

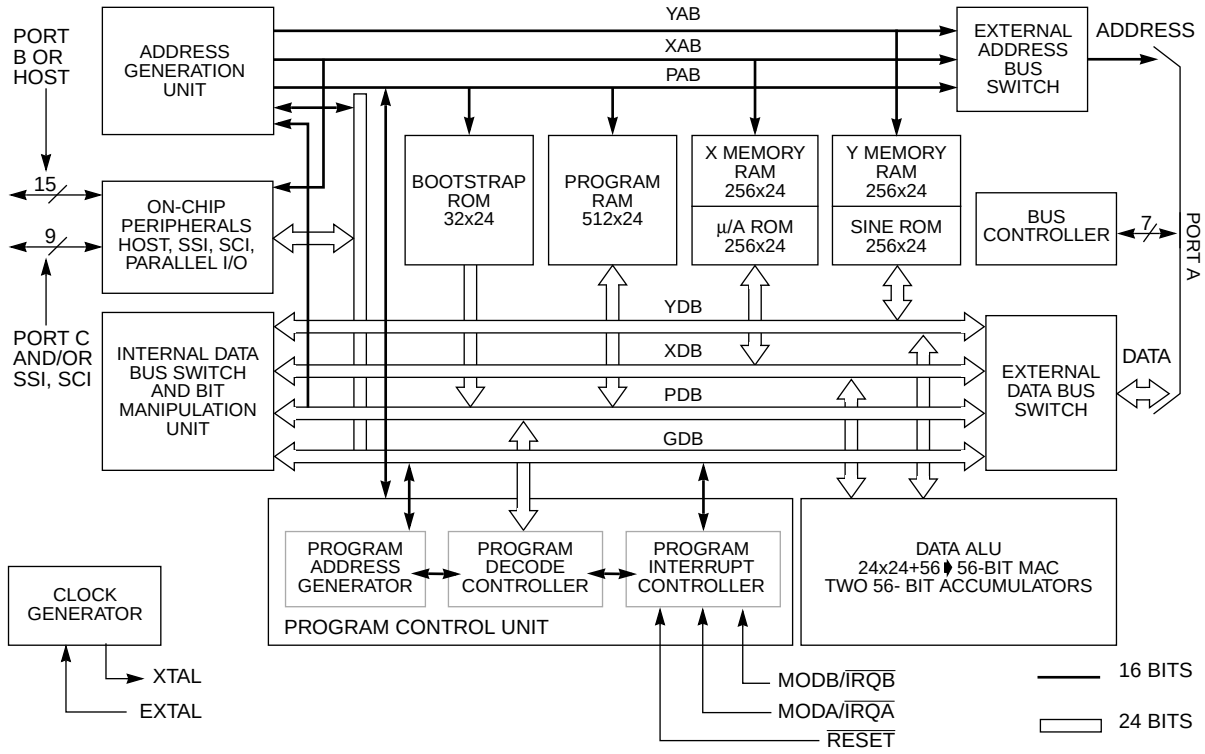


Figure 2-2 DSP56001 Block Diagram

2.1 DATA BUSES

The DSP56000/DSP56001 is organized around the registers of a central processor composed of three independent execution units: the program control unit, the AGU, and the Data ALU.

Data movement on the chip occurs over four, bidirectional, 24-bit buses: the X data bus (XDB), the Y data bus (YDB), the program data bus (PDB), and the global data bus (GDB). The X and Y data buses may also be treated by certain instructions as one 48-bit data bus by concatenation of XDB and YDB. Data transfers between the data ALU and the X data memory or Y data memory occur over XDB and YDB, respectively. XDB and YDB are kept local on the chip to maximize speed and minimize power dissipation. All other data transfers, such as I/O transfers with peripherals, occur over the GDB. Instruction word prefetches occur in parallel over the PDB. The bus structure supports general register-to-register, register-to-memory, and memory-to-register data movement and can transfer up to two 24-bit words and one 56-bit word in the same instruction cycle. Transfers between buses occur in the internal bus switch.

2.2 ADDRESS BUSES

Addresses are specified for internal X data memory and Y data memory on two, unidirectional, 16-bit buses — X address bus (XAB) and Y address bus (YAB). Program memory addresses are specified on the bidirectional program address bus (PAB). External memory spaces are addressed via a single 16-bit, unidirectional address bus driven by a three-input multiplexer that can select the XAB, the YAB, or the PAB. Only one external memory access can be made in an instruction cycle. There is no speed penalty if only one external memory space is accessed in an instruction cycle. If two or three external memory spaces are accessed in a single instruction, there will be a one- or two-instruction-cycle execution delay, respectively. A bus arbitrator controls external access.

2.2.1 Internal Bus Switch

Transfers between buses occur in the internal bus switch. The internal bus switch, which is similar to a switch matrix, can connect any two internal buses without adding any pipeline delays. This flexibility simplifies programming.

2.2.2 Bit Manipulation Unit

The bit manipulation unit is physically located in the internal bus switch block because the internal data bus switch can access each memory space. The bit manipulation unit performs bit manipulation operations on memory locations, address registers, control registers, and data registers over the XDB, YDB, and GDB.

2.3 Data ALU

The data ALU has been designed to process signals which have a wide dynamic range. Special circuitry handles data overflows and roundoff errors.

The data ALU performs all of the arithmetic and logical operations on data operands. It consists of four 24-bit input registers, two 48-bit accumulator registers, two 8-bit accumulator extension registers, an accumulator shifter, two data bus shifter/limiter circuits, and a parallel, single-cycle, nonpipelined multiply-accumulator (MAC) unit. Data ALU operations use fractional twos-complement arithmetic.

Data ALU registers may be read or written over XDB and YDB as 24- or 48-bit operands. The data ALU can perform any of the following operations in a single instruction cycle — multiplication, multiply-accumulate with positive or negative accumulation, convergent rounding, multiply-accumulate with positive or negative accumulation and convergent rounding, addition, subtraction, a divide iteration, a normalization iteration, shifting, and logical operations.

Data ALU source operands, which may be 24, 48, or, in some cases, 56 bits, always originate from data ALU registers. Arithmetic operations always have a 56-bit result stored in an accumulator. Logical operations are performed on 24-bit operands and yield 24-bit results in one of the two accumulators.

The 24-bit data word provides 144 dB of dynamic range, which is sufficient for most real-world applications, since the majority of data converters are 16 bits or less — and certainly not greater than 24 bits. The 56-bit accumulation inside the data ALU provides 336 dB of internal dynamic range so no loss of precision occurs due to intermediate processing.

The data shifter/limiter circuits perform special postprocessing on data read from the ALU accumulator registers A and B out to the XDB or YDB. The data shifters can shift data one bit to the left or one bit to the right as well as pass the data unshifted. Each data shifter has a 24-bit output with overflow indication. The data shifters are controlled by the scaling mode bits in the status register. These shifters permit dynamic scaling of fixed-point data without modifying the program code, which allows block floating-point algorithms to be implemented in a regular fashion. For example, fast Fourier transform (FFT) routines can use this feature to selectively scale each butterfly pass.

“Overflow” occurs when a source operand requires more bits for accurate representation than are available in the destination. To minimize error due to overflow, the DSP56000 writes the maximum (or ‘‘limited’’) signed value the destination can assume when an overflow condition is detected.

In the DSP56000/DSP56001, the data ALU accumulators A and B have extension registers that are used when more than 48-bit accuracy is needed. Therefore, when the extension registers are in use, and either A or B is the source being read over XDB or YDB, data limiters place a ‘‘limited’’ value on XDB or YDB. Such limiting is performed on

the contents of A or B after the contents have been shifted in the shifter. Two limiters allow two-word operands to be limited independently in the same instruction cycle. The two limiters can also be concatenated to form one 48-bit data limiter for long-word operands.

2.4 ADDRESS GENERATION UNIT

All of the address storage and address calculations necessary to indirectly address data operands in memory occur in the AGU. This unit operates in parallel with other chip resources to minimize address generation overhead. The AGU contains eight address registers (R0–R7), eight offset registers (N0–N7), and eight modifier registers (M0–M7). Rn are 16-bit registers that may contain an address or data. The contents of each Rn may be sent to the XAB (65,536 locations), YAB (65,536 locations), or PAB (65,536 locations); thus, 196,608 24-bit data words can be directly addressed. Nn and Mn, which are 16-bit registers normally used in updating or modifying Rn registers, can also be used to store 16-bit data. The AGU registers may be read or written via the GDB as 16-bit operands.

The AGU has two identical address arithmetic units that can generate two 16-bit addresses every instruction cycle — one for any two of the XAB, YAB, or PAB buses. Each of the arithmetic units can perform three types of arithmetic: linear, modulo, and reverse-carry.

2.5 X DATA MEMORY

The on-chip X data random-access memory (RAM), a 24-bit-wide internal static memory, occupies the lowest 256 (0 - 255) locations in X memory space. The on-chip X data read-only memory (ROM) occupies locations 256–511. On the DSP56001, the X data ROM has been programmed as positive Mu-law (128 locations) and A-law (128 locations) 24-bit companding tables useful in telecommunication applications. On the DSP56000, the X data ROM is user defined.

Three on-chip peripherals exist on the DSP56000/DSP56001: an 8-bit parallel host microprocessor unit/direct memory access (MPU/DMA) interface, an SCI, and an SSI. The on-chip peripherals occupy the top 64 locations in X data memory space. Addresses are received from the XAB, and data transfers to the data ALU occur on the XDB. X data memory may be expanded off-chip so that a total of 65,536 locations can be addressed.

2.6 Y DATA MEMORY

The on-chip Y data RAM, a 24-bit-wide internal static memory, occupies the lowest 256 (0 - 255) locations in Y memory space. The on-chip Y data ROM occupies locations 256–511. On the DSP56001, the Y data ROM has been programmed as a full, four-quadrant, 24-bit sine table. On the DSP56000, the Y data ROM is user defined. The off-chip peripheral registers should be mapped into the top 64 locations in Y data memory space.

Addresses are received from the YAB, and data transfers to the data ALU occur on the YDB. Y memory may be expanded off-chip so that a total of 65,536 locations can be

addressed.

2.7 PROGRAM MEMORY

The on-chip program memory consists of a 3.75K-word by 24-bit ROM for the DSP56000 or a 512-word by 24-bit RAM for the DSP56001. Addresses are received from the program control logic (usually the program counter). The interrupt vector addresses for the on-chip resources are located in the bottom 64 locations of program memory. Program memory may be expanded off-chip so that a total of 65,536 locations can be addressed.

Bootstrap ROM is a 32-word by 24-bit factory-programmed ROM used only in the bootstrap mode (operating mode 1). It is available only on the DSP56001; it is not available on the DSP56000. More detailed information on bootstrap ROM is discussed in the DSP56001 Advance Information Data Sheet (ADI1290).

2.8 PROGRAM CONTROL UNIT

The program control unit performs instruction prefetch, instruction decoding, hardware DO loop control, and exception processing. It contains a 15-level by 32-bit system stack memory and the following six directly addressable registers: the program counter (PC), loop address (LA), loop counter (LC), status register (SR), operating mode register (OMR), and stack pointer (SP). The 16-bit PC can address 65,536 locations in program memory space.

2.9 INPUT/OUTPUT

The I/O capability of the DSP56000/DSP56001 is extensive and advanced.Its structure facilitates interfacing into a variety of system configurations, including multiple DSP56000/DSP56001 systems (with or without a host processor), global bus systems with bus arbitration, and many serial configurations, all with minimal additional ‘‘glue’’ logic.

Each I/O interface, which has its own control, status, and data registers, is treated as memory-mapped I/O by the DSP56000/DSP56001. Each interface has several dedicated interrupt vector addresses and control bits to enable/disable interrupts, which minimizes the overhead associated with servicing the device. The interrupt sources can be programmed to one of three maskable priority levels.

The I/O structure consists of a flexible, 47-pin expansion port (Port A) and 24 additional I/O pins. These pins may operate as general-purpose I/O pins, called port B and port C, or they may be allocated to on-chip peripherals (MPU/DMA, SCI, and SSI) under software control.

Port B is a 15-bit I/O interface that may function as general-purpose I/O pins or as host MPU/DMA interface pins.

Port C is a 9-bit I/O interface that may be used as general-purpose I/O pins or as SCI and

SSI pins. These interfaces are described in the following paragraphs.

2.9.1 Expansion Port (Port A)

DSP56000/DSP56001 expansion port is designed to synchronously interface over a common 24-bit data bus which has a wide variety of memory and peripheral devices. These devices include high-speed static RAMs, slower memory devices, and other DSPs and MPUs in master/slave configurations. This variety is possible because the expansion bus timing is programmable.

Two pins can be defined with a control bit to operate as either master processor controls (called “bus strobe” and “bus wait” in this configuration) or as slave processor controls (called “bus request” and “bus grant”).

The expansion bus timing can also be controlled by a bus control register (BCR). The BCR controls the timing of bus interface signals RD and WR, as well as the data output lines. Each of the four memory spaces, X data, Y data, program data, and I/O, has its own 4-bit register in the BCR that can be programmed for inserting up to 15 wait states (one wait state is equal to a clock period or equivalently one-half of an instruction cycle). Thus, external bus timing can be tailored to match the speed requirements of the different memory spaces.

2.9.2 General-Purpose I/O (Ports B and C)

Each Port B and Port C pin may be programmed as a general-purpose I/O pin or as a dedicated, on-chip peripheral pin under software control. A 9-bit port C control register (PCC) allows each port C pin to be programmed for one of these two functions. The port control register associated with port B (PBC) contains only one bit, which programs all 15 pins. Also associated with each general-purpose port is a data direction register, which programs the direction of each pin, and a data register for data I/O. All these registers are memory mapped and read/write, which makes the use of bit manipulation instructions extremely effective.

2.9.3 Host Interface

The host interface is a byte-wide, full-duplex, parallel port that can be connected directly to the data bus of a host processor. The host processor may be any of a number of industry-standard microcomputers or MPUs, another DSP, or DMA hardware. To control data transfers the DSP56000/DSP56001 host interface has an 8-bit, bidirectional data bus: H0–H7 (PB0–PB7); and seven dedicated control lines: HA0, HA1, HA2, HR/$\overline{W}$, $\overline{HEN}$, $\overline{HREQ}$, and $\overline{HACK}$ (PB8–PB14).

The host interface appears as a memory-mapped peripheral occupying eight bytes in the host-processor address space. Separate double buffered transmit and receive data registers allow the DSP56000/DSP56001 and host processor to efficiently transfer data at high speed. Standard, host-processor data move instructions and addressing modes facilitate communication with the host interface. Handshake flags are provided for polled or interrupt-driven data transfers with the host processor. DMA hardware may be used with the handshake flags to efficiently transfer data without using address lines HA0–HA2.

One of the most innovative features of the host interface is the host command feature. With this feature, the host processor can issue vectored exception requests to the

DSP56000/DSP56001. The host may select any one of 32 DSP56000/DSP56001 exception routines to be executed by writing a vector address register in the host interface. This flexibility allows the host programmer to execute up to 32 preprogrammed functions inside the DSP56000/DSP56001. For example, host exception routines allow the host processor to read or write DSP56000/DSP56001 registers, X, Y, or program memory locations, force exception handlers (e.g., SSI, SCI, $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$ exception routines), and perform control and debugging operations.

2.9.4 Serial Communication Interface

The SCI provides a full-duplex port for 8-bit data serial communication to other DSPs, MPUs, or peripherals such as modems. The communication can be either direct or over RS232C-type lines. This interface uses three dedicated pins — transmit data (TXD), receive data (RXD), and SCI serial clock (SCLK). It supports industry-standard asynchronous bit rates and protocols as well as high-speed (up to 2.5 Mbits/sec) synchronous data transmission.

The asynchronous protocols include a multidrop mode for master/slave operation. The SCI consists of separate transmit and receive sections having operations that can be asynchronous with respect to each other by using the internal clock for one and an external clock for the other. A programmable baud-rate generator is included to generate the transmit and receive clocks. An enable and interrupt vector are included so that the baud-rate generator can function as a general-purpose timer when it is not being used by the SCI peripheral.

2.9.5 Synchronous Serial Interface

The SSI is a flexible, full-duplex serial interface that allows the DSP56000/DSP56001 to communicate with a variety of serial devices, including one or more industry-standard codecs, other DSPs, MPUs, and peripherals.

The user can independently define the following characteristics of the SSI: the number of bits per word, the protocol, the clock, and the transmit/receive synchronization.

The user can select three modes: normal, on-demand, and network. The normal mode is typically used to interface with devices on a regular or periodic basis. The data-driven on-demand mode is intended to be used to communicate with devices on a nonperiodic basis. The network mode provides time slots in addition to a bit clock and frame synchronization pulse.

The SSI functions with a range of 2 to 32 words of I/O per frame in the network mode. This mode is typically used in star or ring time division multiplex (TDM) networks with other DSP56000s and/or codecs. The clock can be programmed to be continuous or gated. Since the transmitter and receiver sections of the SSI are independent, they can be programmed to be synchronous (using a common clock) or asynchronous with respect to each other.

The SSI supports a subset of the Motorola SPI. The SSI requires up to six pins, depending on its operating mode. The most common minimum configuration is three pins: transmit data (STD), receive data (SRD), and clock (SCK).

2.10 SIGNAL DESCRIPTION

The DSP56000/DSP56001 is available in an 88-pin pin-grid array package or surface mount. The input and output signals are organized into the following seven functional groups which are shown in Figure 2-1:

1. Port A Address and Data Buses
2. Port A Bus Control
3. Interrupt and Mode Control
4. Power and Clock
5. Host Interface or Port B I/O
6. SCI or Port C I/O
7. SSI or Port C I/O

The signals are discussed in the following paragraphs.

2.10.1 Port A Address and Data Bus

The following signals relate to the Port A address and data bus.

2.10.1.1 Address (A0–A15)

These three-state output pins specify the address for external program and data memory accesses. To minimize power dissipation, A0–A15 do not change state when external memory spaces are not being accessed.

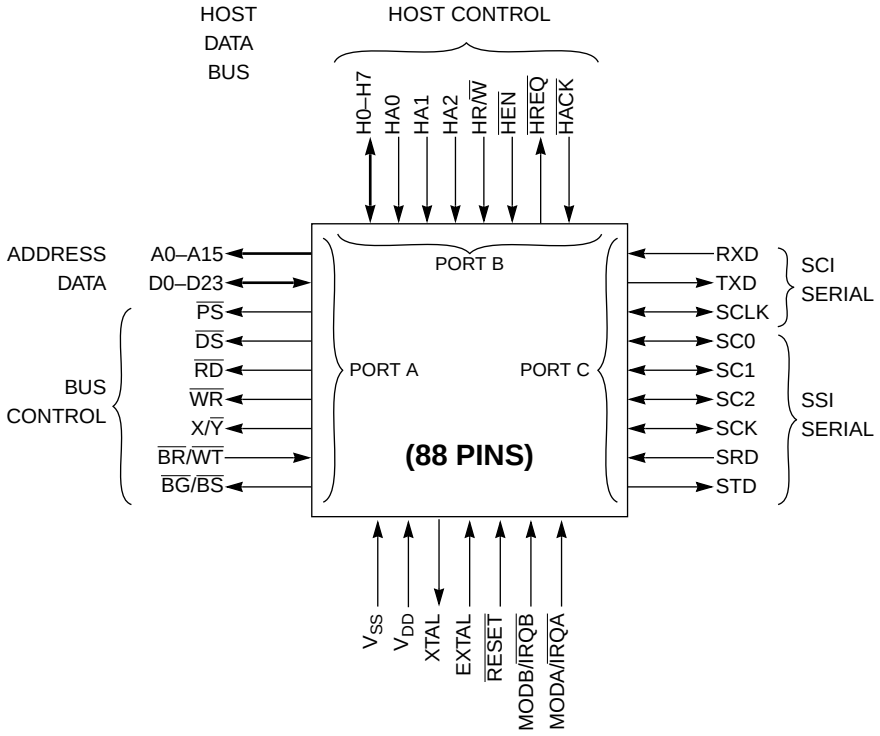


Figure 2-1 DSP56000/DSP56001 Functional Signal Groups

2.10.1.2 Data (D0–D23) These pins provide the bidirectional data bus for external program and data memory accesses. D0–D23 are in the high-impedance state when the bus grant signal is asserted.

2.10.2 Port A Bus Control The Port A bus control signals are discussed in the following paragraphs.

2.10.2.1 Program Memory Select ($\overline{\text{PS}}$) This three-state output is asserted only when external program memory is referenced (see Table 2-1)**.**

Table 2-1 Program and Data Memory Select Encoding

$\overline{\text{PS}}$	$\overline{\text{DS}}$	X/$\overline{\text{Y}}$	External Memory Reference
1	1	1	No Activity
1	0	1	X Data Memory on Data Bus
1	0	0	Y Data Memory on Data Bus
0	1	1	Program Memory on Data Bus (Not Exception)
0	1	0	External Exception Fetch: Vector or Vector +1 (Development Mode Only)
0	0	X	Reserved
1	1	0	Reserved

2.10.2.2 Data Memory Select ($\overline{\text{DS}}$) This three-state output is asserted only when external data memory is referenced (see Table 2-1).

2.10.2.3 X/$\overline{\text{Y}}$ Select (X/$\overline{\text{Y}}$) This three-state output selects which external data memory space (X or Y) is referenced by $\overline{\text{DS}}$ (see Table 2-1l).

2.10.2.4 Read Enable ($\overline{\text{RD}}$) This three-state output is asserted to read external memory on the data bus (D0–D23).

2.10.2.5 Write Enable ($\overline{\text{WR}}$) This three-state output is asserted to write external memory on the data bus (D0–D23).

2.10.2.6 Bus Request/Wait ($\overline{\text{BR}}$/$\overline{\text{WT}}$) The bus request input ($\overline{\text{BR}}$) allows another device such as a processor or DMA controller to become the master of the external data bus (D0–D23) and external address bus (A0–A15). When bit 7 of the operating mode register (OMR)

is clear and $\overline{BR}$ is asserted, the DSP56000/DSP56001 will always release Port A, including A0–A15, D0–D23, and the bus control pins by placing them in the high-impedance state after execution of the current instruction has been completed.

If OMR bit 7 is set, $\overline{BR}$/$\overline{WT}$ acts as an input that allows an external device to force wait states during an external Port A operation for as long as $\overline{WT}$ is asserted.

2.10.2.7 Bus Grant/Bus Strobe ($\overline{BG}$/$\overline{BS}$) If OMR bit 7 is clear, this output is asserted to grant an external bus request after Port A has been released. If OMR bit 7 is set, this pin assumes bus strobe and is asserted when the DSP accesses Port A.

2.10.3 Interrupt and Mode Control

The signals described in the following paragraphs are the interrupt and mode control signals for the DSP56000/DSP56001.

2.10.3.1 Mode Select A/External Interrupt Request A (MODA/$\overline{IRQA}$) and Mode Select B/External Interrupt Request B (MODB/$\overline{IRQB}$) These two inputs have dual functions: 1) to select the initial chip operating mode and 2) to receive an interrupt request from an external source.

MODA and MODB are read and internally latched in the DSP when the processor exits the reset state. After leaving the reset state, the MODA and MODB pins automatically change to external interrupt requests, $\overline{IRQA}$ and $\overline{IRQB}$.

After leaving the reset state, the chip operating mode can be changed by software. $\overline{IRQA}$ and $\overline{IRQB}$ can be programmed to be level sensitive or negative edge triggered. When edge triggered, triggering occurs at a voltage level and is not directly related to the fall time of the interrupt signal. However, as the fall time of the interrupt signal increases, theprobability for noise on $\overline{IRQA}$ or $\overline{IRQB}$ to generate multiple interrupts also increases.

2.10.3.2 Reset ($\overline{RESET}$) This Schmitt-trigger input pin is used to reset the DSP56000/DSP56001. When $\overline{RESET}$ is asserted, the DSP56000/DSP56001 is initialized and placed in the reset state. When $\overline{RESET}$ is deasserted, the initial chip operating mode is latched from the MODA and MODB pins. When coming out of $\overline{RESET}$, deassertion occurs at a voltage level and is not directly related to the rise time of the $\overline{RESET}$ signal; however, the probability of noise on $\overline{RESET}$ generating multiple resets increases with increasing rise time of the $\overline{RESET}$ signal.

2.10.4 Power and Clock

The power and clock signals are presented in the following paragraphs.

2.10.4.1 Power (V_{CC}), Ground (GND)

There are five sets of power and ground pins: two pairs for internal logic; one power, and two ground for Port A address and control pins; one power and two ground for Port A data pins; and one pair for peripherals.

2.10.4.2 External Clock/Crystal Input (EXTAL)

EXTAL interfaces the internal crystal oscillator input to an external crystal or an external clock.

2.10.4.3 Crystal Output (XTAL)

This output connects the internal crystal oscillator output to an external crystal. If an external clock is used, XTAL should not be connected.

2.10.5 Host Interface

The following paragraphs discuss the host interface signals.

2.10.5.1 Host Data Bus (H0–H7)

This bidirectional data bus transfers data between the host processor and the DSP56000/DSP56001. This bus is an input unless enabled by a host processor read. It is high impedance when $\overline{\text{HEN}}$ is deasserted. H0–H7 can be programmed as general-purpose parallel I/O pins (PB0–PB7) when the host interface is not being used.

2.10.5.2 Host Address (HA0–HA2)

These inputs provide the address selection for each host interface register. HA0–HA2 can be programmed as general-purpose parallel I/O pins (PB8–PB10) when the host interface is not being used.

2.10.5.3 Host Read/Write (HR/$\overline{\text{W}}$)

This input selects the direction of data transfer for each host processor access. HR/$\overline{\text{W}}$ can be programmed as a general-purpose I/O pin (PB11) when the host interface is not being used.

2.10.5.4 Host Enable ($\overline{\text{HEN}}$)

This input enables a data transfer on the host data bus. When $\overline{\text{HEN}}$ is asserted and HR/$\overline{\text{W}}$ is high, H0–H7 become outputs and DSP56000/DSP56001 data may be read by the host processor. When $\overline{\text{HEN}}$ is asserted and HR/$\overline{\text{W}}$ is low, H0–H7 become inputs, and host data is latched inside the DSP. When $\overline{\text{HEN}}$ is deasserted, the host data bus is three-stated. Normally, a chip select signal derived from host address decoding and an enable clock are used to generate $\overline{\text{HEN}}$. $\overline{\text{HEN}}$ can be programmed as a general-purpose I/O pin (PB12) when the host interface is not being used.

2.10.5.5 Host Request ($\overline{\text{HREQ}}$)

This open-drain output signal is used by the DSP56000/DSP56001 host interface to request service from the host processor, DMA controller, or a simple external controller. $\overline{\text{HREQ}}$ can be programmed as a general-purpose (not open-drain) I/O pin (PB13) when the host interface is not being used.

2.10.5.6 Host Acknowledge ($\overline{\text{HACK}}$)

This input has two functions: 1) to provide a host acknowledge handshake signal for DMA transfers and 2) to receive a host interrupt acknowledge compatible with M68000 Family processors. $\overline{\text{HACK}}$ may be programmed as a general-purpose I/O pin (PB14) when the host interface is not being used.

2.10.6 Serial Communications Interface

The following signals relate to the SCI.

2.10.6.1 Receive Data (RXD)

This input receives byte-oriented serial data and transfers the data to the SCI receive shift register. RXD can be programmed as a general-purpose I/O pin (PC0) when the SCI RXD function is not being used.

2.10.6.2 Transmit Data (TXD)

This output transmits serial data from the SCI transmit shift register. TXD can be programmed as a general-purpose I/O pin (PC1) when the SCI TXD function is not being used.

2.10.6.3 SCI Serial Clock (SCLK)

This bidirectional pin provides an input or output clock from which the transmit and/or receive baud rate is derived in the asynchronous mode, and from which data is transferred in the synchronous mode. SCLK can be programmed as a general-purpose I/O pin (PC2) when the SCI SCLK function is not being used.

2.10.7 Synchronous Serial Interface

The SSI signals are presented in the following paragraphs.

2.10.7.1 Serial Clock Zero (SC0)

The SSI uses this bidirectional pin for control by the SSI as a flag or receiver clock. SC0 can be programmed as a general-purpose I/O pin (PC3) when the SSI SC0 function is not being used.

2.10.7.2 Serial Control One (SC1)

The SSI uses this bidirectional pin to control flag or frame synchronization. SC1 can be programmed as a general-purpose I/O pin (PC4) when the SSI SC1 function is not being used.

2.10.7.3 Serial Control Two (SC2)

The SSI uses this bidirectional pin to control frame synchronization only. SC2 can be programmed as a general-purpose I/O pin (PC5) when the SSI SC2 function is not being used.

2.10.7.4 SSI Serial Clock (SCK)

This bidirectional pin provides the serial bit rate clock for the SSI. SCK can be programmed as a general-purpose I/O pin (PC6) when SCK is not being used.

2.10.7.5 SSI Receive Data (SRD)

This input pin receives serial data into the SSI receive shift register. SRD can be programmed as a general-purpose I/O pin (PC7) when SRD is not being used.

2.10.7.6 SSI Transmit Data (STD)

This output pin transmits serial data from the SSI transmit shift register. STD can be programmed as a general-purpose I/O pin (PC8) when the SSI STD function is not being used.

SECTION 3
MEMORY SPACES

This section is divided into two major subsections, the DSP56000 and DSP56001. Each subsection describes the memory spaces available and the operating modes that redefine these memory spaces.

3.1 OVERVIEW

The memory of the DSP56000/DSP56001 can be partitioned in several ways to provide high-speed parallel operation and additional off-chip memory expansion. Program and data memory are separate, and the data memory is, in turn, divided into two separate memory spaces, X and Y. Both the program and data memories can be expanded off-chip. There are also two on-chip data read-only memories (ROMs) that can overlay a portion of the X and Y data memories and a bootstrap ROM (DSP56001 only) that can overlay part of the program random-access memory (RAM). The data memories are divided into two independent spaces to work with the two address arithmetic logic units (ALUs) to feed two operands simultaneously to the data ALU.

3.2 DSP56000 MEMORY INTRODUCTION

The three independent memory spaces of the DSP56001, X data, Y data, and program, are shown in Figure 3-1. The memory spaces are configured by control bits in the operating mode register (OMR). The operating mode control bits (MA and MB) in the OMR control the program memory map and select the reset vector address. The data ROM enable (DE) bit in the OMR controls the X and Y data memory maps and enables/disables the internal X and Y data ROMs. The bootstrap memory on the DSP56000 is used only for factory testing and should not be invoked by the user.

3.2.1 X Data Memory

The on-chip X data RAM is a 24-bit-wide, internal, static memory occupying the lowest 256 locations (0–255) in X memory space. The on-chip X data ROM (factory programmed to user specifications like the program ROM) occupies locations 256–511 in the X data memory space and is controlled by the DE bit in the OMR. The on-chip peripheral registers occupy the top 64 locations of the X data memory ($FFC0–$FFFF). The 16-bit addresses are received from the XAB, and 24-bit data transfers to the data ALU occur on the XDB. The X memory may be expanded to 64K off-chip.

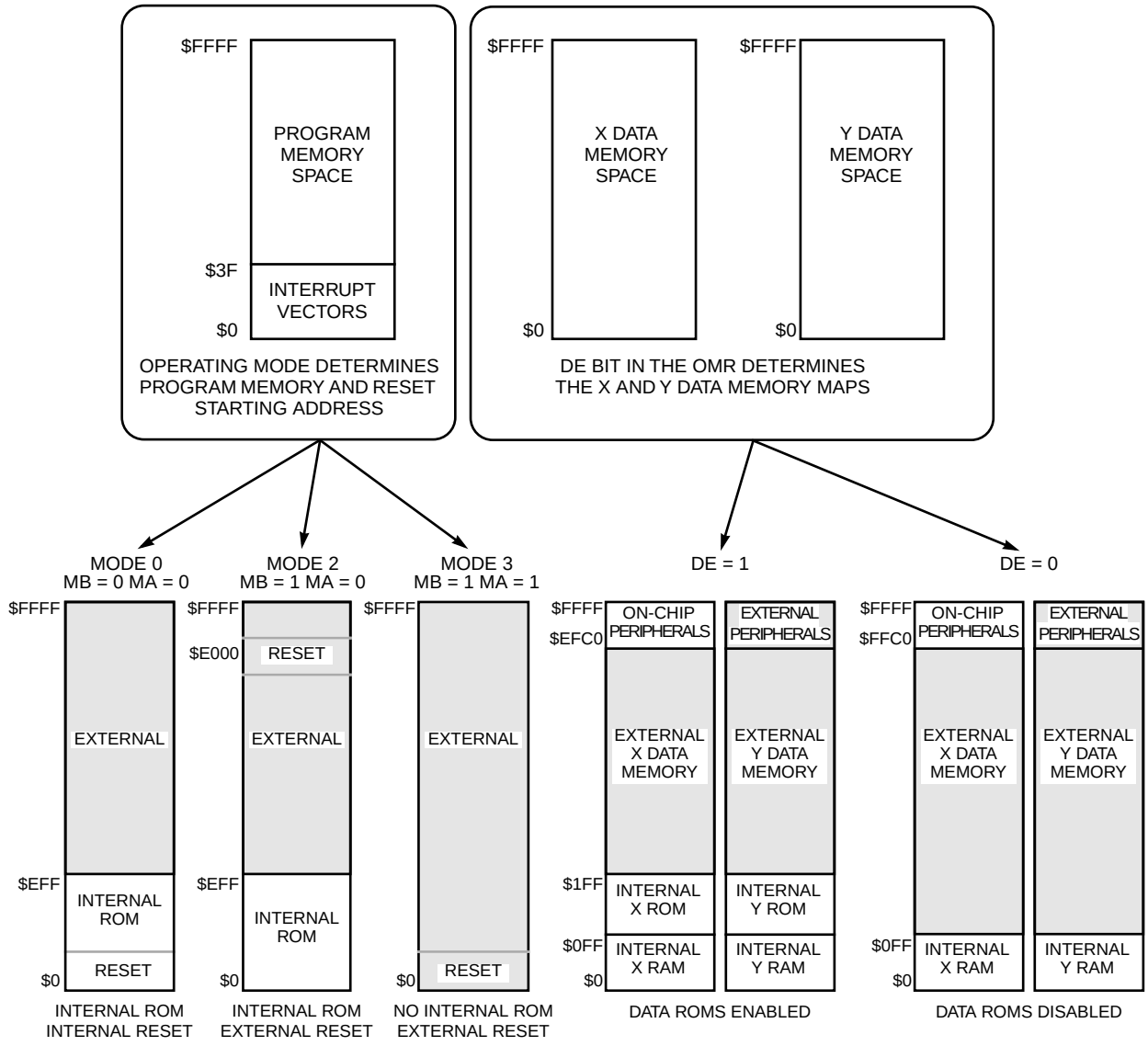


Figure 3-1 DSP56000 Memory Map

3.2.2 Y Data Memory

The on-chip Y data RAM is a 24-bit-wide, internal, static memory occupying the lowest 256 locations (0–255) in the Y memory space. The on-chip Y data ROM (factory programmed to user specifications like the program ROM) occupies locations 256–511 in Y data memory space and is controlled by the DE bit in the OMR. The off-chip peripheral registers should be mapped into the top 64 locations ($FFC0–$FFFF) to take advantage of the move peripheral data (MOVEP) instruction. The 16-bit addresses are received from the YAB, and 24-bit data transfers to the data ALU occur on the YDB. Y memory may be

expanded to 64K off-chip.

3.2.3 Program Memory

On-chip program memory consists of a 3840-location by 24-bit, high-speed ROM (3.75K x 24) that is enabled/disabled by the MA and MB bits in the OMR. When the on-chip program memory is disabled, either off-chip memory or a special mode 1 ROM is selected for program memory.

NOTE: The mode 1 ROM is used only for test purposes on the DSP56000 and should not be invoked by the user.

Addresses are received from the program control logic (usually the program counter) over the PAB. Off-chip program memory may be written using move program memory (MOVEM) instructions. The interrupt vectors for the on-chip resources are located in the bottom 64 locations ($0000–$003F) of program memory. Program memory may be expanded to 64K off-chip.

3.2.4 Chip Operating Modes

The DSP operating modes determine the memory maps for program and data memories and the startup procedure when the DSP leaves the reset state. The MODA and MODB pins are sampled as the DSP leaves the reset state, and the initial operating mode of the DSP is set accordingly. When the reset state is exited, the MODA and MODB pins become general-purpose interrupt pins, IRQA and IRQB. One of three initial operating modes is selected: single chip, normal expanded, or development. Chip operating modes can be changed by writing the operating mode bits (MB, MA) in the OMR. Changing operating modes does not reset the DSP. It is desirable to disable interrupts immediately before changing the OMR to prevent an interrupt from going to the wrong memory location. Also, one no-operation (NOP) instruction should be included after changing the OMR to allow for remapping to occur.

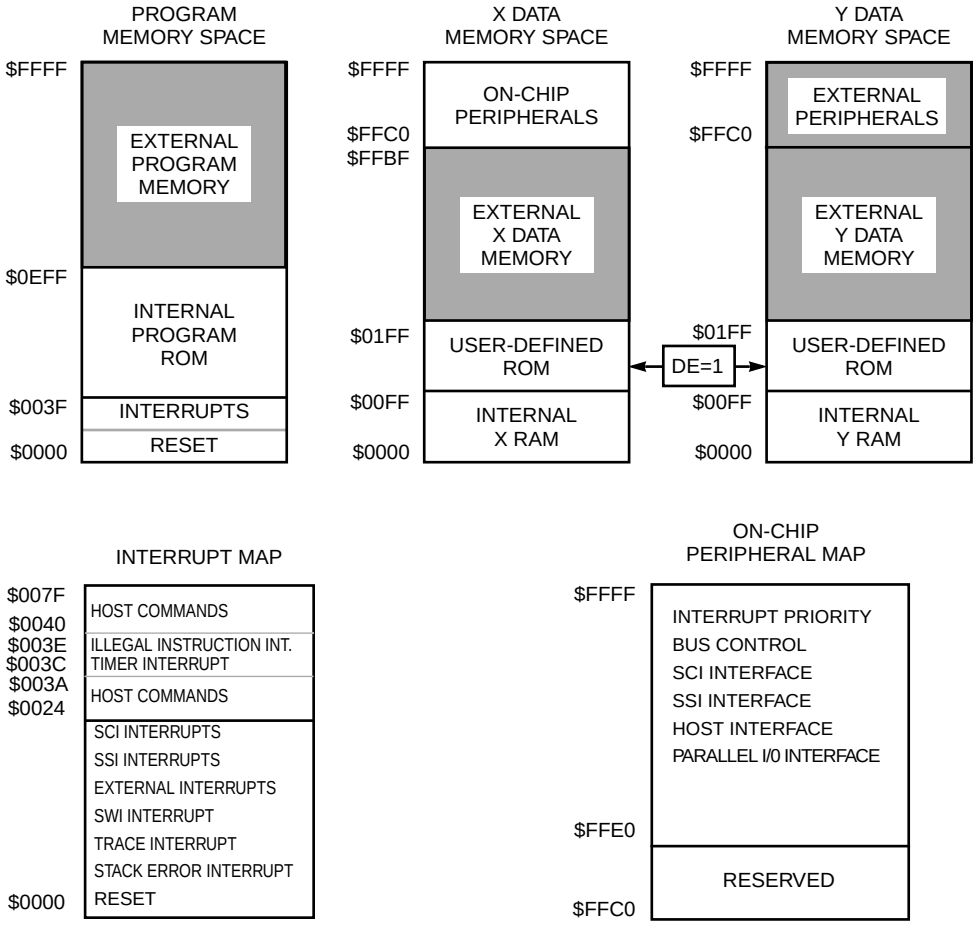
Some pins on the DSP are mode independent; whereas, the use of others depends on the particular operating mode. Specifically, external address bus, data bus, and bus con-

Table 3-1 Initial DSP56000 Operating Mode Summary

Operating Mode	MOD B	MODA	Description
0	0	0	Single-Chip Mode
1	0	1	Single-Chip Mode
2	1	0	Normal Expanded Mode
3	1	1	Development Mode

trol pins are affected by the particular operating mode. Table 3-1 shows the mode assignments.

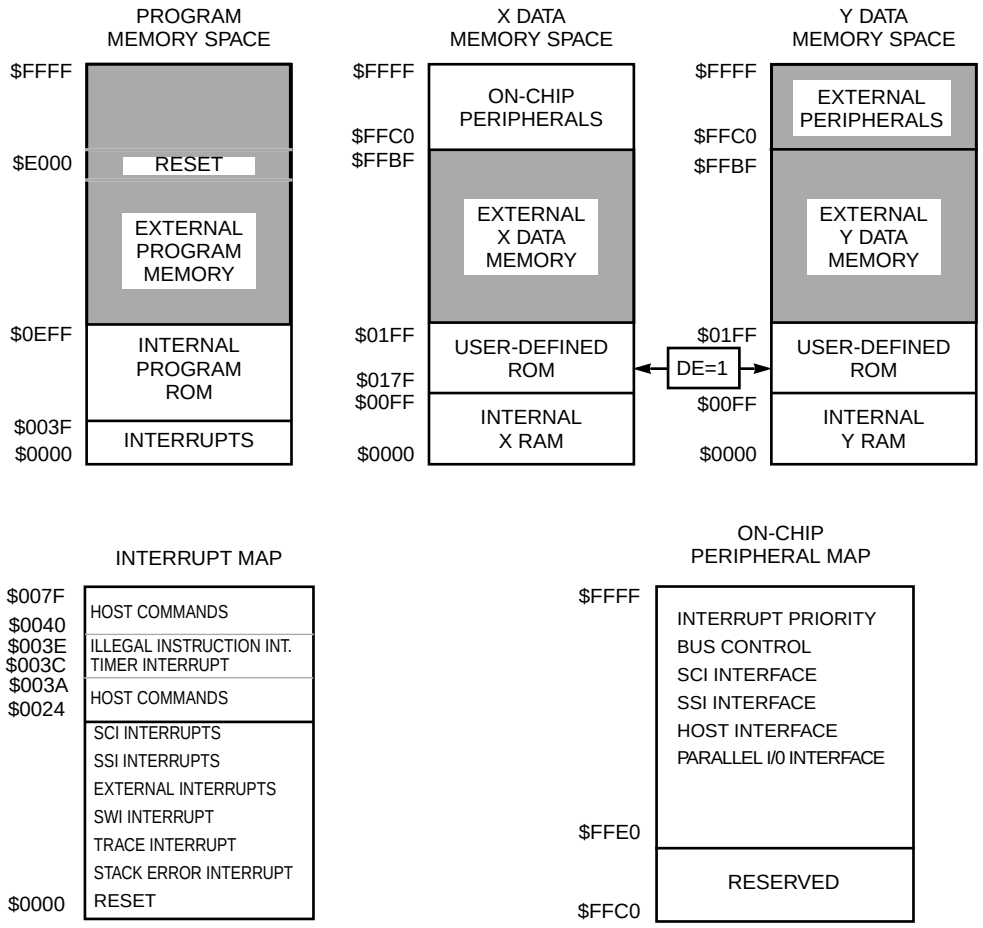
3.2.4.1 Single-Chip Mode (Mode 0). In the single-chip mode, all internal program and data RAM memories are enabled. A hardware reset causes the DSP to jump to internal program memory location $0000 ($=hexadecimal notation) and resume execution. The memory map for this mode is shown in Figure 3-2. The memory maps for mode 0 and mode 2 (see Figure 3-3) are identical. The difference between the two modes is that reset vectors to program memory location $0000 in mode 0 and vectors to location $E000 in mode 2.



NOTE: Addresses $FFC0–$FFFF in X data memory are NOT available externally.

Figure 3-2 Memory Map for DSP56000 Mode 0: Single-Chip Mode

3.2.4.2 Mode 1. Mode 1 is the same as Mode 0 on the DSP56000. It is recommended that



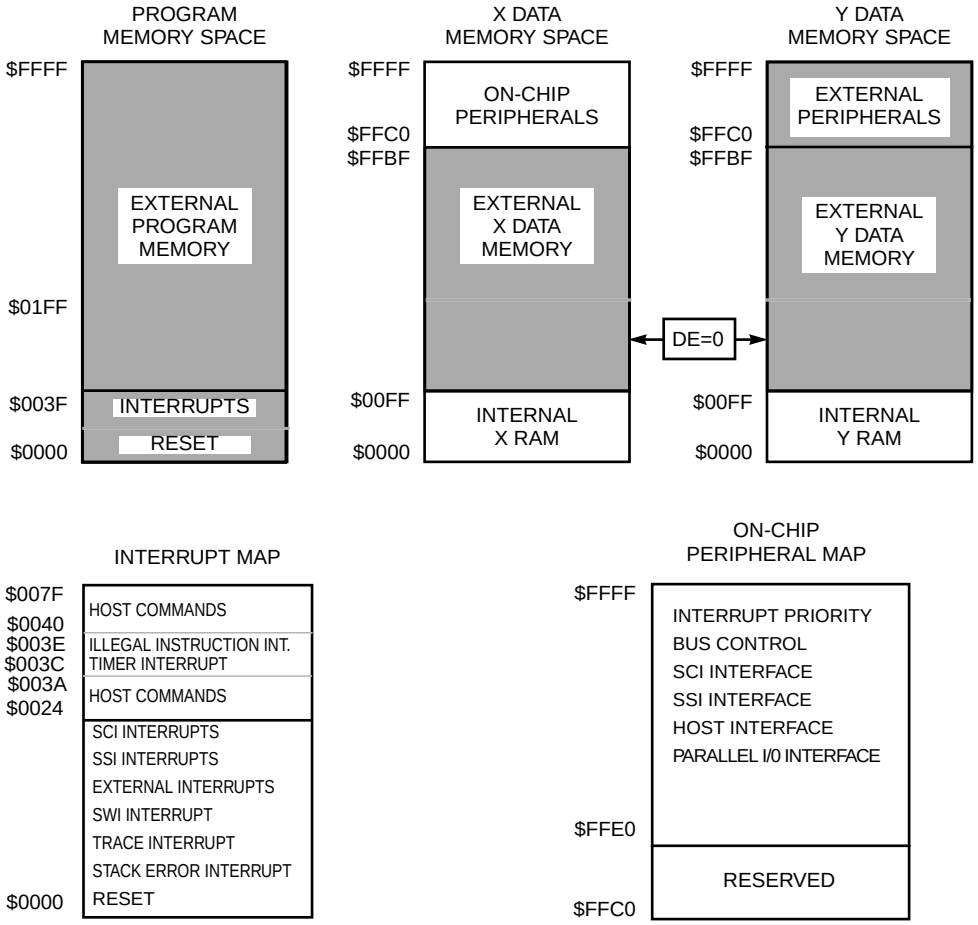
NOTE: Addresses $FFC0–$FFFF in X data memory are NOT available externally.

Figure 3-3 Memory Map for DSP56000 Mode 2: Normal Expanded Mode

this mode not be invoked by the user.

3.2.4.3 Normal Expanded Mode (Mode 2). Mode 2 is almost identical to mode 0 (see **3.2.4.1 Single-Chip Mode (Mode 0). In the single-chip mode, all internal program and data RAM memories are enabled. A hardware reset causes the DSP to jump to internal program memory location $0000 ($=hexadecimal notation) and resume execution. The memory map for this mode is shown in Figure 3-2. The memory maps for mode 0 and mode 2 (see Figure 3-3) are identical. The difference between the two modes is that reset vectors to program memory location $0000 in mode 0 and vectors to location $E000 in mode 2.** for further information).

3.2.4.4 Development Mode (Mode 3). The development mode is similar to the normal expanded mode except that internal program memory is disabled. All references to program memory space are directed to external program memory, which is accessed on the external data bus. The memory map for this mode is shown in Figure 3-4. DSP56000 chips with bad



NOTE: Addresses $FFC0–$FFFF in X data memory are NOT available externally.

Figure 3-4 Memory Map for DSP56000 Mode 3: Development Mode

or obsolete internal program ROM code can be used with external program memory in the development mode. The memory map in Figure 3-4is shown with DE arbitrarily set to zero.

3.2.5 Security ROM Version (DSP56000)[1]

The security ROM version of the DSP56000 is a standard DSP56000 that has been modified to prevent unauthorized access to the program contained in the DSP program ROM. This protection is accomplished in two ways. First, the DSP is forced into the single-chip mode at reset. The chip powers up in single-chip mode, and it is not possible to enter any other mode on powerup. The MODA/IRQA and MODB/IRQB pins are configured only as IRQA and IRQB and cannot be used to change the mode. Second, the programmer must avoid fetches from external program memory =m i.e., the user code must be placed only in internal

1. For additional information concerning this part, contact the Motorola field office.

program ROM. This placement prevents the execution of unauthorized code that might be used to dump the contents of the program ROM.

3.3 DSP56001 MEMORY INTRODUCTION

The three independent memory spaces of the DSP56001, X data, Y data, and program, are shown in Figure 3-5. The memory spaces are configured by control bits in the OMR. The MA and MB control bits in the OMR control the program memory map and select the reset vector address. The DE bit in the OMR controls the X and Y data memory maps and enables/disables the internal X and Y data ROMs. One additional memory available on the DSP56001 is the bootstrap memory that overlays the program memory in mode 1.

3.3.1 X Data Memory

The on-chip X data RAM is a 24-bit-wide, static, internal memory occupying the lowest 256 locations (0–255) in X memory space. The on-chip X data ROM occupies locations 256–511 in the X data memory space when enabled by setting DE to one in the OMR. The X data ROM is factory programmed with positive Mu-law and A-law expansion tables, which are useful in telecommunication applications. The on-chip peripheral registers occupy the top 64 locations of the X data memory (locations $FFC0–$FFFF). The 16-bit addresses are received from the XAB, and 24-bit data transfers to the data ALU occur on the XDB. The X memory may be expanded to 64K off-chip.

3.3.2 Y Data Memory

The on-chip Y data RAM is a 24-bit-wide, static, internal memory occupying the lowest 256 locations (0–255) in the Y memory space. The on-chip Y data ROM occupies locations 256–511 in Y data memory space when enabled by setting DE to one in the OMR. The Y data ROM is factory programmed with a full, four-quadrant, sine-wave table (see DSP56001 Advance Information Data Sheet(ADI1290)), which is useful for fast Fourier transforms, discrete Fourier transforms, and waveform generation. The off-chip peripheral registers should be mapped into the top 64 locations ($FFC0–$FFFF) to take advantage of the MOVEP instruction. The 16-bit addresses are received from the YAB, and 24-bit data transfers to the data ALU occur on the YDB. Y memory may be expanded to 64K off-chip.

3.3.3 Program Memory

On-chip program memory consists of a 512-location by 24-bit, high-speed, static RAM that is enabled/disabled by the MA and MB bits in the OMR. When the on-chip program memory is disabled, either off-chip memory or a special bootstrap ROM is selected for program memory.

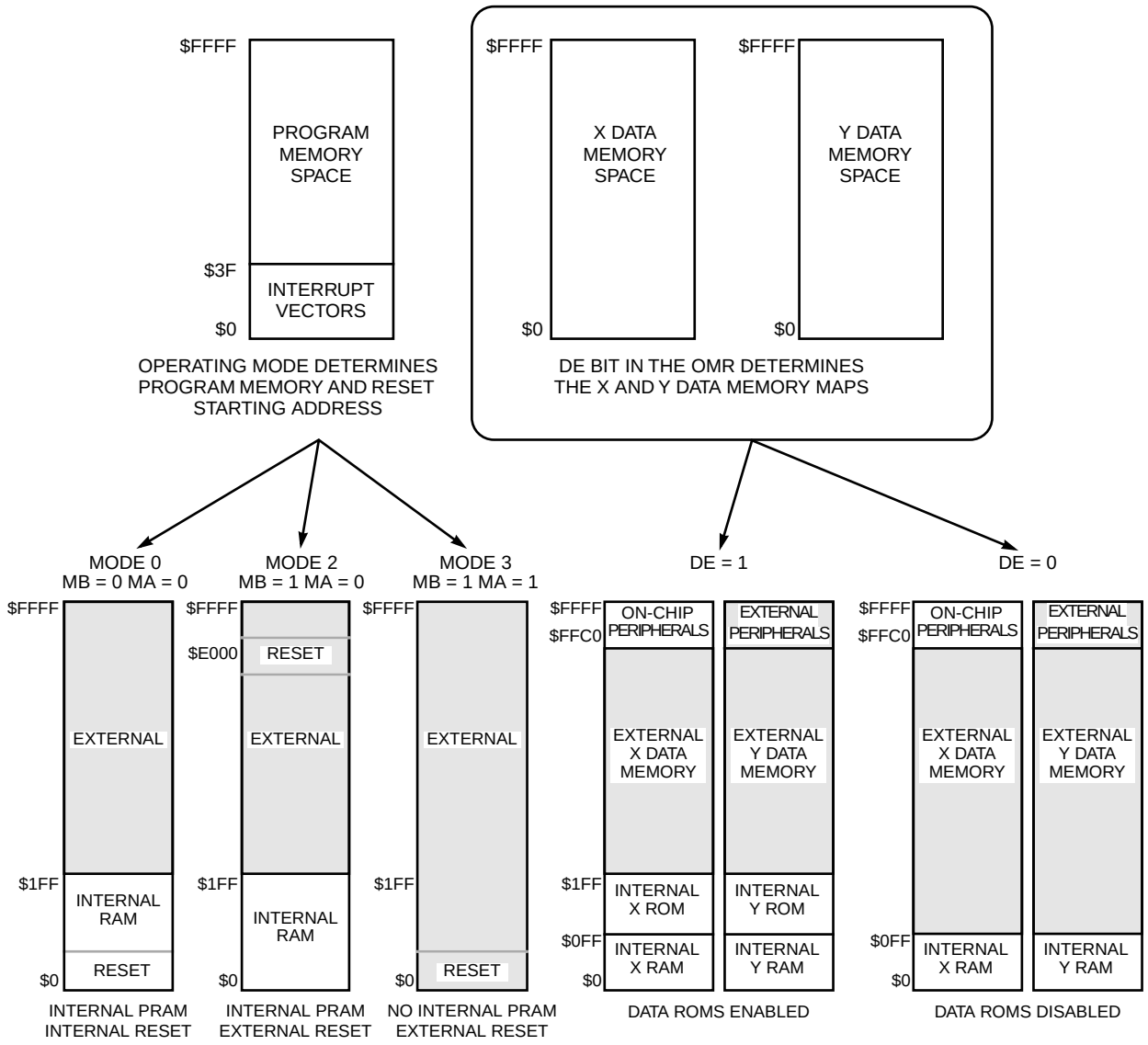


Figure 3-5 DSP56001 Memory Map

Addresses are received from the program control logic (usually the program counter) over the PAB. Program memory may be written using MOVEM instructions. The interrupt vectors for the on-chip resources are located in the bottom 64 locations ($0000–$003F) of program memory. Program memory may be expanded to 64K off-chip.

Program RAM provides a method of developing code efficiently, and programs can be changed dynamically, allowing efficient overlaying of DSP software algorithms. In this way, the on-chip program RAM operates as a fixed cache, thereby minimizing contention with accesses to external data memory spaces.

The bootstrap mode overlays the program memory in mode 1 and provides a convenient,

low-cost method of loading the DSP56001 program RAM with a program after power-on reset. The bootstrap mode also allows loading the program RAM from a single, inexpensive EPROM through port A or via the host interface using a host processor.

3.3.4 Bootstrap ROM (DSP56001 Only)

Factory programmed to perform the bootstrap operation from the memory expansion port (port A) or from the host interface, the 32-word on-chip ROM is invoked while the processor is in operating mode 1. Users have no access to the bootstrap ROM other than through the bootstrap process.

3.3.5 Chip Operating Modes

The DSP operating modes determine the memory maps for program and data memories and the startup procedure when the DSP leaves the reset state. The MODA and MODB pins are sampled as the DSP leaves the reset state, and the initial operating mode of the DSP is set accordingly. When the reset state is exited, the MODA and MODB pins become general-purpose interrupt pins, IRQA and IRQB. One of four initial operating modes is selected: single chip, special bootstrap, normal expanded, or development. Chip operating modes can be changed by writing the operating mode bits (MB, MA) in the OMR. Changing operating modes does not reset the DSP. It is desirable to disable interrupts immediately before changing the OMR to prevent an interrupt from going to the wrong memory location. For example, if the user changed to the bootstrap mode and an interrupt occurred, he would execute the bootstrap code out of order. Also, one NOP instruction must be included after changing the OMR to allow for remapping to occur.

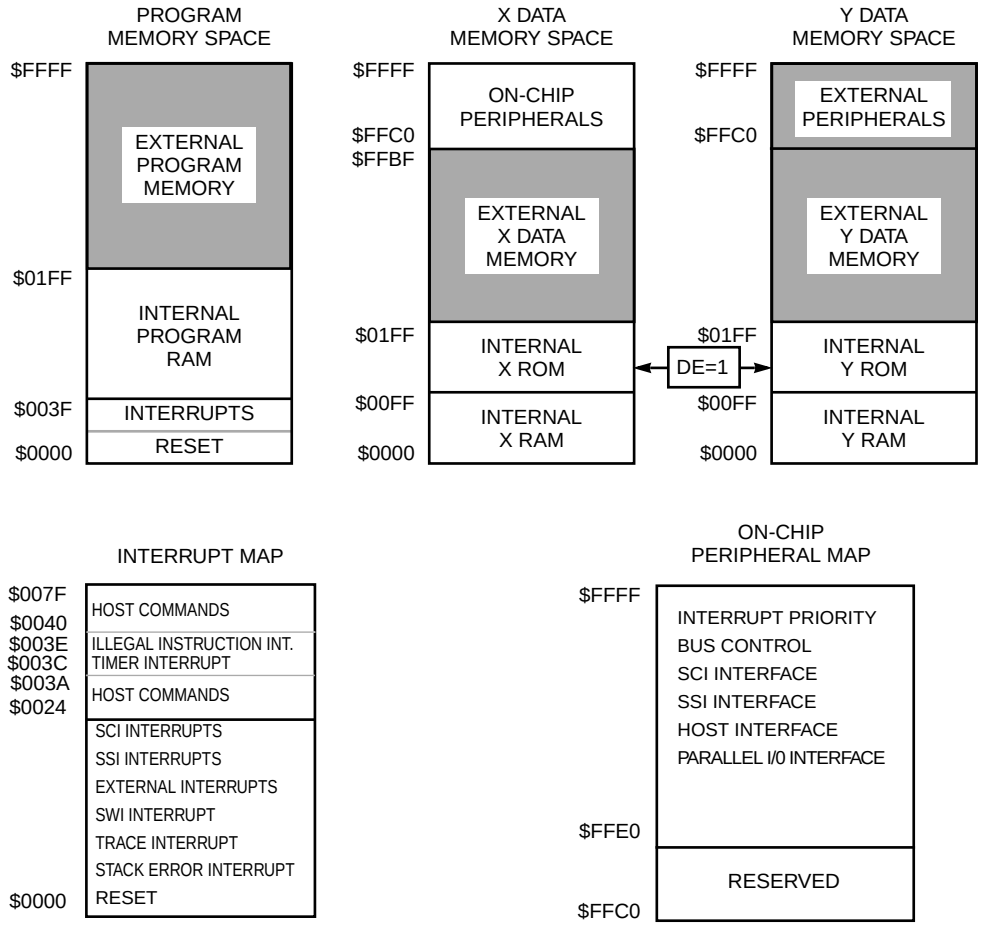
Some pins on the DSP are mode independent; whereas, others depend on the particular operating mode. Specifically, external address bus, data bus, and bus control pins are affected by the particular operating mode. Table 3-2 depicts the mode assignments.

Table 3-2 Initial DSP56001 Operating Mode Summary

Operating Mode	MODB	MODA	Description
0	0	0	Single-Chip Mode
1	0	1	Special Bootstrap Mode
2	1	0	Normal Expanded Mode
3	1	1	Development Mode

3.3.5.1 Single-Chip Mode (Mode 0)

In the single-chip mode, all internal program and data RAM memories are enabled. A hardware reset causes the DSP to jump to internal program memory location $0000 and



NOTE: Addresses $FFC0–$FFFF in X data memory are NOT available externally.

Figure 3-6 Memory Map for DSP56001 Mode 0: Single-Chip

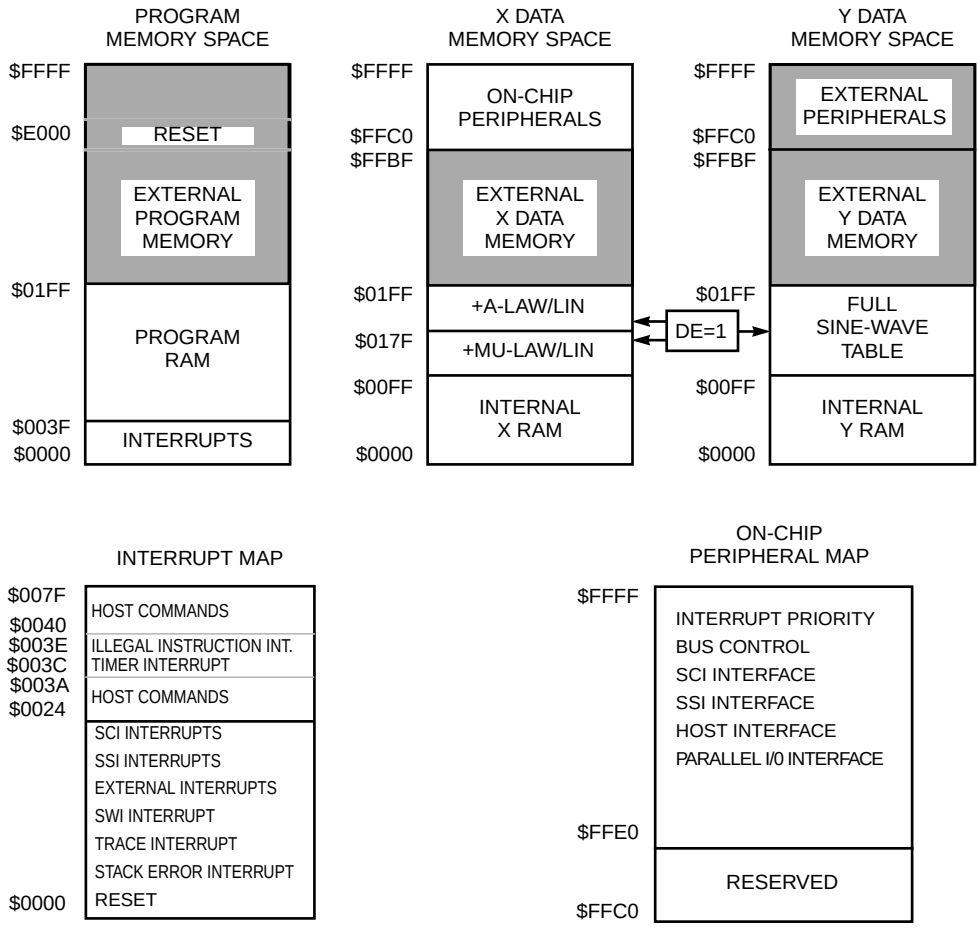
resume execution. The memory map for this mode is shown in Figure 3-6. The memory maps for mode 0 and mode 2 (see Figure 3-7) are identical. The difference between the two modes is that reset vectors to program memory location $0000 in mode 0 and vectors to location $E000 in mode 2.

3.3.5.2 Special Bootstrap Mode (Mode 1)

The bootstrap mode is a special mode that loads internal program RAM either from a byte-wide external memory such as EPROM or from the host interface. After loading the internal memory, the DSP switches to the single-chip mode and begins program execution at on-chip program memory location $0000.

One method of selecting mode 1 is to assert the reset pin on the DSP56001. When the DSP leaves the reset state (RESET goes high), the MODB and MODA pins are sampled (they should be set to zero and one, respectively), and the initial operating mode of the DSP is

set accordingly. The following actions occur once the processor comes out of the reset state.



NOTE: Addresses $FFC0–$FFFF in X data memory are NOT available externally.

Figure 3-7 Memory Map for DSP56001 Mode 2: Normal Expanded Mode

1. The control logic maps the bootstrap ROM into the internal DSP program memory space starting at location $0000.
2. The control logic causes program reads to come from the bootstrap ROM (only address bits 4–0 are significant) and all writes go to the program RAM (all address bits are significant). This condition allows the bootstrap program to load the user program from $0000–$01FF.
3. Program execution begins at location $0000 in the bootstrap ROM. The bootstrap ROM program can load program RAM through either the memory expansion port or through the host interface. The choice is made by looking at bit 23 of P:$C000. The processor loads from the host interface if bit 23 is a zero; if bit 23 is a one, it loads from a byte-wide memory starting at P:$C000.
4. The bootstrap ROM program executes the following sequence to end the bootstrap operation and begin executing the user program. First, operating mode 2 is entered by writing to the OMR. This action will be timed to remove the bootstrap ROM from the program memory map and re-enable read/write access to the program RAM.

Second, the change to mode 2 is exactly timed to allow the bootstrap program to execute a single-cycle instruction (clear status register), then a JMP #<00, and begin execution of the user program at location $0000.

The bootstrap mode may also be selected by writing zero to MB and one to MA in the OMR. This selection initiates a timed operation to map the bootstrap ROM into the program address space after a delay to allow execution of a single-cycle instruction and then a JMP #<00 to begin the bootstrap process previously described. This technique allows the DSP56001 user to reboot the system (with a different program, if desired). The code to enter the bootstrap mode is as follows:

```
MOVEP     #0,X:$FFFF     ;Disable interrupts.
MOVEC     #1,OMR         ;The bootstrap ROM is mapped ;into
                         the lowest 32 locations
                         ;in program memory.
NOP                      ;Allow one cycle delay for the
                         ;remapping.
JMP       <$0            ;Begin bootstrap.
```

The interrupts are disabled before executing the bootstrap code; otherwise, an interrupt could cause the DSP to execute the bootstrap code out of sequence because the bootstrap program overlays the interrupt vectors.

The bootstrap ROM contains the bootstrap firmware program that performs initial loading of the DSP56001 program RAM.

Written in DSP56001 assembly language, the program contains two separate methods of initializing the program RAM: loading from a byte-wide memory starting at location P:$C000 or loading through the host interface. The particular method used is selected by the level of program memory location P:$C000 bit 23.

If location P:$C000 bit 23 is read as a one, the external bus version of the bootstrap program will be selected. Typically, a byte-wide EPROM will be connected to the DSP56001 address and data bus. The data contents of the EPROM must be organized as shown in

Table 3-3:

Table 3-3 Organization of EPROM Data Contents

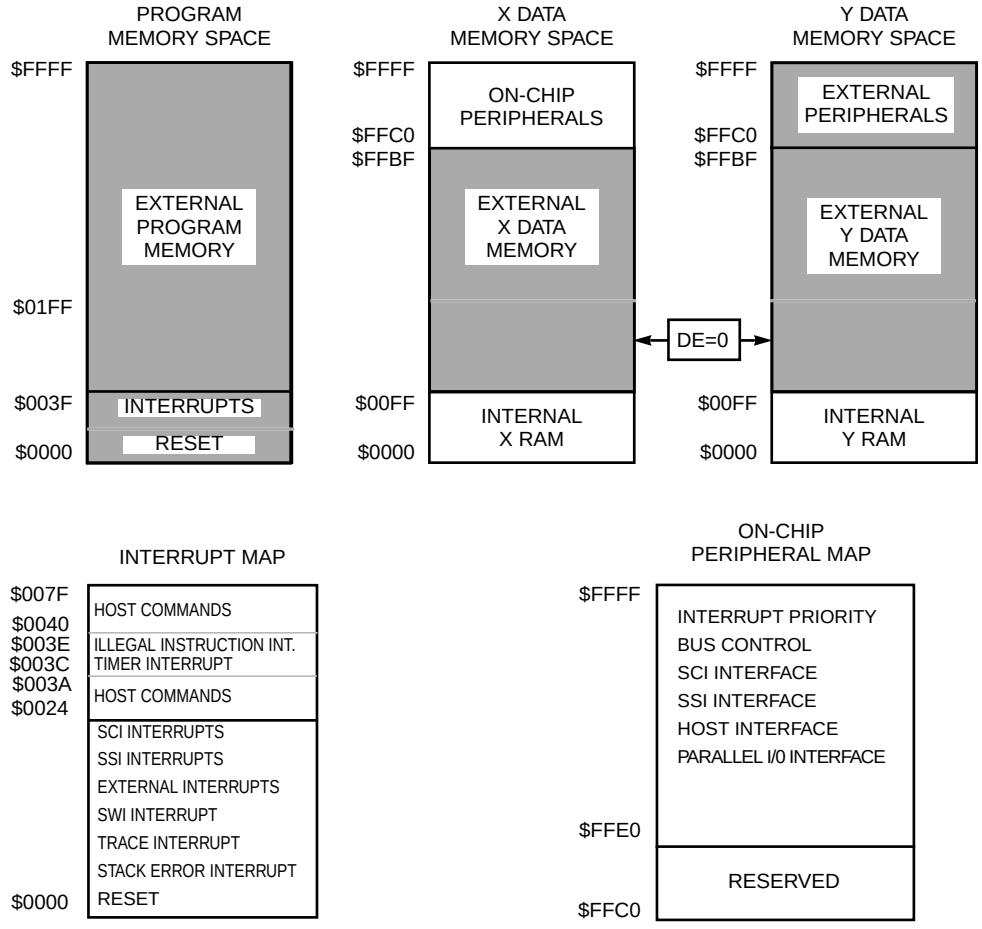
Address of External Byte-Wide Memory:	Contents Loaded to Internal Program RAM at:	
P:$C000	P:$0000	low byte
P:$C001	P:$0000	mid byte
P:$C002	P:$0000	high byte
•	•	
•	•	
•	•	
P:$C5FD	P:$01FF	low byte
P:$C5FE	P:$01FF	mid byte
P:$C5FF	P:$01FF	high byte

If location P:$C000 bit 23 is read as a zero, the host interface version of the bootstrap program will be selected. Typically, a host microprocessor will be connected to the DSP56001 host interface. The host microprocessor must write the host interface byte-wide registers TXH, TXM, and then TXL with the desired contents of program RAM from location P:$0000 up to P:$01FF. If less than 512 words are to be loaded, the host programmer can exit the bootstrap program and force the DSP56001 to begin executing at location P:$0000 by setting HF0 to one in the host interface control register. In most systems, the DSP56001 response is so fast that handshaking between the DSP56001 and the host is not necessary.

3.3.5.3 Normal Expanded Mode (Mode 2)

Mode 2 is almost identical to mode 0 (see 3.3.5.1 Single-Chip Mode (Mode 0) for details).

3.3.5.4 Development Mode (Mode 3). The development mode is similar to the normal expanded mode except that internal program memory is disabled. All references to program memory space are directed to external program memory, which is accessed on the external data bus. The reset vector points to location $0000. The memory map for this mode is shown in Figure 3-8. The memory map in Figure 3-8 is shown with DE arbitrarily set to zero.



NOTE: Addresses $FFC0–$FFFF in X data memory are NOT available externally.

Figure 3-8 Memory Map for DSP56001 Mode 3: Development Mode

SECTION 4
DATA ARITHMETIC LOGIC UNIT

This section describes the operation of the data arithmetic logic unit (ALU) registers and hardware. The data representation, rounding, and saturation arithmetic used within the data ALU are also presented. This section concludes with a discussion of the programming model.

4.1 OVERVIEW AND DATA ALU ARCHITECTURE

The DSP56000/DSP56001 central processor is composed of three execution units that operate in parallel. They are the data ALU, address generation unit (AGU), and the program control unit (see Figure 4-1). These three units are register oriented rather than bus

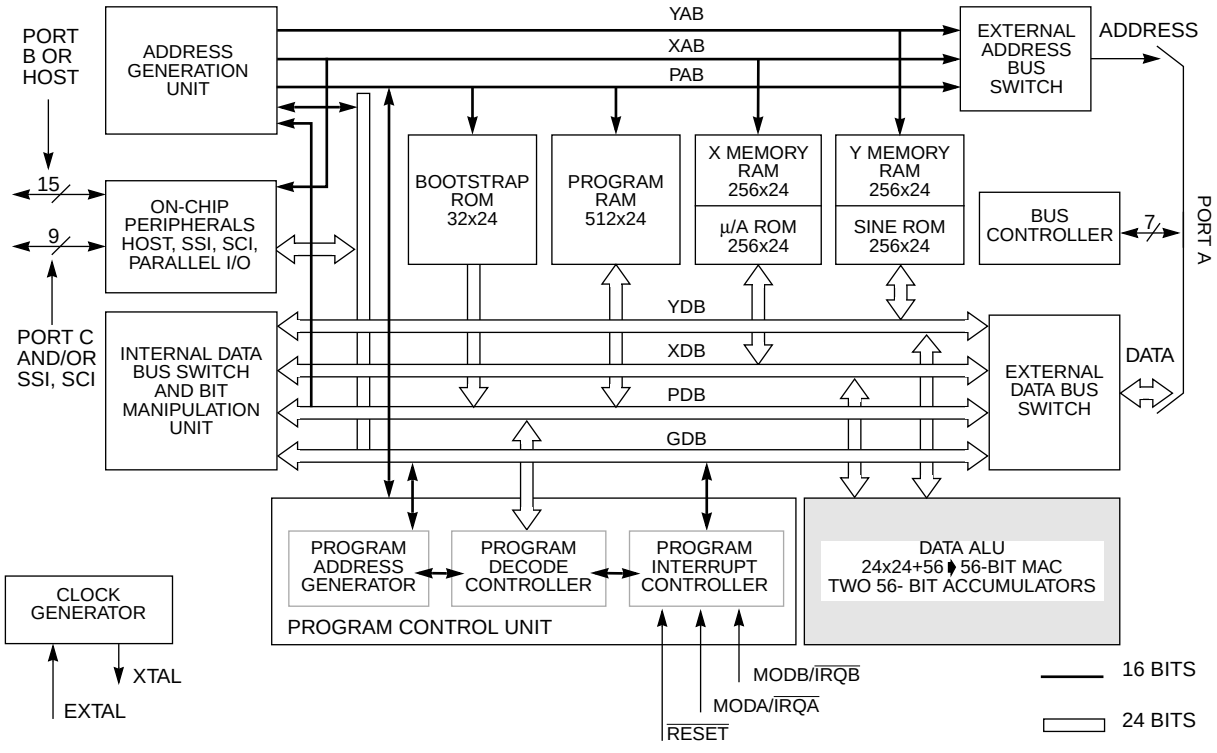


Figure 4-1 DSP56001 Block Diagram

oriented and are designed to interface over the system buses with memory and memory-mapped I/O devices. The DSP56000/DSP56001 instruction set has been designed to allow flexible control of these parallel processing resources. Many instructions allow the programmer to keep each unit busy, thus enhancing performance. It was possible to make the programming model like that of conventional microprocessor units (MPUs), eliminating the need to refer to the detailed chip architecture when programming the DSP56000/DSP56001 because the parallel execution units appear to execute their operations in a nonpipelined manner.

The data ALU (see Figure 4-2r) is the first of these execution units to be presented. The data ALU, which has been designed to be fast and yet provide the capability to process signals having a wide dynamic range, performs all the arithmetic and logical operations on data operands in the DSP56000/DSP56001.

The data ALU registers may be read or written over the XDB and the YDB as 24- or 48-bit operands. The source operands for the data ALU, which may be 24, 48, or 56 bits, always originate from data ALU registers. The results of all data ALU operations are stored in an accumulator.

The 24-bit data words provide 144 dB of dynamic range. This range is sufficient for most real-world applications since the majority of data converters are 16 bits or less, and certainly not greater than 24 bits. The 56-bit accumulator internal to the data ALU provides 336 dB of internal dynamic range so that no loss of precision will occur due to intermediate processing. Circuitry has been provided to facilitate handling data overflows and roundoff errors.

Any of the following operations can be performed by the data ALU in a single instruction cycle: multiplication, multiply-accumulate with positive or negative accumulation, convergent rounding, multiply-accumulate with positive or negative accumulation and convergent rounding, addition, subtraction, a divide iteration, a normalization iteration, shifting, and logical operations.

The components of the data ALU are as follows:

- Four 24-bit input registers
- A parallel, single-cycle, nonpipelined multiply-accumulator/logic unit (MAC)
- Two 48-bit accumulator registers
- Two 8-bit accumulator extension registers
- An accumulator shifter
- Two data bus shifter/limiter circuits

Each of these components is described in the following paragraphs as well as a description of data representation, rounding, and saturation arithmetic.

4.1.1 Data ALU Input Registers (X1, X0, Y1, Y0)

X1, X0, Y1, and Y0 are four 24-bit, general-purpose data registers. They can be treated as four independent, 24-bit registers or as two 48-bit registers called X and Y, developed by the concatenation of X1:X0 and Y1:Y0, respectively. X1 is the most significant word in X and Y1 is the most significant word in Y. The registers serve as input buffer registers between the XDB or YDB and the MAC unit. They are used as data ALU source operands, allowing new operands to be loaded for the next instruction while the register con-

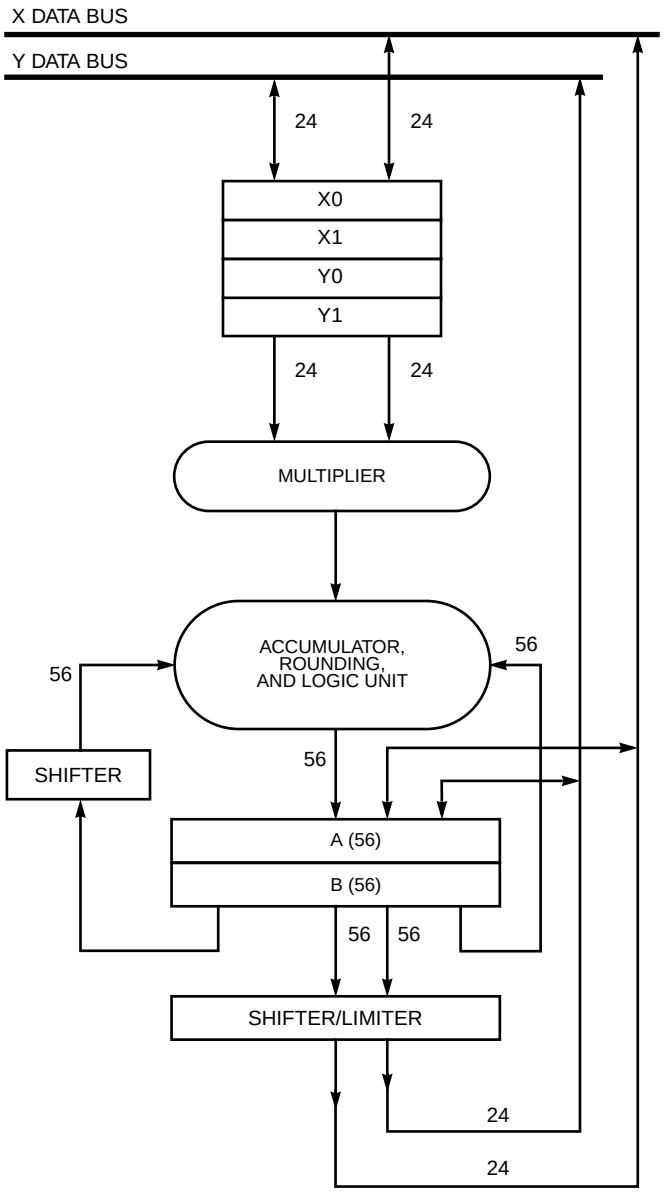


Figure 4-2 Data ALU

tents are used by the current instruction. The registers may also be read back out to the appropriate data bus to implement memory-delay operations and save/restore operations for interrupt service routines.

4.1.2 MAC and Logic Unit

The MAC and logic unit comprise the main arithmetic processing unit of the DSP and perform all of the calculations on data operands. In the case of arithmetic instructions, the unit accepts up to three input operands and outputs one 56-bit result of the following form, extension:most significant product:least significant product (EXT:MSP:LSP). The operation of the MAC unit occurs independently and in parallel with XDB and YDB activity, and its registers facilitate buffering for both data ALU inputs and outputs. Latches are provided on the MAC unit input to permit writing an input register, which is the source for a data ALU operation in the same instruction.

The arithmetic unit contains a multiplier and two accumulators. The input to the multiplier can only come from the X or Y registers (X1, X0, Y1, Y0). The multiplier executes 24-bit x 24-bit, parallel, twos-complement fractional multiplies. The 48-bit product is right justified and added to the 56-bit contents of either the A or B accumulator. The 56-bit sum is stored back in the same accumulator (see Figure 4-3r). An 8-bit adder, which is used as an extension accumulator for the MAC array, accommodates overflow of up to 256 and allows the two 56-bit accumulators to be added and subtracted from each other. The extension adder output is the EXT portion of the MAC unit output. This multiply/accumulate operation is not pipelined but rather is a single-cycle operation. If a multiply without accumulation (MPY) is specified in the instruction, the MAC clears the accumulator and then adds the contents to the product.

In summary, the results of all arithmetic instructions are valid (sign-extended and zero-filled) 56-bit operands in the form of EXT:MSP:LSP or A2:A1:A0 or B2:B1:B0. When a 56-bit result is to be stored as a 24-bit operand, the LSP can be simply truncated, or it can be rounded (using convergent rounding) into the MSP.

Convergent rounding (round-to-nearest) is performed when adding the multiplier's product to the contents of the accumulator if specified in the DSP instruction (e.g., the signed multiply-accumulate and round (MACR) instruction). The bit in the accumulator that is rounded is specified by the scaling mode bits in the status register.

The logic unit performs the logical operations, AND, OR, EOR, and NOT, on data ALU registers. This unit is 24 bits wide and operates on data in the MSP portion of the accumulator. The LSP and EXT portions of the accumulator are not affected.

4.1.3 Data ALU Accumulator Registers (A2, A1, A0, B2, B1, B0)

The six data ALU registers (A2, A1, A0, B2, B1, and B0) form two general-purpose, 56-

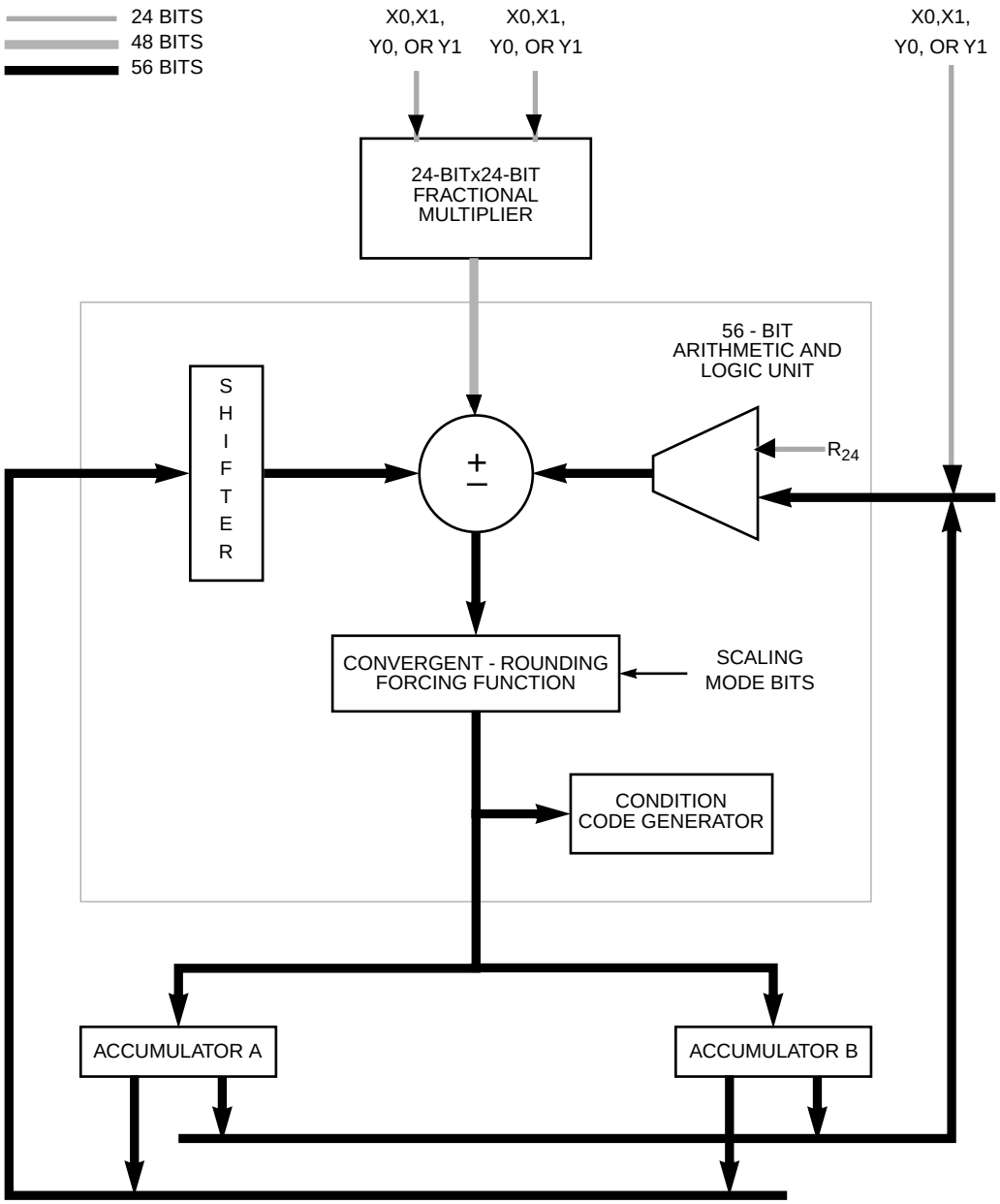


Figure 4-3 MAC Unit

bit accumulators, A and B. Each of these two registers consists of three concatenated registers (A2:A1:A0 and B2:B1:B0, respectively). The 24-bit MSP is stored in A1 or B1; the 24-bit LSP is stored in A0 or B0. The 8-bit EXT is stored in A2 or B2.

The 8-bit extension registers offer protection against overflow. On the DSP56000/ DSP56001, the extreme values that a word operand can assume are - 1 and +

0.9999998. If the sum of two numbers is less than - 1 or greater than + 0.9999998, the result (which cannot be represented in a word operand =m i.e., 24 bits) has underflowed or overflowed. The 8-bit extension registers can accurately represent the result of 255 overflows or 255 underflows. Whenever the accumulator extension registers are in use, the V bit in the status register is set.

Automatic sign extension is provided when writing to the 56-bit accumulators A or B with a 48- or 24-bit operand. When a 24-bit operand is written, the low-order portion will be automatically zero filled to form a valid 56-bit operand. The registers may also be written without sign extension or zero fill by specifying the individual register name. When accumulator registers A or B are read, they may be optionally scaled one bit left or one bit right for block floating-point arithmetic.

Reading the A or B accumulators over the XDB and YDB is protected against overflow by substituting a limiting constant for the data that is being transferred. The content of A or B is not affected should limiting occur; only the value transferred over the XDB or YDB is limited. This overflow protection is performed after the contents of the accumulator have been shifted according to the scaling mode. Shifting and limiting will be performed only when the entire 56-bit A or B register is specified as the source for a parallel data move over the XDB or YDB. When A0, A1, A2, B0, B1, or B2 are specified as the source for a parallel data move, shifting and limiting are not performed. The accumulator registers serve as buffer registers between the MAC unit and the XDB and/or YDB. These registers are used as both data ALU source and destination operands.

Automatic sign extension of the 56-bit accumulators is provided when the A or B register is written with a smaller operand. Sign extension can occur when writing A or B from the XDB and/or YDB or with the results of certain data ALU operations (such as the transfer conditionally (Tcc) or transfer data ALU register (TFR) instructions). If a word operand is to be written to an accumulator register (A or B), the MSP (A1 or B1) portion of the accumulator is written with the word operand, the LSP (A0 or B0) portion is zero filled, and the EXT (A2 or B2) portion is sign extended from MSP. Long-word operands are written into the low-order portion, MSP:LSP, of the accumulator register, and the EXT portion is sign extended from MSP. No sign extension is performed if an individual 24-bit register is written (A1, A0, B1, or B0). Test logic is included in each accumulator register to support operation of the data shifter/limiter circuits. This test logic is used to detect overflows out of the data shifter so that the limiter can substitute one of several constants to minimize errors due to the overflow. This process is commonly referred to as saturation arithmetic.

4.1.4 Accumulator Shifter

The accumulator shifter (see Figure 4-3) is an asynchronous parallel shifter with a 56-bit input and a 56-bit output that is implemented immediately before the MAC accumulator

input. The source accumulator shifting operations are as follows:

- No Shift (Unmodified)
- 1-Bit Left Shift (Arithmetic or Logical) ASL, LSL, ROL
- 1-Bit Right Shift (Arithmetic or Logical) ASR, LSR, ROR
- Force to zero

4.1.5 Data Shifter/Limiter

The data shifter/limiter circuits (see Figure 4-3) provide special postprocessing on data read from the ALU accumulator registers A and B out to the XDB or YDB. There are two independent shifter/limiter circuits (one for XDB and one for the YDB); each consists of a shifter followed by a limiting circuit.

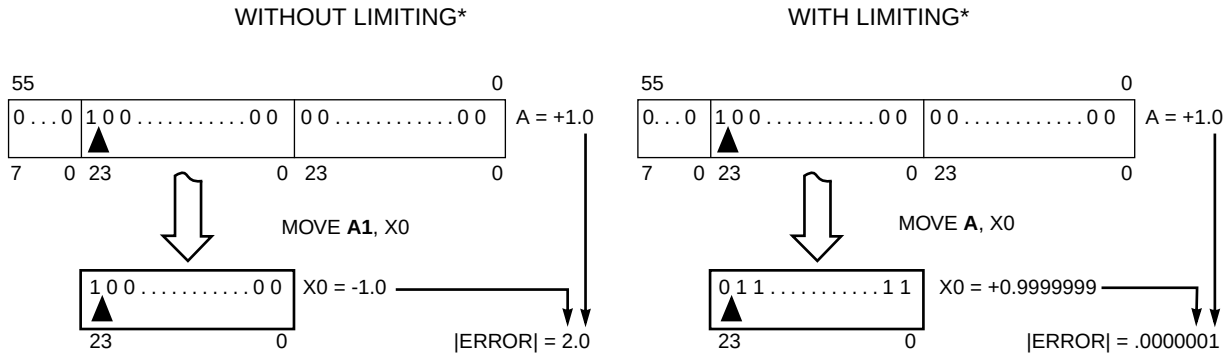
4.1.5.1 Limiting (Saturation Arithmetic)

In the DSP56000/DSP56001, the data ALU accumulators A and B have eight extension bits. Limiting will occur when the extension bits are in use and either A or B is the source being read over XDB or YDB. The limiters in the DSP56000/DSP56001 place a shifted and limited value on XDB or YDB without changing the contents of the A or B registers. Having two limiters allows two-word operands to be limited independently in the same instruction cycle. The two data limiters can also be combined to form one 48-bit data limiter for long-word operands.

If the contents of the selected source accumulator can be represented without overflow in the destination operand size (i.e., accumulator extension register not in use), the data limiter is disabled, and the operand is not modified. If 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 having maximum magnitude (saturated) and having the same sign as the source accumulator contents: $7FFFFF for 24-bit or $7FFFFF FFFFFF for 48-bit positive numbers, $800000 for 24-bit or $800000 000000 for 48-bit negative numbers. This process is called saturation arithmetic. The value in the accumulator register is not shifted and can be reused within the data ALU. When limiting does occur, a flag is set and latched in the status register.

For example, if the source operand were 01.100 (+ 1.5 decimal) and the destination register were only four bits, the destination register would contain 1.100 (- 1.5 decimal) after the transfer, assuming signed fractional arithmetic. This is clearly in error as overflow has occurred. To minimize the error due to overflow, it is preferable to write the maximum (‘‘limited’’) value the destination can assume. In the example, the limited value would be 0.111 (+ 0.875 decimal), which is clearly closer to + 1.5 than - 1.5 and therefore introduces less error.

Figure 4-4 shows the effects of saturation arithmetic on a move from register A1 to register X0. The instruction ‘‘MOVE A1,X0’’ causes a move without limiting, and the instruc-



* Limiting automatically occurs when the 56 - bit operands A or B (not A2, A1, A0, B2, B1, or B0) are read. The contents of A or B are **NOT** changed.

Figure 4-4 Saturation Arithmetic

tion ‘‘MOVE A,X)’’ causes a move of the same 24 bits with limiting. The error without limiting is 2.0; whereas, it is 0.0000001 with limiting. Table 4-1 shows a more complete set of limiting situations.

4.1.5.2 Scaling

The data shifters are capable of shifting data one bit to the left or one bit to the right as

Table 4-1 Limited Data Values

Destination Memory Reference	Source Operand	Accumulator Sign	Limited Value (Hexadecimal)		Type of Access
			XDB	YDB	
X	X:A X:B	+ -	7FFFFF 800000	— —	One 24 bit
Y	Y:A Y:B	+ -	— —	7FFFFF 800000	One 24 bit
X and Y	X:A Y:A X:A Y:B X:B Y:A X:B Y:B L:AB L:BA	+ - + - + -	7FFFFF 800000 7FFFFF 800000 7FFFFF 800000	7FFFFF 800000 7FFFFF 800000 7FFFFF 800000	Two 24 bit
L (X:Y)	L:A L:B	+ -	7FFFFF 800000	FFFFFF 000000	One 48 bit

well as passing the data unshifted. Each data shifter has a 24-bit output with overflow indication and is controlled by the scaling mode bits in the status register. These shifters permit dynamic scaling of fixed-point data without modifying the program code. For example, this permits block floating-point algorithms such as fast Fourier transforms to be implemented in a regular fashion.

4.2 DATA REPRESENTATION AND ROUNDING

The DSP56000/DSP56001 uses a fractional data representation for all data ALU operations. Figure 4-5 shows the bit weighting of words, long words, and accumulator oper-

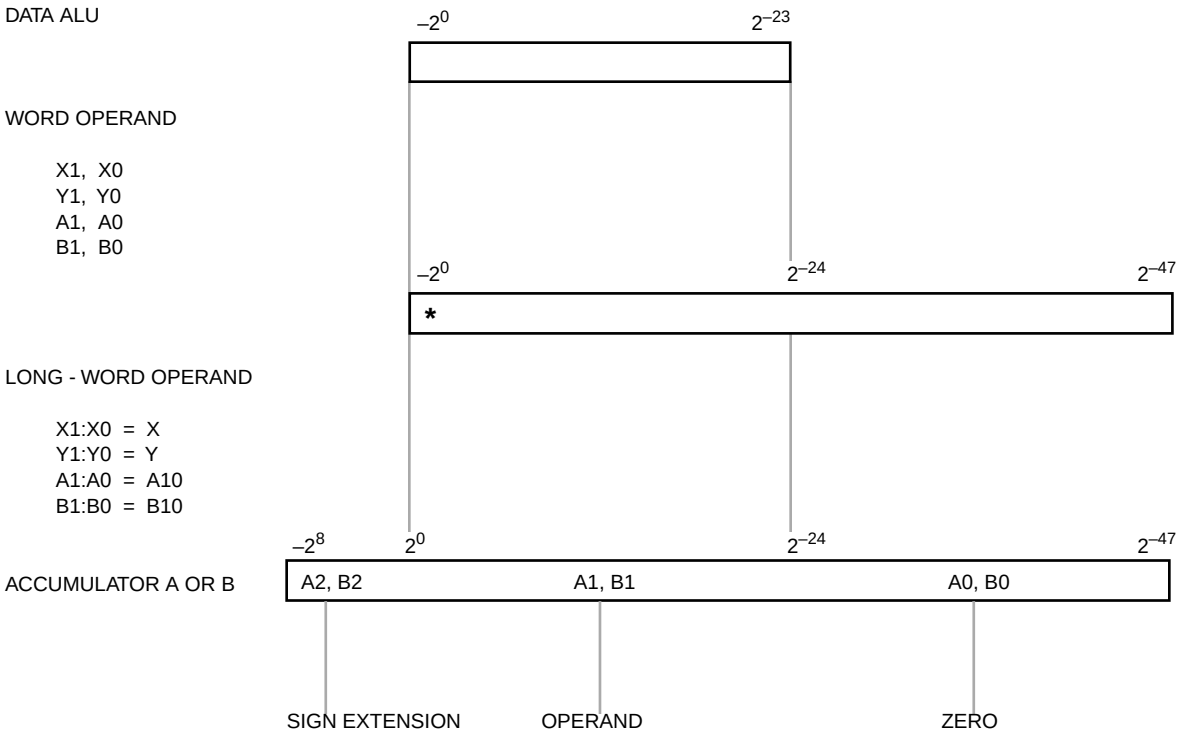


Figure 4-5 Bit Weighting and Alignment of Operands

ands for this representation. The decimal points are all aligned and are left justified.

Data must be converted to a fractional number by scaling before being used by the DSP56000/DSP56001, or the user will have to be very careful in how the DSP manipulates the data. Moving $3F to a 24-bit data ALU register does not result in the contents being $00003F as might be expected. Assuming numbers are fractional, the DSP left justifies rather than right justifies. As a result, storing $3F in a 24-bit register results in the contents being $3F0000. The simplest example of scaling is to convert all integer num-

bers to fractional numbers by shifting the decimal 24 places to the left (see Figure 4-6).

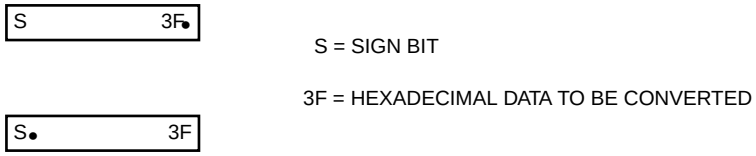


Figure 4-6 Integer-to-Fractional Data Conversion

Thus, the data has not changed; only the position of the decimal has moved.

For words and long words, the most negative number that can be represented is; -1 whose internal representation is $800000 and $800000000000, respectively. The most positive word is $7FFFFF or 1 - 2 - 23 and the most positive long word is $7FFFFFFFFFFF or 1 - 2 - 47. These limitations apply to all data stored in memory and to data stored in the data ALU input buffer registers. The extension registers associated with the accumulators allow word growth so that the most positive number that can be used is approximately 256 and the most negative number is approximately; -256. 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 most significant bit (MSB) of the extension register).

To maintain alignment of the binary point when a word operand is written to accumulator A or B, the operand is written to the most significant accumulator register (A1 or B1), and its MSB is automatically sign extended through the accumulator extension register. The least significant accumulator register is automatically cleared. When a long-word operand is written to an accumulator, the least significant word of the operand is written to the least significant accumulator register (see Figure 4-7).

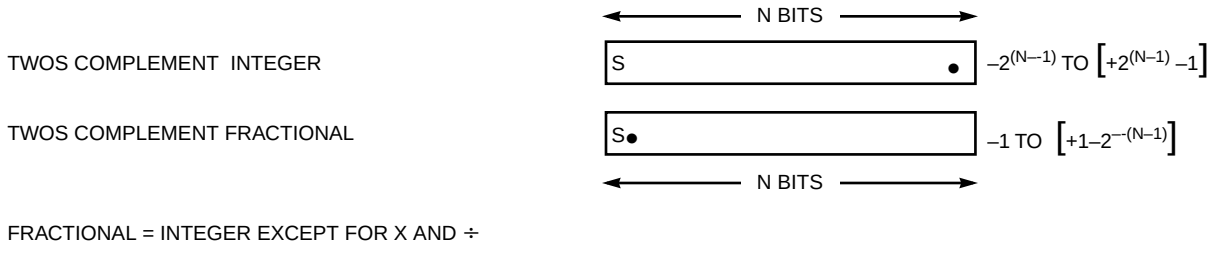


Figure 4-7 Integer/Fractional Number Comparison

A comparison between integer and fractional number representation is shown in Figure 4-7. The number representation for integers is between $\pm 2^{(N-1)}$; whereas, the fractional representation is limited to numbers between ±1. To convert from an integer to a fractional number, the integer must be multiplied by a scaling factor so the result will always be between ±1. The representation of integer and fractional numbers is the same if the numbers are added or subtracted but is different if the numbers are multiplied or divided. An example of two numbers multiplied together is given in Figure 4-8. The key difference

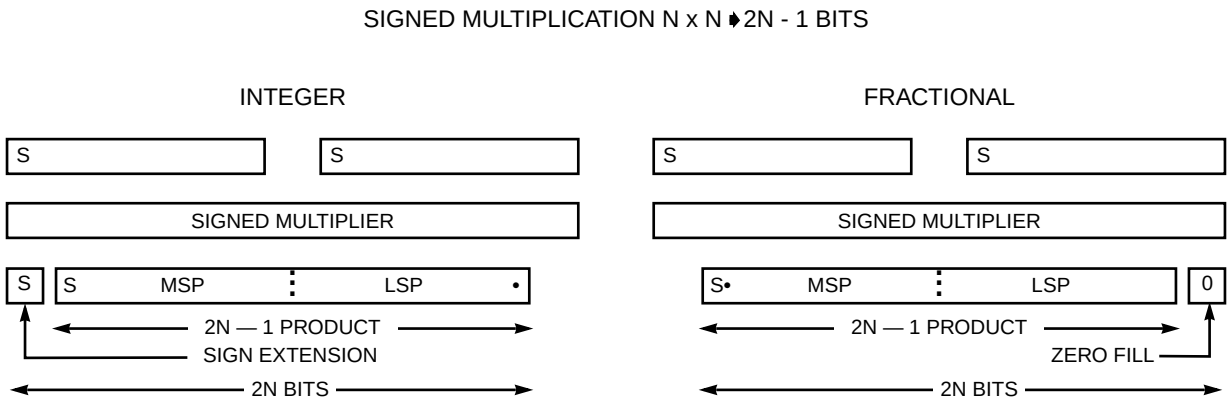


Figure 4-8 Integer/Fractional Multiplication Comparison

is that the extra bit in the integer multiplication is used as a duplicate sign bit and as the least significant bit (LSB) in the fractional multiplication. 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.

A significant bit is not lost through sign extension.

Conversion to floating-point representation is easier because the industry-standard floating-point formats use fractional mantissas.

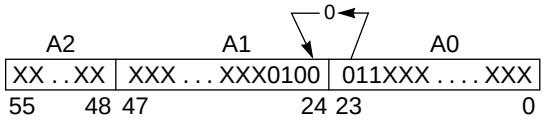
Coefficients for most digital filters are derived as fractions by the high-level language programs used in digital-filter design packages, which implies that the results can be used without the extensive data conversions that other formats require.

Should integer arithmetic be required in an application, shifting a one or zero, depending on the sign, into the MSB converts a fraction to an integer.

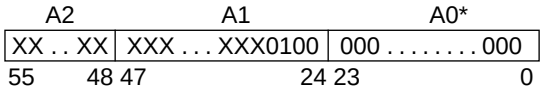
The data ALU MAC performs rounding of the accumulator register to single precision if

CASE I: IF A0 < $800000 (1/2), THEN ROUND DOWN (ADD NOTHING)

BEFORE ROUNDING

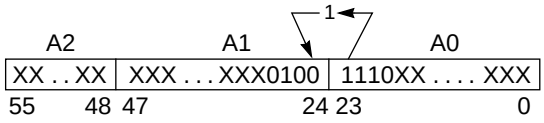


AFTER ROUNDING

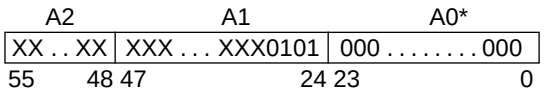


CASE II: IF A0 > $800000 (1/2), THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING

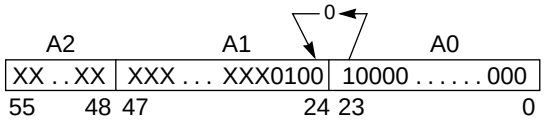


AFTER ROUNDING

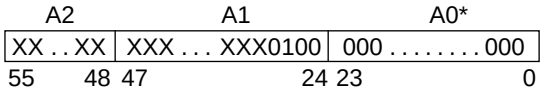


CASE III: IF A0 = $800000 (1/2), AND THE LSB OF A1 = 0,THEN ROUND DOWN (ADD NOTHING)

BEFORE ROUNDING

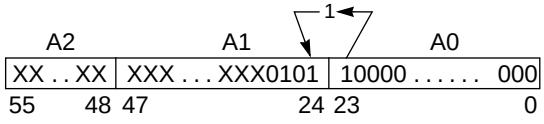


AFTER ROUNDING

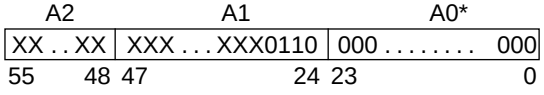


CASE IV: IF A0 = $800000 (1/2), AND THE LSB = 1, THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING



AFTER ROUNDING



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

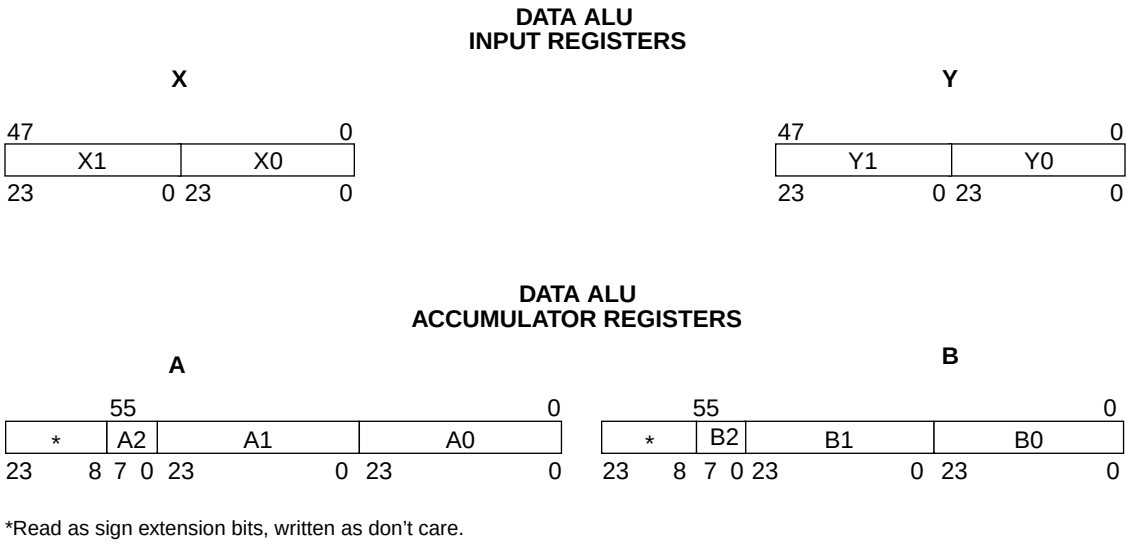
Figure 4-9 Convergent Rounding

requested in the instruction (the A1 or B1 register is rounded according to the contents of the A0 or B0 register). The rounding method used is called round-to-nearest (even) number, sometimes referred to as convergent rounding. The usual rounding method rounds up any value above one-half and rounds down any value below one-half. The question arises as to which way one-half should be rounded. If it is always rounded one way, the

results will eventually be a bias in that direction. Convergent rounding solves the problem by rounding down if the number is odd (LSB=0) and rounding up if the number is even (LSB=1). Figure 4-9 shows the four cases for rounding a number in the A1 (or B1) register. If scaling is set in the status register, the resultant number will be rounded as it is put on the data bus. However, the contents of the register are not scaled.

4.3 DATA ALU PROGRAMMING MODEL

The data ALU features 24-bit input/output data registers that can be concatenated to accommodate 48-bit data and two 56-bit accumulators, which are segmented into three 24-bit pieces that can be transferred over the buses. Figure 4-10 illustrates how the registers in the programming model are grouped.



*Read as sign extension bits, written as don't care.

Figure 4-10 DSP56000/DSP56001 Programming Model

4.4 DATA ALU SUMMARY

The data ALU is optimized for arithmetic operations involving multiply and accumulate operations with two separate data spaces. The data ALU, which executes all instructions in one machine cycle, is not pipelined. The two 24-bit numbers being multiplied can come from the X registers (X0 or X1) or Y registers (Y0 or Y1). After multiplication, they are added (or subtracted) with one of the 56-bit accumulators and can be convergently rounded to 24 bits. The convergent-rounding forcing function detects the $800000 condition in the LSP and makes the correction as necessary. The final result is then stored in one of the accumulators as a valid 56-bit number. The condition code bits are set based on the rounded output of the logic unit.

SECTION 5
ADDRESS GENERATION UNIT AND ADDRESSING MODES

This section contains three major subsections. The first subsection describes the hardware architecture of the address generation unit (AGU); the second subsection describes the programming model. The third subsection describes the addressing modes, illustrating how the Rn, Nn, and Mn registers work together to form a memory address.

5.1 AGU ARCHITECTURE

The AGU is one of the three execution units on the DSP56000/DSP56001 shown in Figure 5-1. The AGU performs the effective address calculations (using integer arithmetic)

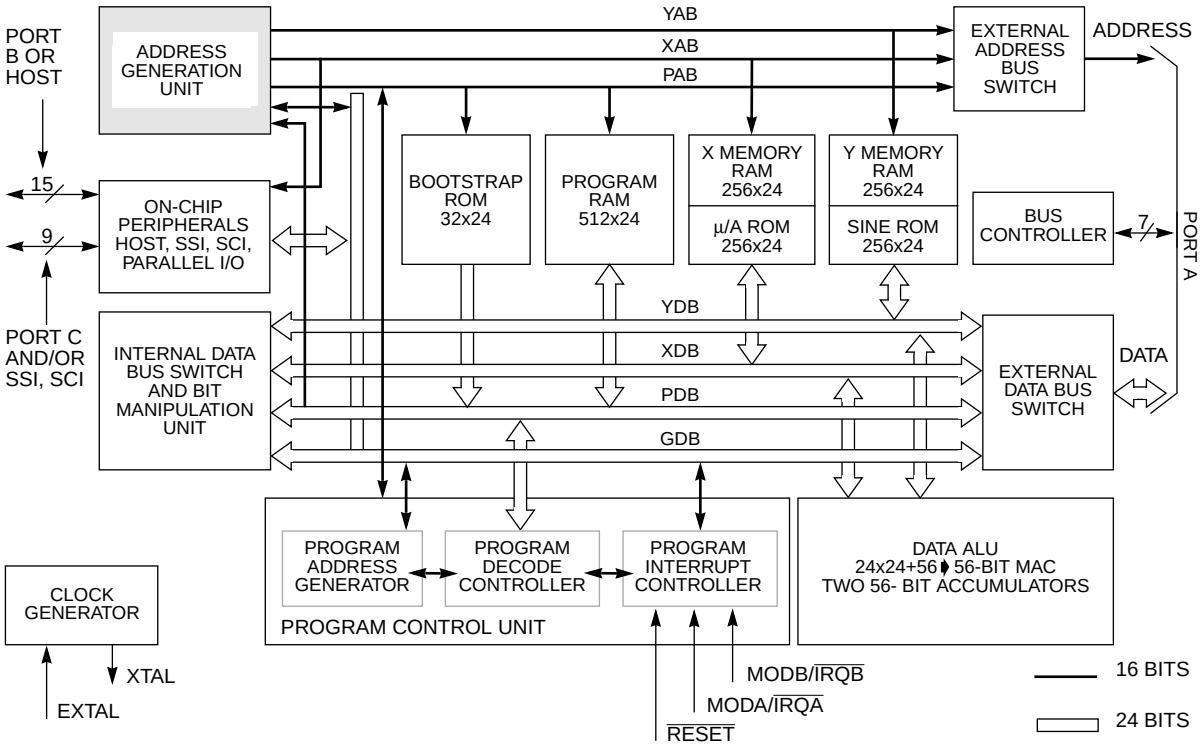


Figure 5-1 DSP56001 Block Diagram

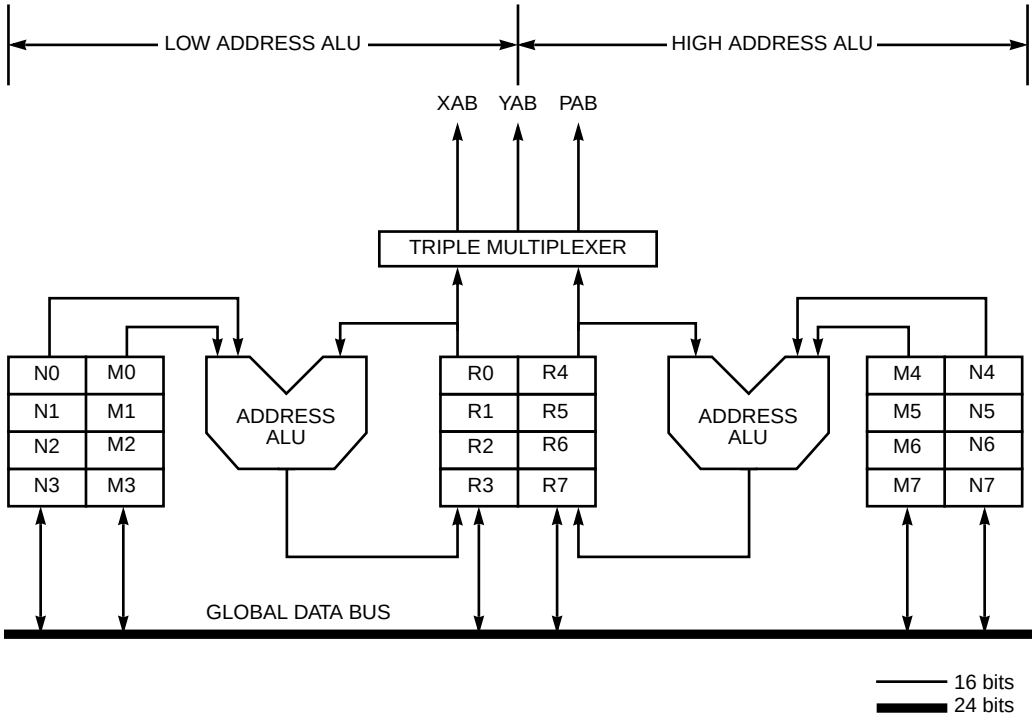


Figure 5-2 AGU Block Diagram

necessary to address data operands in memory and contains the registers used to generate the addresses. It implements three types of arithmetic, linear, modulo, and reverse-carry, and operates in parallel with other chip resources to minimize address-generation overhead. The AGU is divided into two identical halves, each of which has an address arithmetic logic unit (ALU) and four sets of three registers (see Figure 5-2).

These registers are the address registers (R0 - R3 and R4 - R7), offset registers (N0 - N3 and N4 - N7), and the modifier registers (M0 - M3 and M4 - M7). The eight Rn, Nn, and Mn registers are treated as register triplets — e.g., only N2 and M2 can be used to update R2. The eight triplets are R0:N0:M0, R1:N1:M1, R2:N2:M2, R3:N3:M3, R4:N4:M4, R5:N5:M5, R6:N6:M6, and R7:N7:M7.

The two arithmetic units can generate two 16-bit addresses every instruction cycle — one for any two of the XAB, YAB, or PAB. The AGU can directly address 65,536 locations on the XAB, 65,536 locations on the YAB, and 65,536 locations on the PAB. The two independent address ALUs work with the two data memories to feed the data ALU two operands in a single cycle. Each operand may be addressed by an Rn, Nn, and Mn triplet.

5.1.1 Address Register Files (Rn)

Each of the two address register files (see Figure 5-2) consists of four 16-bit registers. The two files contain address registers R0 - R3 and R4 - R7, which usually contain addresses

used as pointers to memory. Each register may be read or written by the global data bus (GDB). When read by the GDB, 16-bit registers are written into the two least significant bytes of the GBD, and the most significant byte is set to zero. When written from the GBD, only the two least significant bytes are written, and the most significant byte is truncated. Each address register can be used as input to its associated address ALU for a register update calculation. Each register can also be written by the output of its respective address ALU. One Rn register from the low address ALU and one Rn register from the high address ALU can be accessed in a single instruction.

5.1.2 Offset Register Files (Nn)

Each of two offset register files, shown in Figure 5-2, consists of four 16-bit registers. The two files contain offset registers N0 - N3 and N4 - N7, which contain either offset values used to update address pointers or data. Each offset register can be read or written by the GDB. When read by the GDB, the contents of a register are placed in the two least significant bytes, and the most significant byte on the GDB is zero extended. When a register is written, only the least significant 16 bits of the GDB are used; the upper portion is truncated.

5.1.3 Modifier Register Files (Mn)

Each of two modifier register files, shown in Figure 5-2, consists of four 16-bit registers. The two files contain modifier registers M0 - M3 and M4 - M7, which specify the type of arithmetic used during address register update calculations or contain data. Each modifier register can be read or written by the GDB. When read by the GDB, the contents of a register are placed in the two least significant bytes, and the most significant byte on the GDB is zero extended. When a register is written, only the least significant 16 bits of the GDB are used; the upper portion is truncated. Each modifier register is preset to $FFFF during a processor reset.

5.1.4 Address ALU

The two address ALUs are identical (see Figure 5-2) in that each contains a 16-bit full adder (called an offset adder), which can add 1) plus one, 2) minus one, 3) the contents of the respective offset register N, or 4) the twos complement of N to the contents of the selected address register. A second full adder (called a modulo adder) adds the summed result of the first full adder to a modulo value, M or minus M, where M is stored in the respective modifier register. A third full adder (called a reverse-carry adder) can add 1) plus one, 2) minus one, 3) the offset N (stored in the respective offset register), or 4) minus N to the selected address register with the carry propagating in the reverse direction — i.e., from the most significant bit (MSB) to the least significant bit (LSB). The offset adder and the reverse-carry adder are in parallel and share common inputs. The only difference between them is that the carry propagates in opposite directions. Test logic determines

which of the three summed results of the full adders is output.

Each address ALU can update one address register, Rn, from its respective address register file during one instruction cycle and is capable of performing linear, reverse-carry, and modulo arithmetic. The contents of the selected modifier register specify the type of arithmetic to be used in an address register update calculation. The modifier value is decoded in the address ALU.

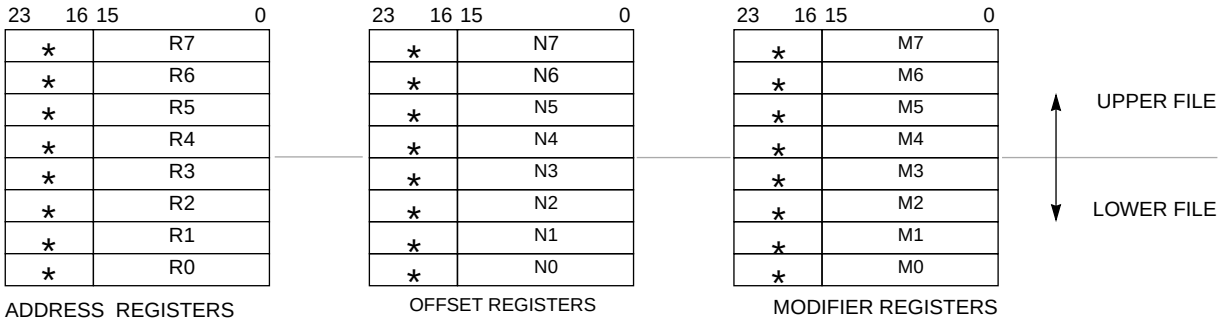
The output of the offset adder gives the result of linear arithmetic (e.g., Rn ± 1; Rn ± N) and is selected as the modulo arithmetic unit output for linear arithmetic addressing modifiers. The reverse-carry adder performs the required operation for reverse-carry arithmetic and its result is selected as the address ALU output for reverse-carry addressing modifiers. Reverse-carry arithmetic is useful for 2^k-point fast Fourier transform (FFT) addressing. For modulo arithmetic, the modulo arithmetic unit will perform the function (Rn ± N) modulo M, where N can be one, minus one, or the contents of the offset register Nn. If the modulo operation requires wraparound for modulo arithmetic, the summed output of the modulo adder gives the correct updated address register value; if wraparound is not necessary, the output of the offset adder gives the correct result.

5.1.5 Address Output Multiplexers

The address output multiplexers (see Figure 5-2) select the source for the XAB, YAB, and PAB. These multiplexers allow the XAB, YAB, or PAB outputs to originate from R0 - R3 or R4 - R7.

5.2 PROGRAMMING MODEL

The programmer's view of the AGU is eight sets of three registers (see Figure 5-3). These registers can be used as temporary data registers and indirect memory pointers. Automatic updating is available when using address register indirect addressing. The Rn registers can be programmed for linear addressing, modulo addressing, and bit-reverse addressing.



* Written as don't care; read as zero

Figure 5-3 AGU Programming Model

5.2.1 Address Register Files (R0 - R3 and R4 - R7)

The eight 16-bit address registers, R0 - R7, can contain addresses or general-purpose data. The 16-bit address in a selected address register is used in the calculation of the effective address of an operand. When supporting parallel X and Y data memory moves, the address registers must be thought of as two separate files, R0 - R3 and R4 - R7. The contents of an Rn may point directly to data or may be offset. In addition, Rn can be pre-updated or post-updated according to the addressing mode selected. If an Rn is updated, modifier registers, Mn, are always used to specify the type of update arithmetic. Offset registers, Nn, are used for the update-by-offset addressing modes. The address register modification is performed by one of the two modulo arithmetic units. Most addressing modes modify the selected address register in a read-modify-write fashion; the address register is read, its contents are modified by the associated modulo arithmetic unit, and the register is written with the appropriate output of the modulo arithmetic unit. The form of address register modification performed by the modulo arithmetic unit is controlled by the contents of the offset and modifier registers discussed in the following paragraphs.

5.2.2 Offset Register Files (N0 - N3 and N4 - N7)

The eight 16-bit offset registers, N0 - N7, can contain offset values used to increment/decrement address registers in address register update calculations or can be used for 16-bit general-purpose storage. For example, the contents of an offset register can be used to step through a table at some rate (e.g., five locations per step for waveform generation), or the contents can specify the offset into a table or the base of the table for indexed addressing. Each address register, Rn, has its own offset register, Nn, associated with it.

5.2.3 Modifier Register Files (M0 - M3 and M4 - M7)

The eight 16-bit modifier registers, M0 - M7, define the type of address arithmetic to be performed for addressing mode calculations, or they can be used for general-purpose storage. The address ALU supports linear, modulo, and reverse-carry arithmetic types for all address register indirect addressing modes. For modulo arithmetic, the contents of Mn also specify the modulus. Each address register, Rn, has its own modifier register, Mn, associated with it. Each modifier register is set to $FFFF on processor reset, which specifies linear arithmetic as the default type for address register update calculations.

5.3 ADDRESSING

The DSP56000/DSP56001 provides three different addressing modes: register direct, address register indirect, and special (see Table 5-1). Since the register direct and special addressing modes do not necessarily use the AGU registers, they are described in **SECTION**

Table 5-1 Address Register Indirect Summary

Address Register Indirect	Uses Mn Modifier	Operand Reference									Assembler Syntax
		S	C	D	A	P	X	Y	L	XY	
No Update	No					X	X	X	X	X	(RN)
Postincrement by 1	Yes					X	X	X	X	X	(RN)+
Postdecrement by 1	Yes					X	X	X	X	X	(RN)–
Postincrement by Offset Nn	Yes					X	X	X	X	X	(RN)+Nn

NOTE:
S = System Stack Reference
C = Program Control Unit Register Reference
D = Data ALU Register Reference
A = Address ALU Register Reference
P = Program Memory Reference
X = X Memory Reference
Y = Y Memory Reference
L = L Memory Reference
XY = XY Memory Reference

7 INSTRUCTION SET SUMMARY. The address register indirect addressing modes use the registers in the AGU and are described in the following paragraphs.

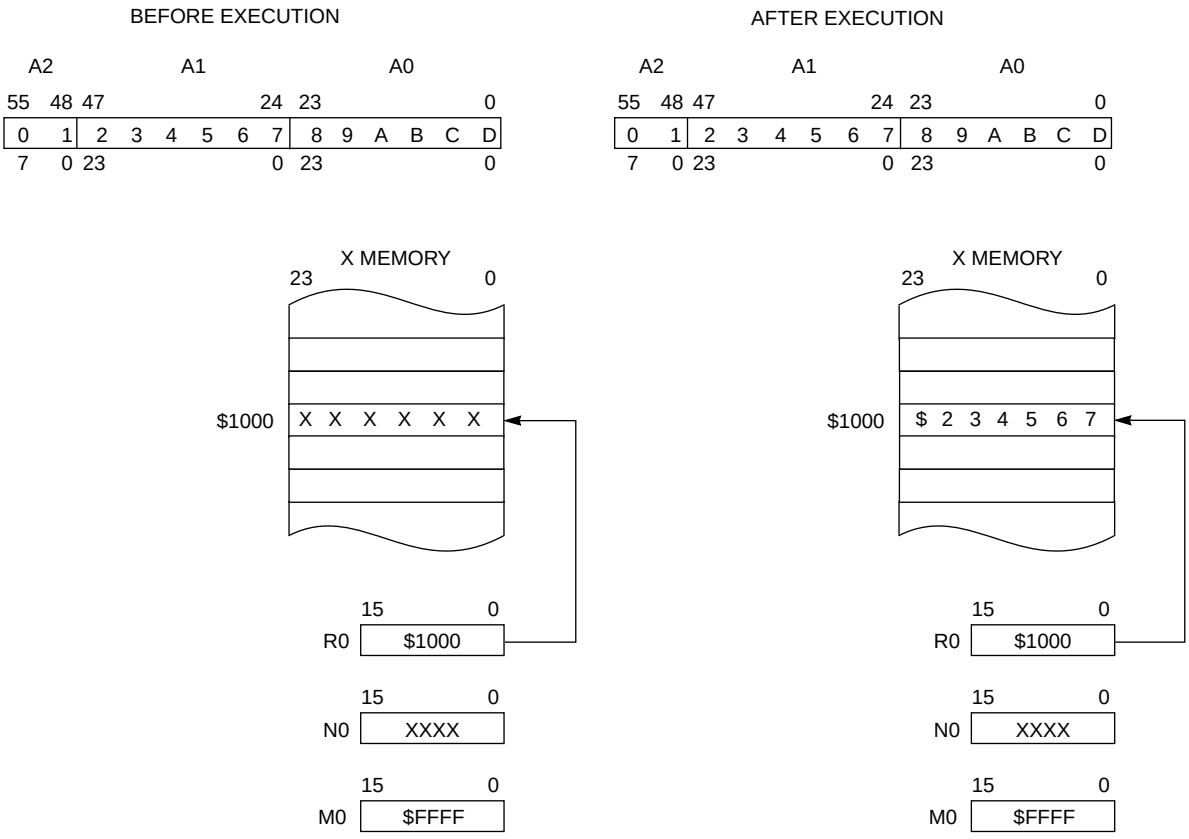
5.3.1 Address Register Indirect Modes

When an address register is used to point to a memory location, the addressing mode is called address register indirect (see Table 5-1). The term indirect is used because the register contents are not the operand itself, but rather the address of the operand. These addressing modes specify that an operand is in memory and specify the effective address of that operand.

A portion of the data bus movement field in the instruction specifies the memory space to be referenced. The contents of specific AGU registers that determine the effective address are modified by arithmetic operations performed in the AGU. The type of address arithmetic used is specified by the address modifier register, Mn. The offset register, Nn, is only used when the update specifies an offset.

Not all possible combinations are available, e.g., + (Rn). The 24-bit instruction word size of the DSP56000/DSP56001 is not large enough to allow a completely orthogonal instruction set for all instructions used by the processor.

EXAMPLE: MOVE A1,X: (R0)



Assembler Syntax: (Rn)
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

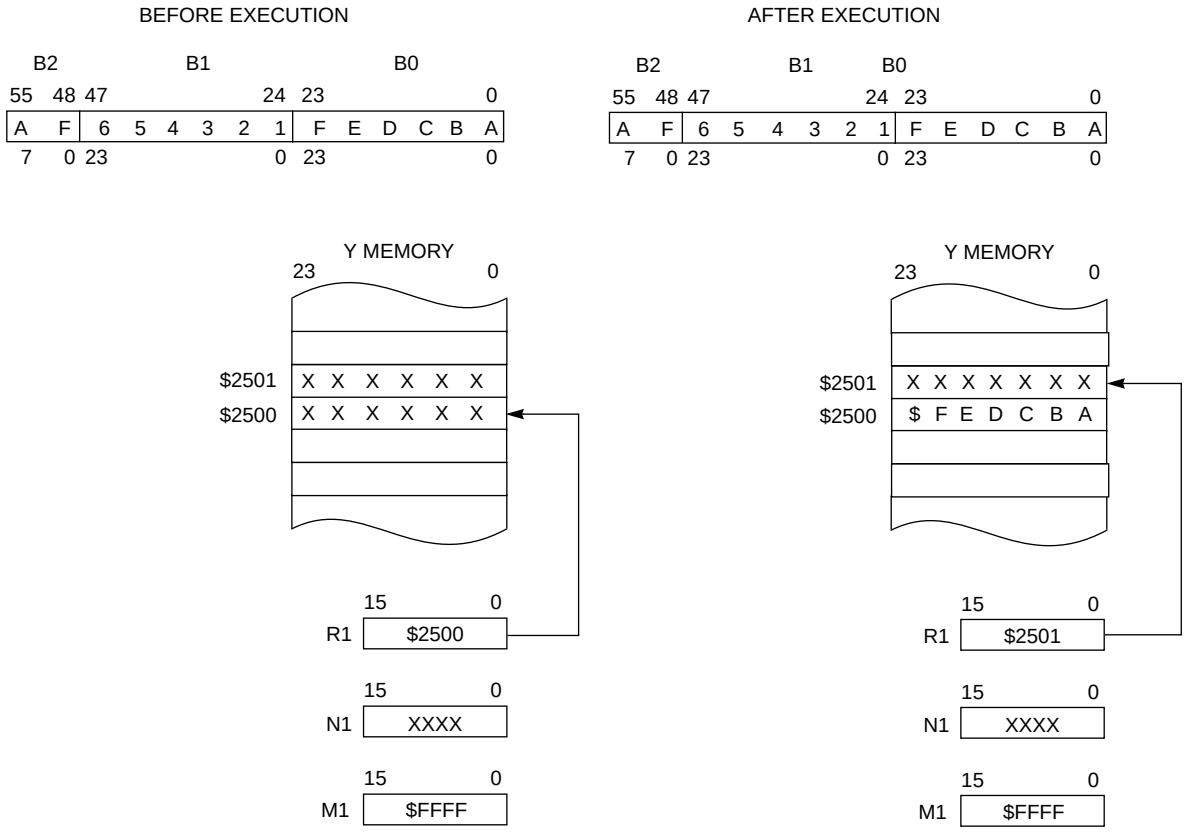
Figure 5-4 Address Register Indirect — No Update

An example and description of each mode is given in the following paragraphs. **SECTION 7 INSTRUCTION SET SUMMARY** and **APPENDIX A INSTRUCTION SET DETAILS** give a complete description of the instruction syntax used in these examples. In particular, XY: memory references refer to instructions in which an operand in X memory and an operand in Y memory are referenced in the same instruction.

5.3.1.1 No Update

The address of the operand is in the address register, Rn (see Table 5-1). The contents of the Rn register are unchanged by executing the instruction. Figure 5-4 shows a MOVE instruction using address register indirect addressing with no update. This mode can be used for making XY: memory references.

EXAMPLE: MOVE B0,Y: (R1)+



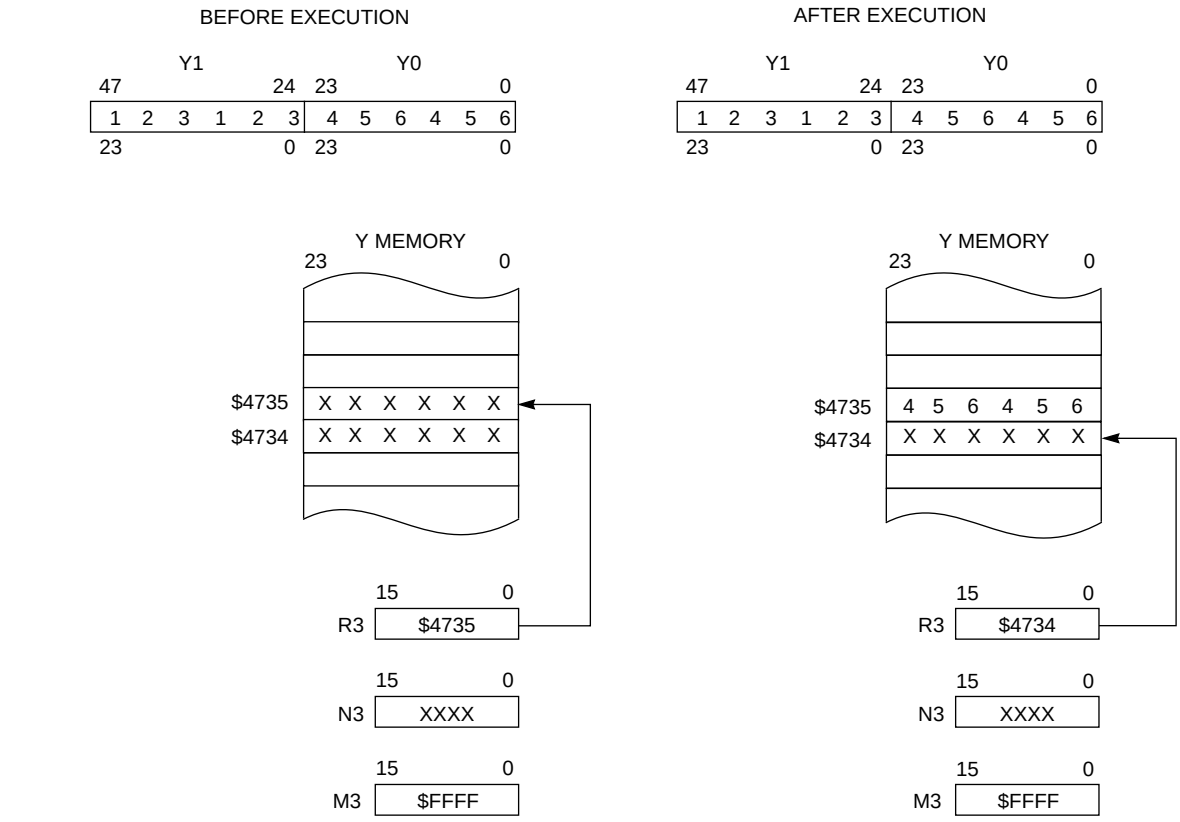
Assembler Syntax: (Rn)+
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 5-5 Address Register Indirect — Postincrement

5.3.1.2 Postincrement By 1

The address of the operand is in the address register, Rn (see Table 5-1 and Figure 5-5). After the operand address is used, it is incremented by 1 and stored in the same address register. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.

EXAMPLE: MOVE Y0,Y: (R3)-



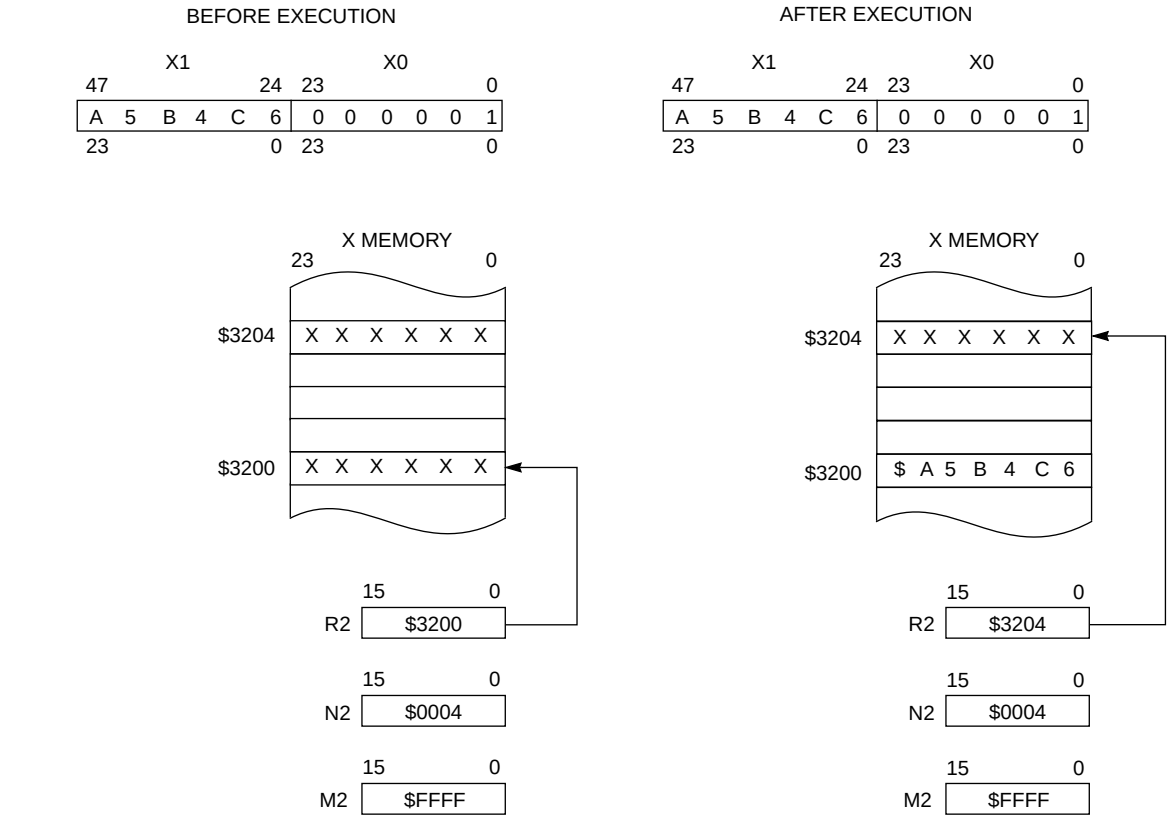
Assembler Syntax: (Rn)–
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 5-6 Address Register Indirect — Postdecrement

5.3.1.3 Postdecrement By 1

The address of the operand is in the address register, Rn (see Table 5-1 and Figure 5-6). After the operand address is used, it is decremented by 1 and stored in the same address register. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.

EXAMPLE: MOVE X1,X: (R2)+N2



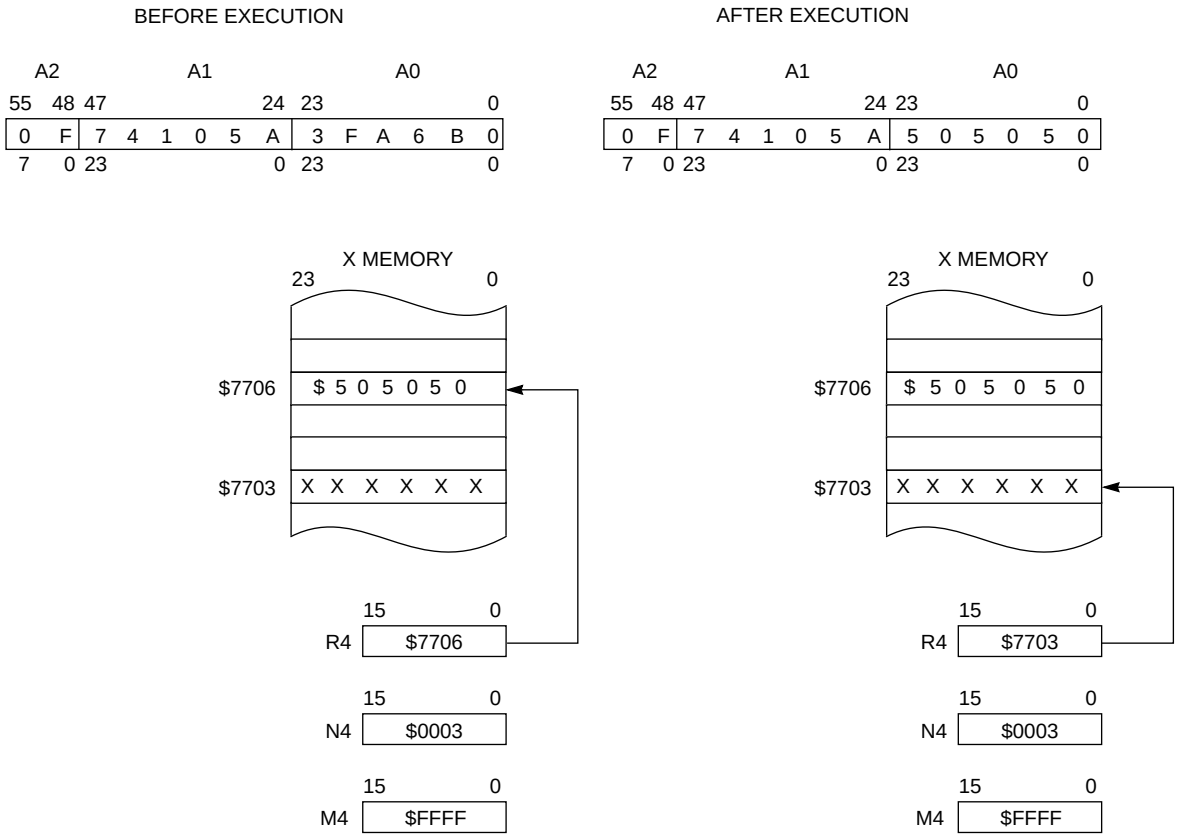
Assembler Syntax: (Rn)+Nn
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 5-7 Address Register Indirect — Postincrement by Offset Nn

5.3.1.4 Postincrement By Offset Nn

The address of the operand is in the address register, Rn (see Table 5-1 and Figure 5-7). After the operand address is used, it is incremented by the contents of the Nn register and stored in the same address register. The contents of the Nn register are unchanged. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.

EXAMPLE: MOVE X:(R4)–N4,A0



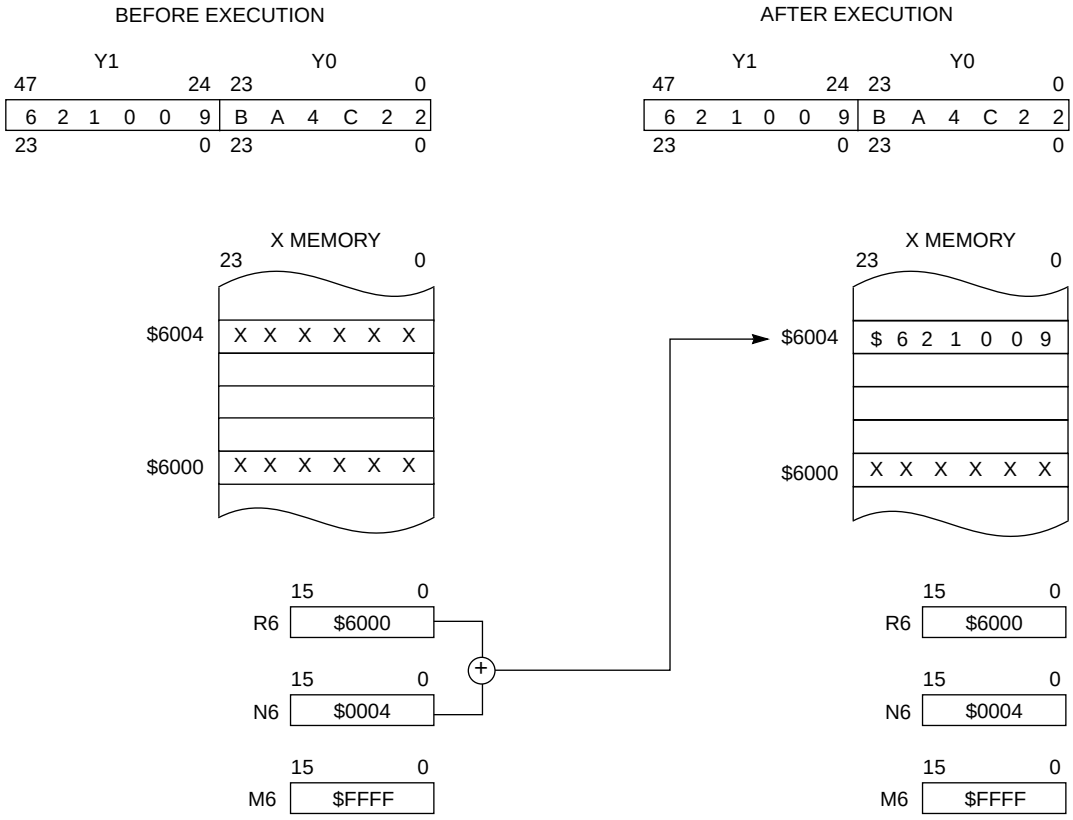
Assembler Syntax: (Rn)–Nn
Memory Spaces: P:, X:, Y:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 5-8 Address Register Indirect — Postdecrement by Offset Nn

5.3.1.5 Postdecrement By Offset Nn

The address of the operand is in the address register, Rn (see Table 5-1 and Figure 5-8). After the operand address is used, it is decremented by the contents of the Nn register and stored in the same address register. The contents of the Nn register are unchanged. This mode cannot be used for making XY: memory references, but it can be used to modify the contents of Rn without an associated data move.

EXAMPLE: MOVE Y1,X: (R6+N6)



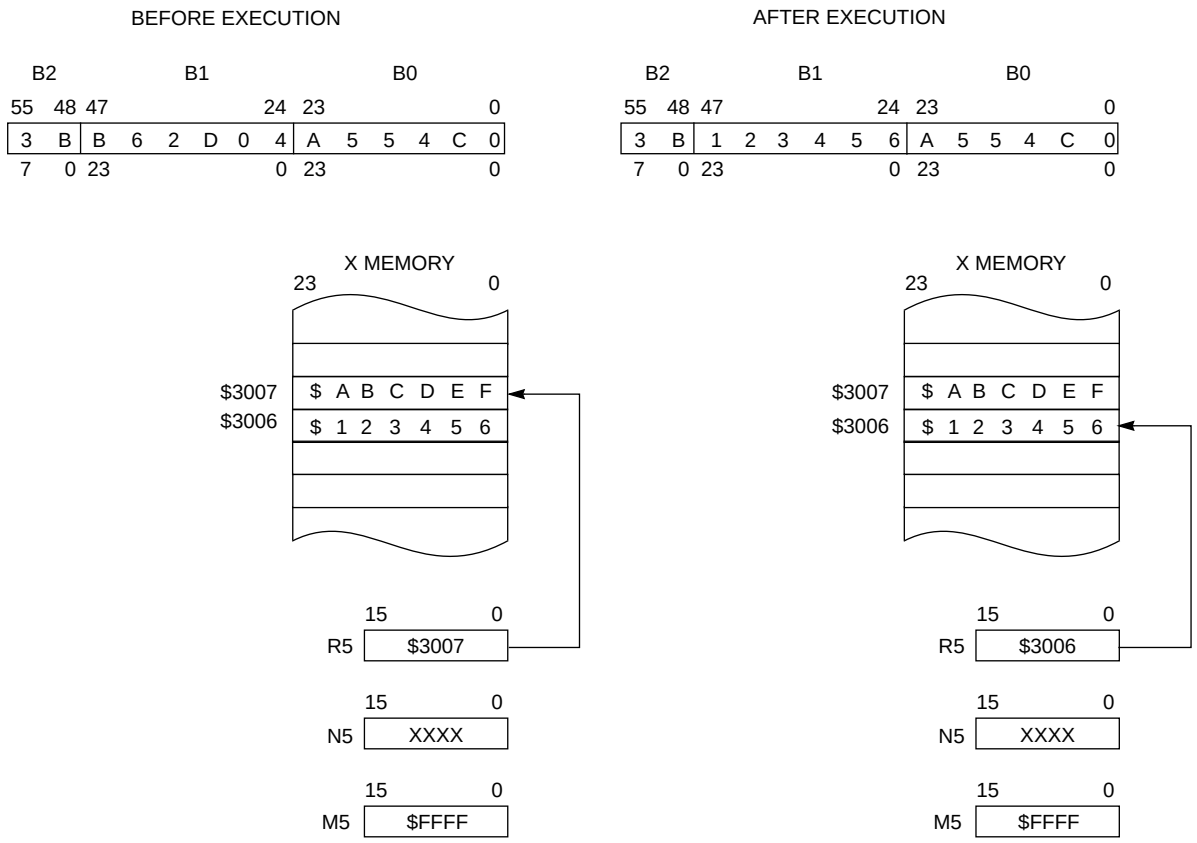
Assembler Syntax: (Rn+Nn)
Memory Spaces: P:, X:, Y:, L:
Additional Instruction Execution Time (Clocks): 2
Additional Effective Address Words: 0

Figure 5-9 Address Register Indirect — Indexed by Offset Nn

5.3.1.6 Indexed By Offset Nn

The address of the operand is the sum of the contents of the address register, Rn, and the contents of the address offset register, Nn (see Table 5-1 and Figure 5-9). The contents of the Rn and Nn registers are unchanged. This addressing mode, which requires an extra instruction cycle, cannot be used for making XY: memory references.

EXAMPLE: MOVE X: –(R5),B1



Assembler Syntax: –Rn
Memory Spaces: P:, X:, Y:, L:
Additional Instruction Execution Time (Clocks): 2
Additional Effective Address Words: 0

Figure 5-10 Address Register Indirect — Predecrement

5.3.1.7 Predecrement By 1

The address of the operand is the contents of the address register, Rn, decremented by 1 before the operand address is used (see Table 5-1 and Figure 5-10). The contents of Rn are decremented and stored in the same address register. This addressing mode requires an extra instruction cycle. This mode cannot be used for making XY: memory references, nor can it be used for modifying the contents of Rn without an associated data move.

5.3.2 Address Modifier Types

The DSP56000/DSP56001 address ALU supports linear, modulo, and reverse-carry arithmetic types for all address register indirect modes. These arithmetic types easily allow the creation of data structures in memory for FIFOs (queues), delay lines, circular buffers, stacks, and bit-reversed FFT buffers. Data is manipulated by updating address registers (pointers) rather than moving large blocks of data. The contents of the address modifier register, Mn, define the type of arithmetic to be performed for addressing mode calculations; for modulo arithmetic, the contents of Mn also specify the modulus. All address register indirect modes can be used with any address modifier. Each address register, Rn, has its own modifier register, Mn, associated with it.

5.3.2.1 Linear Modifier (Mn=$FFFF)

Address modification is performed using normal 16-bit linear (modulo 65,536) arithmetic (see Table 5-2). A 16-bit offset, Nn, and + or –1 can be used in the address calculations. The range of values can be considered as signed (Nn from –32,768 to + 32,767) or unsigned (Nn from 0 to + 65,535) since there is no arithmetic difference between these two data representations. Addresses are normally considered unsigned, and data is normally considered signed.

Table 5-2 Linear Address Modifiers

Modifier Mn Value	Addressing Mode Arithmetic
0	Reverse Carry (Bit Reverse)
1	Modulo 2
2	Modulo 3
:	:
:	Modulo (Mn+1)
:	:
32766	Modulo 32767
32767	Modulo 32768

5.3.2.2 Modulo Modifier (Mn=MODULUS–1)

The address modification is performed modulo M, where M ranges from 2 to + 32,768 (see Table 5-3).

Table 5-3 Modulo Address Modifiers

Modifier Mn Value	Addressing Mode Arithmetic
0	Reverse Carry (Bit Reverse)
1	**Modulo 2**
2	**Modulo 3**
:	:
:	**Modulo (Mn+1)**
:	:
32766	**Modulo 32767**
32767	**Modulo 32768**
:	Reserved
65535	Linear (Modulo 65536)

Modulo M arithmetic causes the address register value to remain within an address range of size M, defined by a lower and upper address boundary (see Figure 5-11).

The value m=M–1 is stored in the modifier register, Mn. The lower boundary (base address) value must have zeros in the k LSBs, where $2^k \geq M$, and therefore must be a multiple of 2^k. The upper boundary is the lower boundary plus the modulo size minus one (base address plus M–1). Since $M \leq 2^k$, once M is chosen, a sequential series of memory blocks (each of length 2^k) is created where these circular buffers can be located. If $M < 2^k$, there will be a space between sequential circular buffers of (2^k)–M.

For example, to create a circular buffer of 21 stages, M is 21, and the lower address boundary must have its five LSBs equal to zero ($2^k \geq 21$, thus $k \geq 5$). The Mn register is loaded with the value 20. The lower boundary may be chosen as 0, 32, 64, 96, 128, 160, etc. The upper boundary of the buffer is then the lower boundary plus 21. There will be an unused space of 11 memory locations between the upper address and next usable lower address. The address pointer is not required to start at the lower address boundary or to end on the upper address boundary; it can initially point anywhere within the defined modulo address range. Neither the lower nor the upper boundary of the modulo region is stored; only the size of the modulo region is stored in Mn. The boundaries are determined

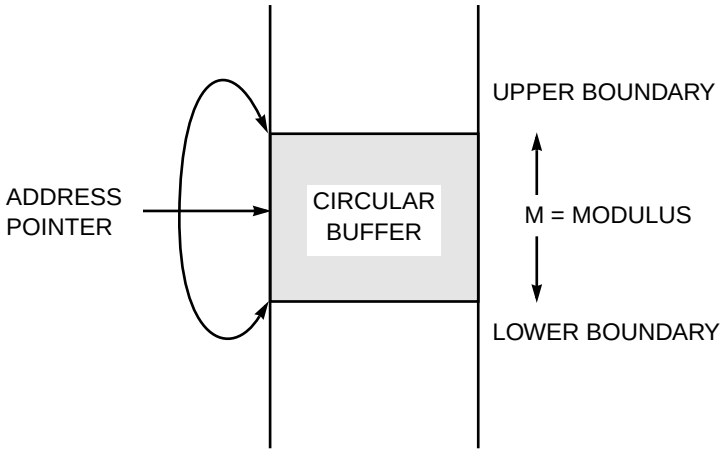


Figure 5-11 Circular Buffer

by the contents of Rn. Assuming the (Rn)+ indirect addressing mode, if the address register pointer increments past the upper boundary of the buffer (base address plus M–1), it will wrap around through the base address (lower boundary). Alternatively, assuming the (Rn)- indirect addressing mode, if the address decrements past the lower boundary (base address), it will wrap around through the base address plus M–1 (upper boundary).

If an offset, Nn, is used in the address calculations, the 16-bit absolute value, |Nn|, must be less than or equal to M for proper modulo addressing. If Nn>M, the result is data dependent and unpredictable, except for the special case where Nn=P x 2^k, a multiple of the block size where P is a positive integer. For this special case, when using the (Rn)+ Nn addressing mode, the pointer, Rn, will jump linearly to the same relative address in a new buffer, which is P blocks forward in memory (see Figure 5-12).

Similarly, for (Rn)–Nn, the pointer will jump P blocks backward in memory. This technique is useful in sequentially processing multiple tables or N-dimensional arrays. The range of values for Nn is –32,768 to + 32,767. The modulo arithmetic unit will automatically wrap around the address pointer by the required amount. This type address modification is useful for creating circular buffers for FIFOs (queues), delay lines, and sample buffers up to 32,768 words long as well as for decimation, interpolation, and waveform generation. The special case of (Rn) $\pm$ Nn mod M with Nn=P x 2^k is useful for performing the same algorithm on multiple blocks of data in memory — e.g., parallel infinite impulse response (IIR) filtering.

An example of address register indirect modulo addressing is shown in Figure 5-13 Starting at location 64, a circular buffer of 21 stages is created. The addresses generated are offset by 15 locations. The lower boundary = L x (2^k) where $2^k \geq 21$; therefore, k=5 and the lower address boundary must be a multiple of 32. The lower boundary may be chosen

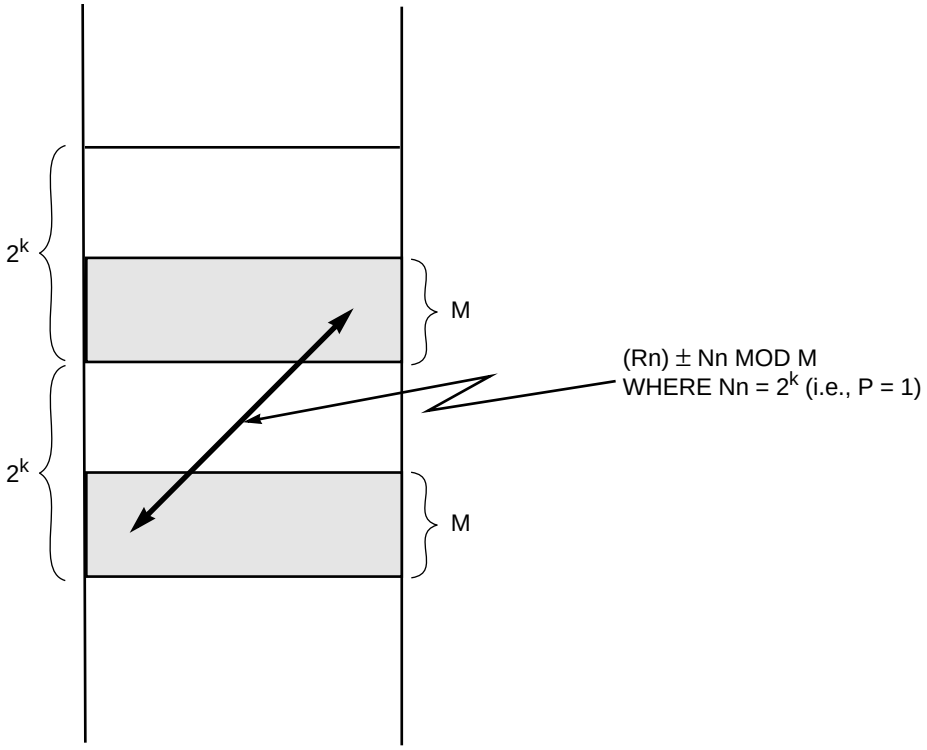


Figure 5-12 Linear Addressing with a Modulo Modifier

as 0, 32, 64, 96, 128, 160, etc. For this example, L is arbitrarily chosen to be 2, making the lower boundary 64. The upper boundary of the buffer is then 84 (the lower boundary plus 20 (M–1)). The Mn register is loaded with the value 20 (M–1). The offset register is arbitrarily chosen to be 15 (Nn≤M). The address pointer is not required to start at the lower address boundary and can begin anywhere within the defined modulo address range — i.e., within the lower boundary + (2^k) address region. The address pointer, Rn, is arbitrarily chosen to be 75 in this example. When R2 is postincremented by the offset by the MOVE instruction, instead of pointing to 90 (as it would in the linear mode) it wraps around to 69. If the address register pointer increments past the upper boundary of the buffer (base address plus M–1), it will wrap around to the base address. If the address decrements past the lower boundary (base address), it will wrap around to the base address plus M–1.

If Rn is outside the valid modulo buffer range and an operation occurs that causes Rn to be updated, the contents of Rn will be updated according to modulo arithmetic rules. For example, a MOVE B0,X:(R0)+ N0 instruction (where R0=6, M0=5, and N0=0) would apparently leave R0 unchanged since N0=0. However, since R0 is above the upper boundary, the AGU calculates R0+ N0–M0–1 for the new contents of R0 and sets R0=0.

The MOVE instruction in Figure 5-13 takes the contents of the X0 register and moves it to a location in the X memory pointed to by (R2), and then (R2) is updated modulo 21. The

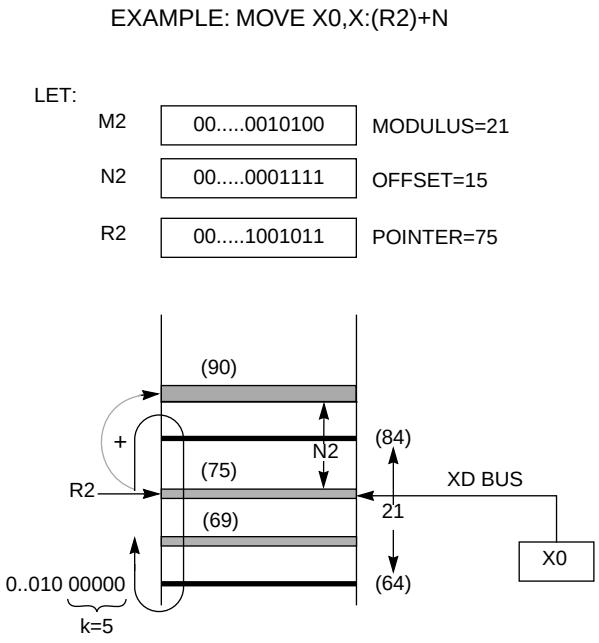


Figure 5-13 Modulo Modifier Example

new value of R2 is not 90 (75+ 15), which would be the case if linear arithmetic had been used, but rather is 69 since modulo arithmetic was used.

5.3.2.3 Reverse-Carry Modifier (Mn=$0000)

Reverse carry is selected by setting the modifier register to zero (see Table 5-4). The address modification is performed in hardware by propagating the carry in the reverse direction — i.e., from the MSB to the LSB. Reverse carry is equivalent to bit reversing the contents of Rn (i.e., redefining the MSB as the LSB, the next MSB as bit 1, etc.) and the offset value, Nn, adding normally, and then bit reversing the result. If the + Nn addressing mode is used with this address modifier and Nn contains the value $2^{(k-1)}$ (a power of two), this addressing modifier is equivalent to bit reversing the k LSBs of Rn, incrementing Rn by 1, and bit reversing the k LSBs of Rn again. This address modification is useful for addressing the twiddle factors in 2k-point FFT addressing and to unscramble 2^k-point FFT data. The range of values for Nn is 0 to + 32K (i.e., $Nn=2^{15}$), which allows bit-reverse addressing for FFTs up to 65,536 points.

Table 5-4 Reverse-Carry Address Modifiers

Modifier Mn Value	Addressing Mode Arithmetic
0	**Reverse Carry (Bit Reverse)**
1	Modulo 2
2	Modulo 3
:	:
:	Modulo (Mn+1)
:	:
32766	Modulo 32767
32767	Modulo 32768
:	Reserved
65535	Linear (Modulo 65536)

To make bit-reverse addressing work correctly for a 2^k point FFT, the following procedures must be used:

1. Set Mn=0; this selects reverse-carry arithmetic.
2. Set Nn=$2^{(k-1)}$.
3. Set Rn between the lower boundary and upper boundary in the buffer memory. The lower boundary is L x (2^k), where L is an arbitrary whole number. This boundary gives a 16-bit binary number ‘‘xx . . . xx00 . . . 00’’, where xx . . . xx=L and 00 . . . 00 equals k zeros. The upper boundary is L x (2^k)+ ((2^k)–1). This boundary gives a 16-bit binary number ‘‘xx . . . xx11 . . . 11’’, where xx . . . xx=L and 11 . . . 11 equals k ones.
4. Use the (Rn)+ Nn addressing mode.

As an example, consider a 1024-point FFT with real data stored in the X memory and imaginary data stored in the Y memory. Since 1,024=210, k=10. The modifier register (Mn) is zero to select bit-reverse addressing. Offset register (Nn) contains the value 512 ($2^{(k-1)}$), and the pointer register (Rn) contains 3,072 (L x (2^k)=3 x (2^{10})), which is the lower boundary of the memory buffer that holds the results of the FFT. The upper boundary is 4,095 (lower boundary + (2^k)–1=3,072+ 1,023).

Postincrementing by + N generates the address sequence (0, 512, 256, 768, 128, 640,...),

which is added to the lower boundary. This sequence (0, 512, etc.) is the scrambled FFT data order for sequential frequency points from 0 to 2 x pi. Table 5-5 shows the successive contents of Rn when using (Rn)+ Nn updates.

Table 5-5 Bit-Reverse Addressing Sequence Example

Rn Contents	Offset From Lower Boundary
3072	0
3584	512
3328	256
3840	768
3200	128
3712	640

The reverse-carry modifier only works when the base address of the FFT data buffer is a multiple of 2^k, such as 1,024, 2,048, 3,072, etc. The use of addressing modes other than postincrement by + Nn is possible but may not provide a useful result.

The term bit reverse with respect to reverse-carry arithmetic is descriptive. The lower boundary that must be used for the bit-reverse address scheme to work is L x (2^k). In the previous example shown in Table 5-5, L=3 and k=10. The first address used is the lower boundary (3072); the calculation of the next address is shown in Figure 5-14. The k LSBs of the current contents of Rn (3,072) are swapped:

- Bits 0 and 9 are swapped.
- Bits 1 and 8 are swapped.
- Bits 2 and 7 are swapped.
- Bits 3 and 6 are swapped.
- Bits 4 and 5 are swapped.

The result is incremented (3,073), and then the k LSBs are swapped again:

- Bits 0 and 9 are swapped.
- Bits 1 and 8 are swapped.
- Bits 2 and 7 are swapped.
- Bits 3 and 6 are swapped.
- Bits 4 and 5 are swapped.

The result is Rn equals 3,584.

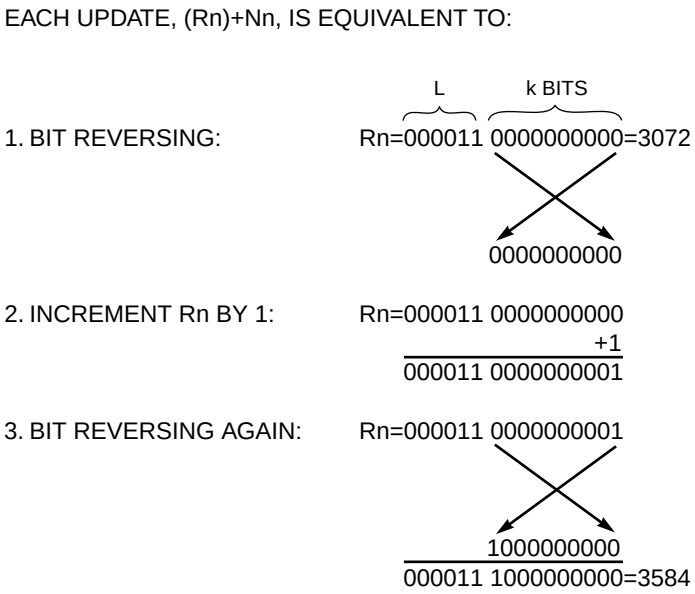


Figure 5-14 Bit-Reverse Address Calculation Example

5.3.2.4 Address-Modifier-Type Encoding Summary

Table 5-6 is a summary of the address modifier types discussed in the previous paragraphs. There are three modifier types:

- Linear Addressing
- Reverse-Carry Addressing
- Modulo Addressing

Bit-reverse addressing is useful for 2^k-point FFT addressing. Modulo addressing is useful for creating circular buffers for FIFOs (queues), delay lines, and sample buffers up to 32,768 words long. The linear addressing is useful for general-purpose addressing. There is a reserved set of modifier values (from 32,768 to 65,534) that should not be used.

Figure 5-15 gives examples of the three addressing modifiers using 8-bit registers for simplification (all AGU registers in the DSP56000/DSP56001 are 16 bit). The addressing mode used in the example, postincrement by offset Nn, adds the contents of the offset register to the contents of the address register after the address register is accessed. The

results of the three examples are as follows:

- The linear address modifier addresses every fifth location since the offset register contains $5.
- Using the bit-reverse address modifier causes the postincrement by offset Nn addressing mode to use the address register, bit reverse the four LSBs, increment by 1, and bit reverse the four LSBs again.
- The modulo address modifier has a lower boundary at a predetermined location, and the modulo number plus the lower boundary establishes the upper boundary. This boundary creates a circular buffer so that, if the address register is pointing within the boundaries, addressing past a boundary causes a circular wraparound to the other boundary.

Table 5-6 Address-Modifier-Type Encoding Summary

Modifier Mn	Rn Update Arithmetic
0	Reverse Carry (Bit Reverse) Addressing
1	Modulo 2
2	Modulo 3
:	:
:	Modulo (Mn+1) Addressing
:	:
32767	Modulo 32768

LINEAR ADDRESS MODIFIER

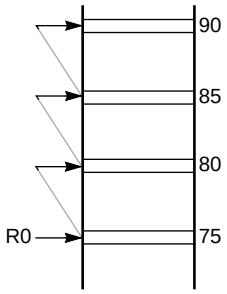
M0 = 255 = 11111111 FOR LINEAR ADDRESSING WITH R0

ORIGINAL REGISTERS: N0 = 5, R0 = 75 = 0100 1011

POSTINCREMENT BY OFFSET N0: R0 = 80 = 0101 0000

POSTINCREMENT BY OFFSET N0: R0 = 85 = 0101 0101

POSTINCREMENT BY OFFSET N0: R0 = 90 = 0101 1010



MODULO ADDRESS MODIFIER

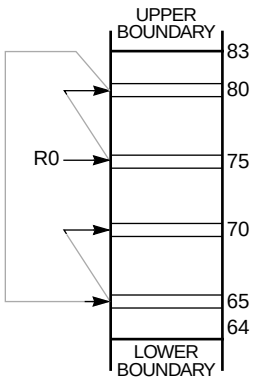
M0 = 19 = 0001 0011 FOR MODULO 20 ADDRESSING WITH R0

ORIGINAL REGISTERS: N0 = 5, R0 = 75 = 0100 1011

POSTINCREMENT BY OFFSET N0: R0 = 80 = 0101 0000

POSTINCREMENT BY OFFSET N0: R0 = 65 = 0100 0001

POSTINCREMENT BY OFFSET N0: R0 = 70 = 0100 0110



REVERSE-CARRY ADDRESS MODIFIER

M0 = 0= 0000 0000 FOR REVERSE-CARRY ADDRESSING WITH R0

ORIGINAL REGISTERS: N0 = 8, R0 = 64 = 0100 0000

POSTINCREMENT BY OFFSET N0: R0 = 72 = 0100 1000

POSTINCREMENT BY OFFSET N0: R0 = 68 = 0100 0100

POSTINCREMENT BY OFFSET N0: R0 = 76 = 0100 1100

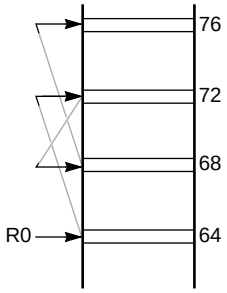


Figure 5-15 Address Modifier Summary

SECTION 6
PROGRAM CONTROL UNIT

This section describes the hardware of the program control unit and concludes with a description of the programming model. The instruction pipeline description is also included since understanding the pipeline is particularly important in understanding the DSP56000/DSP56001.

6.1 OVERVIEW

The program control unit (one of the three concurrent execution units in the central processor) performs program address generation (instruction prefetch), instruction decoding, hardware DO loop control, and exception processing (see Figure 6-1). The programmer views the program control unit as consisting of six registers and a hardware system stack (SS) as shown in Figure 6-2. In addition to the standard program flow-control resources, such as a program counter (PC), complete status register (SR), and SS, the program control unit features registers (loop address (LA) and loop counter (LC) dedicated to supporting the hardware DO loop instruction.

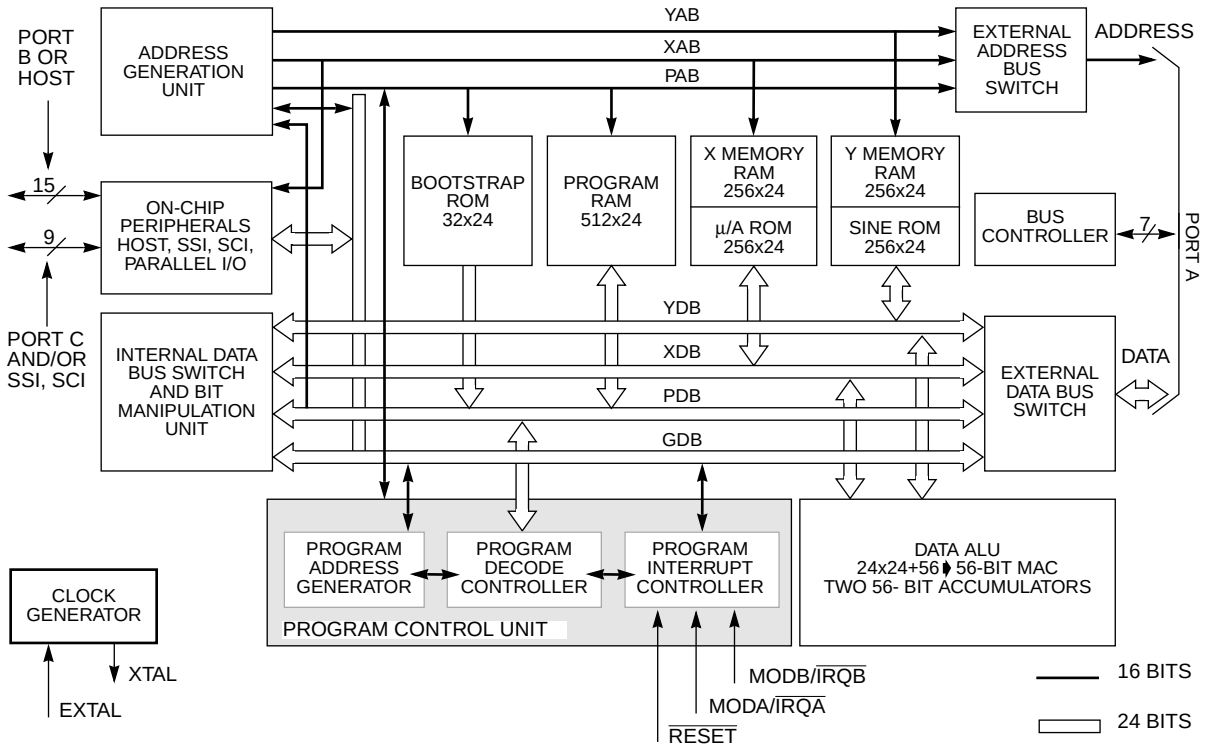


Figure 6-1 DSP56001 Block Diagram

The SS is a 15-level by 32-bit separate internal memory used to store the PC and SR

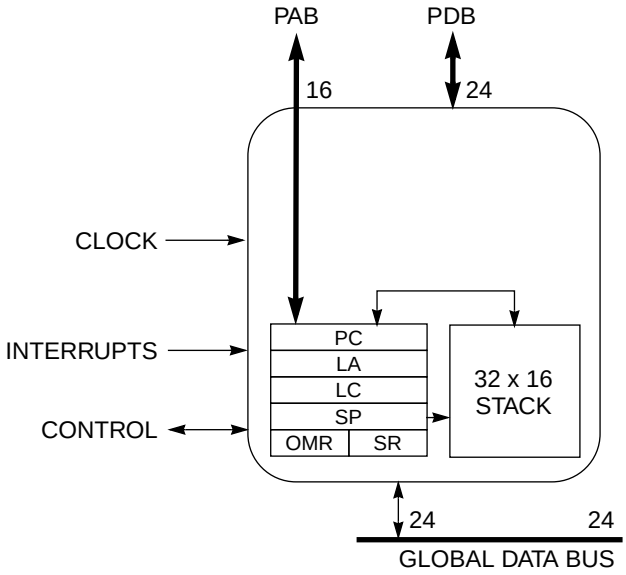


Figure 6-2 DSP56000/DSP56001 Program Control Unit

during subroutine calls and long interrupts. The SS will also store the LC and LA registers in addition to the PC and SR registers for program looping. Each location in the SS is addressable as 16-bit registers, system stack high (SSH) and system stack low (SSL), which are pointed to by the stack pointer (SP). Thus, SS management is under software control.

All registers are read/write to facilitate system debugging. Although none of the program control unit registers are 24 bits, they are read or written over 24-bit buses. When they are read, the least significant bits (LSBs) are significant, and the most significant bits (MSBs) are zeroed as appropriate. When they are written, only the appropriate LSBs are significant, and the MSBs are written as don't care. The program control unit implements a three-stage (prefetch, decode, execute) pipeline and controls the five processing states of the DSP56000/DSP56001: normal, exception, reset, wait, and stop.

6.2 PROGRAM CONTROL UNIT ARCHITECTURE

The program control unit consists of three hardware blocks: the program decode controller (PDC), the program address generator (PAG), and the program interrupt controller (PIC) (see Figure 6-1).

6.2.1 Program Decode Controller

The PDC contains the program logic array decoders, the register address bus generator, the loop state machine, the repeat state machine, the condition code generator, the inter-

rupt state machine, the instruction latch, and the backup instruction latch. The PDC decodes the 24-bit instruction loaded into the instruction latch and generates all signals necessary for pipeline control. The backup instruction latch stores a duplicate of the prefetched instruction to optimize execution of the repeat (REP) and jump (JMP) instructions.

6.2.2 Program Address Generator

The PAG contains the PC, the SP, the SS, the operating mode register (OMR), the SR, the LC register, and the LA register. Loops, which are frequent constructs in digital signal processing (DSP) algorithms, are supported by dedicated hardware on the DSP56000/ DSP56001. Executing a DO instruction loads the LC register with 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 during one loop pass), and asserts the loop flag in the SR. Executing the DO instruction also causes the contents of the LA, LC, and SR to be stacked prior to the execution of the DO instruction, thereby supporting nesting of DO loops. Under control of the loop state machine, the address of the first instruction in the loop is also stacked so the loop can be repeated with no overhead. While the loop flag 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 SS is read to update the PC with the address of the first instruction in the loop, effectively executing an automatic branch. If the LC is equal to one, then the LC, LA, and the loop flag in the SR are restored with the stack contents, while instruction fetches continue at the incremented PC value (LA + 1).

Block data moves can be accomplished using the repeat feature. The REP instruction loads the LC with the number of times the next instruction is to be repeated. Since the instruction to be repeated is only fetched once, throughput is increased by reducing external bus contention. However, REP instructions are not interruptable since they are fetched only once. A single-instruction DO loop can be used in place of an REP if interrupts must be allowed.

6.2.3 Program Interrupt Controller

The PIC receives all interrupt requests, arbitrates among all of them each cycle, and generates the interrupt vector address. There are four external and 16 internal interrupt sources that may generate interrupts.

The interrupts are organized in a flexible priority structure. Each interrupt has associated with it an interrupt priority level (IPL) that can be from zero to three. Levels 0 (lowest level), 1, and 2 are maskable. Level 3 is the highest IPL and is not maskable. Two interrupt mask bits in the SR reflect the current processor IPL and indicate the level needed for an interrupt source to interrupt the processor. Interrupts are inhibited for all IPLs less than the current processor priority. Level 3 interrupts can always interrupt the processor. All interrupt sources and their IPLs are listed in Table 6-1. Each interrupt source is vectored (one of 32 vectors) to a separate, fixed, two-word service routine located in the lowest 64 words of program memory. If some of this space is not used, it may be used for program storage.

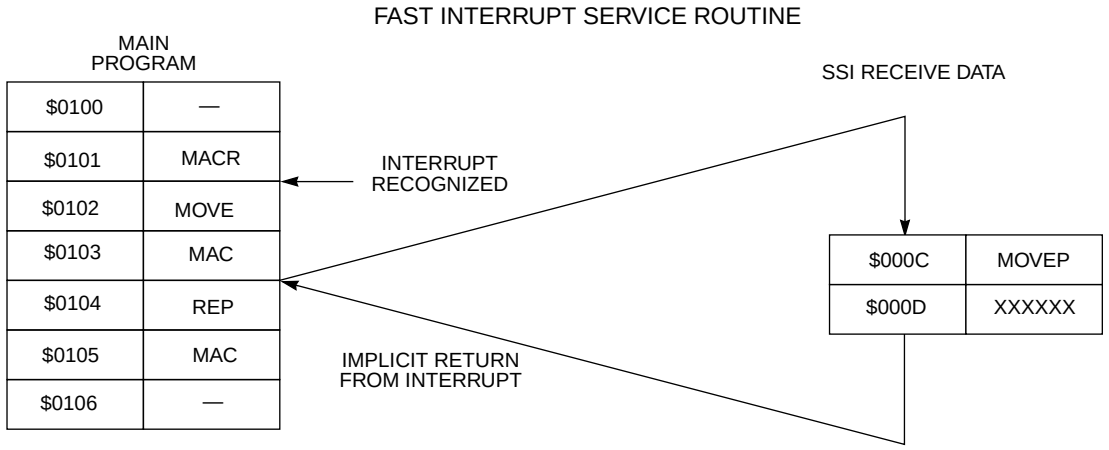
Upon entering the exception processing state, the current instruction in decode will execute normally, unless it is the first word of a two-word instruction, in which case it will be aborted and refetched at the completion of exception processing. The next two fetch addresses are supplied by the PIC. During these fetches, the PC is not updated. The PIC generates an interrupt instruction fetch address, which points to the first instruction word of a two-word fast-interrupt routine. All interrupts begin as fast interrupts (Figure 6-3 (a)). During fast interrupt servicing, the two instruction words at the interrupt vector addresses are jammed into the instruction stream without any overhead or stack usage. If one of the two words is a jump to subroutine (JSR), the fast interrupt routine becomes a long interrupt routine (see Figure 6-3 (b)). The long interrupt service is the traditional context switch in which the stack is used for saving the status and return address. Subroutines and interrupts can be nested using the 15-level stack. The stack can be extended in memory by

Table 6-1 Interrupt Sources

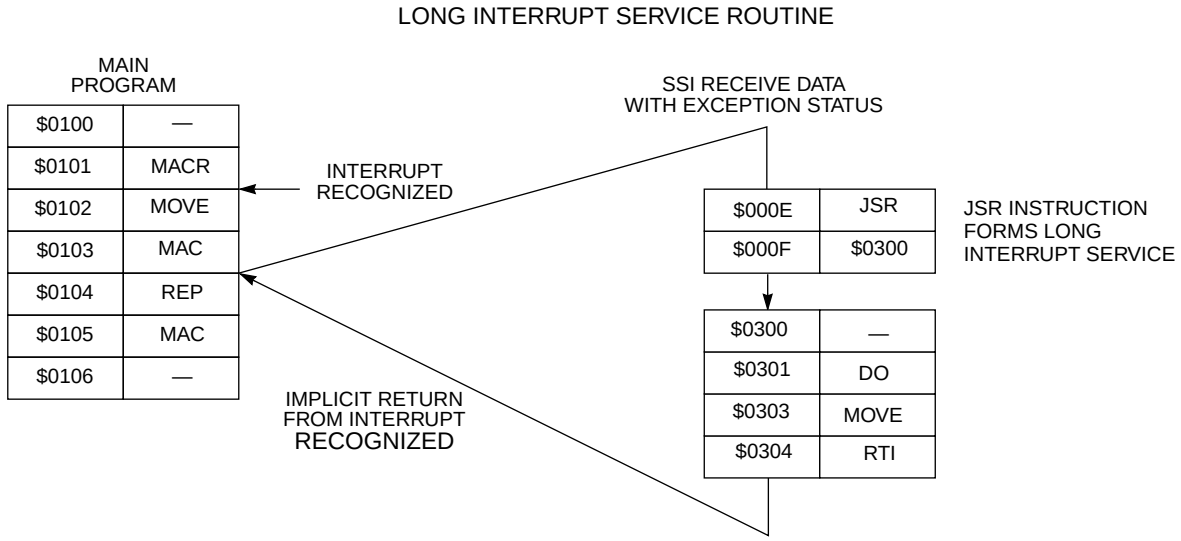
Interrupt Starting Address	IPL	Interrupt Source
P:$0000 or P:$E000	3	Hardware $\overline{\text{RESET}}$ (External)
P:$0002	3	Stack Error
P:$0004	3	Trace
P:$0006	3	SWI (Software Interrupt)
P:$0008	0-2	$\overline{\text{IRQA}}$ (External)
P:$000A	0-2	$\overline{\text{IRQB}}$ (External)
P:$000C	0-2	SSI Receive Data
P:$000E	0-2	SSI Receive Data with Exception Status
P:$0010	0-2	SSI Transmit Data
P:$0012	0-2	SSI Transmit Data with Exception Status
P:$0014	0-2	SCI Receive Data
P:$0016	0-2	SCI Receive Data with Exception Status
P:$0018	0-2	SCI Transmit Data
P:$001A	0-2	SCI Idle Line
P:$001C	0-2	SCI Timer
P:$001E	3	NMI — Reserved for Hardware Development (External)
P:$0020	0-2	Host Receive Data
P:$0022	0-2	Host Transmit Data
P:$0024	0-2	Host Command (Default)
P:$0026	0-2	Available for Host Command
P:$0028	0-2	Available for Host Command
P:$002A	0-2	Available for Host Command
P:$002C	0-2	Available for Host Command
P:$002E	0-2	Available for Host Command
P:$0030	0-2	Available for Host Command
P:$0032	0-2	Available for Host Command
P:$0034	0-2	Available for Host Command
P:$0036	0-2	Available for Host Command
P:$0038	0-2	Available for Host Command
P:$003A	0-2	Available for Host Command
P:$003C	0-2	Available for Host Command
P:$003E	0-2	Illegal Instruction

using software to access the SSH and SSL registers. The exception processing state is described in more detail in SECTION 8 PROCESSING STATES.

Two external interrupt request inputs, IRQA and IRQB, can be defined as either level sensitive or negative edge triggered. One other external interrupt source is available. The nonmaskable interrupt (NMI) is edge sensitive and is generated on the first transition to 10



(a) DSP56000/DSP56001 Fast Interrupt



(b) DSP56000/DSP56001 Long Interrupt

Figure 6-3 Fast and Long Interrupt Examples

V on the IRQB pin after the last time that the NMI interrupt was serviced or the chip was reset. The NMI is a priority level 3 interrupt and cannot be masked. Only RESET and Illegal Instruction have higher priority than NMI. NMI is reserved for hardware development and should not be used as a general-purpose interrupt pin. Continued use of this interrupt can cause damage to the chip (see the DSP56001 Advance Information Data Sheet (ADI1290)). NMI has been provided strictly as an aid to the developer. The hardware reset address vector may point to internal (P:$0000) or external (P:$E000) program memory, determined by the value of the MODA and MODB pins when the RESET pin is deasserted.

The NMI, trace, and software interrupt (SWI) instructions are used for debugging and development purposes. The SWI instruction is useful for implementing breakpoints. Tracing is entered after turning on the trace flag in the SR. During tracing, a trace interrupt will be generated after each instruction is executed, thereby creating a single-step feature.

Internally, the peripheral registers are accessed through the global data bus. All on-chip peripherals use the same interrupt request interface mechanism. Each peripheral provides a single interrupt request line to the PIC and receives two lines: vector read and interrupt acknowledge. Each peripheral possesses more than one interrupt source (see Table 6-1); therefore, interrupt arbitration between internal peripherals must be handled by the peripheral according to its own predefined IPL. The PIC arbitrates between the different I/O peripherals; when one of them is selected, the peripheral supplies the correct vector address to the PIC. The host command vector in the host interface (see CHAPTER 10 PORT B) can be programmed to point to any of the 32 starting addresses, including 13 routines designated specifically as host commands and located at locations P:$0024–P:$003C. The default value set in the host command vector register during a reset is $0024.

6.2.4 Instruction Pipeline

The program control unit implements a three-level pipelined architecture in which concurrent instruction fetch, decode, and execution occur. The fact that the pipelined operation remains essentially hidden from the user makes programming straightforward. The pipeline is illustrated in Figure 6-4. The first instruction, I1, should be interpreted as follows: multiply the contents of X0 by the contents of Y0, add the product to the contents already in accumulator A, round the result to the “nearest even,” store the result back in accumulator A, move the contents in X data memory (pointed to by R0) into X0; postincrement R0; move the contents in Y data memory (pointed to by R4) into Y1; postincrement R4. The second instruction, I2, should be interpreted as follows: clear accumulator A; move the contents in X0 into the location in X data memory pointed to by R0; postincrement R0; before the clear operation, move the contents in accumulator A into the location in Y data memory pointed to by R4; postdecrement R4. The third instruction, I3, is the same as I1, except a rounding operation is not performed. The operations of each of the execution units and all initial conditions necessary to follow the execution of the instruction sequence

are depicted in Figure 6-4.

6.3 CLOCK OSCILLATOR

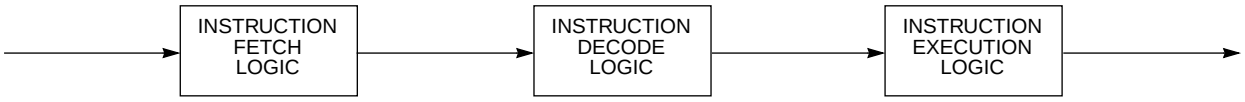
The DSP56000/DSP56001 uses a four-phase clock for instruction execution; therefore, the clock runs at twice the instruction execution rate. The clock can be provided by an internal oscillator (see Figure 6-1) by connecting an external crystal between XTAL and EXTAL or by an external oscillator connected to EXTAL.

6.4 PROGRAMMING MODEL

The program control unit features LA and LC registers dedicated to supporting the hardware DO loop instruction in addition to the standard program flow-control resources, such as a PC, complete SR, and SS. With the exception of the PC, all registers are read/write to facilitate system debugging. Figure 6-5 shows the program control unit programming model with the six registers and SS. The following paragraphs give a detailed description of each register.

6.4.1 Program Counter

This 16-bit register contains the address of the next location to be fetched from program memory space. The PC can point to instructions, data operands, or addresses of operands. References to this register are always inherent and are implied by most instructions.



EXAMPLE: PROGRAM SEGMENT

```
11   MACR    X0,Y1,A    X:(R0)+,X0    Y:(R4)+,Y1
12   CLR     A          X0,X:(R0)+    A,Y:(R4)-
13   MAC     X0,Y1,A    X:(R0)+,X0    Y:(R4)+,Y1
```

INSTRUCTION FETCH INSTRUCTION DECODE INSTRUCTION EXECUTION		11	12 11	13 12 11	14 13 12	15 14 13
PARALLEL OPERATIONS	INITIAL CONDITIONS					
ADDRESS UPDATE (AGU)	R0=$0005 R4=$0008	→	→	R0=5+1 R4=8+1	R0=6+1 R4=9–1	R0=7+1 R4=8+1
INSTRUCTION EXECUTION	A: A2=$00 A1=$000066 A0=$000000	→	→	A: A2=$00 A1=$0000A2 A0=$000000	A: A2=$00 A1=$000000 A0=$000000	A: A2=$00 A1=$000000 A0=$000050
(DATA ALU)	X0=$400000 Y1=$000077	→	→	X0=$000005 Y1=$000008	X0=$000005 Y1=$000008	X0=$000007 Y1=$000008
X MEMORY AT ADDRESS $0005 $0006 $0007	DATA $000005 $000006 $000007	→	→	$000005 $000006 $000007	$000005 $000005 $000007	$000005 $000005 $000007
Y MEMORY AT ADDRESS $0008 $0009	DATA $000008 $000009	→	→	$000008 $000009	$000008 $0000A2	$000008 $0000A2

Figure 6-4 Three-Stage Pipeline

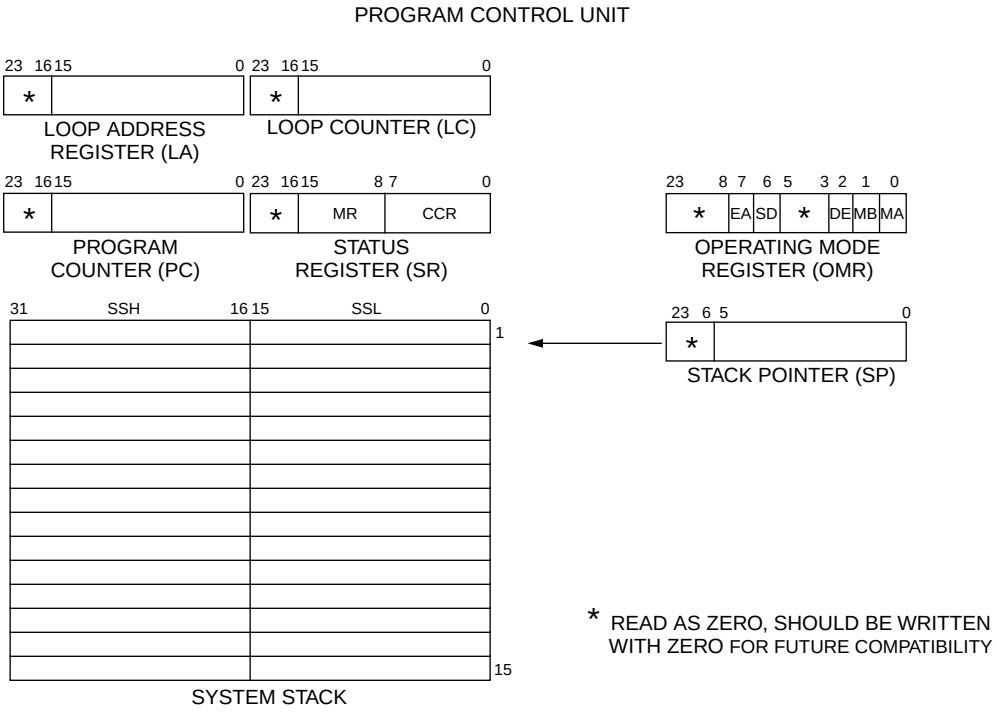


Figure 6-5 Program Control Unit Programming Model

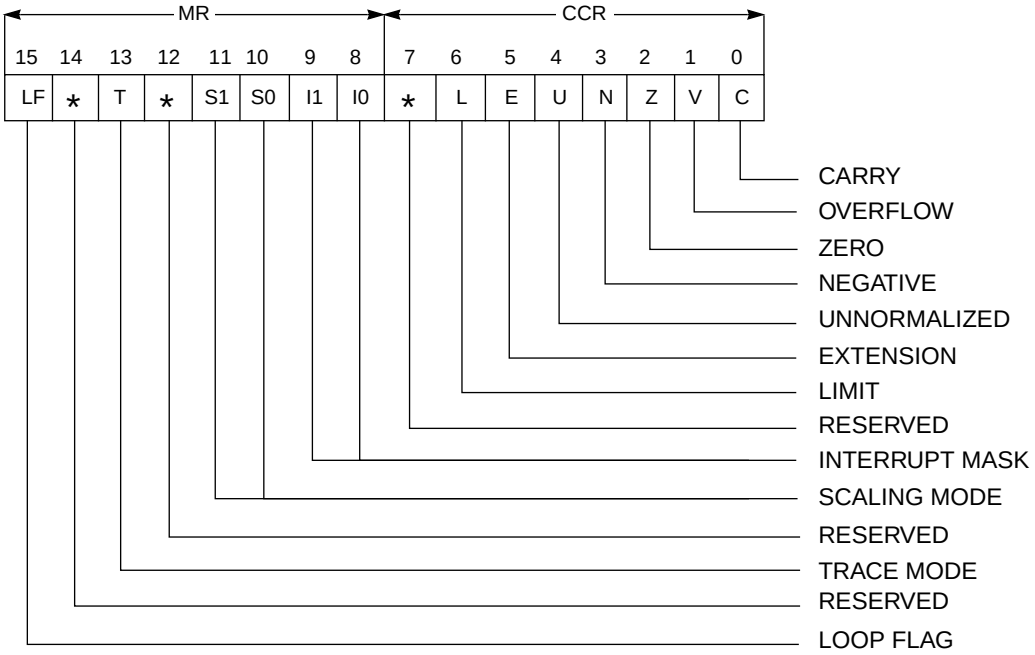
This special-purpose address register is stacked when program looping is initialized, when a JSR is performed, or when interrupts occur (except for no-overhead fast interrupts).

6.4.2 Status Register

The 16-bit SR consists of a mode register (MR) in the high-order eight bits and a condition code register (CCR) in the low-order eight bits. The SR is stacked when program looping is initialized, when a JSR is performed, or when interrupts occur, (except for no-overhead fast interrupts). The SR format is shown in Figure 6-6.

The MR is a special-purpose control register defining the current system state of the processor. The MR bits are affected by processor reset, exception processing, the DO, end current DO loop (ENDDO), return from interrupt (RTI), and SWI instructions and by instructions that directly reference the MR register =m OR immediate to control register (ORI) and AND immediate to control register (ANDI). During processor reset, the interrupt mask bits of the MR will be set; the scaling mode bits, loop flag, and trace bit will be cleared.

The CCR is a special-purpose control register that defines the current user state of the processor. The CCR bits are affected by data arithmetic logic unit (ALU) operations, parallel move operations, and by instructions that directly reference the CCR (ORI and ANDI). The CCR bits are not affected by parallel move operations unless data limiting occurs when reading the A or B accumulators. During processor reset, all CCR bits are



* Written as don't care; read as zero

Figure 6-6 Status Register Format

cleared.

6.4.2.1 Carry (Bit 0)

The carry (C) bit is set if a carry is generated out of the MSB of the result in an addition. This bit is also set if a borrow is generated in a subtraction. The carry or borrow is generated from bit 55 of the result. The carry bit is also affected by bit manipulation, rotate, and shift instructions. Otherwise, this bit is cleared.

6.4.2.2 Overflow (Bit 1)

The overflow (V) bit is set if an arithmetic overflow occurs in the 56-bit result. This bit indicates that the result cannot be represented in the accumulator register; thus, the register has overflowed. Otherwise, this bit is cleared.

6.4.2.3 Zero (Bit 2)

The zero (Z) bit is set if the result equals zero; otherwise, this bit is cleared.

6.4.2.4 Negative (Bit 3)

The negative (N) bit is set if the MSB (bit 55) of the result is set; otherwise, this bit is cleared.

6.4.2.5 Unnormalized (Bit 4)

The unnormalized (U) bit is set if the two MSBs of the most significant product (MSP) portion of the result are identical. Otherwise, this bit is cleared. The MSP portion of the A or B accumulators, which is defined by the scaling mode and the U bit, is computed as

follows:

S1	S0	Scaling Mode	U Bit Computation
0	0	No Scaling	U = $\overline{(\text{Bit 47} \oplus \text{Bit 46})}$
0	1	Scale Down	U = $\overline{(\text{Bit 48} \oplus \text{Bit 47})}$
1	0	Scale Up	U = $\overline{(\text{Bit 46} \oplus \text{Bit 45})}$

6.4.2.6 Extension (Bit 5)

The extension (E) bit is cleared if all the bits of the integer portion of the 56-bit result are all ones or all zeros; otherwise, this bit is set. The integer portion, defined by the scaling mode and the E bit, is computed as follows:

S1	S0	Scaling Mode	Integer Portion
0	0	No Scaling	Bits 55,54.....48,47
0	1	Scale Down	Bits 55,54.....49,48
1	0	Scale Up	Bits 55,54.....47,46

If the E bit is cleared, then the low-order fraction portion contains all the significant bits; the high-order integer portion is just sign extension. In this case, the accumulator extension register can be ignored. If the E bit is set, it indicates that the accumulator extension register is in use.

6.4.2.7 Limit (Bit 6)

The limit (L) bit is set if the overflow bit is set. The L bit is also set if the data shifter/limiter circuits perform a limiting operation; otherwise, it is not affected. The L bit is cleared only by a processor reset or by an instruction that specifically clears it, which allows the L bit to be used as a latching overflow bit (i.e., a “sticky” bit). L is affected by data movement operations that read the A or B accumulator registers.

6.4.2.8 Interrupt Masks (Bits 8 and 9)

The interrupt mask bits, I1 and I0, reflect the current IPL of the processor and indicate the IPL needed for an interrupt source to interrupt the processor. The current IPL of the processor can be changed under software control. The interrupt mask bits are set during hardware reset but not during software reset.

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

6.4.2.9 Scaling Mode (Bits 10 and 11)

The scaling mode bits, S1 and S0, specify the scaling to be performed in the data ALU shifter/limiter and the rounding position in the data ALU multiply-accumulator (MAC). The

scaling modes are shown in the following table:

S1	S0	Rounding Bit	Scaling Mode
0	0	23	No Scaling
0	1	24	Scale down (1-Bit Arithmetic Right Shift)
1	0	22	Scale Up (1-Bit Arithmetic Left Shift)
1	1	—	Reserved for Future Expansion

The shifter/limiter scaling mode affects data read from the A or B accumulator registers out to the XDB and YDB. Different scaling modes can be used with the same program code to allow dynamic scaling. One application of dynamic scaling is to facilitate block floating-point arithmetic. The scaling mode also affects the MAC rounding position to maintain proper rounding when different portions of the accumulator registers are read out to the XDB and YDB. The scaling mode bits, which are cleared at the start of a long interrupt service routine, are also cleared during a processor reset.

6.4.2.10 Trace Mode (Bit 13)

The trace mode (T) bit specifies the tracing function of the DSP. If the T bit is set at the beginning of any instruction execution, a trace exception will be generated after the instruction execution is completed. If the T bit is cleared, tracing is disabled and instruction execution proceeds normally. If a long interrupt is executed during a trace exception, the SR having the trace bit set will be stacked, and the trace bit in the SR is cleared (see **SECTION 8 PROCESSING STATES** for a complete description of a long interrupt operation). The T bit is also cleared during processor reset.

6.4.2.11 Reserved Status (Bits 7, 12, 14)

These bits, which are reserved for future expansion, will read as zero during DSP read operations.

6.4.2.12 Loop Flag (Bit 15)

The loop flag (LF) bit, set when a program loop is in progress, enables the detection of the end of a program loop. The LF 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, allow the nesting of program loops. At the start of a long interrupt service routine, the SR (including the LF) is pushed on the SS and the

SR LF is cleared. When returning from the long interrupt with an RTI instruction, the SS is pulled and the LF is restored. During a processor reset, the LF is cleared.

6.4.3 Operating Mode Register

The OMR is a 24-bit register (only five bits are defined) that sets the current operating mode of the processor (i.e., the memory maps for program and data memories as well as the start-up procedure). The OMR bits are only affected by processor reset and by instructions directly referencing the OMR: ANDI, ORI, and MOVEC. During processor reset, the chip operating mode bits, MB and MA, will be loaded from the external mode select pins B and A, respectively. The data ROM enable (DE) bit will be cleared, disabling the X and Y on-chip lookup-table ROMs. The OMR format is shown in Figure 6-7. Table 6-5 summarizes the DSP56000 operating modes and Table 6-3 summarizes the DSP56001 operating modes and their respective effects on the memory map. Table 6-4 shows how the DE bit in the OMR affects the X and Y memory maps.

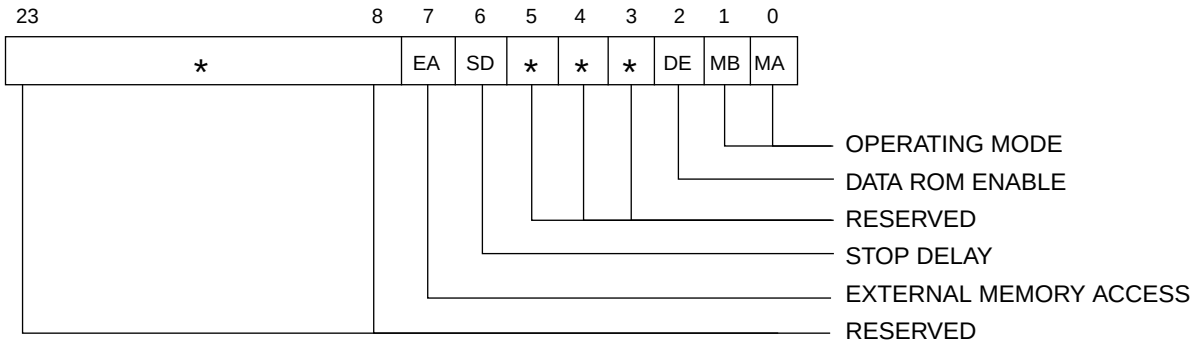


Figure 6-7 OMR Format

Table 6-5 DSP56000/DSP56001 Operating Mode Summary

Operating Mode	MB	MA	DSP56000 Program Memory Map		
			Internal ROM	External	Reset
0	0	0	$0000 — $0EFF	$0200 — $FFFF	Internal — $0000
1	0	1	Mode 1 is not a valid mode for the DSP56000. Attempting to put the DSP56000 in mode 1 will put it into mode 0.		
2	1	0	$0000 — $0EFF	$0FFF — $FFFF	External — $E000
3	1	1	—	$0000 — $FFFF	External — $0000

Table 6-4 DSP56000/56001 DE Memory Control

DE	Data Memory Map	
	Y Memory	X Memory

6.4.3.1 Chip Operating Mode (Bits 0 and 1)

The chip operating mode bits, MB and MA, indicate the bus expansion mode of the DSP56000/DSP56001. On processor reset, these bits are loaded from the external mode select pins, MODB and MODA, respectively. After the DSP leaves the reset state, MB and MA can be changed under program control. The “secure DSP56000” is an exception. The external mode select pins, MODB and MODA, are disabled on the “secure DSP56000” and are only used for interrupts as IRQA and IRQB. The operating modes are shown in the following table:

MB	MA	Chip Operating Mode
0	0	Single-Chip Nonexpanded
0	1	Special Bootstrap (DSP56001 Only)
1	0	Normal Expanded
1	1	Development Expanded

6.4.3.2 Data ROM Enable (Bit 2)

The DE bit enables the two, on-chip, 256;ts24 data ROMs located at addresses $0100–$01FF in the X and Y memory spaces. When DE is cleared, the $0100–$01FF address space is part of the external X and Y data spaces, and the on-chip data ROMs are dis-

abled.

6.4.3.3 Stop Delay (Bit 6)

The STOP instruction causes the DSP56000/DSP56001 to indefinitely suspend processing in the middle of the STOP instruction (see **SECTION 8 PROCESSING STATES**). When exiting the stop state, if the stop delay bit is zero, a 64K clock cycle delay (i.e., 131,072 T states) is selected before continuing the stop instruction cycle. However, if the stop delay bit is one, the delay before continuing the instruction cycle is 16 T states. The long delay allows a clock stabilization period for the internal clock to begin oscillating and to stabilize. When a stable external clock is used, the shorter delay allows faster start-up of the DSP.

6.4.3.4 External Memory Access (Bit 7)

The external memory access mode bit selects the function of two of the port A control pins. The DSP56000/DSP56001 comes out of reset with these pins defined as bus request/bus grant (BR/BG) =m i.e., bit 7 is cleared. When bit 7 is clear, wait states are only introduced into the port A timing by using the bus control register (BCR). Additional information on the BCR can be found in CHAPTER 10 PORT B. When bit 7 is set under program control (using ANDI, ORI, or MOVEC), these pins are defined as bus strobe (BS) and wait (WT). In this mode, wait states are introduced into port A timing by using either the BCR or asserting WT. BR and BG allow the DSP56000/DSP56001 to give the external bus to an external device, thus preventing bus conflicts. BS and WT allow the DSP56000/DSP56001 to work with asynchronous devices (bus arbitrators) on port A.

Table 6-6 DSP56001 Operating Mode Summary

Operating Mode	MB	MA	DSP56001 Program Memory Map		
			Internal RAM	External	Reset
0	0	0	$0000 — $01FF	$0200 — $FFFF	Internal — $0000
1	0	1	Special bootstrap mode; after program RAM loading, mode 2 is automatically selected but PC = $0000		
2	1	0	$0000 — $01FF	$0200 — $FFFF	External — $E000
3	1	1	—	$0000 — $FFFF	External — $0000

The definition of the control pins is summarized in the following table:

OMR Bit 7	$\overline{\text{BR}}$ Pin (Input)	$\overline{\text{BG}}$ Pin (Output)
0 (Default)	Bus Request ($\overline{\text{BR}}$)	Bus Grant ($\overline{\text{BG}}$)
1	Wait ($\overline{\text{WT}}$)	Bus Strobe ($\overline{\text{BS}}$)

6.4.3.5 Reserved OMR Bits (Bits 3–5 and 8–23)

These OMR bits, reserved for future expansion, will read as zero during DSP read operations.

6.4.4 Loop Address Register

The contents of the LA register indicate the location of the last instruction word in a program loop. This register is stacked into the SSH by a DO instruction and is unstacked by end-of-loop processing or by execution of an ENDDO instruction. When the instruction at the address contained in this register is fetched, the contents of the LC register are checked. If the contents are not one, the LC is decremented, and the next instruction is taken from the address at the top of the SS; otherwise, the PC is incremented, the loop flag is restored (pulled from the SS), the SS is purged, the LA and LC registers are pulled from the SS and restored, and instruction execution continues normally. The LA register, a read/write register, is written by a DO instruction and read by the SS when stacking the register. Since the LC register can be accessed under program control, the number of times a loop has been executed can be determined.

6.4.5 Loop Counter Register

The LC register is a special 16-bit counter used to specify the number of times a hardware program loop is to be repeated. This register is stacked into the SSL by a DO instruction and unstacked by end-of-loop processing or by execution of an ENDDO instruction. When the end of a hardware program loop is reached, the contents of the LC register are tested for one. If the LC is one, the program loop is terminated, and the LC register is loaded with the previous LC contents stored on the SS. If LC is not one, it is decremented and the program loop is repeated. The LC can be read under program control, which allows the number of times a loop will be executed to be monitored/changed dynamically. The LC is also used in the REP instruction.

6.4.6 System Stack

The SS is a separate 15x32-bit internal memory divided into two banks: SSH and SSL, each 16 bits wide. The SSH stores the PC contents, and the SSL stores the SR contents for subroutine calls and long interrupts. The SS will also store the LA and LC registers in addition to the PC and SR registers for program looping. The SS is in stack memory

space; its address is always inherent and implied by the current instruction.

The contents of the PC and SR register are pushed on the top location of the SS when a subroutine call or long interrupt occurs. When a return from subroutine (RTS) occurs, the contents of the top location in the SS are pulled and put in the PC; the SR is not affected. When an RTI occurs, the contents of the top location in the SS are pulled to both the PC and SR.

The SS is also used to implement no-overhead nested hardware DO loops. When the DO instruction is executed, the LA:LC are pushed on the SS, then the PC:SR are pushed on the SS. Since each SS location can be addressed as separate 16-bit registers (SSH and SSL), software stacks can be created for unlimited nesting.

Up to 15 long interrupts, seven DO loops, 15 JSRs, or combinations of these can be accommodated by the SS. When the SS limit is exceeded, a nonmaskable stack error interrupt occurs, and the PC is pushed to SS location zero, which is not implemented in hardware. The PC will be lost, and there will be no SP from the stack interrupt routine to the program that was executing when the error occurred.

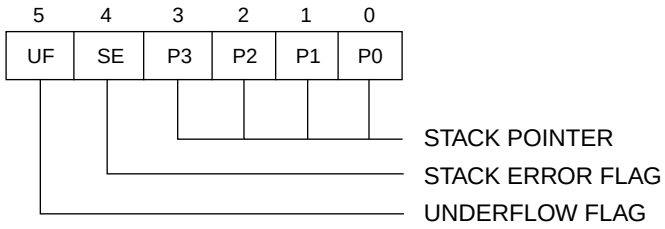


Figure 6-8 SP Register Format

6.4.7 Stack Pointer Register

The 6-bit SP register indicates the location of the top of the SS and the status of the SS (underflow, empty, full, and overflow). The SP register is referenced implicitly by some instructions (DO, REP, JSR, RTI, etc.) or directly by the MOVEC instruction. The SP register format, shown in Figure 6-8, is described in the following paragraphs. The SP register is implemented as a 6-bit counter that addresses (selects)a 15-location stack with its four LSBs. The possible SP values, shown in Figure 6-9, are described in the following paragraphs

6.4.7.1 Stack Pointer (Bits 0–3)

The SP points to the last used location on the SS. Immediately after hardware reset,

these bits are cleared (SP=0), indicating that the SS is empty.

Data is pushed onto the SS by incrementing the SP, then writing data to the location pointed to by the SP. An item is pulled off the stack by copying it from the location pointed to by the SP and then by decrementing SP.

6.4.7.2 Stack Error Flag (Bit 4)

The stack error flag indicates that a stack error has occurred, and the transition of the stack error flag from zero to one causes a priority level-3 stack error exception (see Section 6.4.7.1 for additional information).

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

Any implied pull operation with SP equal to zero will cause a stack error exception, and the SP will read 111111 (or 111110 if an implied double pull occurs). The stack error bit is set as shown in Figure 6-9.

The stack error flag is a “sticky bit” which, once set, remains set until cleared by the user. There is a sequence of instructions which can cause a stack overflow which, without the sticky bit, would not be detected because the stack pointer is decremented before the stack error interrupt is taken. The sticky bit keeps the stack error bit set until cleared by the user by writing a zero to SP bit 4. It also latches the overflow/underflow bit so that it cannot be changed by stack pointer increments or decrements as long as the stack error is set. The overflow/underflow bit remains latched until the first move to SP is executed.

Note: When SP is zero (stack empty), instructions that read the stack without SP postdecrement and instructions that write to the stack without SP preincrement do not cause

UF	SE	P3	P2	P1	P0	
1	1	1	1	1	0	← STACK UNDERFLOW CONDITION AFTER DOUBLE PULL
1	1	1	1	1	1	← STACK UNDERFLOW CONDITION
0	0	0	0	0	0	← STACK EMPTY (RESET); PULL CAUSES UNDERFLOW
0	0	0	0	0	1	← STACK LOCATION 1
		• • •				
		• • •				
		• • •				
0	0	1	1	1	0	← STACK LOCATION 14
0	0	1	1	1	1	← STACK LOCATION 15; PUSH CAUSES OVERFLOW
0	1	0	0	0	0	← STACK OVERFLOW CONDITION
0	1	0	0	0	1	← STACK OVERFLOW CONDITION AFTER DOUBLE PUSH

Figure 6-9 SP Register Values

a stack error exception (i.e., 1) DO SSL,xxxx 2) REP SSL 3) MOVEC or move peripheral data (MOVEP) when SSL is specified as a source or destination).

6.4.7.3 Underflow Flag (Bit 5)

The underflow flag is set when a stack underflow occurs. The stack underflow flag is a “sticky bit” when the stack error flag is set i.e., when the stack error flag is set, the underflow flag will not change state. The combination of “underflow=1” and “stack error=0” is an illegal combination and will not occur unless it is forced by the user. If this condition is forced by the user, the hardware will correct itself based on the result of the next stack operation. Also see the description for the stack error flag (Section 6.4.7.2) for additional information.

6.4.7.4 Reserved Stack Pointer Registration (Bits 6–23)

Any unimplemented SP register bits are reserved for future expansion and will read as zero during DSP56000/DSP56001 read operations.

6.4.8 DSP56000/DSP56001 Programming Model Summary

The complete programming model for the DSP56000/DSP56001 central processor is shown in Figure 6-10. SECTION 9 PORT A, SECTION 10 PORT B, and SECTION 11 PORT C describe in detail the programming model for the peripherals and external memory control (number of wait states).

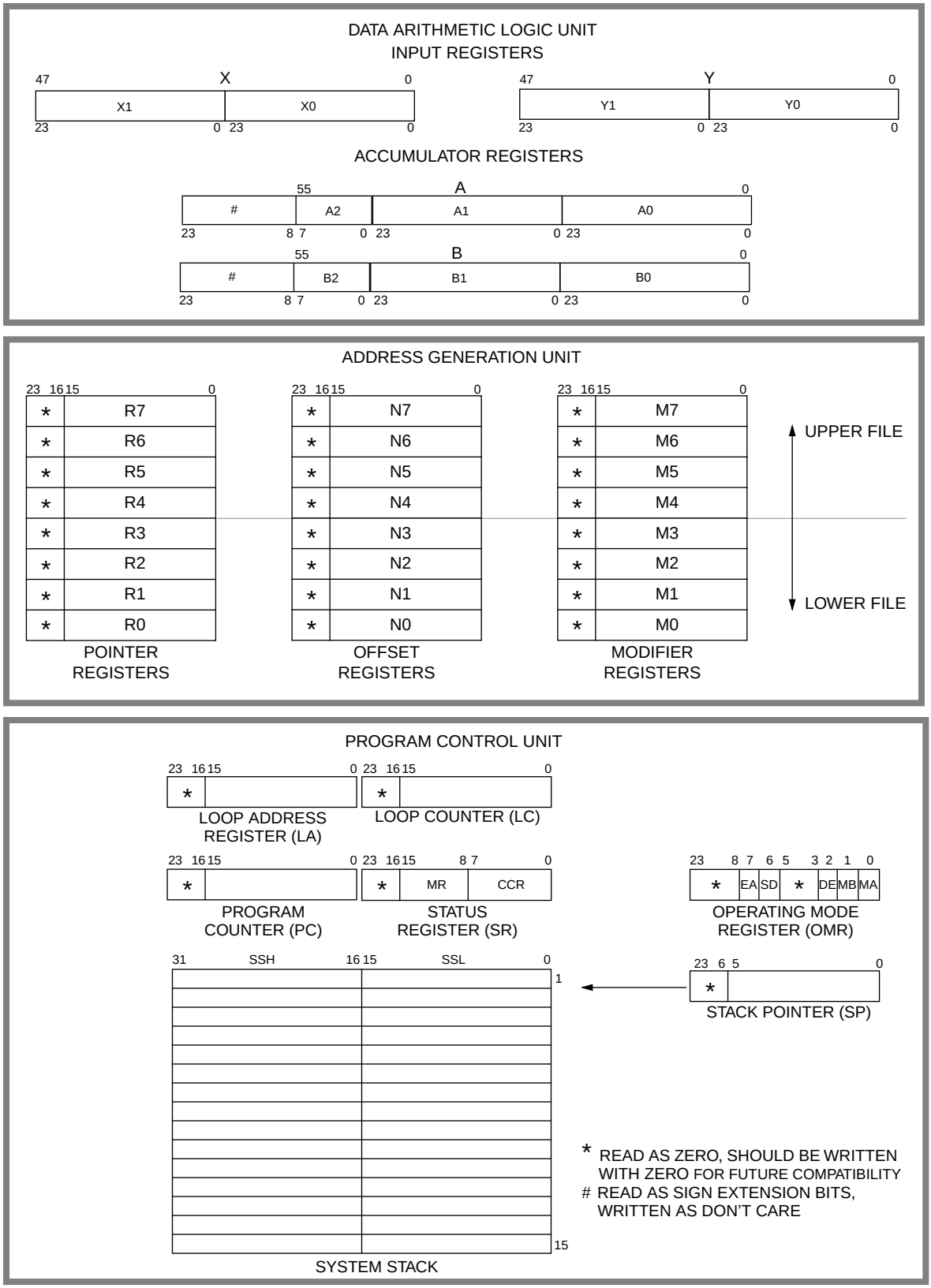


Figure 6-10 DSP56000/DSP56001 Central Processor Programming Model

SECTION 7
INSTRUCTION SET INTRODUCTION

The programming model indicates that the DSP56000/DSP56001 central processor architecture can be viewed as three functional units operating in parallel: data arithmetic logic unit (ALU), address generation unit (AGU), and program control unit (see Figure 7-1). The goal of the instruction set is to provide the capability to keep each of these units busy each instruction cycle, achieving maximum speed and minimum program size.

This section introduces the DSP56000/DSP56001 instruction set and instruction format. The complete range of instruction capabilities combined with the flexible addressing modes used in this processor provide a very powerful assembly language for implementing digital signal processing (DSP) algorithms. The instruction set has been designed to allow efficient coding for DSP high-level language compilers such as the C compiler. Execution time is minimized by the hardware looping capabilities, use of an instruction pipeline, and parallel moves.

7.1 SYNTAX

The instruction syntax is organized into four columns: opcode, operands, and two parallel-move fields. The assembly-language source code for a typical one-word instruction is shown in the following illustration. Because of the multiple bus structure and the parallelism of the DSP, up to three data transfers can be specified in the instruction word – one on the X data bus (XDB), one on the Y data bus (YDB), and one within the data ALU. These transfers are explicitly specified. A fourth data transfer is implied and occurs in the program control unit (instruction word prefetch, program looping control, etc.). Each data transfer involves a source and a destination.

Opcode	**Operands**	**XDB**	**YDB**
MAC	X0,Y0,A	X:(R0)+,X0	Y:(R4)+,Y0

The opcode column indicates the data ALU, AGU, or program control unit operation to be performed and must always be included in the source code. The operands column specifies the operands to be used by the opcode. The XDB and YDB columns specify optional

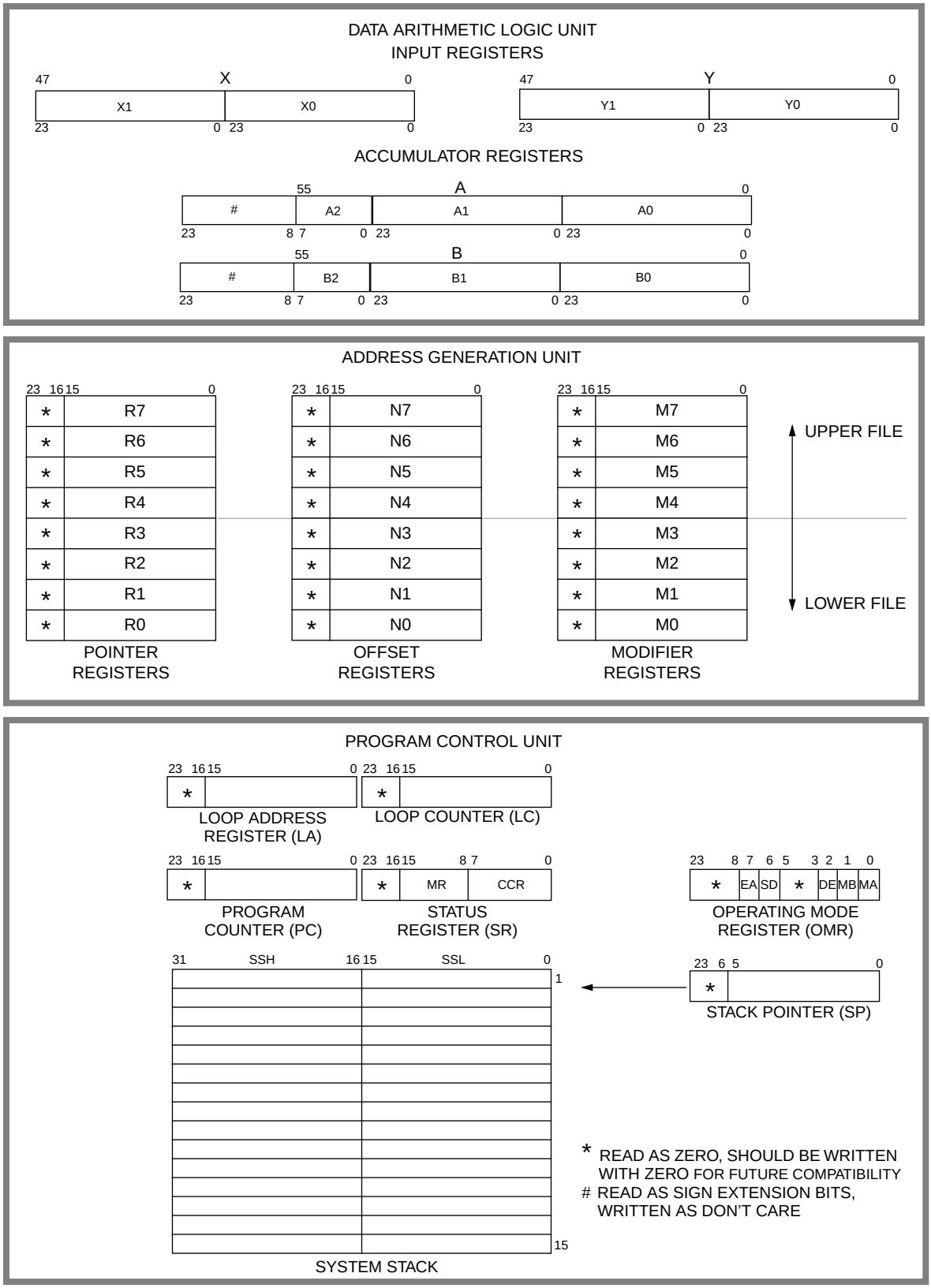


Figure 7-1 DSP56000/DSP56001 Central Processor Programming Model

data transfers over the XDB and/or YDB and the associated addressing modes. The

address space qualifiers (X:, Y:, and L:) indicate which address space is being referenced. Parallel moves are allowed in 30 of the 62 instructions. Additional information is presented in APPENDIX A INSTRUCTION SET DETAILS.

7.2 INSTRUCTION FORMATS

The DSP56000/DSP56001 instructions consist of one or two 24-bit words – an operation word and an optional effective address extension word. The general format of the operation word is shown in Figure 7-2. Most instructions specify data movement on the XDB, YDB, and data ALU operations in the same operation word. The DSP is designed to perform each of these operations in parallel.

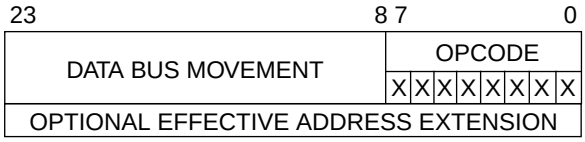


Figure 7-2 General Format of an Instruction Operation Word

The data bus movement field provides the operand reference type, which selects the type of memory or register reference to be made, the direction of transfer, and the effective address(es) for data movement on the XDB and YDB. This field may require additional information to fully specify the operand for certain addressing modes. An effective address extension word following the operation word is used to provide immediate data or an absolute address if required. Examples of operations that may include the extension word include the move operations X:, X:R, Y:, R:Y, and L:. Additional information is presented in APPENDIX A INSTRUCTION SET DETAILS.

The opcode field of the operation word specifies the data ALU operation or the program control unit operation to be performed and any additional operands required by the instruction. Only those data ALU and program control unit operations that can accompany data bus movement will be specified in the opcode field of the instruction. Other data ALU, program control unit operations, and all address ALU operations will be specified in an instruction word with a different format. These formats include operation words containing short immediate data or short absolute addresses.

Encoding the 30 opcodes that allow up to two parallel data moves into 24 bits has used all of the available bits and precluded adding more instructions or instruction variations. The available operation codes form a very versatile microcontroller unit (MCU) style instruction set, providing highly parallel operations in most programming situations.

7.2.1 Operand Sizes

Operand sizes are defined as follows: a byte is 8 bits long, a short word is 16 bits long, a word is 24 bits long, a long word is 48 bits long, and an accumulator is 56 bits long (see Figure 7-3). The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Implicit instructions support some subset of these five sizes.

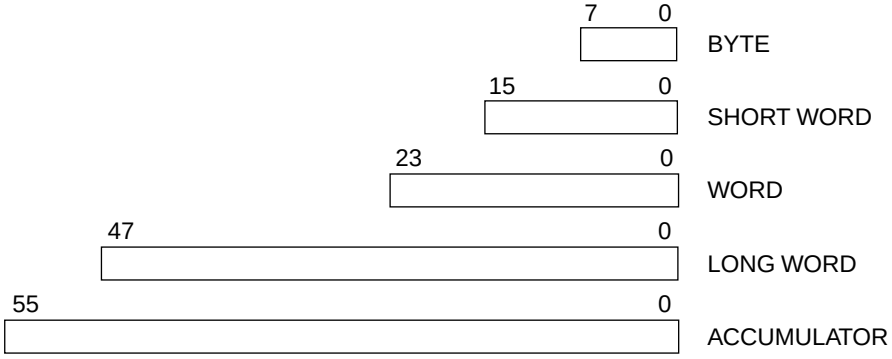


Figure 7-3 Operand Sizes

7.2.2 Data Organization in Registers

The ten data ALU registers support 8- or 24-bit data operands. Instructions also support 48- or 56-bit data operands by concatenating groups of specific data ALU registers. The eight address registers in the AGU support 16-bit address or data operands. The eight AGU offset registers support 16-bit offsets or may support 16-bit address or data operands. The eight AGU modifier registers support 16-bit modifiers or may support 16-bit address or data operands. The program counter (PC) supports 16-bit address operands. The status register (SR) and operating mode register (OMR) support 8- or 16-bit data operands. Both the loop counter (LC) and loop address (LA) registers support 16-bit address operands.

7.2.2.1 Data ALU Registers

The eight main data registers are 24 bits wide. Word operands occupy one register; long-word operands occupy two concatenated registers. The least significant bit (LSB) is the right-most bit (bit 0); whereas, the most significant bit (MSB) is the left-most bit (bit 23 for word operands and bit 47 for long-word operands). The two accumulator extension registers are eight bits wide. When an accumulator extension register is used as a source operand, it occupies the low-order portion (bits 0–7) of the word; the high-order portion (bits 8–23) is sign extended (see Figure 7-4). When used as a destination operand, this register receives the low-order portion of the word, and the high-order portion is not used. Accumulator operands occupy an entire group of three registers (i.e., A2:A1:A0 or B2:B1:B0). The LSB is the right-most bit (bit 0), and the MSB is the left-most bit (bit 55).

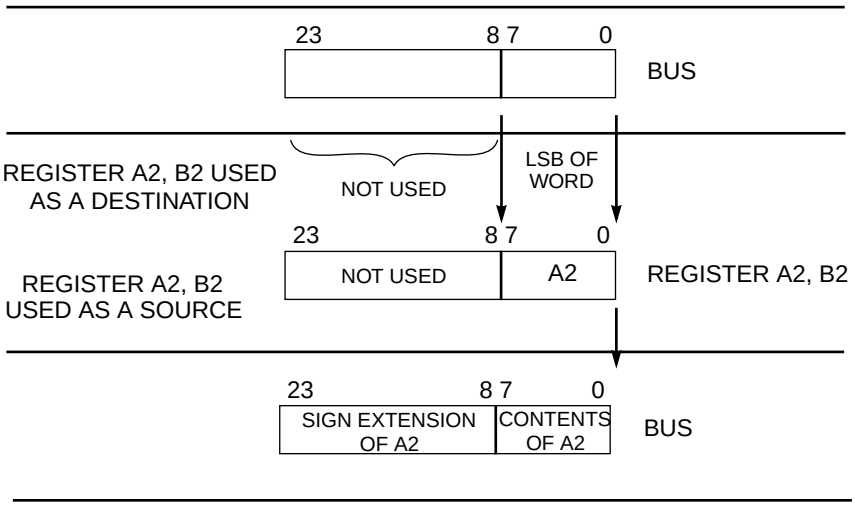


Figure 7-4 Reading and Writing the ALU Extension Registers

7.2.2.2 AGU Registers

The 24 AGU registers, which are 16 bits wide, may be accessed as word operands for address, address modifier, and data storage. When used as a source operand, these registers occupy the low-order portion of the 24-bit word; the high-order portion is read as zeros (see Figure 7-5). When used as a destination operand, these registers receive the low-order portion of the word; the high-order portion is not used. The notation Rn is used to designate one of the eight address registers, R0–R7; the notation Nn is used to designate one of the eight address offset registers, N0–N7; and the notation Mn is used to designate one of the eight address modifier registers, M0–M7.

7.2.2.3 Program Control Registers

The 8-bit OMR may be accessed as a word operand; however, not all eight bits are defined. In general, undefined bits are written as ‘‘don’t care’’ and read as zero. The 16-bit SR has the system mode register (MR) occupying the high-order eight bits and the user condition code register (CCR) occupying the low-order eight bits. The SR may be accessed as a word operand. The MR and CCR may be accessed individually as word operands (see Figure 7-6(b)). The LC, LA, system stack high (SSH), and system stack low (SSL) registers are 16 bits wide and may be accessed as word operands (see Figure 7-6(a)). When used as a source operand, these registers occupy the low-order portion of the 24-bit word; the high-order portion is zero. When used as a destination operand, they receive the low-order portion of the 24-bit word; the high-order portion is not used. The system stack pointer (SP) is a 6-bit register that may be accessed as a word operand

.The PC, a special 16-bit-wide program control register, is always referenced implicitly as

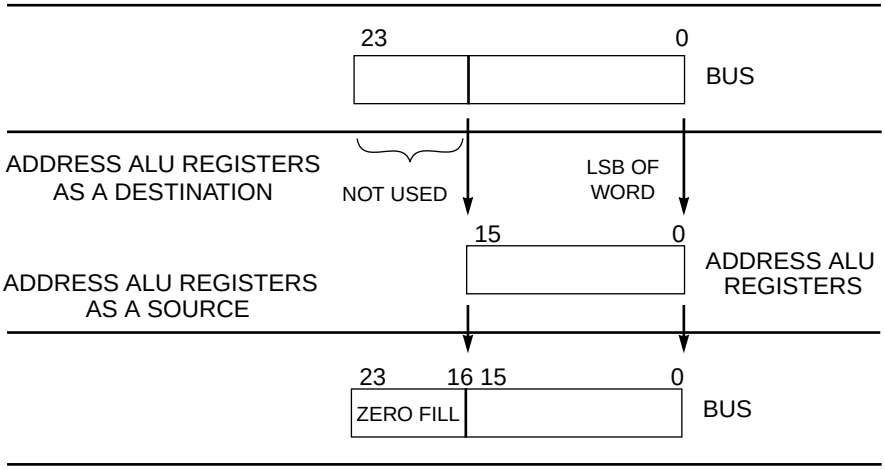


Figure 7-5 Reading and Writing the Address ALU Registers

a short-word operand.

7.2.3 Data Organization in Memory

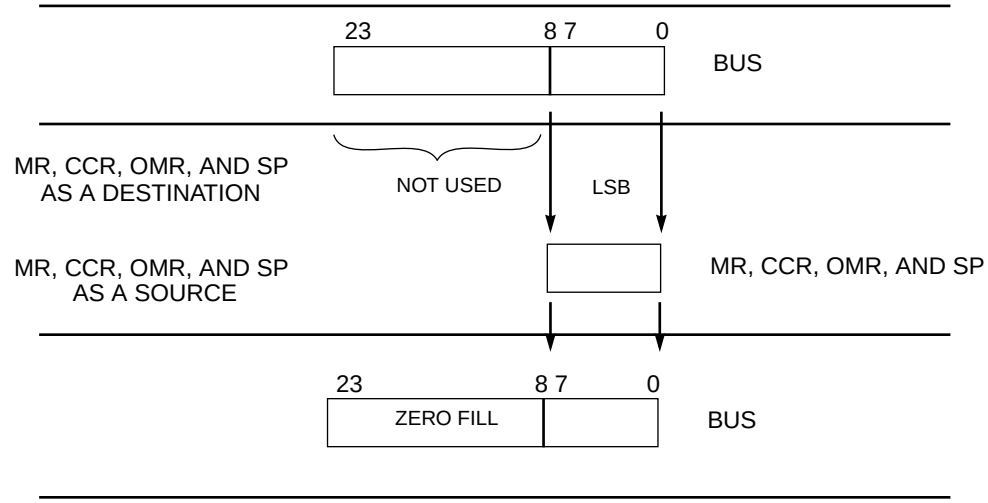
The 24-bit program memory can store both 24-bit instruction words and instruction extension words. The 32-bit system stack (SS) can store the concatenated PC and SR registers (PC:SR) for subroutine calls, interrupts, and program looping. The SS also supports the concatenated LA and LC registers (LA:LC) for program looping. The 24-bit-wide X and Y memories can store word, short-word, and byte operands. Short-word and byte operands, which usually occupy the low-order portion of the X or Y memory word, are either zero extended or sign extended on the XDB or YDB.

The symbols used to abbreviate the various operands and operations in each instruction and their respective meanings are shown in the following list:

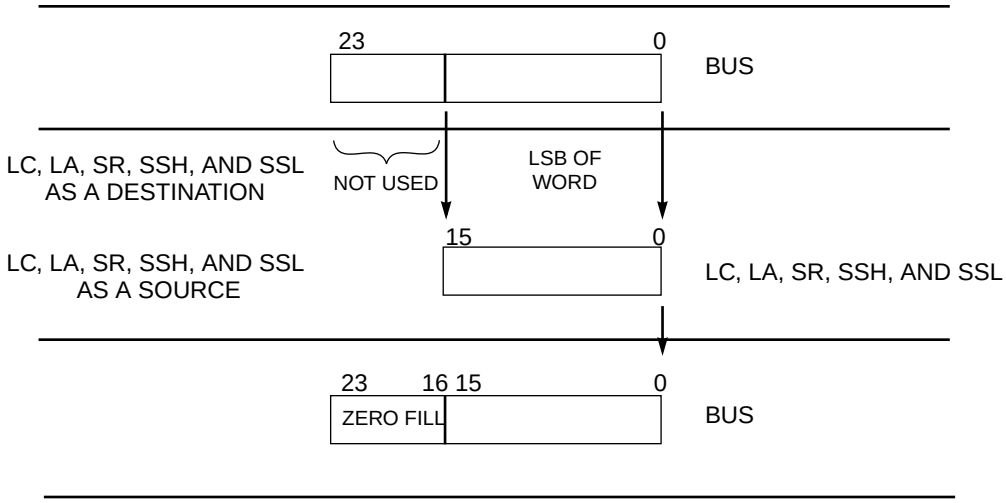
Data ALU

Xn	Input Registers X1, X0 (24 Bits)
Yn	Input Registers Y1, Y0 (24 Bits)
An	Accumulator Registers A2 (8 Bits), A1, A0 (24 Bits)
Bn	Accumulator Registers B2 (8 Bits), B1, B0 (24 Bits)
X	Input Register X (X1:X0, 48 Bits)
Y	Input Register Y (Y1:Y0, 48 Bits)
A	Accumulator A (A2:A1:A0, 56 Bits)*

*Data Move Operations: when specified as a source operand, shifting and limiting are performed. When specified as a destination operand, sign extension and zero filling are performed.



(a) 16 Bit



(b) 8 Bit

Figure 7-6 Reading and Writing Control Registers

B Accumulator B (B2:B1:B0, 56 Bits)

AB	Accumulators A and B (A1:B1, 48 Bits)*
BA	Accumulators B and A (B1:A1, 48 Bits)*
A10	Accumulator A (A1:A0, 48 Bits)
B10	Accumulator B (B1:B0, 48 Bits)

Address ALU

Rn	Address Registers R0–R7 (16 Bits)
Nn	Address Offset Registers N0–N7 (16 Bits)
Mn	Address Modifier Registers M0–M7 (16 Bits)

Program Control Unit

PC	Program Counter (16 Bits)
MR	Mode Register (8 Bits)
CCR	Condition Code Register (8 Bits)
SR	Status Register (MR:CCR, 16 Bits)
OMR	Operating Mode Register (8 Bits)
LA	Hardware Loop Address Register (16 Bits)
LC	Hardware Loop Counter (16 Bits)
SP	System Stack Pointer (6 Bits)
SS	System Stack RAM (15X32 Bits)
SSH	Upper 16 Bits of the Contents of the Current Top of Stack
SSL	Lower 16 Bits of the Contents of the Current Top of Stack

Addresses

ea	Effective Address
xxxx	Absolute Address (16 Bits)
xxx	Short Jump Address (12 Bits)
aa	Absolute Short Address (6 Bits Zero Extended)
pp	I/O Short Address (6 Bits Ones Extended)
< . . . >	Contents of the Specified Address
X:	X Memory Reference
Y:	Y Memory Reference
L:	Long Memory Reference – X Concatenated with Y
P:	Program Memory Reference

Miscellaneous

#xx	Immediate Short Data (8 Bits)
#xxx	Immediate Short Data (12 Bits)
#xxxxxx	Immediate Data (24 Bits)
#n	Immediate Short Data (5 Bits)
S,Sn	Source Operand Register
D,Dn	Destination Operand Register

D[n]	Bit n of D Affected
r	Rounding Constant
I1,I0	Interrupt Priority Level in SR
LF	Loop Flag in SR

7.2.4 Operand References

The DSP separates operand references into four classes: program, stack, register, and memory references. The type of operand reference(s) required for an instruction is specified by both the opcode field and the data bus movement field of the instruction; however, all operand reference types may not be used with all instructions. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Implicit instructions support some subset of the five operand sizes.

7.2.4.1 Program References

Program (P) references, which are references to 24-bit-wide program memory space, are usually instruction reads. Instructions or data operands may be read from or written to program memory space using the move program memory (MOVEM) and move peripheral data (MOVEP) instructions. Depending on the address and the chip operating mode, program references may be internal or external memory references.

7.2.4.2 Stack References

Stack (S) references, which are references to a separate32-bit-wide internal memory space (SS), are used implicitly to store the PC and SR for subroutine calls, interrupts, and returns. In addition to the PC and SR, the LA and LC registers are stored on the stack when a program loop is initiated. S references are always implied by the instruction. Data is written to the stack memory to save the processor state and is read from the stack memory to restore the processor state. In contrast to S references, references to SSL and SSH are always explicit.

7.2.4.3 Register References

Register (R) references are references to the data ALU, AGU, and program control unit registers. Data can be read from one register and written into another register.

7.2.4.4 Memory References

Memory references, which are references to the 24-bit-wide X or Y memory spaces, can be internal or external memory references, depending on the effective address of the operand in the data bus movement field of the instruction. Data can be read or written from any address in either memory space.

7.2.4.4.1 X Memory References

The operand, which is in X memory space, is a word reference. Data can be transferred from memory to a register or from a register to memory.

7.2.4.4.2 Y Memory References

The operand, a word reference, is in Y memory space. Data can be transferred from memory to a register or from a register to memory.

7.2.4.4.3 L Memory References

Long (L) memory space references both X and Y memory spaces with one operand address. The data operand is a long-word reference developed by concatenating the X and Y memory spaces (X:Y). The high-order word of the operand is in the X memory; the low-order word of the operand is in the Y memory. Data can be read from memory to concatenated registers X1:X0, A1:A0, etc. or from concatenated registers to memory.

7.2.4.4.4 YX Memory References

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

7.2.5 Addressing Modes

The DSP instruction set contains a full set of operand addressing modes. To minimize execution time and loop overhead, all address calculations are performed concurrently in the address ALU.

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

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

The addressing modes are grouped into three categories: register direct, address register indirect, and special. These addressing modes are described in the following paragraphs. Refer to Table 7-1 for a summary of the addressing modes and allowed operand references.

7.2.5.1 Register Direct Modes

These effective addressing modes specify that the operand source or destination is one of the data, control, or address registers in the programming model.

7.2.5.1.1 Data or Control Register Direct

The operand is in one, two, or three data ALU register(s) as specified in a portion of the data bus movement field in the instruction. Classified as a register reference, this addressing mode is also used to specify a control register operand for special instructions such as OR immediate to control registers (ORI) and AND immediate to control registers (ANDI).

7.2.5.1.2 Address Register Direct

Classified as a register reference, the operand is in one of the 24 address registers (Rn, Nn, or Mn) specified by an effective address in the instruction.

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

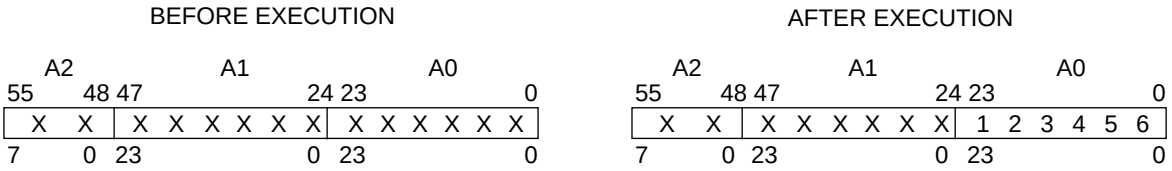
7.2.5.2 ADDRESS REGISTER INDIRECT MODES

The address register indirect mode description is presented in SECTION 5 ADDRESS GENERATION UNIT AND ADDRESSING MODES.

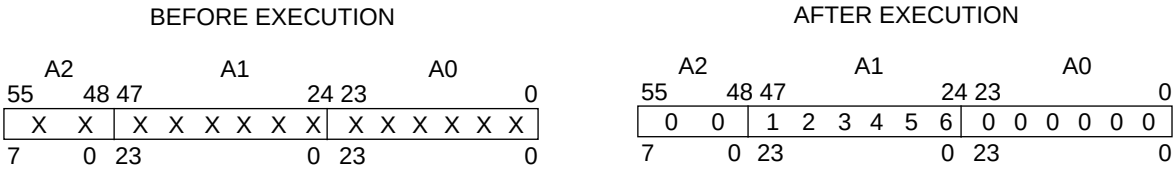
7.2.5.3 SPECIAL ADDRESSING MODES

The special addressing modes do not use specific registers in specifying an effective address. These modes specify the operand or the operand address in a field of the instruction, or they implicitly reference an operand. Figure examples are given for each of

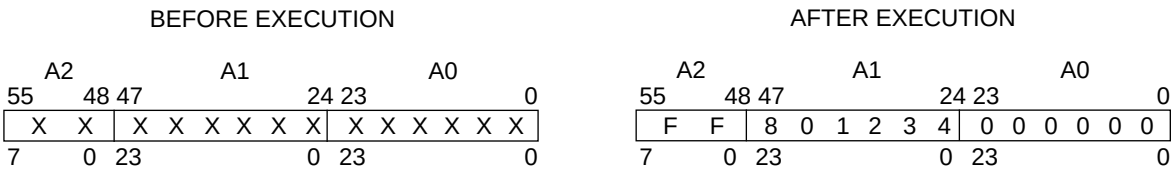
EXAMPLE A: IMMEDIATE INTO 24-BIT REGISTER
(MOVE #$123456,A0)



EXAMPLE B:POSITIVE IMMEDIATE INTO 56-BIT REGISTER
(MOVE #$123456,A)



EXAMPLE C: NEGATIVE IMMEDIATE INTO 56-BIT REGISTER
(MOVE #$801234,A)



Assembler Syntax: #XXXXXX
Memory Spaces: P:
Additional Instruction Execution Time (Clocks): 2
Additional Effective Address Words: 1

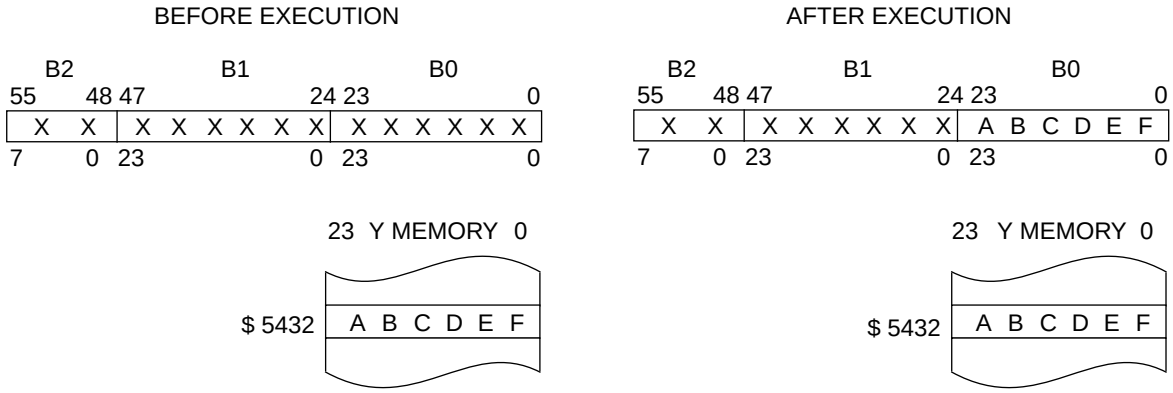
Figure 7-7 Special Addressing – Immediate Data

the special addressing modes discussed in the following paragraphs.

7.2.5.3.1 Immediate Data

Classified as a program reference, this addressing mode requires one word of instruction extension containing the immediate data. Figure 7-7 shows three examples. Example A moves immediate data to register A0 without affecting A1 or A2. Examples B and C zero fill register A0 and sign extend register A2

EXAMPLE: MOVE Y:$5432,B0



Assembler Syntax: #XXXXXX
Memory Spaces: P:
Additional Instruction Execution Time (Clocks): 2
Additional Effective Address Words: 1

Figure 7-8 Special Addressing – Absolute Addressing

7.2.5.3.2 Absolute Address

This addressing mode requires one word of instruction extension containing the absolute address. Figure 7-8 shows that MOVE Y:$5432,B0 copies the contents of address $5432 into B0 without changing memory location $5432, register B1, or register B2. This addressing mode is classified as both a memory reference and program reference. The 16-bit absolute address is stored in the 16 LSBs of the extension word; the eight MSBs are zero filled.

7.2.5.3.3 Immediate Short

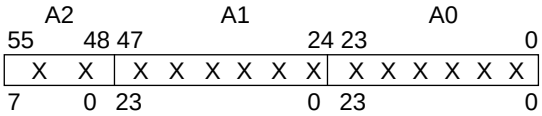
The 8- or 12-bit operand, which is in the instruction operation word, is classified as a program reference. The immediate data is interpreted as an unsigned integer (low-order portion) or signed fraction (high-order portion), depending on the destination register. Figure 7-9 shows the use of immediate short addressing in four examples.

7.2.5.3.4 Short Jump Address

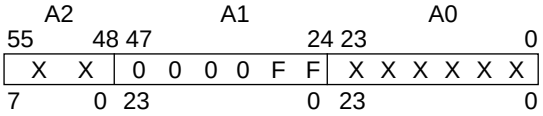
The operand occupies 12 bits in the instruction operation word, which allows addresses $0000–$0FFF to be accessed (see Figure 7-10). The address is zero extended to 16 bits when used to address program memory. This addressing mode is classified as a program reference.

EXAMPLE A: IMMEDIATE SHORT INTO A0, A1, A2, B0, B1, B2, Rn, Nn
(MOVE #$FF,A1)

BEFORE EXECUTION

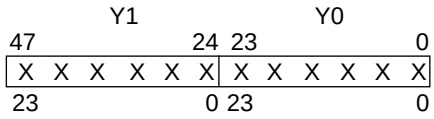


AFTER EXECUTION

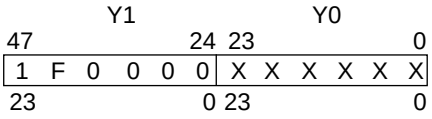


EXAMPLE B:POSITIVE IMMEDIATE SHORT INTO X0, X1, Y0, Y1, A, B
(MOVE #$1F, Y1)

BEFORE EXECUTION

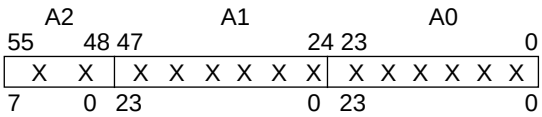


AFTER EXECUTION

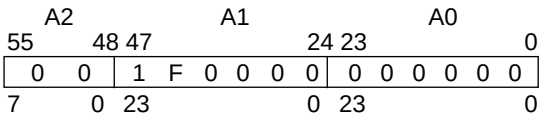


EXAMPLE C: POSITIVE IMMEDIATE SHORT INTO X, Y, A, B
(MOVE #$1F, A)

BEFORE EXECUTION

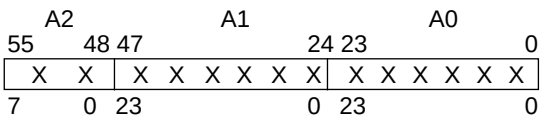


AFTER EXECUTION

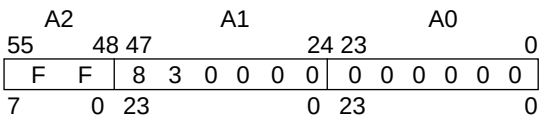


EXAMPLE D: NEGATIVE IMMEDIATE INTO 56-BIT REGISTER
(MOVE #$801234,A)

BEFORE EXECUTION



AFTER EXECUTION

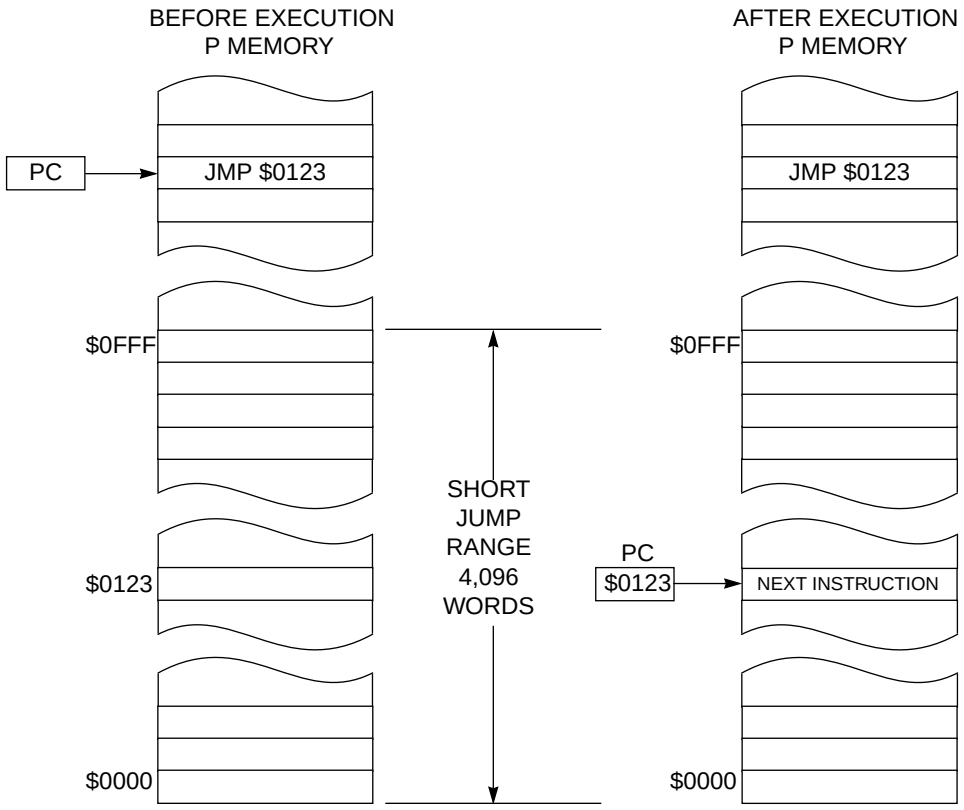


Assembler Syntax: #XX
Memory Spaces: P:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 7-9 Special Addressing – Immediate Short Data

7.2.5.3.5 Absolute Short

EXAMPLE: JMP $123



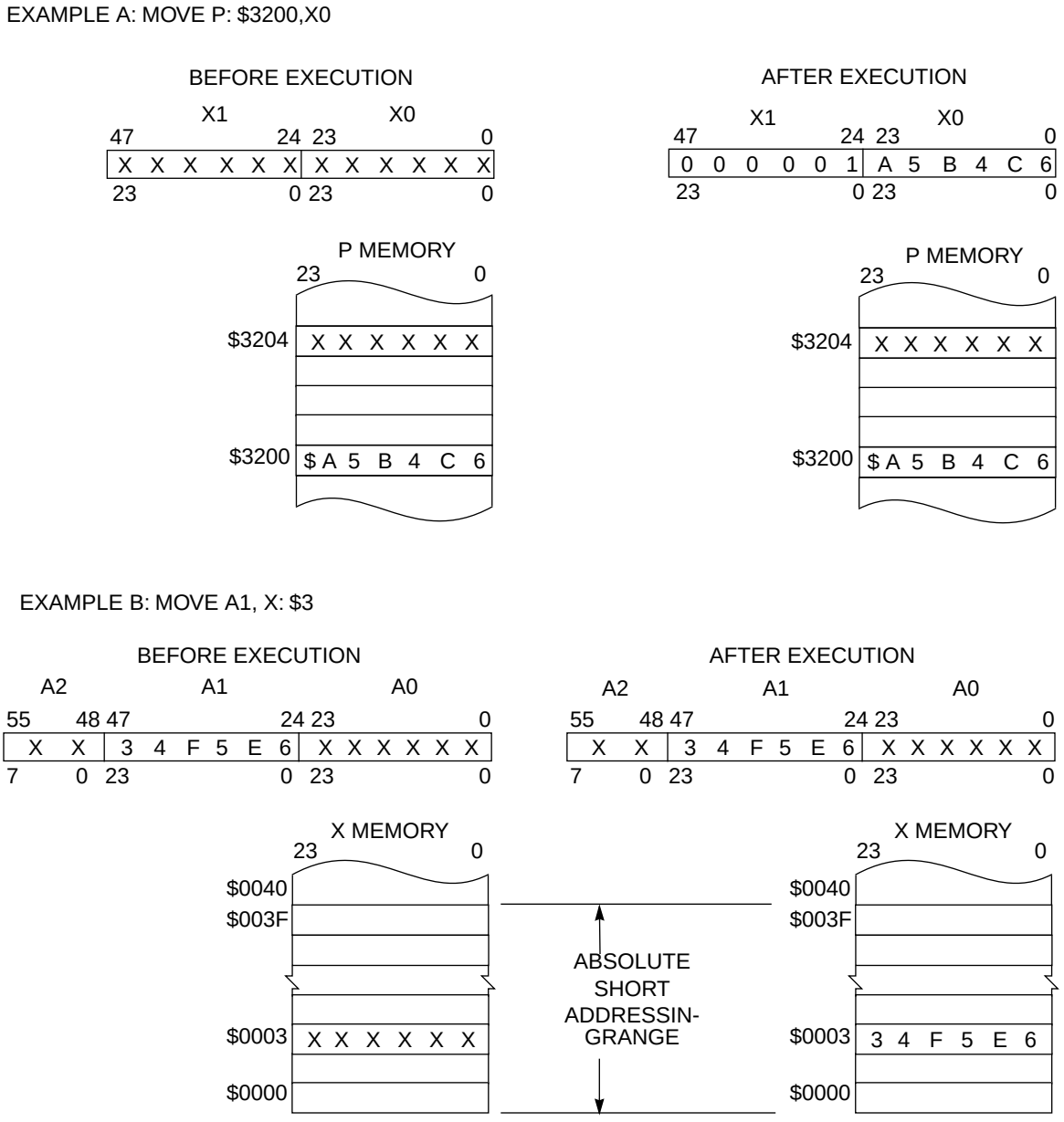
Assembler Syntax: XXX
Memory Spaces: P:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 7-10 Special Addressing – Short Jump Address

The address of the operand occupies six bits in the instruction operation word, allowing addresses $0000–$003F to be accessed (see Figure 7-11). Classified as both a memory reference and program reference, the address is zero extended to 16 bits when used to address an operand or program memory.

7.2.5.3.6 I/O Short

Classified as a memory reference, the I/O short addressing mode is similar to absolute short addressing. The address of the operand occupies six bits in the instruction operation word. I/O short is used with the bit manipulation and MOVEP instructions. The I/O short address is ones extended to 16 bits to address the I/O portion of X and Y memory (addresses $FFC0–$FFFF – see Figure 7-12).



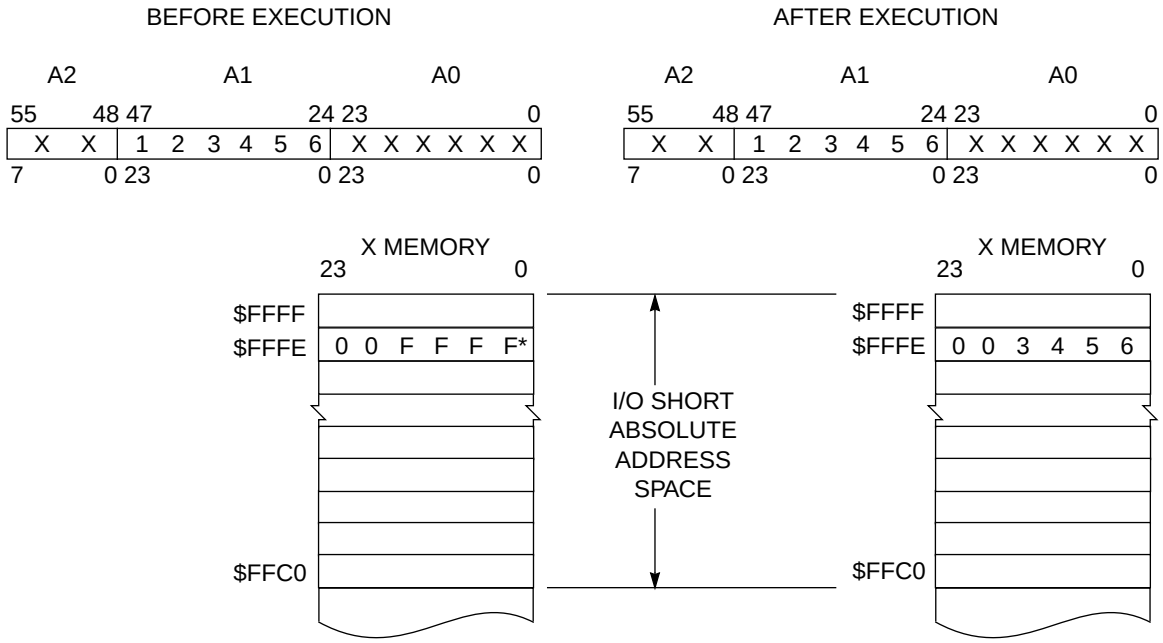
Assembler Syntax: aa
Memory Spaces: P:, X:, Y:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 7-11 Special Addressing – Absolute Short Address

7.2.5.3.7 Implicit Reference

Some instructions make implicit reference to PC, SS, LA, LC, or SR. For example, the jump instruction (JMP) implicitly references the PC; whereas, the repeat next instruction (REP) implicitly references LC. The registers implied and their uses are defined by the

EXAMPLE: MOVEP A1, X:<<$FFFE



*Contents of Bus Control Register (X:$FFFE) After Reset

Assembler Syntax: pp
Operands Referenced: X:, Y Memories
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 7-12 Special Addressing – I/O Short Address

individual instruction descriptions (see APPENDIX A INSTRUCTION SET DETAILS).

7.2.5.4 Addressing Modes Summary. Table 7-1 is a summary of the addressing modes discussed in the previous paragraphs.

Table 7-1 Addressing Modes Summary

Addressing Mode	Modifier MMMM	Operand Reference								
		P	S	C	D	A	X	Y	L	XY
Register Direct										
Data or Control Register	No			X	X					
Address Register	No					X				
Address Modifier Register	No					X				
Address Offset Register	No					X				
Address Register Indirect										
No Update	No	X					X	X	X	X
Postincrement by 1	Yes	X					X	X	X	X
Postdecrement by 1	Yes	X					X	X	X	X
Postincrement by Offset Nn	Yes	X					X	X	X	X
Postdecrement by Offset Nn	Yes	X					X	X	X	
Indexed by Offset Nn	Yes	X					X	X	X	
Predecrement by 1	Yes	X					X	X	X	
Special										
Immediate Data	No	X								
Absolute Address	No	X					X	X	X	
Immediate Short Data	No	X								
Short Jump Address	No	X								
Absolute Short Address	No	X					X	X	X	
I/0 Short Address	No						X	X		
Implicit	No	X	X	X						

Where: MMMM = Address Modifier

P = Program Reference
S = Stack Reference
C = Program Control Unit Register Reference
D = Data ALU Register Reference
A = AGU Register Reference
X = X Memory Reference
Y = Y Memory Reference
L = L Memory Reference
XY = XY Memory Reference

7.3 INSTRUCTION GROUPS

The instruction set is divided into the following groups:

Arithmetic
Logical
Bit Manipulation
Loop
Move
Program Control

Each instruction group is described in the following paragraphs; detailed information on each instruction is given in APPENDIX A INSTRUCTION SET DETAILS.

7.3.1 Arithmetic Instructions

The arithmetic instructions, which perform all of the arithmetic operations within the data ALU, execute in one instruction cycle. These instructions may affect all of the CCR bits. Arithmetic instructions are register based (register direct addressing modes used for operands) so that the data ALU operation indicated by the instruction does not use the XDB, the YDB, or the global data bus (GDB). Optional data transfers may be specified with most arithmetic instructions, which allows for parallel data movement over the XDB and YDB or over the GDB during a data ALU operation. This parallel movement allows new data to be prefetched for use in subsequent instructions and allows results calculated in previous instructions to be stored. The following list contains the arithmetic instructions:

ABS	Absolute Value
ADC	Add Long with Carry
ADD	Addition
ADDL	Shift Left and Add
ADDR	Shift Right and Add
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
CLR	Clear an Operand
CMP	Compare
CMPM	Compare Magnitude
DIV*	Divide Iteration
MAC	Signed Multiply-Accumulate
MACR	Signed Multiply-Accumulate and Round
MPY	Signed Multiply
MPYR	Signed Multiply and Round
NEG	Negate Accumulator
NORM*	Normalize
RND	Round
SBC	Subtract Long with Carry
SUB	Subtract
SUBL	Shift Left and Subtract
SUBR	Shift Right and Subtract
Tcc*	Transfer Conditionally
TFR	Transfer Data ALU Register
TST	Test an Operand

7.3.2 Logical Instructions

The logical instructions, which execute in one instruction cycle, perform all of the logical

*These instructions do not allow parallel data moves.

operations within the data ALU (except ANDI and ORI). They may affect all of the CCR bits and, like the arithmetic instructions, are register based. Optional data transfers may be specified with most logical instructions, allowing parallel data movement over the XDB and YDB or over the GDB during a data ALU operation. This parallel movement allows new data to be prefetched for use in subsequent instructions and allows results calculated in previous instructions to be stored. The following list includes the logical instructions:

AND	Logical AND
ANDI*	AND Immediate to Control Register
EOR	Logical Exclusive OR
LSL	Logical Shift Left
LSR	Logical Shift Right
NOT	Logical Complement
OR	Logical Inclusive OR
ORI*	OR Immediate to Control Register
ROL	Rotate Left
ROR	Rotate Right

7.3.3 Bit Manipulation Instructions

The bit manipulation instructions test the state of any single bit in a memory location and then optionally set, clear, or invert the bit. The carry bit of the CCR will contain the result of the bit test. The following list defines the bit manipulation instructions:

BCLR	Bit Test and Clear
BSET	Bit Test and Set
BCHG	Bit Test and Change
BTST	Bit Test on Memory and Registers

7.3.4 Loop Instructions

The hardware DO loop executes with no overhead cycles – i.e., it runs as fast as straight-line code. Replacing straight-line code with DO loops can significantly reduce program memory. The loop instructions control hardware looping by 1) initiating a program loop and establishing looping parameters or by 2) restoring the registers by pulling the SS when terminating a loop. Initialization includes saving registers used by a program loop (LA and LC) on the SS so that program loops can be nested. The address of the first instruction in a program loop is also saved to allow no-overhead looping. The loop instructions are as follows:

DO	Start Hardware Loop

*These instructions do not allow parallel data moves.

START OF LOOP

1)SP+1 ⇨ SP; LA ⇨ SSH; LC ⇨ SSL; #xxx ⇨ LC
2)SP+1 ⇨ SP; PC ⇨ SSH; SR ⇨ SSL; Expr–1 ⇨ LA
3)1 ⇨ LF

END OF LOOP

1)SSL(LF) ⇨ SR
2)SP–1 ⇨ SP; SSH ⇨ LA; SSL ⇨ LC; SP–1 ⇨ SP
3)PC + 1 ⇨ PC

NOTE:
#xxx=Loop Count Number
Expr=Expression

Figure 7-13 Hardware DO Loop

ENDDO Exit from Hardware Loop

Both static and dynamic loop counts are supported in the following forms:

```
DO      #xxx,Expr       ; (Static)
DO      S,Expr          ; (Dynamic)
```

Expr is an assembler expression or absolute address, and S is a directly addressable register such as X0.

The operation of a DO loop is shown in Figure 7-13. When a program loop is initiated with the execution of a DO instruction, the following events occur:

1. The stack is pushed.
   A. The SP is incremented.
   B. The current 16-bit LA and 16-bit LC registers are pushed onto the SS to allow nested loops.
   C. The LC register is initiated with the loop count value specified in the DO instruction.
2. The stack is pushed again.
   H. The SP is incremented.
   I. The address of the first instruction in the program loop (PC) and the current SR contents are pushed onto the SS.
   J. The LA register is initialized with the value specified in the DO instruction decremented by one.
3. The LF bit in the SR is set. The LF bit is set when a program loop is in progress and enables the end-of-loop detection.

The program loop continues execution until the program address fetched equals the LA register contents (last address of program loop). The contents of the LC are then tested for one. If the LC is not one, it is decremented, and the top location in the stack RAM is

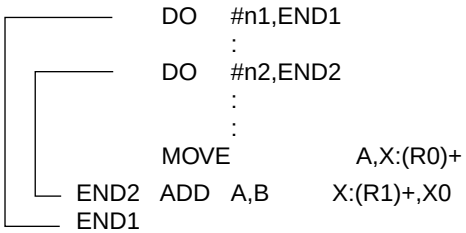


Figure 7-14 Nested DO Loops

read (but not pulled) into the PC to return to the start of the loop. If the LC is one, the program loop is terminated by the following sequence:

1. Reading the previous LF bit from the top location in the SS into the SR
2. Purging the SS (pulling the top location and discarding the contents), pulling the LA and LC registers off the SS, and restoring the respective registers
3. Incrementing the PC

The LF bit (pulled from the SS when a loop is terminated) indicates if the terminated loop was a nested loop. Figure 7-14 shows two DO loops, one nested inside the other. If the stack is managed to prevent a stack overflow, DO loops can be stacked indefinitely.

The ENDDO instruction is not used for normal termination of a DO loop; it is only used to terminate a DO loop before the LC has been decremented to one.

7.3.5 Move Instructions

The move instructions perform data movement over the XDB and YDB or over the GDB. Move instructions do not affect the CCR except the limit bit L if limiting is performed when reading a data ALU accumulator register. An address ALU instruction (LUA) is also included in the following move instructions. The MOVE instruction is the parallel move with a data ALU no-operation (NOP).

LUA	Load Updated Address
MOVE	Move Data Register
MOVEC	Move Control Register
MOVEM	Move Program Memory
MOVEP	Move Peripheral Data

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

	OPCODE/OPERANDS		PARALLEL MOVE EXAMPLES	
IMMEDIATE SHORT DATA	ADD	X0,A	#$05,Y1	
ADDRESS REGISTER UPDATE	ADD	X0,A	(R0)+N0	
REGISTER TO REGISTER	ADD	X0,A	A1,Y0	
X MEMORY	ADD	X0,A	X0,X:(R3)+	
X MEMORY PLUS REGISTER	ADD	X0,A	X:(R4)–,X1	A,Y0
Y MEMORY	ADD	X0,A	Y:(R6)+N6,X0	
Y MEMORY PLUS REGISTER	ADD	X0,A	A,X0	B,Y:(R0)

NOTE: Parallel Move Syntax—Source(Src), Destination(Dst)

Figure 7-15 Classifications of Parallel Data Moves

There are nine classifications of parallel data moves between registers and memory. Figure 7-15 shows seven parallel moves. The source of the data to be moved and the destination are separated by a comma.

Examples of the other two classifications, XY and long (L) moves, are shown in Figure 7-16.Example A illustrates the following steps: 1) register X0 is added to register A and the result is placed in register A; 2) register X0 is moved to the X memory register location pointed to by R3, and R3 is incremented; and 3) the contents of the Y memory location pointed to by R7 is moved to the B register, and R7 is decremented.

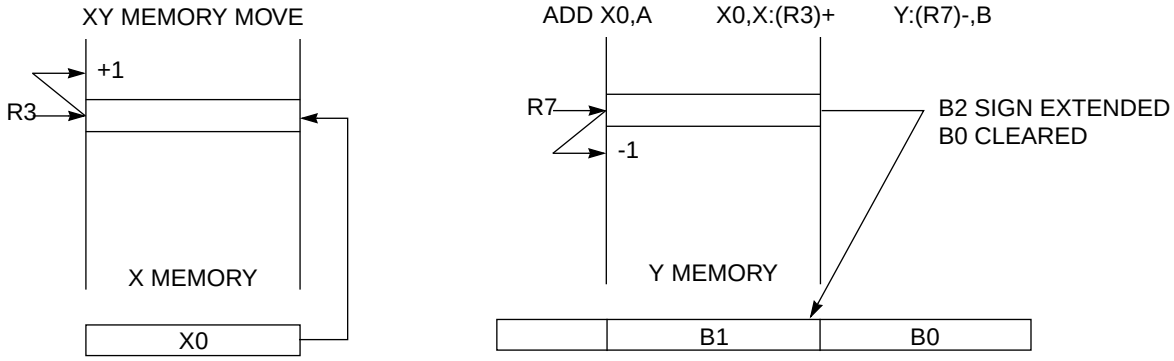
Example B depicts the following sequence: 1) register X0 is added to register A and the result is placed in register A; and 2) registers A and B are moved, respectively, to the locations in memories X and Y pointed to by R2, and then R2 is incremented by N2. The contents of the 56-bit registers A and B were rounded to 24 bits before moving to the 24-bit memory registers.

The DSP offers parallel processing of the data ALU, AGU, and program control unit. For the instruction word above, the DSP will perform the designated operation (data ALU), the data transfers specified with address register updates (AGU), and will decode the next instruction and fetch an instruction from program memory (program control unit) all in one instruction cycle. When an instruction is more than one word in length, an additional instruction execution cycle is required. Most instructions involving the data ALU are register based (all operands are in data ALU registers), thereby allowing the programmer to keep each parallel processing unit busy. An instruction that is memory oriented (such as a bit manipulation instruction) or that causes a control-flow change (such as a JMP) prevents the use of parallel processing resources during its execution.

7.3.6 Program Control Instructions

The program control instructions include jumps, conditional jumps, and other instructions

Example A



Example B

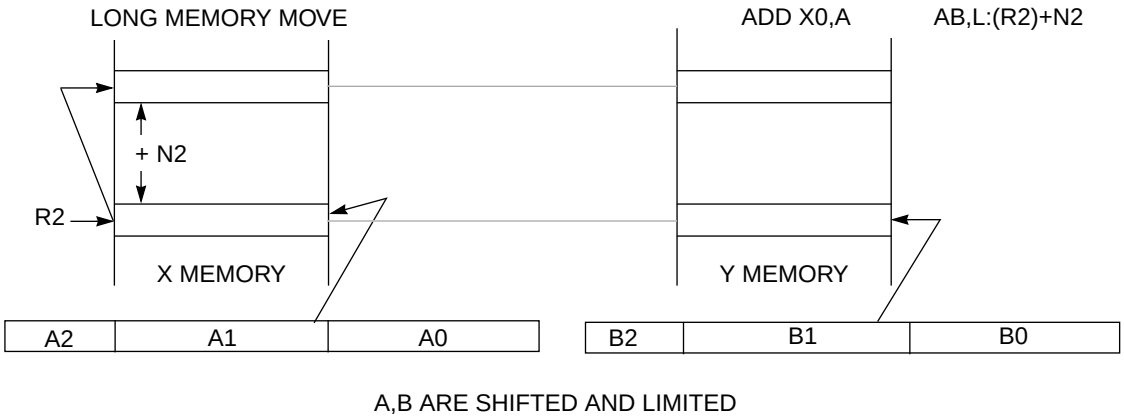


Figure 7-16 Parallel Move Examples

affecting the PC and SS. Program control instructions may affect the CCR bits as specified in the instruction. Optional data transfers over the XDB and YDB may be specified in some of the program control instructions. The following list contains the program control instructions:

Il	Illegal Instruction
Jcc	Jump Conditionally
JMP	Jump
JCLR	Jump if Bit Clear
JSET	Jump if Bit Set
JScc	Jump to Subroutine Conditionally
JSR	Jump to Subroutine
JSCLR	Jump to Subroutine if Bit Clear
JSSET	Jump to Subroutine if Bit Set

NOP	No Operation
REP	Repeat Next Instruction
RESET	Reset On-Chip Peripheral Devices
RTI	Return from Interrupt
RTS	Return from Subroutine
STOP	Stop Processing (Low-Power Standby)
SWI	Software Interrupt
WAIT	Wait for Interrupt (Low-Power Standby)

SECTION 8
PROCESSING STATES

The DSP is always in one of five processing states: normal, exception, reset, wait, and stop. These states are described in the following paragraphs.

8.1 NORMAL PROCESSING STATE

The normal processing state is associated with instruction execution. Details concerning normal processing of the individual instructions can be found in APPENDIX A INSTRUCTION SET DETAILS. Instructions are executed using a three-stage pipeline, which is described in the following paragraphs.

8.1.1 Instruction Pipeline

DSP56000/DSP56001 instruction execution is performed in a three-stage pipeline, allowing most instructions to execute at a rate of one instruction every instruction cycle. However, certain instructions require additional time to execute: instructions longer than one word, instructions using an addressing mode that requires more than one cycle, and instructions causing a control-flow change. In the latter case, a cycle is needed to clear the pipeline.

Instruction pipelining allows overlapping of instruction execution so that the fetch-decode-execute operations of a given instruction occur concurrently with the fetch-decode-execute operations of other instructions. Specifically, 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. Only one word is fetched per cycle so that, if an instruction is two words in length, the additional word will be fetched before the next instruction is fetched. Table 8-1 demonstrates pipelining; F1, D1, and E1 refer to the fetch, decode, and execute operations, respectively, of the first instruction. The third instruction, which contains an instruction extension word, takes two instruction cycles to execute. The extension word will be either an absolute address or immediate data. Although it takes three instruction cycles for the pipeline to fill and the first instruction to execute, an instruction usually executes on each instruction cycle thereafter.

Table 8-1 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	•	•	•

Each instruction requires a minimum of three instruction cycles (12 clock phases) to be fetched, decoded, and executed. This means that there is a delay of three instruction cycles on powerup to fill the pipe. A new instruction may begin immediately following the previous instruction. Two-word instructions require a minimum of four instruction cycles to execute (three cycles for the first instruction word to move through the pipe and execute and one more cycle for the second word to execute). A new instruction may start after two instruction cycles.

The pipeline is normally transparent to the user. However, it will affect program execution in some situations. These situations, which are instruction-sequence dependent, are best described by case studies. Most of these restricted sequences occur because 1) all addresses are formed during instruction decode, or 2) they are the result of contention for an internal resource such as the status register (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. To test for a suspected pipeline effect, compare between the execution of the suspect instruction 1) when it directly follows the previous instruction and 2) when four NOPs are inserted between the two. If there is a difference, it is due to a pipeline effect. The DSP56000/DSP56001 assembler (ASM56000) is designed to flag instruction sequences with potential pipeline effects so that the user can decide if the operation will be as expected.

Case 1: The following two examples show similar code sequences.

1. .No pipeline effect:

```
ORI #xx,CCR        ;Changes CCR at the end of execution time slot
Jcc xxxx           ;Reads condition codes in SR in its execution time slot
```

The Jcc will test the bits modified by the ORI without any pipeline effect in the code segment above.

2. Instruction that started execution during decode:

```
ORI #04,OMR        ;Sets DE bit at execution time slot
MOVE x:$100,a      ;Reads external RAM instead of internal ROM
```

A pipeline effect occurs in example 2 because the address of the MOVE is formed at its decode time before the ORI changes the DE bit (which changes the memory map) in the ORI's execution time slot. The following code produces the expected results of reading the internal ROM:

```
ORI #04,OMR          ;Sets DE bit at execution time slot
NOP                  ;Delays the MOVE so it will read the updated OMR
MOVE x:$100,a        ;Reads internal ROM
```

Case 2: One of the more common sequences where pipeline effects are apparent is as follows:

```
  •                  ;Move a number into register Rn (n=0; - 7).
  •
MOVE #xx,Rn
MOVE X:(Rn),A        ;Use the new contents of Rn to address memory.
  •
  •
```

In this case, before the first MOVE instruction has written Rn during its execution cycle, the second MOVE has accessed the old Rn, using the old contents of Rn. This is because the address for indirect moves is formed during the decode cycle. This overlapping instruction execution in the pipeline causes the pipeline effect. One instruction cycle should be allowed after a register has been written by a MOVE instruction before the new contents are available for use by another MOVE instruction. The proper instruction sequence is as follows:

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

Case 3: A situation related to Case 2 can be seen in the boot ROM code shown in APPENDIX A of the DSP56001 Advance Information Data Sheet (ADI1290). At the end of the bootstrap operation, the operation mode register (OMR) is changed to mode #2, and then the program that was loaded is executed. This process is accomplished in the last three instructions:

```
_BOOTEND    MOVEC    #2,OMR      ;Set the operating mode to 2
                                 ;(and trigger an exit from
                                 ;bootstrap mode).
            ANDI     #$0,CCR     ;Clear SR as if RESET and
                                 ;introduce delay needed for
                                 ;Op. Mode change.
            JMP      <$0         ;Start fetching from PRAM, P:$0000
```

The JMP instruction generates its jump address during its decode cycle. If the JMP instruction followed the MOVEC, the MOVEC instruction would not have changed the OMR before the JMP instruction formed the fetch address. As a result, the jump would fetch the instruction at P:$0000 of the bootstrap ROM (MOVE #$FFE9,R2). The OMR would then change due to the MOVEC instruction, and the next instruction would be the second instruction of the downloaded code at P:$0001 of the internal RAM. However, the ANDI instruction allows the OMR to be changed before the JMP instruction uses it, and the JMP fetches P:$0000 of the internal RAM.

Case 4: An interrupt has two additional control cycles that are executed in the interrupt controller concurrently with the fetch, decode, and execute cycles (see 8.2 EXCEPTION PROCESSING STATE (INTERRUPT PROCESSING) and Figure 8-2). During these two control cycles, the interrupt is arbitrated by comparing the interrupt mask level with the interrupt priority level (IPL) of the interrupt and allowing or disallowing the interrupt. Therefore, if the interrupt mask is changed after an interrupt is arbitrated and accepted as pending but before the interrupt is executed, the interrupt will be executed, regardless of what the mask was changed to. The following examples show that the old interrupt mask is in effect for up to four additional instruction cycles after the interrupt mask is changed. All instructions shown in the examples here are one-word instructions; however, one two-word instruction can replace two one-word instructions except where noted.

1. Program flow with no interrupts after interrupts are disabled:

```
    •
    •
ORI #03,MR      ;Disable interrupts
INST 1
INST 2
INST 3
INST 4
    •
    •
```

2. The four possible variations in program flow that may occur after interrupts are disabled:

```
    •              •              •              •
    •              •              •              •
ORI #03,MR     ORI #03,MR     ORI #03,MR     ORI #03,MR
II (See Note 2)INST 1         INST 1         INST 1
II+1           II             INST 2         INST 2
INST 1         II+1           ll             INST 3 (See Note 1)
INST 2         INST 2         II+1           ll
INST 3         INST 3         INST 3         II+1
INST 4         INST 4         INST 4         INST 4
    •              •              •              •
    •              •              •              •
```

Note 1: INST 3 may be executed at that point only if the preceding instruction (INST 2) was a single-word instruction.

Note 2: II=Interrupt instruction from maskable interrupt.

The following program flow will not occur because the ORI instruction becomes effective after a pipeline latency of four instruction cycles:

```
    •
    •
ORI #03,MR          ;Disable interrupts.
INST 1
INST 2
INST 3
INST 4
II                  ;Interrupts disabled.
II+1                ;Interrupts disabled.
    •
    •
```

1. Program flow without interrupts after interrupts are re-enabled:

```
    •
    •
ANDI #00,MR         ;Enable interrupts
INST 1
INST 2
INST 3
INST 4
    •
    •
```

2. Program flow with interrupts after interrupts are re-enabled:

```
    •
    •
ANDI #00,MR         ;Enable interrupts
INST 1              ;Uninterruptable
INST 2              ;Uninterruptable
INST 3              ;II+1 fetched
INST 4              ;II+1 fetched
II
II+1
    •
    •
```

The DO instruction is another instruction that begins execution during the decode cycle of the pipeline. As a result, there are a number of restrictions concerning access contention with the program controller registers accessed by the DO instruction. The ENDDO instruction has similar restrictions. APPENDIX A INSTRUCTION SET DETAILS contains

additional information on the DO and ENDDO instruction restrictions.

Case 5: A resource contention problem can occur when one instruction is using a register during its decode while the instruction executing is accessing the same resource. One example of this is as follows:

```
MOVEC       X:$100,SSH
DO          #$10,END
```

The problem occurs because the MOVEC instruction loads the contents of X:$100 into the system stack high (SSH) during T3 of its execution cycle. The DO instruction that follows pushes the stack (LA → SSH, LC → SSL) during T3 of its decode cycle. Therefore, the two instructions try writing to the SSH simultaneously and conflict.

8.1.2 Summary of Pipeline-Related Restrictions

A summary of the instruction sequences that cause pipeline effects is given in the following paragraphs. Additional information concerning the individual instructions can be found in APPENDIX A INSTRUCTION SET DETAILS.

DO instruction restrictions:

The DO instruction must not be immediately preceded by any of the following instructions:

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

Restrictions near the end of DO loops:

Proper DO loop operation is guaranteed if no instruction starting at address LA-2, LA-1, or LA specifies the program controller registers SR, SP, SSL, LA, LC, or (implicitly) PC as a destination register or specifies SSH as a source or a destination register. Also, SSH can not be specified as a source register in the DO instruction.

The restricted instructions at LA-2, LA-1, and LA are as follows:

```
DO
BCHG/BCLR/BSET     LA, LC, SR, SP, SSH, or SSL
BTST SSH
JCLR/JSET/JSCLR/JSSET SSH
MOVEC/MOVEM/MOVEP from SSH
MOVEC/MOVEM/MOVEP to LA, LC, SR, SP, SSH, or SSL
ANDI/ORI MR
```

The restricted instructions at LA include the following:

```
Any two-word instruction
```

Jcc, JMP, JScc, JSR,
REP, RESET, RTI, RTS, STOP, WAIT

Other restrictions are

DO SSH,xxxx
JSR/JScc/JSCLR/JSSET to LA, if loop flag is set

ENDDO instruction restrictions:

The ENDDO instruction must not be immediately preceded by any of the following instructions:

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

RTI and RTS instruction restrictions:

The RTI instruction must not be immediately preceded by any of the following instructions:

BCHG/BCLR/BSET SR, SSH, SSL, or SP
MOVEC/MOVEM to SR, SSH, SSL, or SP
MOVEC/MOVEM from SSH
ANDI MR, ANDI CCR
ORI MR, ORI CCR

The RTS instruction must not be immediately preceded by any of the following instructions:

BCHG/BCLR/BSET SSH, SSL, or SP
MOVEC/MOVEM to SSH, SSL, or SP
MOVEC/MOVEM from SSH

SP and SSH/SSL register manipulation restrictions:

In addition to all the above restrictions concerning SP, SSH, and SSL, the following instruction sequences are illegal:

1. BCHG/BCLR/BSET SP
2. MOVEC/MOVEM/MOVEP from SSH or SSL

and

1. MOVEC/MOVEM to SP
2. MOVEC/MOVEM/MOVEP from SSH or SSL

and

1. MOVEC/MOVEM to SP
2. JCLR/JSET/JSCLR/JSSET SSH or SSL

and

1. BCHG/BCLR/BSET SP
2. JCLR/JSET/JSCLR/JSSET SSH or SSL

Also the instruction MOVEC SSH,SSH is illegal.

Rn, Nn, and Mn register restrictions:

If an address register (R0 – R7, N0 – N7, or M0 – M7) is changed with a move-type instruction (LUA, Tcc, MOVE, MOVEM, MOVEC, or parallel move), the new contents will not be available for use as a pointer until the second following instruction. This restriction does not apply to registers updated as part of an indirect addressing mode.

Fast interrupt routines:

SWI, STOP, and WAIT may not be used in a fast interrupt routine.

8.2 EXCEPTION PROCESSING STATE (INTERRUPT PROCESSING)

The exception processing state is associated with interrupts that can be generated by conditions inside the DSP or from external sources. There are many sources for interrupts on the DSP56000/DSP56001; some of these sources can generate more than one interrupt. A prioritized interrupt vector scheme with 32 vectors is used to provide fast interrupt service. The following list outlines how interrupts are processed by the DSP56000/DSP56001:

9. A hardware interrupt is synchronized with the DSP clock, and the interrupt pending flag for that particular hardware interrupt is set. An interrupt source can have only one interrupt pending at any given time.
10. All pending interrupts (external and internal) are arbitrated to select which interrupt will be processed. The arbiter automatically ignores any interrupts with an IPL lower than the interrupt mask level in the SR and selects the remaining interrupt with the highest IPL.
11. The interrupt controller then freezes the program counter (PC) and fetches two instructions at the two interrupt vector addresses associated with the selected interrupt.
12. The interrupt controller jams the two instructions into the instruction stream and releases the PC, which is used for the next instruction fetch. The next interrupt arbitration is then begun.

If neither instruction is a change of program-flow instruction (i.e., a JSR), the state of the machine is not saved on the stack, and a fast interrupt is executed. A long interrupt is formed if one of the interrupt instructions fetched is a JSR instruction. The PC is immedi-

ately released, the SR and the PC are saved in the stack, and the jump instruction controls where the next instruction is fetched from. While either an unconditional jump or conditional jump can be used to form a long interrupt, they do not store the PC on the stack; therefore, there is no return path.

In digital signal processing, one of the main uses of interrupts is to transfer data between DSP memory or registers and a peripheral device. When such an interrupt occurs, a limited context switch with minimum overhead is often desirable. This limited context switch is accomplished by a fast interrupt. The long interrupt is used when a more complex task must be accomplished to service the interrupt..

The second and third activities require two additional control cycles, which effectively make the interrupt pipeline five levels deep.

8.2.1 Interrupt Sources

Exceptions may originate from any of the 32 vector addresses listed in Table 8-2. The corresponding interrupt starting address for each interrupt source is shown. These addresses are located in the first 64 locations of program memory. When an interrupt is serviced, the instruction at the interrupt starting address is fetched first. Because the program flow is directed to a different starting address for each interrupt, the interrupt structure of the DSP56000/DSP56001 is said to be vectored. A vectored interrupt structure has low overhead execution. If it is known a priori that certain interrupts will not be used, those interrupt vector locations can be used for program or data storage.

The 32 interrupts are prioritized into four levels. Level 3, the highest priority level, is not maskable. Levels 0 – 2 are maskable. The interrupts within each level are prioritized according to a predefined priority. The level-3 interrupts (reset, illegal instruction, nonmaskable interrupt (NMI), stack error, trace, and software interrupt (SWI) are discussed individually.

8.2.1.1 Hardware Interrupt Source

There are two types of hardware interrupts in the DSP: internal and external. The internal interrupts include all of the on-chip peripheral devices (host interface (HI), synchronous serial interface (SSI), and serial communications interface (SCI). Each internal interrupt source is latched and serviced if it is not masked. When it is serviced, the interrupt is cleared. Each internal hardware source has independent enable control.

The external hardware interrupts include $\overline{\text{RESET}}$, NMI, $\overline{\text{IRQA}}$, and $\overline{\text{IRQB}}$. The $\overline{\text{RESET}}$ interrupt, which is level sensitive, is the highest level interrupt (IPL 3). The $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ interrupts can be programmed to be level sensitive or edge sensitive. Since the level-sensitive interrupts will not be cleared automatically when they are serviced, they must be cleared by other means to prevent multiple interrupts. The edge-sensitive inter-

Table 8-2 Interrupt Sources

Interrupt Starting Address	IPL	Interrupt Source
$0000	3	Hardware $\overline{\text{RESET}}$
$0002	3	Stack Error
$0004	3	Trace
$0006	3	SWI
$0008	0 - 2	$\overline{\text{IRQA}}$
$000A	0 - 2	$\overline{\text{IRQB}}$
$000C	0 - 2	SSI Receive Data
$000E	0 - 2	SSI Receive Data With Exception Status
$0010	0 - 2	SSI Trasmit Data
$0012	0 - 2	SSI Transmit Data with Exception Status
$0014	0 - 2	SCI Receive Data
$0016	0 - 2	SCI Receive Data with Exception Status
$0018	0 - 2	SCI Transmit Data
$001A	0 - 2	SCI Idle Line
$001C	0 - 2	SCI Timer
$001E	3	NMI — Reserved for Hardware Development
$0020	0 - 2	Host Receive Data
$0022	0 - 2	Host Transmit Data
$0024	0 - 2	Host Command (Default)
$0026	0 - 2	Available for Host Command
$0028	0 - 2	Available for Host Command
$002A	0 - 2	Available for Host Command
$002C	0 - 2	Available for Host Command
$002E	0 - 2	Available for Host Command
$0030	0 - 2	Available for Host Command
$0032	0 - 2	Available for Host Command
$0034	0 - 2	Available for Host Command
$0036	0 - 2	Available for Host Command
$0038	0 - 2	Available for Host Command
$003A	0 - 2	Available for Host Command
$003C	0 - 2	Available for Host Command
$003E	3	Illegal Instruction

rupts are latched as pending on the high-to-low transition of the interrupt input and are

automatically cleared when the interrupt is serviced. $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ can be programmed to one of three priority levels: 0, 1, or 2, all of which are maskable. Additionally, both of these interrupts have independent enable control.

When the $\overline{\text{IRQA}}$ or $\overline{\text{IRQB}}$ interrupts are disabled in the interrupt priority register, the pending request will be ignored, regardless of whether the interrupt input was defined as level sensitive or edge sensitive. Additionally, if the interrupt is defined as edge sensitive, its edge-detection latch will remain in the reset state as long as the interrupt is disabled; if the interrupt is defined as level sensitive, its edge-detection latch will remain in the reset state. If the level-sensitive interrupt is disabled while the interrupt is pending, the pending interrupt will be cancelled. However, if the interrupt has been fetched, it normally will not be cancelled.

Interrupt service, which begins by fetching the instruction word in the first vector location, is considered finished when the instruction word in the second vector location is fetched. In the case of an edge-triggered interrupt, the internal latch is automatically cleared when the second vector location is fetched. The fetch of the first vector location does not guarantee that the second location will be fetched. Figure 8-1 illustrates the one case where the second vector location is not fetched. In Figure 8-1, the SWI instruction discards the fetch of the first interrupt vector to ensure that the SWI vectors will be fetched. Instruction n4 is decoded as an SWI while ii1 is being fetched. Execution of the SWI requires that ii1 be discarded and the two SWI vectors (ii3 and ii4) be fetched instead.

INTERRUPT CONTROL CYCLE 1		i		i*							
INTERRUPT CONTROL CYCLE 2			i		i*						
FETCH	n3	n4	n5	ii1		ii3	ii4	sw1	sw2	sw3	sw4
DECODE	n2	n3	SWI	—	—	—	JSR	—	sw1	sw2	sw3
EXECUTE	n1	n2	n3	SWI	NOP	NOP	NOP	JSR	—	sw1	sw2
INSTRUCTION BEING DECODED	1										

i = INTERRUPT REQUEST
i* = INTERRUPT REQUEST GENERATED BY SWI
ii1 = FIRST VECTOR OF INTERRUPT i
ii3 = FIRST SWI VECTOR (ONE-WORD JSR)
ii4 = SECOND SWI VECTOR
n = NORMAL INSTRUCTION WORD
n4 = SWI
sw = INSTRUCTIONS PERTAINING TO THE SWI LONG INTERRUPT ROUTINE

Figure 8-1 Interrupting an SWI

CAUTION

On all level-sensitive interrupts, the interrupt must be externally released before interrupts are internally re-enabled, or the processor will be interrupted repeatedly until the interrupt is released.

The edge-sensitive NMI is generated on the first transition to 10 V on the $\overline{\text{IRQB}}$ pin after the last time the NMI interrupt was serviced or the DSP was reset. The NMI is a priority 3 interrupt and cannot be masked. Only $\overline{\text{RESET}}$ and illegal instruction have higher priority than NMI. NMI is reserved for hardware development and should not be used in an application. Repeated use may damage the integrated circuit.

8.2.1.2 Software Interrupt Source

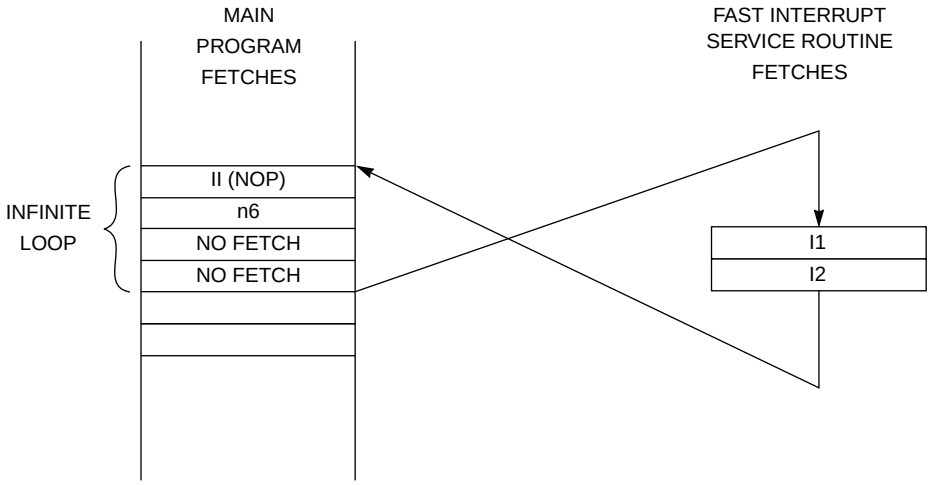
There are two software interrupt sources — illegal instruction interrupt (III) and SWI. The III is a nonmaskable interrupt (IPL 3), which is serviced immediately following the execution of the illegal instruction or the attempted execution of an illegal instruction (any undefined operation code). IIIs are fatal errors. Only a long interrupt routine should be used for the III routine; RTI or RTS should not be used at the end of the interrupt routine since return from the III to the main code should not be attempted. During the III service, the JSR located in the III vector will normally stack the address of the illegal instruction (this is the reason why return should not be attempted (see Figure 8-2). The user may examine the stack (using MOVE SSH,dest) to locate the offending illegal instruction. The illegal instruction is useful for triggering the illegal interrupt service to see if the III routine is capable of recovery from illegal instructions.

There are two cases in which the stacked address will not point to the illegal instruction:

1. If the illegal instruction is one of the two instructions at an interrupt vector location and is fetched during a regular interrupt service, the processor will stack the address of the next sequential instruction in the normal instruction flow (the regular return address of the interrupt routine that had the illegal opcode in its vector).

2. If the illegal instruction follows an REP instruction (see Figure 8-3), the DSP will effectively execute the illegal instruction as a repeated NOP and the interrupt vector will then be inserted in the pipeline. The next instruction will be fetched but will not be decoded or executed. The processor will stack the address of the next sequential instruction, which is two instructions after the illegal instruction.

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



(a) Instruction Fetches from Memory

ILLEGAL INSTRUCTION INTERRUPT RECOGNIZED AS PENDING

ILLEGAL INSTRUCTION INTERRUPT RECOGNIZED AS PENDING

INTERRUPT CONTROL CYCLE 1								i						
INTERRUPT CONTROL CYCLE 2									i					
FETCH		n1	n2	n3	n4	n5	n6	—	—	ii1	ii2	n5		
DECODE			n1	n2	n3	n4	II	—	—	—	i1	ii2	II	
EXECUTE				n1	n2	n3	n4	NOP	—	—	—	ii1	ii2	NOP
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

Figure 8-2 Illegal Instruction Interrupt Serviced by a Fast Interrupt

it is the next sequential instruction in the flow). The loop state machine will again decrement LC because the LA instruction is being executed. At this point, the illegal instruction will trigger the III. The result is that the loop state machine decrements LC twice in one loop due to the presence of the illegal opcode at the LA location.

SWI is a nonmaskable interrupt (IPL 3), which is serviced immediately following the SWI

ILLEGAL INSTRUCTION INTERRUPT
RECOGNIZED AS PENDING

INTERRUPT CONTROL CYCLE 1										i						
INTERRUPT CONTROL CYCLE 2											i					
FETCH		n1	n2	n3	n4	n5	n6	n7	—	—	—	ii1	ii2	n8		
DECODE			n1	n2	n3	n4	REP	II	—	—	—	—	ii1	ii2	n8	
EXECUTE				n1	n2	n3	n4	REP	REP	NOP	—	—	—	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

Figure 8-3 Repeated Illegal Instruction

instruction execution. A long interrupt service routine is usually used. The difference between an SWI and a JSR instruction is that the SWI sets the interrupt mask to prevent interrupts below IPL 3 from being serviced. Masking out lower level interrupts makes the SWI very useful for setting breakpoints in monitor programs. The JSR instruction does not affect the interrupt mask.

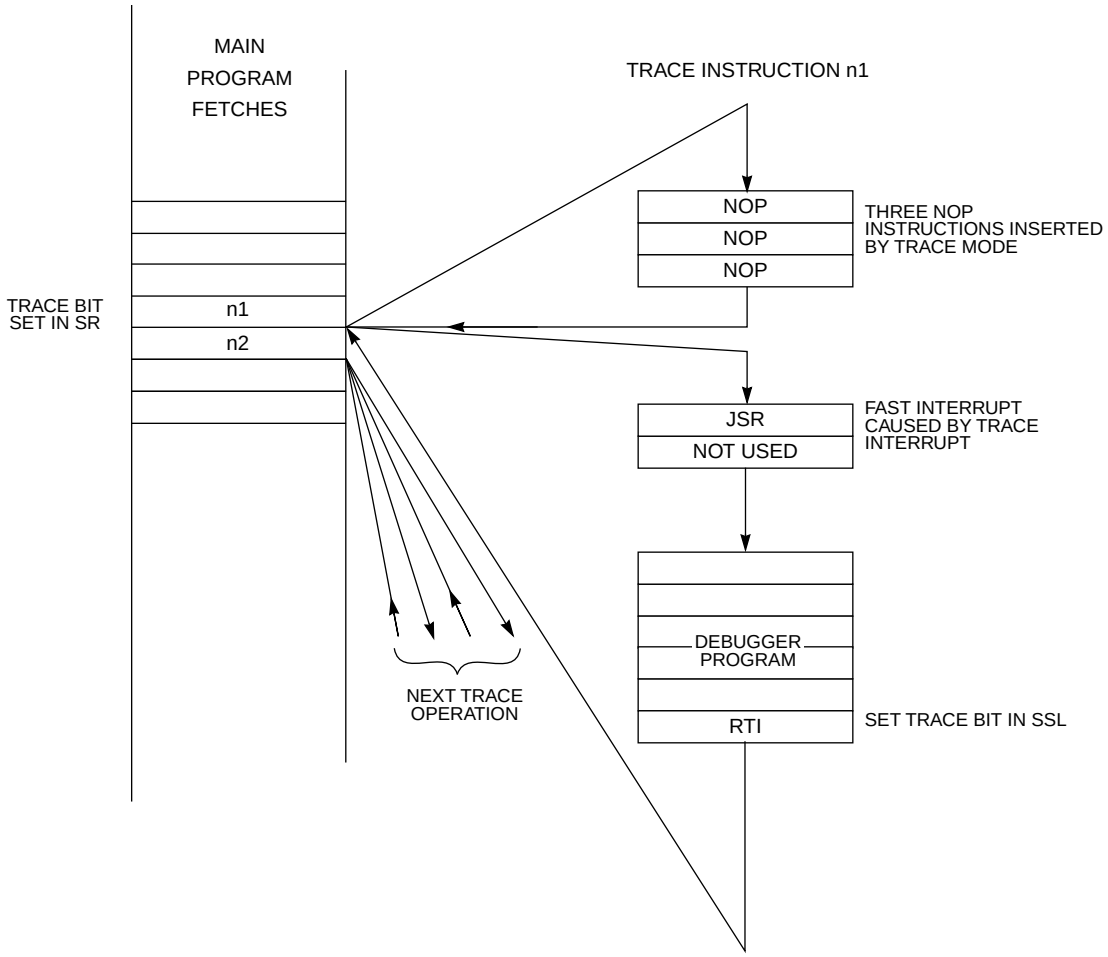
8.2.1.3 Other Interrupt Sources

Other interrupt sources include the stack error interrupt and trace interrupt (IPL3 interrupts).

An overflow or underflow of the system stack (SS) causes a stack error interrupt (see SECTION 6 PROGRAM CONTROL UNIT for additional information on the stack error flag). The stack error interrupt is caused by a nonrecoverable error condition and is vectored to P:$0002. Since the stack error is nonrecoverable, a long interrupt should be used to service the interrupt, and the service routine should not end in an RTI. Executing an RTI instruction ‘‘pops’’ the stack, which has been corrupted.

The DSP56000/DSP56001 includes a facility for instruction-by-instruction tracing as a program development aid. This trace mode (entered by setting the trace bit in the SR) generates a trace exception after each instruction executed (see Figure 8-4), which can be used by a debugger program to monitor the execution of a program.

The trace mode is entered by setting the trace bit in the SR. A trace exception is generated after executing each instruction executed while the trace bit is set. When servicing



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPT CONTROL CYCLE 1	i												i					
INTERRUPT CONTROL CYCLE 2		i													i			
FETCH		n1	NOP	NOP	NOP	JSR	—	TRACE PROGRAM			RTI	—	n2	NOP	NOP	NOP		
DECODE			n1	NOP	NOP	NOP	JSR	NOP	TRACE PROGRAM			RTI	NOP	n2	NOP	NOP	NOP	
EXECUTE				n1	NOP	NOP	NOP	JSR	NOP	TRACE PROGRAM			RTI	NOP	n2	NOP	NOP	NOP
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
II = ILLEGAL INSTRUCTION
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 8-4 Trace Exception

the trace exception, it is expected that a JSR will be encountered in the trace vector loca-

tions, thereby forming a long interrupt routine. The JSR causes the SR to be stacked and the trace bit in the SR to be cleared (clearing the trace bit in the SR prevents tracing while executing the trace exception service routine). This service routine should end with an RTI instruction, which restores the SR (with the trace bit set) from the SS, causing the next instruction to be traced. The pipeline must be flushed to allow each sequential instruction to be traced. Three instruction cycles are appended by the tracing facility to the end of each instruction traced (these are the three NOP instructions shown in Figure 8-4) flushing the pipeline and allowing the next trace interrupt to follow the next sequential interrupt.

During tracing, the REP instruction and the instruction being repeated are considered a single two-word instruction. That is, only after executing the REP instruction and all the repeats of the next instruction will the trace exception be generated.

Fast interrupts can not be traced because they are uninterruptable. Long interrupts will not be traced (unless the trace mode is entered in the subroutine) because the SR is pushed on the stack and the trace bit is cleared. Tracing is resumed upon returning from a long interrupt because the trace bit is restored when the SR is restored. Interrupts are not likely to occur during tracing because only an interrupt with a higher IPL can interrupt during a trace operation. While executing the program being traced, the trace interrupt will always be pending and will win the interrupt arbitration. During the trace interrupt, the interrupt mask is set to reject interrupts below IPL3.

8.2.2 Interrupt Priority Structure

Four levels of interrupt priority are provided. IPLs numbered 0, 1, and 2 are maskable (level 0 is the lowest level). Level 3 (highest level) is nonmaskable. The only IPL 3 interrupts are reset, III, NMI, stack error, trace, and SWI. The interrupt mask bits (I1, I0) in the SR reflect the current processor priority level and indicate the IPL needed for an interrupt source to interrupt the processor (see Table 8-3). Interrupts are inhibited for all priority levels less than the current processor priority level. However, level 3 interrupts are not maskable and therefore can always interrupt the processor.

Table 8-3 Status Register Interrupt Mask Bits

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

8.2.2.1 Interrupt Priority Levels

The IPL for each on-chip peripheral device (HI, SSI, SCI) and for each external interrupt source ($\overline{IRQA}$, $\overline{IRQB}$) can be programmed under software control. Each on-chip or external peripheral device can be programmed to one of the three maskable priority levels (IPL 0, 1, or 2). IPLs are set by writing to the interrupt priority register shown in Figure 8-5. This read/write register specifies the IPL for each of the interrupting devices (HI, SSI, SCI, $\overline{IRQA}$, $\overline{IRQB}$). In addition, this register specifies the trigger mode of both external interrupt

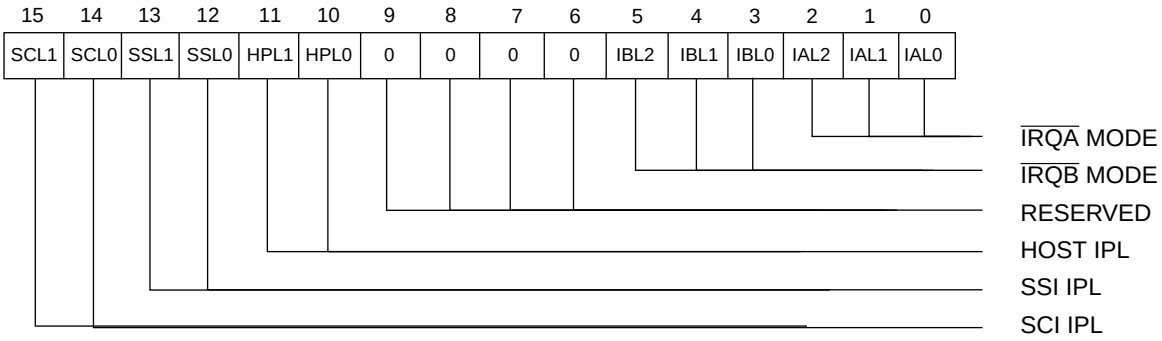


Figure 8-5 Interrupt Priority Register (Addr X:$FFFF)

sources and is used to enable or disable the individual external interrupts. This register is cleared on $\overline{RESET}$ or by the reset instruction. Table 8-4 defines the IPL bits. Table 8-5 defines the external interrupt trigger mode bits.

Table 8-4 Interrupt Priority Level

xxL1	xxL0	Enabled	IPL
0	0	No	—

Table 8-5 External Interrupt

8.2.2.2 Exception Priorities within an IPL

If more than one exception is pending when an instruction is executed, the interrupt with the highest priority level is serviced first. When multiple interrupt requests having the same IPL are pending, a second fixed-priority structure within that IPL determines which interrupt is serviced. The fixed priority of interrupts within an IPL and the interrupt enable bits for all interrupts are shown in Table 8-6. The interrupt enable bits for the HI, SSI, and SCI are located in the control registers associated with their respective on-chip peripherals.

8.2.3 Instructions Preceding the Interrupt Instruction Fetch

The following one-word instructions are aborted when they are fetched in the cycle preceding the fetch of the first interrupt instruction word — REP, STOP, WAIT, RESET, RTI, RTS, Jcc, JMP, JScc, and JSR.

Two-word instructions are aborted when the first interrupt instruction word fetched will replace the fetch of the second word of the two-word instruction. Aborted instructions are refetched again when program control returns from the interrupt routine. The PC is adjusted appropriately before the end of the decode cycle of the aborted instruction.

If the first interrupt word fetch occurs in the cycle following the fetch of a one-word instruction not previously listed or the second word of a two-word instruction, that instruction will complete normally before the start of the interrupt routine.

The following cases have been identified where service of an interrupt might encounter an extra delay:

1. If a long interrupt routine is used to service an SWI, then the processor priority level is set to 3. Thus,all interrupts except other level-3 interrupts are disabled until the SWI service routine terminates with an RTI (unless the SWI service routine software lowers the processor priority level).
2. While servicing an interrupt, the next interrupt service will be delayed according to the following rule: after the first interrupt instruction word reaches the instruction decoder, at least three more instructions will be decoded before

Table 8-6 Exception Priorities within an IPL

Priority	Exception	Enabled By	Bit No.	X Data Memory Address
Level 3 (Nonmaskable)				
Highest	Hardware $\overline{\text{RESET}}$	—	—	—
	III	—	—	—
	NMI	—	—	—
	Stack Error	—	—	—
	Trace	—	—	—
Lowest	SWI	_	—	—
Levels 0, 1, 2 (Maskable)				
Highest	$\overline{\text{IRQA}}$ (External Interrupt)	$\overline{\text{IRQA}}$ Mode Bits	0 and 1	$FFFF
	$\overline{\text{IRQB}}$ (External Interrupt)	$\overline{\text{IRQB}}$ Mode Bits	3 and 4	$FFFF
	Host Command Interrupt	HCIE	2	$FFE8
	Host Receive Data Interrupt	HRIE	0	$FFE8
	Host Transmit Data Interrupt	HTIE	1	$FFE8
	SSI RX Data with Exception Interrupt	RIE	15	$FFED
	SSI RX Data Interrupt	RIE	15	$FFED
	SSI TX Data with Exception Interrupt	TIE	14	$FFED
	SSI TX Data Interrupt	TIE	14	$FFED
	SCI RX Data with Exception Interrupt	RIE	11	$FFF0
	SCI RX Data Interrupt	RIE	11	$FFF0
	SCI TX Data Interrupt	TIE	12	$FFF0
	SCI Idle Line Interrupt	ILIE	10	$FFF0
Lowest	SCI Timer Interrupt	TMIE	13	$FFF0

decoding the next first interrupt instruction word. If any one pair of instructions-being counted is the REP instruction followed by an instruction to be repeated, then the combination is counted as two instructions independent of the number of repeats done. Sequential REP combinations will cause pending interrupts to be rejected and can not be interrupted until the sequence of REP

combinations ends.

3. The following instructions are not interruptable: SWI, STOP, WAIT, and RESET.
4. The REP instruction and the instruction being repeated are not interruptable.
5. If the trace bit in the SR is set, the only interrupts that will be processed are the hardware RESET, III,NMI, stack error, and trace. Peripheral and external interrupt requests will be ignored. The interrupt generated by the SWI instruction will be ignored.

During an interrupt instruction fetch, two instruction words are fetched — the first from the interrupt starting address and the second from the interrupt starting address +1 locations.

8.2.4 Interrupt Types

Two types of interrupt routines may be used: fast and long. The fast routine consists of the two automatically inserted interrupt instruction words. These words can contain any unrestricted, single two-word instruction or any two one-word instructions (see A.8 INSTRUCTION SEQUENCE RESTRICTIONS for a list of restrictions). Fast interrupt routines are never interruptable.

CAUTION

Status is not preserved during a fast interrupt routine; therefore, instructions that modify status should not be used at the interrupt starting address and interrupt starting address +1.

If one of the instructions in the fast routine is a JSR, then a long interrupt routine is formed. The following actions occur during execution of the JSR instruction when it occurs in the interrupt starting address or in the interrupt starting address +1:

1. The PC (containing the return address) and the SR are stacked.
2. The loop flag is reset.
3. The scaling mode bits are reset.
4. The IPL is raised to disallow further interrupts at the same or lower levels (except that hardware $\overline{\text{RESET}}$, NMI, stack error, trace, and SWI can always interrupt).
5. The trace bit in the SR is cleared.

The long interrupt routine should be terminated by an RTI. Long interrupt routines are interruptable by higher priority interrupts.

8.2.5 Interrupt Arbitration

External interrupts are internally synchronized with the processor clock (takes up to three T cycles) before their interrupt-pending flags are set. Each external interrupt and internal interrupt has its own flag. After each instruction is executed, all interrupts are arbitrated — i.e., all hardware interrupts that have been latched into their respective interrupt-pending flags and all internal interrupts. During arbitration, each interrupt's IPL is compared with the interrupt mask in the SR, and the interrupt is either allowed or disallowed. The remaining interrupts are prioritized according to the priority shown in Table 8-6, and the highest priority interrupt is chosen. The interrupt vector is then calculated so that the program interrupt controller can fetch the first interrupt instruction. Interrupt arbitration and control, which occurs concurrently with the fetch-decode-execute cycle, takes two instruction cycles. Interrupts from a given source are not buffered. The interrupt-pending flag for the chosen interrupt is not cleared until the second interrupt vector of the chosen interrupt is being fetched. A new interrupt from the same source will not be accepted for the next interrupt arbitration until that time.

The internal interrupt acknowledge signal is used to clear the edge-triggered interrupt flags, the HC bit in the host port, the SCI timer interrupt, and the internal latches of the stack error, NMI, SWI, and trace interrupts. Peripheral interrupt requests that need a read/write action to some register do not receive this signal, and those interrupts will remain pending until their registers are read/written. Also, level-triggered interrupts will not be cleared. The acknowledge signal will be generated after generation of the interrupt vectors, not before.

8.2.6 Interrupt Instruction Fetch

The interrupt controller generates an interrupt instruction fetch address, which points to the first instruction word of a two-word interrupt routine. This address is used for the next instruction fetch, instead of the contents of the PC, and the interrupt instruction fetch address +1 is used for the subsequent instruction fetch. While the interrupt instructions are being fetched, the PC is inhibited from being updated. After the two interrupt words have been fetched, the PC is used for any subsequent instruction fetches.

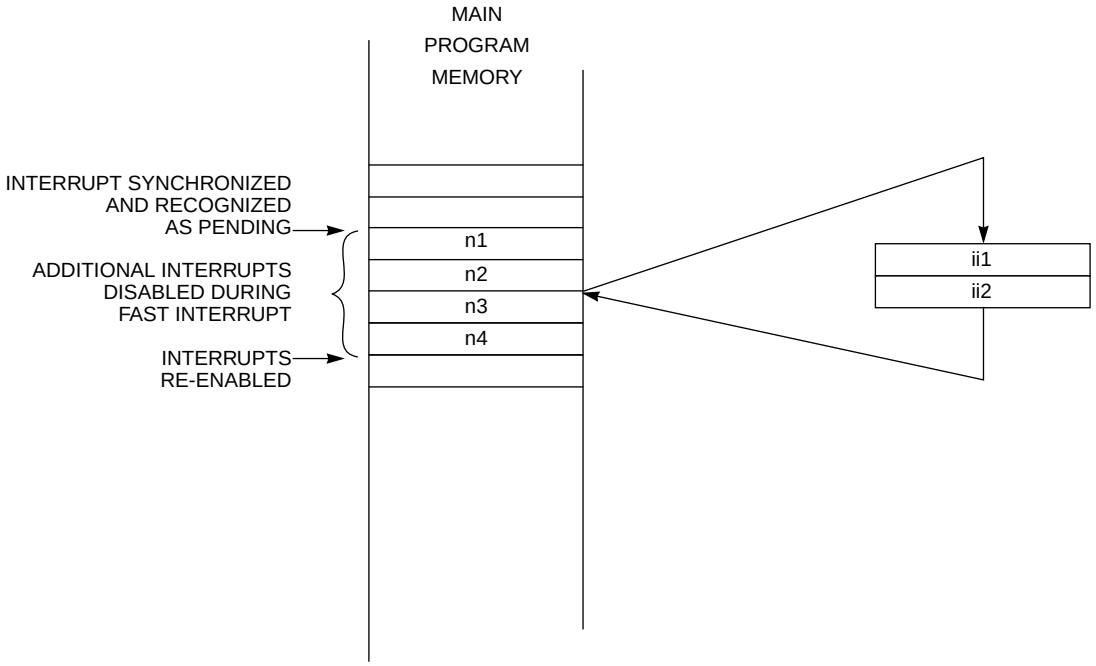
After both interrupt vectors have been fetched, they are guaranteed to be executed. This is true even if the instruction that is currently being executed is a change-of-flow instruction (i.e., JMP, JSR, etc.) that would normally ignore the instructions in the pipe. After the interrupt instruction fetch, the PC will point to the instruction that would have been fetched if the interrupt instructions had not been inserted.

8.2.7 Interrupt Instruction Execution

Interrupt instruction execution is considered "fast" if neither of the instructions of the interrupt service routine cause a change of flow. A JSR within a fast interrupt routine forms a long interrupt, which is terminated with an RTI instruction to restore the PC and SR from the stack and return to normal program execution. Reset is a special exception, which will normally contain only a JMP instruction at the exception start address. At the programmer's option, almost any instruction can be used in the fast interrupt routine. The restricted instructions include SWI, STOP, and WAIT. Figure 8-6 and Figure 8-8 show the fast and the long interrupt service routines. The fast interrupt executes only two instructions and then automatically resumes execution of the main program; whereas, the long interrupt must be told to return to the main program by executing an RTI instruction.

Figure 8-6 illustrates the effect of a fast interrupt routine in the stream of instruction fetches.

Figure 8-7 shows the sequence of instruction decodes between two fast interrupts. Four decodes occur between the two interrupt decodes (two after the first interrupt and two preceding the second interrupt). The requirement for these four decodes establishes the



ii = INTERRUPT INSTRUCTION
n = NORMAL INSTRUCTION

(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPTS RE-ENABLED

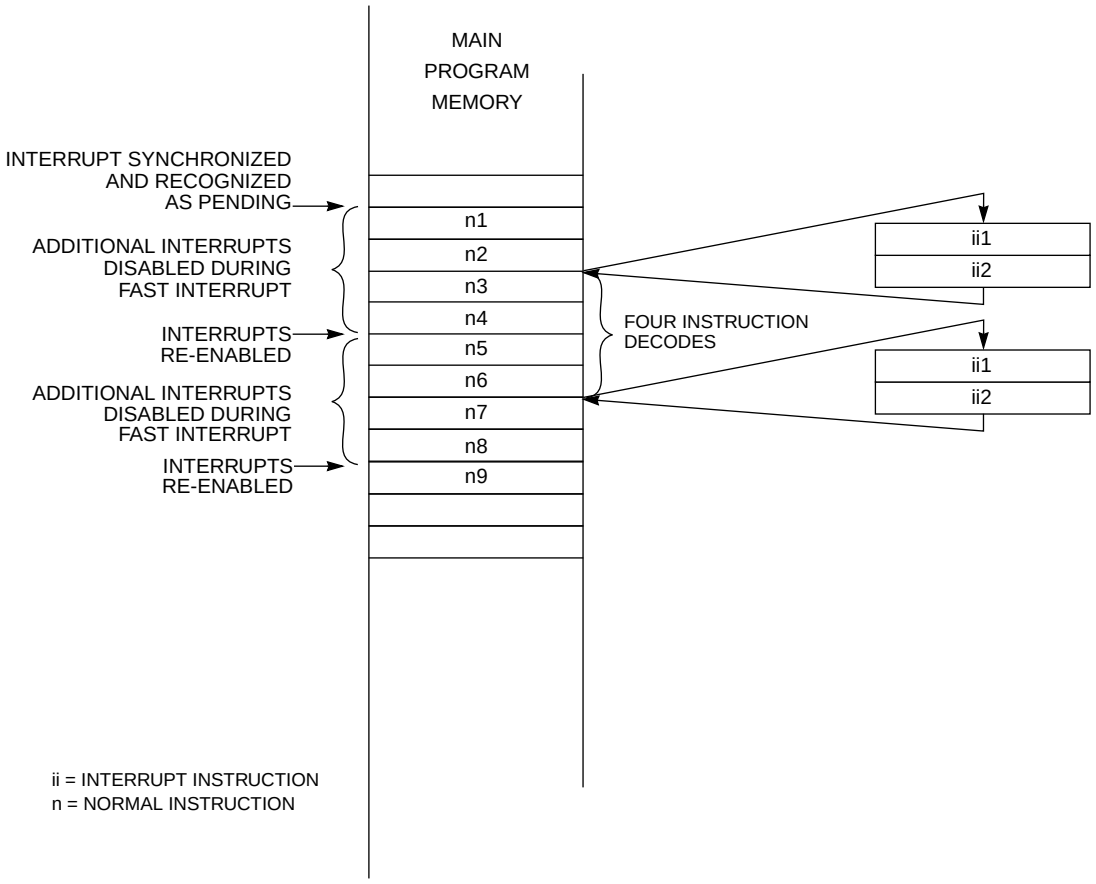
INTERRUPT CONTROL CYCLE 1	i							
INTERRUPT CONTROL CYCLE 2		i						
FETCH	n1	n2	ii1	ii2	n3	n4		
DECODE		n1	n2	ii1	ii2	n3	n4	
EXECUTE			n1	n2	ii1	ii2	n3	n4
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

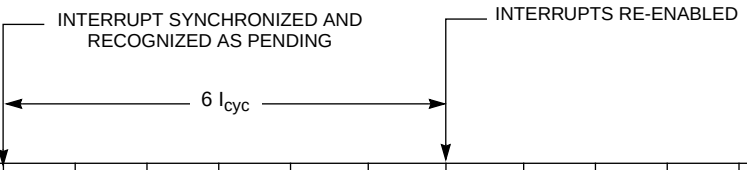
(b) Program Controller Pipeline

Figure 8-6 Fast Interrupt Service Routine

maximum rate at which the DSP56000/DSP56001 will respond to interrupts — namely, one interrupt every six instructions (six instruction cycles if all six instructions are one instruction cycle each). Since some instructions take more than one instruction cycle, the



(a) Instruction Fetches from Memory

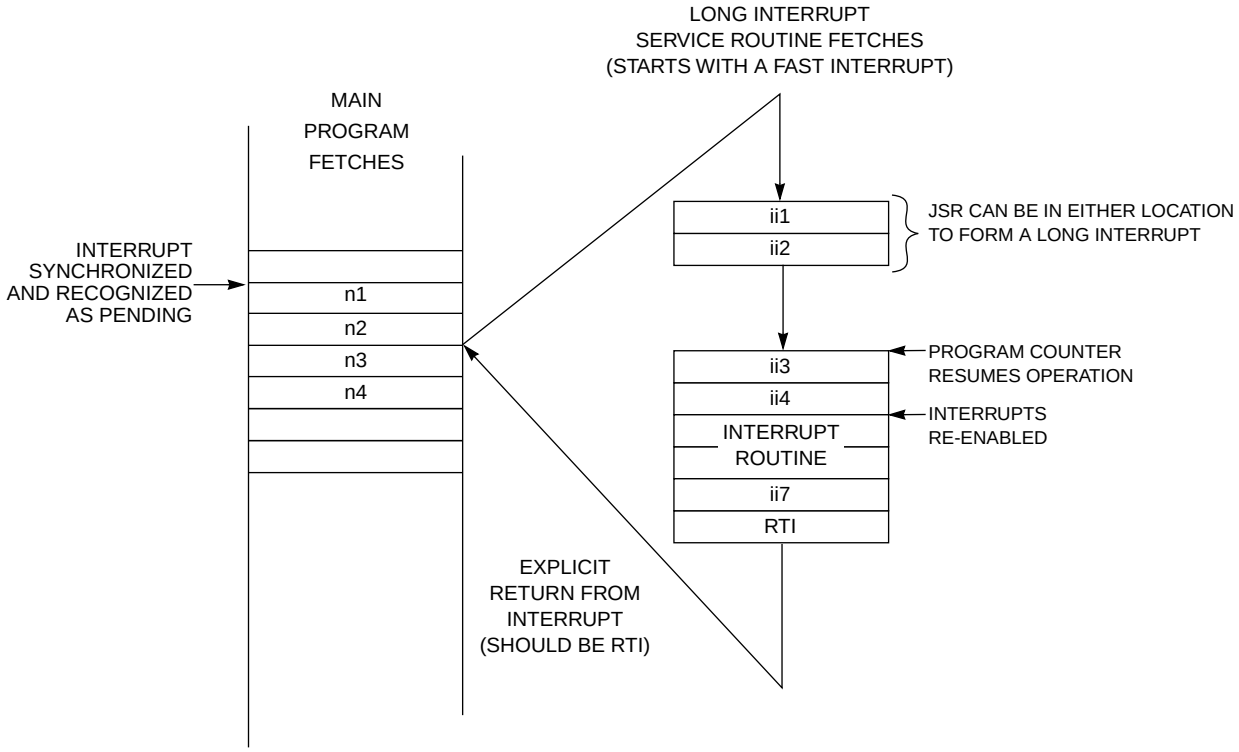


INTERRUPT CONTROL CYCLE 1	i						i					
INTERRUPT CONTROL CYCLE 2		i						i				
FETCH	n1	n2	ii1	ii2	n3	n4	n5	n6	ii1	ii2		
DECODE		n1	n2	ii1	ii2	n3	n4	n5	n6	ii1	ii2	
EXECUTE			n1	n2	ii1	ii2	n3	n4	n5	n6	ii1	ii2
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 8-7 Two Consecutive Fast Interrupts



(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	ii1	ii2	ii3	ii4	ii5	ii6	ii7	RTI	—	n3	n4		
DECODE		n1	n2	ii1	ii2	ii3	ii4	ii5	ii6	ii7	RTI	NOP	n3	n4	
EXECUTE			n1	n2	ii1	ii2	ii3	ii4	ii5	ii6	ii7	RTI	NOP	n3	n4
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 8-8 Long Interrupt Service Routine

minimum number of instructions between two interrupts may be more than six instruction cycles.

Execution of a fast interrupt routine always conforms to the following rules:

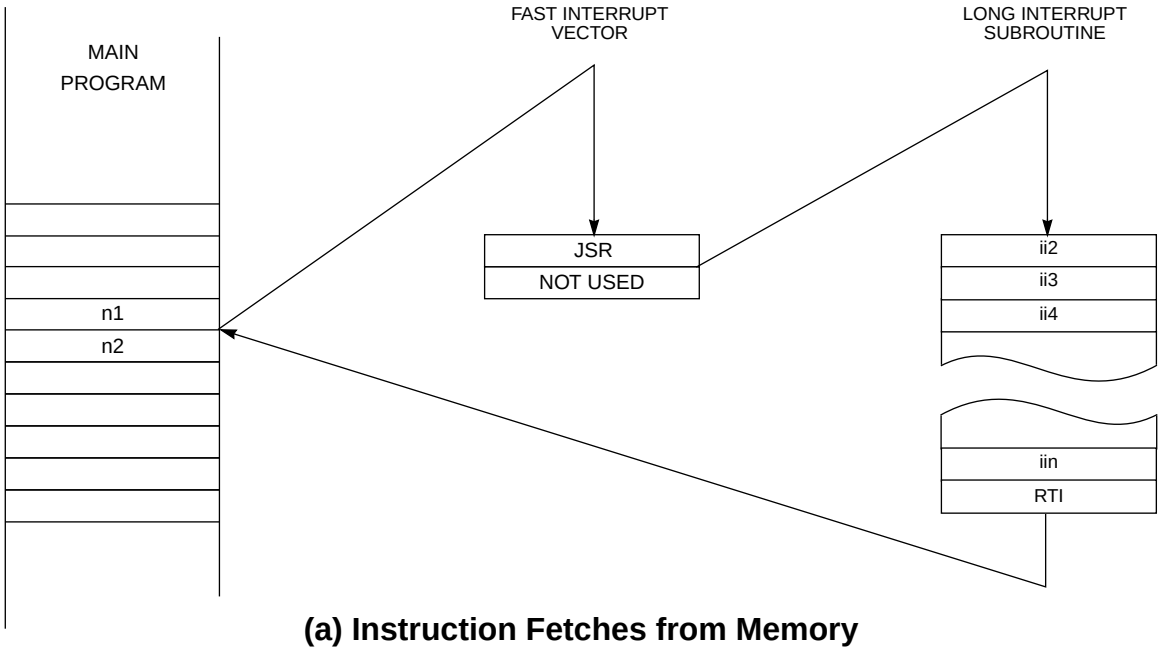
1. A JSR to the starting address of the interrupt service routine is not located at one of the two interrupt vector addresses.
2. The processor status is not saved.
3. The fast interrupt routine may (but should not) modify the status of the normal instruction stream.
4. The fast interrupt routine may contain any single two-word instruction or any two one-word instructions except SWI, STOP, and WAIT.
5. The PC, which contains the address of the next instruction to be executed in normal processing remains unchanged during a fast interrupt routine.
6. The fast interrupt returns without an RTI.
7. Normal instruction fetching resumes using the PC following the completion of the fast interrupt routine.
8. A fast interrupt is not interruptable.
9. A JSR instruction within the fast interrupt routine forms a long interrupt routine.
10. The primary application is to move data between memory and I/O devices.

Execution of a long interrupt routine always adheres to the following rules:

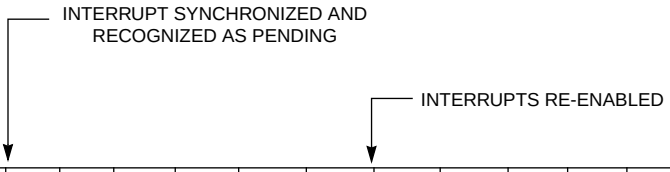
1. A JSR to the starting address of the interrupt service routine is located at one of the two interrupt vector addresses.
2. During execution of the JSR instruction, the PC and SR are stacked. The interrupt mask bits of the SR are updated to mask interrupts of the same or lower priority. The loop flag, trace bit, and scaling mode bits are reset.
3. The first instruction word of the next interrupt service (of higher IPL) will reach the decoder only after the decoding of at least four instructions following the decoding of the first instruction of the previous interrupt.
4. The interrupt service routine can be interrupted — i.e., nested interrupts are supported.
5. The long interrupt routine, which can be any length, should be terminated by an RTI, which restores the PC and SR from the stack.

Figure 8-8 illustrates the effect of a long interrupt routine on the instruction pipeline. A short JSR (a JSR with 12-bit absolute address) is used to form the long interrupt routine. For this example, word 6 of the long interrupt routine is an RTI. The point at which interrupts are re-enabled and subsequent interrupts are allowed is shown to illustrate the noninterruptable nature of the early instructions in the long interrupt service routine.

Either one of the two instructions of the fast interrupt can be the JSR instruction that forms the long interrupt. Figure 8-9 and Figure 8-10 show the two possible cases. If the first fast interrupt vector instruction is the JSR, the second instruction is never used.



(a) Instruction Fetches from Memory



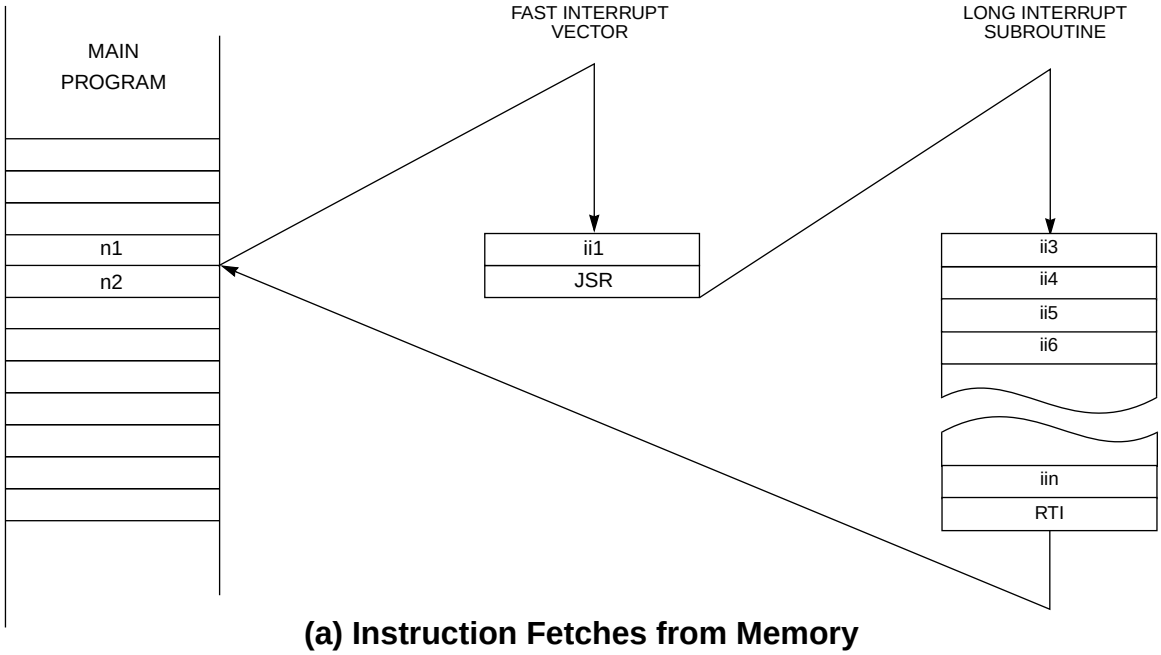
INTERRUPT CONTROL CYCLE 1	i												
INTERRUPT CONTROL CYCLE 2		i											
FETCH		n1	JSR	—	ii2	ii3	ii4	iin	RTI	—	n2		
DECODE			n1	JSR	NOP	ii2	ii3	ii4	iin	RTI	NOP	n2	
EXECUTE				n1	JSR	NOP	ii2	ii3	ii4	iin	RTI	NOP	n2
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 8-9 JSR First Instruction of a Fast Interrupt

An REP instruction is 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 8-11). During the execution of n2 in Figure 8-11, no interrupts will be serviced. When LC finally decrements to one, the fetches are reinitiated, and pending interrupts can be serviced.



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

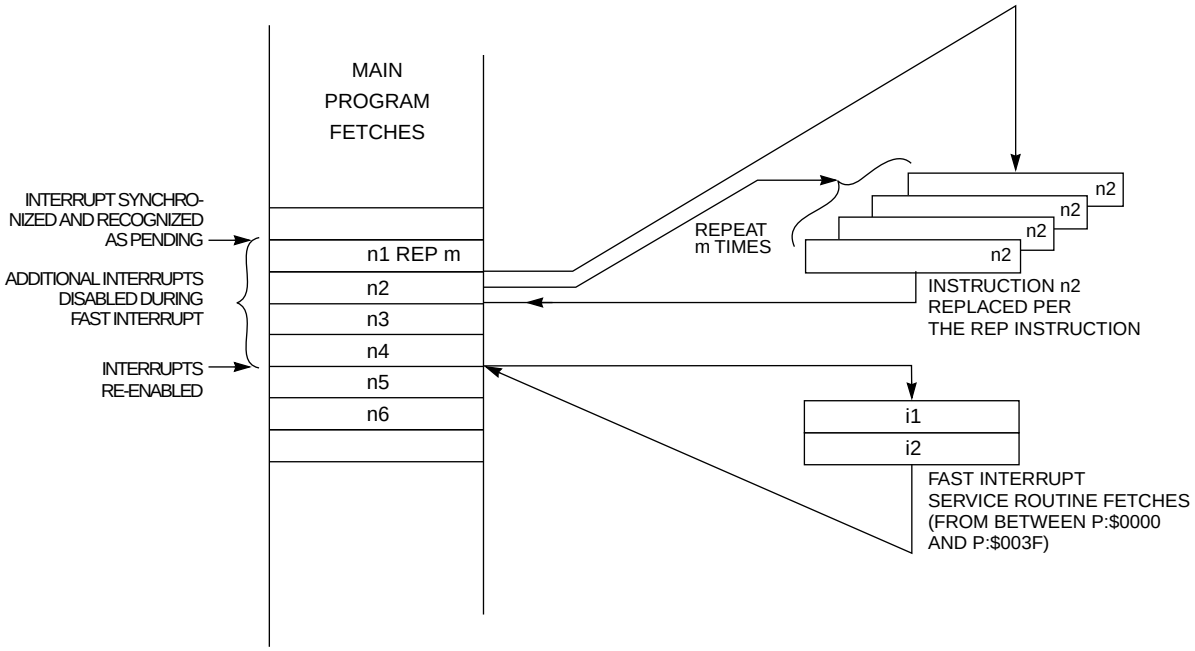
INTERRUPTS RE-ENABLED

NTERRUPT CONTROL CYCLE 1	i														
NTERRUPT CONTROL CYCLE 2		i													
ETCH		n1	ii1	JSR	—	ii3	ii4	ii5		iin	RTI	—	n2		
ECODE			n1	ii1	JSR	NOP	ii3	ii4	ii5	ii6	iin	RTI	NOP	n2	
XECUTE				n1	ii1	JSR	NOP	ii3	ii4	ii5	ii6	iin	RTI	NOP	n2
NSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

= INTERRUPT
i = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 8-10 JSR Second Instruction of a Fast Interrupt



i = INTERRUPT INSTRUCTION
n = NORMAL INSTRUCTION

(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPTS RE-ENABLED

INTERRUPT CONTROL CYCLE 1	i						i					
INTERRUPT CONTROL CYCLE 2		i%						i				
FETCH	REP	n2	n3					n4	ii1	ii2	n5	n6
DECODE		REP	NOP	n2	n2	n2	n2	n3	n4	ii1	ii2	n5
EXECUTE			REP	NOP	n2	n2	n2	n2	n3	n4	ii1	ii2
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

Figure 8-11 Interrupting an REP Instruction

Sequential REP packages will cause pending interrupts to be rejected until the sequence

of REP packages ends. REP packages are not interruptable because the instruction being repeated is not refetched. While that instruction is repeating, no instructions are fetched or decoded, and an interrupt can not be inserted. For example, in Figure 8-12, if n1, n3, and n5 are all REP instructions, no interrupts will be serviced until the last REP instruction (n5 and its repeated instruction, n6) completes execution.

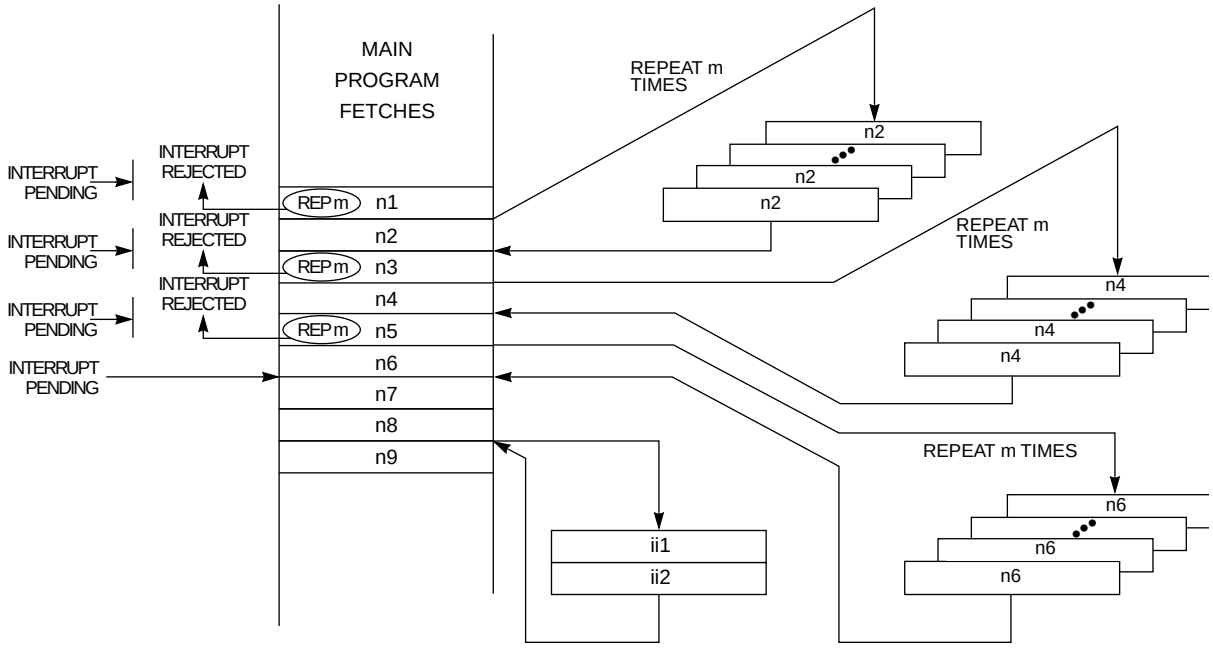
8.3 RESET PROCESSING STATE

The reset processing state is entered in response to the external RESET pin being asserted (a hardware reset). Upon entering the reset state (see Figure 8-13): 1) internal peripheral devices are reset, and their pins revert to general-purpose I/O pins; 2) the modifier registers are set to $FFFF; 3) the interrupt priority register is cleared; 4) the BCR is set to $FFFF, thereby inserting 15 wait states in all external memory accesses; 5) the stack pointer is cleared; 6) the scaling mode, trace mode, loop flag, and condition code bits of the SR are cleared, and the interrupt mask bits of the SR are set; 7) the data ROM enable bit, the stop delay bit, and the memory strobe bit are cleared; and 8) the DSP remains in the reset state until RESET is deasserted. Upon leaving the reset state 9), the chip operating mode bits of the OMR are loaded from the external mode select pins (MODA, MODB), and 10) program execution begins at program memory address $E000 in normal expanded mode or at $0000 in all other operation modes. The first instruction must be fetched and then decoded before executing. Therefore, the first instruction execution is two instruction cycles after the first instruction fetch.

Figure 8-14 is a copy of the output from the DSP56000/DSP56001 simulator showing all of the DSP56000/DSP56001 registers before the hardware reset and showing only the registers that were written by the hardware reset after the reset occurred. The instructions executed are as follows:

1. Reset s — Resets the simulator.
2. Change OMR 0 — Puts the DSP56000/DSP56001 in mode 0.
3. Display all — Displays all registers. Note that OMR=$00.
4. Reset d — Is a hardware reset.
5. Display w — Causes the display command to only display the registers that were written in the last instruction.
6. Display — Displays the contents of the registers that were written by the hardware reset.

The OMR changed from $00 to $02, which is mode 2, because the MODA/$\overline{\text{IRQA}}$ and MODB/$\overline{\text{IRQB}}$ pins are set to a one and zero, respectively (binary 2) in the simulator. If the DSP had been in any other mode, the result would have been the same. The X: memory locations written to are the memory locations of the peripheral registers. The internal peripheral registers are memory mapped between X:$FFC0 and X:$FFFF.



(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	REP				n4	REP				n6	n7				n8	ii1	ii2	n9		
DECODE		REP	NOP	n2	n2	n2	REP	NOP	n4	n4	n4	REP	NOP	n6	n6	n6	n7	n8	ii1	ii2	n9	
EXECUTE			REP	NOP	n2	n2	n2	REP	NOP	n4	n4	n4	REP	NOP	n6	n6	n6	n7	n8	ii1	ii2	n9	
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD
i% = INTERRUPT REJECTED

(b) Program Controller Pipeline

Figure 8-12 Interrupting Sequential REP Instructions

The internal peripheral devices (HI, SSI, SCI, and ports A, B, and C) can be reset by sev-

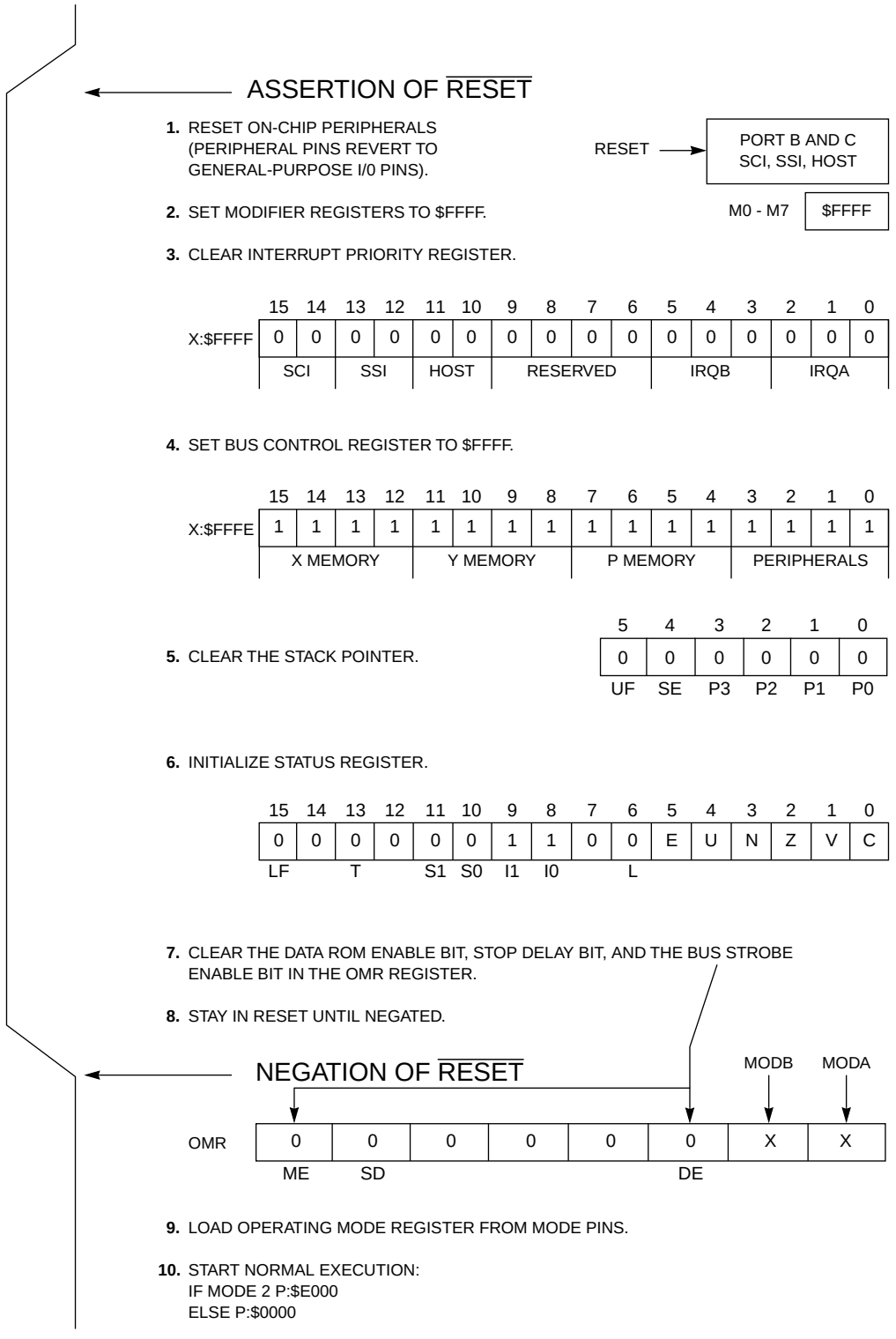


Figure 8-13 Reset Sequence

eral methods — hardware (HW) reset, software (SW) reset, individual (I) reset, and stop

```
reset s
change omr 0
display all


  x=          $000000000000  y=          $000000000000
  a=      $00000000000000  b=      $00000000000000
              x1=    $000000   x0=    $000000    r7=      $0000     n7= $0000     m7= $FFFF
              y1=    $000000   y0=    $000000    r6=      $0000     n6= $0000     m6= $FFFF
 a2=    $00 a1=    $000000   a0=    $000000    r5=      $0000     n5= $0000     m5= $FFFF
 b2=    $00 b1=    $000000   b0=    $000000    r4=      $0000     n4= $0000     m4= $FFFF
                                               r3=      $0000     n3= $0000     m3= $FFFF
  pc=  $E00       sr=  $0300 omr=       $00    r2=      $0000     n2= $0000     m2= $FFFF
   la= $0000      lc=  $0000                   r1=      $0000     n1= $0000     m1= $FFFF
 ssh= $0000      ssl=  $0000  sp=        $00    r0=      $0000     n0= $0000     m0= $FFFF
 pbc=     $0 pbddr=  $0000 pbd=       $0000    pcd=     $0000  pcddr= $0000     pcc= $FFFF
  ipr= $0000     bcr=  $FFFF  htx=     $000000  hrx=     $000000   hsr=   $02     hcr=   $00
  icr=   $00     cvr=    $12  isr=       $06    ivr=       $0F
 rxh=   $00     rxm=    $00  rxl=       $00    txh=       $00     txm=   $00     txl=   $00
 ssr=   $03     scr=  $0000  stx=       $00    srx=       $00    sccr= $0000    stxa=   $00
  tsr=   $00  ssisr=    $40   tx=     $000000  rx=     $000000    cra= $0000     crb= $0000
cyc=000000    ictr= 000000 cnt1=       000000  cnt2=     000000  cnt3=000000   cnt4=000000
P:$E000 000000          =NOP

reset d
display w
display

                                                                               m7= $FFFF
                                                                               m6= $FFFF
                                                                               m5= $FFFF
                                                                               m4= $FFFF
                                                                               m3= $FFFF
  pc= $E000      sr=  $0300 omr=       $02                                     m2= $FFFF
                                                                               m1= $FFFF
                                    sp=        $00                             m0= $FFFF
 pbc=     $0 pbddr=  $0000                                          pcddr= $0000     pcc= $0000
  ipr= $0000     bcr=  $FFFF                                          hsr=   $02     hcr=   $00
  icr=   $00     cvr=    $12  isr=       $06    ivr=       $0F
 ssr=   $03     scr=  $0000                                         sccr= $0000
                ssisr=    $40                                        cra= $0000     crb= $0000
 X:$FFE3       $000000
 X:$FFE8       $000000   $000002
 X:$FFEC       $000000   $000000       $000040
 X:$FFE0       $000000   $000003       $000000
 X:$FFFF       $000000
 P:$E000 000000          =NOP
```

Figure 8-14 Reset When OMR=0

(ST) reset. Depending on the type of reset, the registers of these devices will be affected differently (see SECTION 9 PORT A, SECTION 10 PORT B, and SECTION 11 PORT C for additional information on the internal peripherals). Tables 8-7 – 8-11 show how each bit in these registers is affected by the various resets. The HI is programmed for both the DSP56000/DSP56001 side of the interface and the host processor side of the interface.

The symbols used are as follows:

HW – Hardware reset is caused by asserting the external pin $\overline{\text{RESET}}$.
SW – Software reset is caused by executing the RESET instruction.

I – Individual reset is caused by all of the I/O pins for a given internal I/O device being configured for general-purpose I/O. These I/O devices are the HI, SSI, and SCI. The conditions for these resets are:

1. SSI individual reset occurs when port C control register bits 3 – 8 are set to zero.
2. SCI individual reset occurs when port C control register bits 0 – 2 are set to zero.
3. HI individual reset occurs when port B control register bit 0 is set to zero.

ST – Stop reset is caused by executing the STOP instruction.

1 – The bit is set during the xx reset.

0 – The bit is clear during the xx reset.

— – The bit is not changed during the xx reset.

The definitions for individual reset for the ports A, B, and C register settings during indi-

Table 8-7 HI Reset Effects — DSP56000/ DSP56001Programming Model

Register Name	Register Data Bits	HW Reset	SW Reset	I Reset	ST Reset
HCR X:$FFE8	HF(3-2)	0	0	—	—
	HCIE	0	0	—	—
	HTIE	0	0	—	—
	HRIE	0	0	—	—
	DMA	0	0	0	0
	HF (1-0)	0	0	0	0

Table 8-8 HI Reset Effects — Host Processor Programming Model

Register Name	Register Data Bits	HW Reset	SW Reset	I Reset	ST Reset
ICR $0	INIT	0	0	0	0
	HM(1-0)	0	0	0	0
	TREQ	0	0	0	0
	RREQ	0	0	0	0
	HF(1-0)	0	0	0	0
CVR $1	HC	0	0	0	0
	HV (4-0)	$12	$12	$12	$12
ISR $2	HREQ	0	0	0	0
	DMA	0	0	0	0
	HF(3-2)	0	0	—	—
	TRDY	1	1	1	1
	TXDE	1	1	1	1
	RXDF	0	0	0	0
IVR $3	IV(7-0)	$0F	$0F	—	—
RX $5, 6, 7	RXH(23-16)	—	—	—	—
	RXM(15-8)	—	—	—	—
	RXL	—	—	—	—
TX $5, 6, 7	TXH(23-16)	—	—	—	—
	TXM(15-8)	—	—	—	—
	TXL(7-0)	—	—	—	—

vidual reset are shown in Table 8-11.

Table 8-9 SSI Reset Effects

Register Name	Register Data Bits	HW Reset	SW Reset	I Reset	ST Reset
CRA X:$FFEC	WL(2-0)	0	0	—	—
	PSR	0	0	—	—
	DC(4-0)	0	0	—	—
	PM(7-0)	0	0	—	—
CRB X:$FFED	RIE	0	0	—	—
	TIE	0	0	—	—
	RE	0	0	—	—
	TE	0	0	—	—
	MOD	0	0	—	—
	GCK	0	0	—	—
	SYN	0	0	—	—
	FSL0	0	0	—	—
	FSL1	0	0	—	—
	SCKD	0	0	—	—
	SCD(2-0)	0	0	—	—
	OF(1-0)	0	0	—	—
SR X:$FFEE	RDF	0	0	0	0
	TDE	1	1	1	1
	ROE	0	0	0	0
	TUE	0	0	0	0
	RFS	0	0	0	0
	TFS	0	0	0	0
	IF(1-0)	0	0	0	0
RX X:$FFEF	RDR(23-0)	—	—	—	—
TX X:$FFEF	TDR(23-0)	—	—	—	—
SRSR*	RDR(23-0)	—	—	—	—
STSR**	RDR(23-0)	—	—	—	—

*SRSR—SSI serial receive shift register
**STSR—SSI serial transmit shift register

Table 8-10 SCI Reset Effects

Register Name	Register Data Bits	HW Reset	SW Reset	I Reset	ST Reset
SCR X:$FFF0	SCKP	0	0	—	—
	TMIE	0	0	—	—
	TIE	0	0	—	—
	RIE	0	0	—	—
	ILIE	0	0	—	—
	TE	0	0	—	—
	RE	0	0	—	—
	WOMS	0	0	—	—
	RWU	0	0	—	—
	WAKE	0	0	—	—
	SBK	0	0	—	—
	SSFTD	0	0	—	—
	WDS(2-0)	0	0	—	—
SSR X:$FFF1	R8	0	0	0	0
	FE	0	0	0	0
	PE	0	0	0	0
	OR	0	0	0	0
	IDLE	0	0	0	0
	RDRF	0	0	0	0
	TDRE	1	1	1	1
	TRNE	1	1	1	1
SCCR X:$FFF2	TCM	0	0	—	—
	RCM	0	0	—	—
	SCP	0	0	—	—
	COD	0	0	—	—
	CD(11-0)	0	0	—	—
SRX X:$FFF4 X:$FFF5 X:$FFF6	STX(23-0) LOW MID HIGH	—	—	—	—
STX X:$FFF4 X:$FFF5 X:$FFF6 X:$FFF3	SRX(23-0) LOW MID HIGH STXA	—	—	—	—
SRSH*	SRSH(23-0)	—	—	—	—
STSH**	STSH(23-0)	—	—	—	—

*SRSH—SCI serial receive shift register
**STSH—SCI serial transmit shift register

Table 8-11 Ports A, B, and C Reset Effects

Register Name	Register Data Bits	HW Reset	SW Reset	I Reset	ST Reset	Comments

8.4 WAIT PROCESSING STATE

The wait processing state is a low power-consumption state entered by execution of the WAIT instruction. In the wait state, the internal clock is disabled from all internal circuitry except the internal peripherals (e.g., the interrupt controller, the SCI, SSI, and HI). All internal processing is halted until an unmasked interrupt occurs or until the DSP is reset. The $\overline{BR}$/$\overline{BG}$ circuits remain active during the wait state.

The wait state is one of two low power-consumption states. As a general rule, the normal operating current for the DSP56000/DSP56001 is typically less than 100 ma for a 20.5-MHz clock. The current is typically reduced to less than 10 ma (for a 20.5-MHz clock) in the wait state and to less than 1.0 ma (independent of the clock frequency) in the stop state. See the DSP56001 Advance Information Data Sheet (ADI1290) for exact figures. There are several other ways that power can be reduced. Power consumption varies linearly with both clock frequency and power-supply voltage. Changing clock frequency from 20 MHz[lz to 4 MHz can reduce power consumption 75 percent (i.e., linearly with decreasing frequency). Changing the memory wait states from 0 to 15 can reduce power consumption by more than half during external memory accesses.

Figure 8-15 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 8-15 shows the result of a fast interrupt bringing the pro-

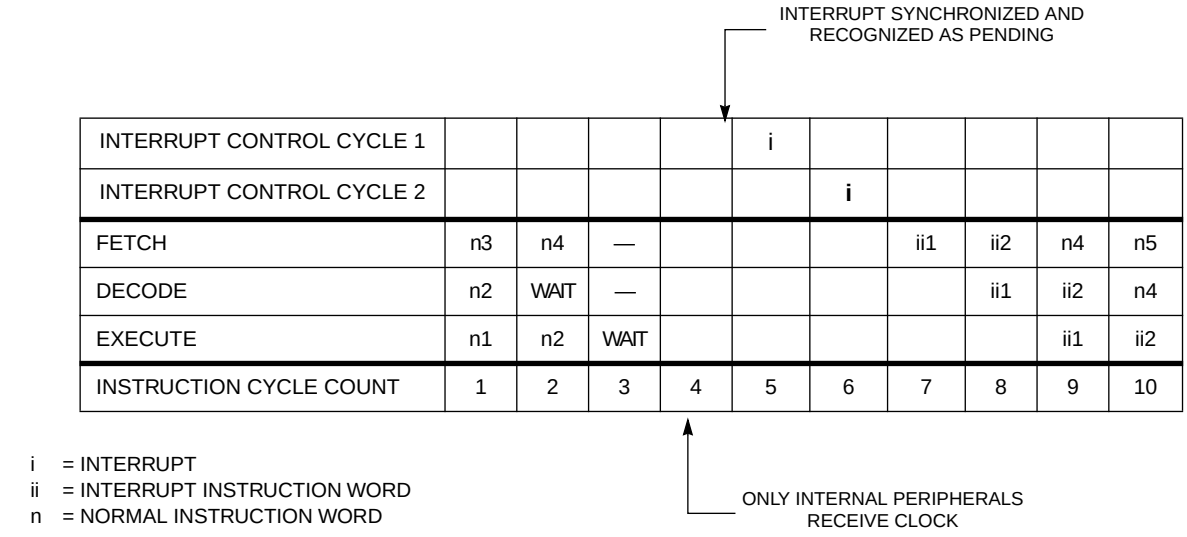
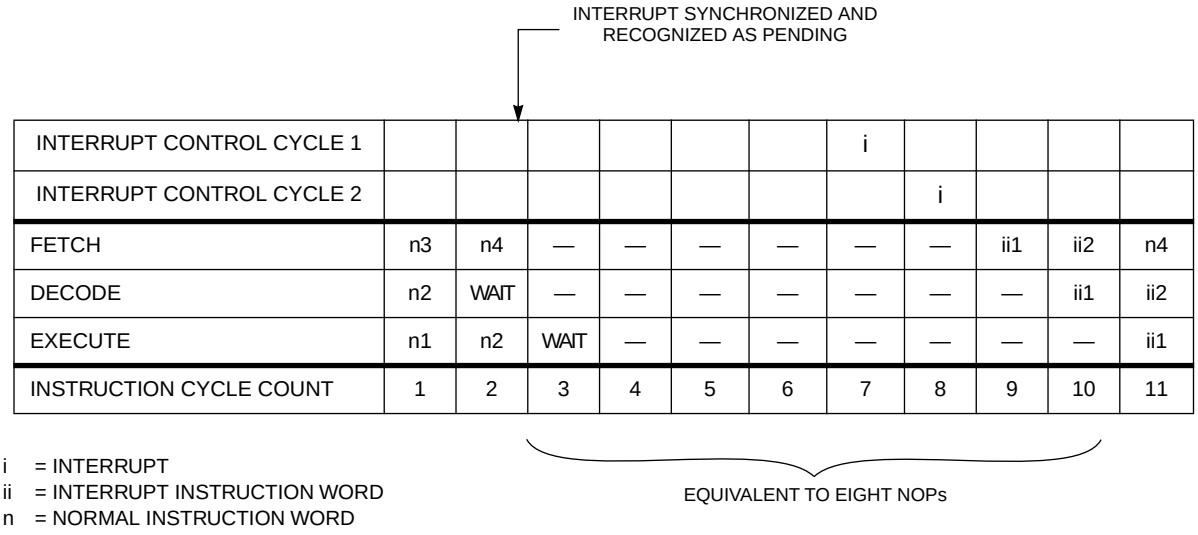


Figure 8-15 Wait Instruction Timing

cessor 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 8-16 shows an example of the WAIT instruction being executed at the same time that an interrupt is pending. Instruction n4 is aborted as before. There is a five-instruction-cycle delay caused by the WAIT instruction; then the interrupt is processed normally.



i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

Figure 8-16 Simultaneous Wait Instruction and Interrupt

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.

During the wait state, the $\overline{BR}$/$\overline{BG}$ circuits remain active. Before $\overline{BR}$ is asserted (see Table 8-12), all port A signals are driven. While the port is inactive, the control signals are deasserted, the data signals are inputs, and the address signals remain as the last address

Table 8-12 $\overline{BR}$/$\overline{BG}$ During WAIT

Signal	Before $\overline{BR}$ Asserted	While $\overline{BG}$ Asserted	After $\overline{BR}$ Deasserted	After Return to Normal State	After First External Access
$\overline{PS}$	Driven	Three-state	Three-state	Driven	Driven
$\overline{DS}$	Driven	Three-state	Three-state	Driven	Driven
X/$\overline{Y}$	Driven	Three-state	Three-state	Driven	Driven
$\overline{RD}$	Driven	Three-state	Driven	Driven	Driven
$\overline{WR}$	Driven	Three-state	Driven	Driven	Driven
Data	Driven	Three-state	Three-state	Three-state	Driven
Address	Driven	Three-state	Three-state	Three-state	Driven

read or written. The signal timing during a read or write is given in the timing diagrams in the DSP56001 Advance Information Data Sheet (ADI1290). When $\overline{BG}$ is asserted, all signals are three-stated (high impedance). Immediately after $\overline{BR}$ is deasserted, the RD and WR signals are driven and are deasserted; all other signals remain in the high-impedance state. During the first T0 clock state following the exit from the wait state, control signals PS, DS, and X/Y are again driven; the data and address signals remain in the high-impedance state. During the first external access, all signals return to their normal operating mode.

8.5 STOP PROCESSING STATE

The stop processing state, which is the lowest power-consumption state, is entered by the execution of the STOP instruction. In the stop state, the clock oscillator is gated off; whereas, in the wait mode, the clock oscillator remains active. The chip clears all peripheral interrupts (HI, SSI, and SCI) and external interrupts ($\overline{IRQA}$, $\overline{IRQB}$, and NMI) when entering the stop state. Trace or stack errors that were pending, remain pending. The priority levels of the peripherals remain as they were before the STOP instruction was executed. The SCI, SSI, and HI are held in their respective individual reset states while in the stop state.

All activity in the processor is halted until one of the following actions occurs:

1. A low level is applied to the $\overline{IRQA}$ pin.

2. A low level is applied to the $\overline{\text{RESET}}$ pin.

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

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

The STOP instruction is fetched in stop cycle 1 of Figure 8-17, 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

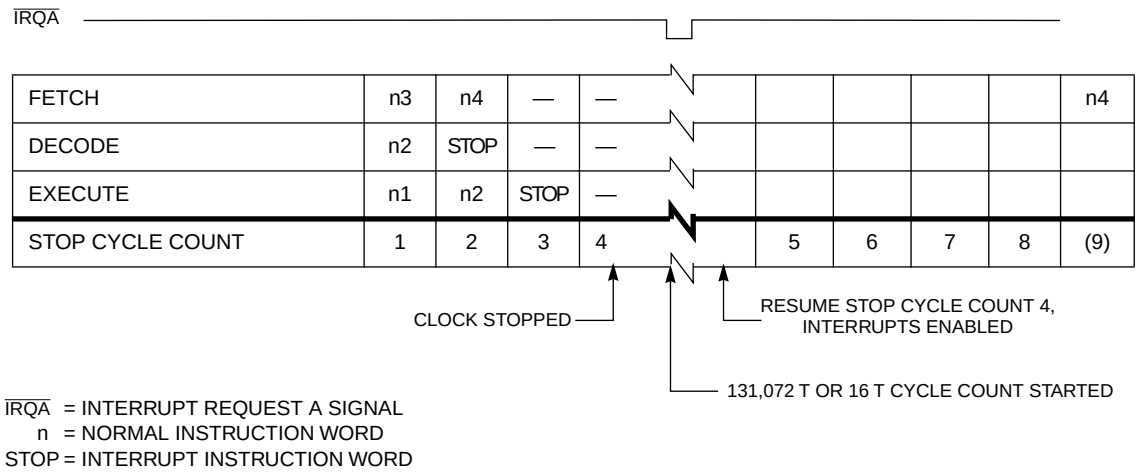


Figure 8-17 Simultaneous Wait Instruction and Interrupt

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 8-18 illustrates restarting the system by asserting the IRQA signal. If the exit from stop state was caused by a low level on the $\overline{\text{IRQA}}$ pin, then the processor will service the highest priority pending interrupt. If no interrupt is pending, then the processor resumes

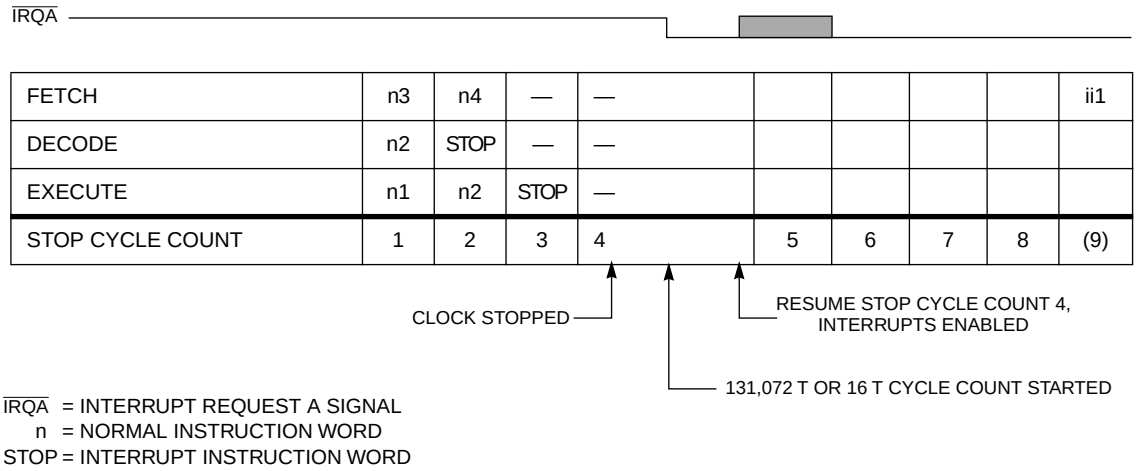


Figure 8-18 STOP Instruction Sequence

at the instruction following the STOP instruction that caused the entry into the stop state.

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. If the internal clock oscillator is used, the SD bit should be set to zero, which enables a delay count of 128K T cycles (131,072 T cycles) to allow the clock oscillator to stabilize. If a stable external clock is used, the SD bit may be set to one, which enables a 16 T cycle delay.

The following description assumes that SD=0 (the 128K T counter is used). During the 128K T count, interrupts are ignored until the last few count cycles. At this time, the interrupts are synchronized. At the end of the 128K T cycle delay period, the chip restarts instruction processing, stop cycle 4 is completed (interrupt arbitration occurs at this time), and stop cycles 5, 6, 7, and 8 are executed (it takes 17T from the end of the 128K T delay to the first instruction fetch). If the $\overline{\text{IRQA}}$ signal is released (pulled high) after a minimum of 4T but less than 128K T cycles, no $\overline{\text{IRQA}}$ interrupt will occur, and the instruction fetched after stop cycle 8 will be the next sequential instruction (n4 in Figure 8-18). An $\overline{\text{IRQA}}$ interrupt will be serviced (as shown in Figure 8-18) if 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, and 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 (see Figure 8-18), the processor will recognize the interrupt and fetch and execute the instructions at P:$0008 and P:$0009 (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.
2. Define $\overline{\text{IRQA}}$ priority as higher than the other sources and higher than the program priority.
3. Ensure that no stack error or trace interrupts are pending.
4. Execute the STOP instruction and enter the stop state.
5. Recover from the stop state by asserting the $\overline{\text{IRQA}}$ pin and holding it asserted for the whole clock recovery time. If it is low, the IRQA vector will be fetched. Also, the user must ensure that NMI will not be asserted during these last three cycles; otherwise, NMI will be serviced before $\overline{\text{IRQA}}$ because NMI priority is higher.
6. The exact elapsed time for clock recovery is unpredictable. The external device that asserts $\overline{\text{IRQA}}$ must wait for some positive feedback, such as specific memory access or a change in some predetermined I/O pin, before deasserting $\overline{\text{IRQA}}$.

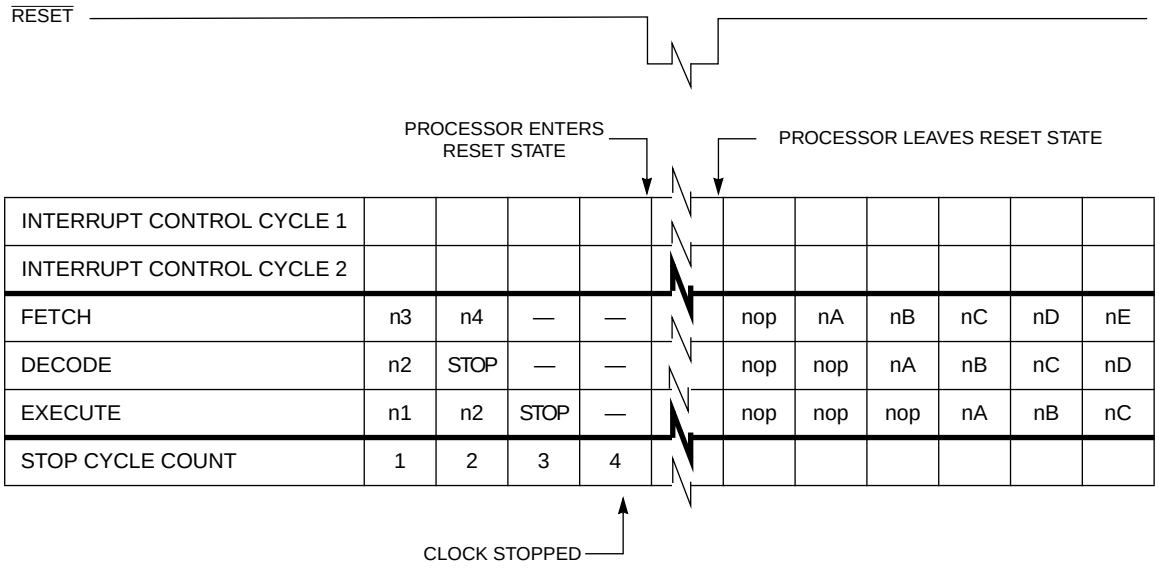
The STOP sequence totals 131,104 T cycles (if SD=0) or 48 T cycles (if SD=1) in addition to the period with no clocks from the stop fetch to the $\overline{\text{IRQA}}$ vector fetch (or next instruction). However, there is an additional delay if the internal oscillator is used. An indeterminant period 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.

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 looks like a 131,072 + 35 T cycle (or 51 T cycle) NOP, since the STOP instruction itself is eight instruction cycles long (32 T) and synchronization of $\overline{\text{IRQA}}$ is 3T which equals 35T.

A trace or stack error interrupt pending before entering the stop state is not cleared and will remain pending. During the clock stabilization delay, all peripheral and external interrupts are cleared and ignored (includes all SCI, SSI, HI, $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, and NMI interrupts, but not trace or stack error). If the SCI, SSI, or HI have interrupts enabled in 1) their respective control registers and 2) in the interrupt priority register, then interrupts like SCI transmitter empty will be immedi-

ately pending after the clock recovery delay and will be serviced before continuing with the next instruction. If peripheral interrupts must be disabled, the user should disable them with either the control registers or the interrupt priority register before the STOP instruction is executed.

If $\overline{\text{RESET}}$ is used to restart the processor (see Figure 8-19), the 128K T cycle delay counter



$\overline{\text{IRESET}}$ = INTERRUPT
n = NORMAL INSTRUCTION WORD
nA, nB, nC = INSTRUCTIONS IN RESET ROUTINE
STOP = INTERRUPT INSTRUCTION WORD

Figure 8-19 STOP Instruction Sequence Recovering with $\overline{\text{RESET}}$

would not be used, all pending interrupts would be discarded, and the processor would immediately enter the reset processing state as described in **8.3 RESET PROCESSING STATE**. The recommended stabilization time suggested in the data sheet for the clock ($\overline{\text{RESET}}$ should be asserted for this time) is only 50 T for a stabilized external clock but is the same 150,000 T for the internal oscillator. These stabilization times are recommended times but are not imposed by internal timers or time delays. The DSP fetches instructions immediately after exiting reset. If the user wishes to use the 128K T (or 16 T) delay counter, it can be started by asserting $\overline{\text{IRQA}}$ for a short time (about two clock cycles).

During the stop mode, the port A bus is frozen. The state of each pin immediately before executing the STOP instruction will be held until the DSP leaves the stop state. Port A is not three-stated, and the $\overline{\text{BR}}$/$\overline{\text{BG}}$ circuits are not operational. However, port A will remain three-stated if $\overline{\text{BG}}$ was asserted before the STOP instruction was executed. One way to release the port A bus for use while the DSP is in the stop state is to use a port B or port C pin to initiate a bus request before executing the STOP instruction.

SECTION 9
PORT A

Port A is the memory expansion port that can be used for either memory expansion or for memory-mapped I/O (see 2.9.1 Expansion Port (Port A)). A number of features make port A versatile and easy to use. These features provide a low-parts-count connection with fast memories, slow memories/devices, and multiple bus master systems.

The port A data bus is 24 bits wide with a separate 16-bit address bus capable of a sustained rate of one memory access per machine cycle (using no-wait-state memory). External memory is divided into three 64K-word X 24-bit spaces – X:, Y:, and P:. An internal wait-state generator can be programmed to insert up to15 wait states if access to slower memory or I/O devices is required. A bus wait signal allows an external device to control the number of wait states inserted in a bus access operation. Bus arbitration signals allow an external device (e.g., a DMA controller or another processor) use of the bus while internal operations continue using the internal memories. Two power-reduction features are specific to port A. The first power-reduction feature is that accessing the internal memory spaces does not toggle the external memory signals, eliminating unneeded switching current. The second power-reduction feature is that, if lower memory speed is acceptable, wait states can be added to external memory accesses to significantly reduce power while accessing those memories.

9.1 PORT A INTERFACE

One or more of the digital signal processor (DSP) memory sources (X data memory, Y data memory, and program memory) can be accessed during the execution of an instruction. Each of these memory sources may be either internal or external to the DSP. Three address buses (XAB, YAB, and PAB) and four data buses (XDB, YDB, PDB, and GDB) are available for internal memory accesses during one instruction cycle, but only one address bus and one data bus (port A) are available for external memory accesses. If all memory sources are internal to the DSP, one or more of the three memory sources may be accessed in one instruction cycle (i.e., program memory access or program memory access plus an X, Y, XY, or L memory reference). However, when one or more of the memories are external to the DSP56000/DSP56001, memory references may require

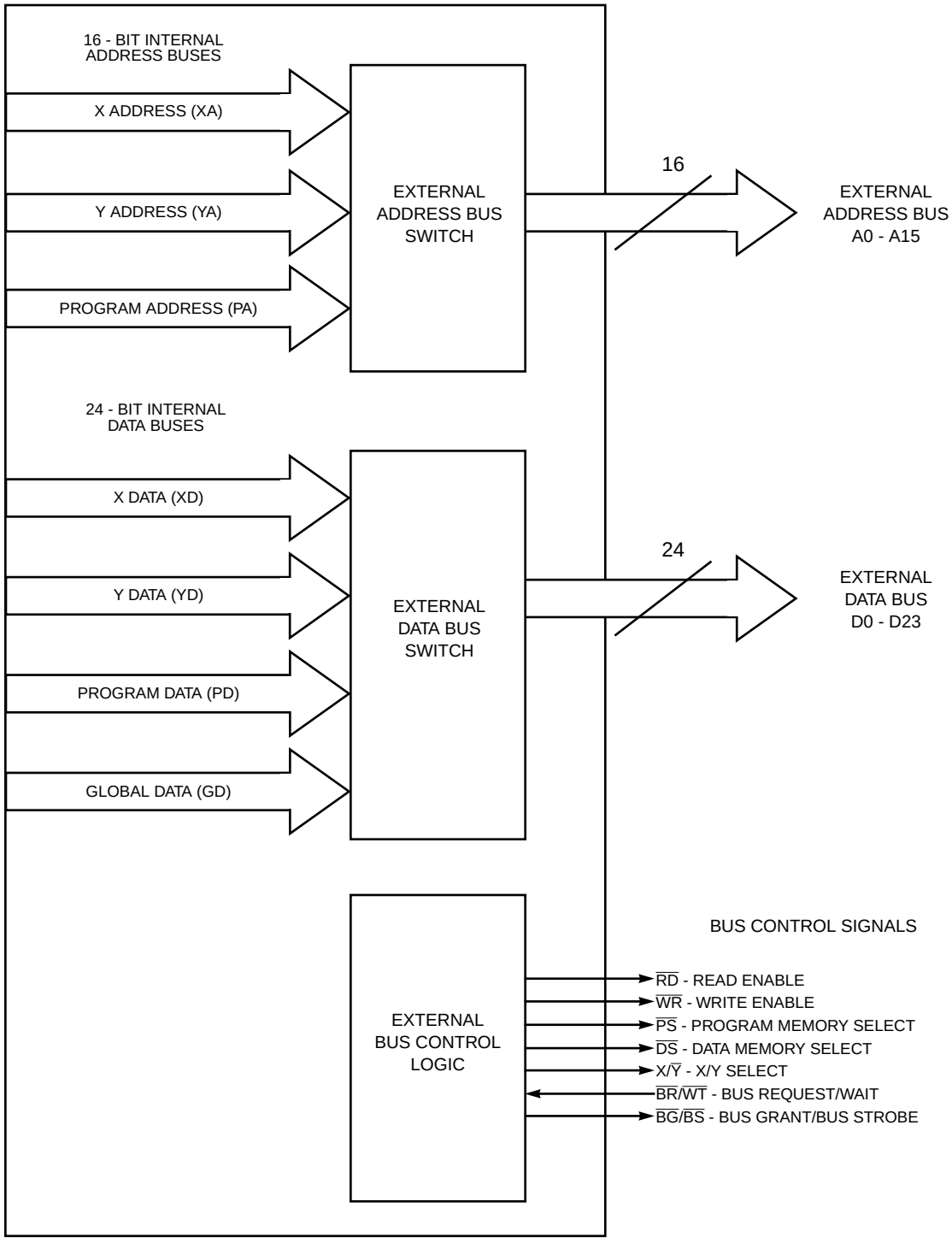


Figure 9-1 Port A Signals

additional instruction cycles because only one external memory access can occur per instruction cycle.

If more than one external access is required in one instruction cycle, the accesses will be made in the following priority: X memory, Y memory, and program memory. It takes one

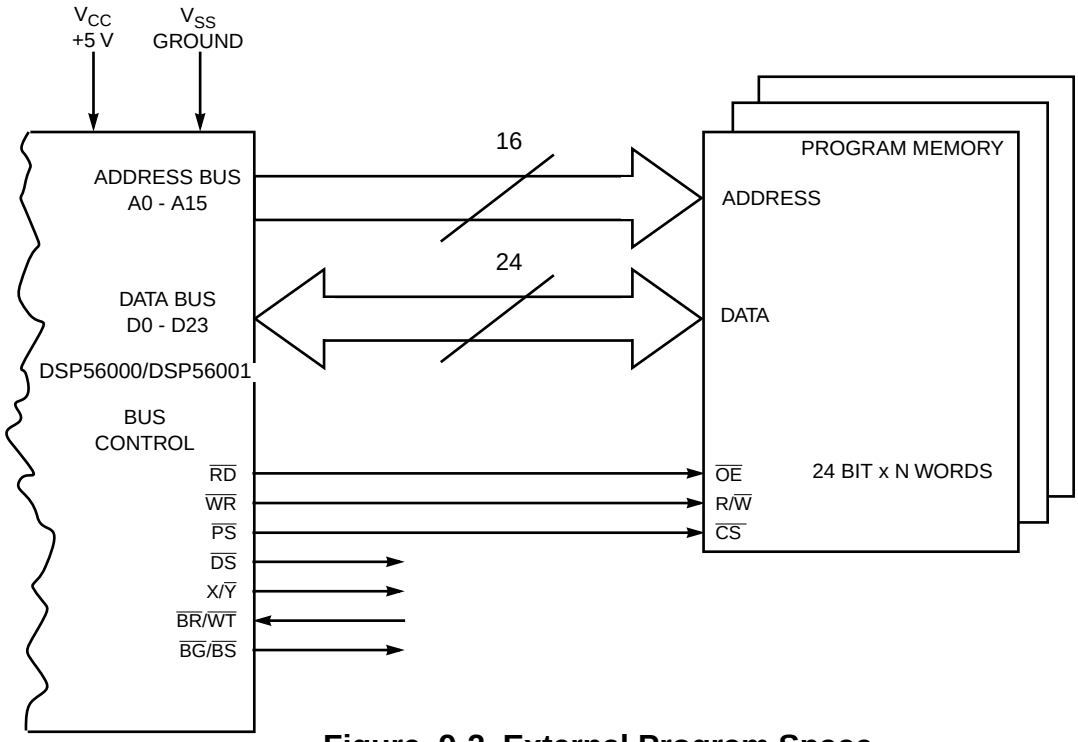


Figure 9-2 External Program Space

instruction cycle for each external memory access – i.e., one access can be executed in one instruction cycle, two accesses take two instruction cycles, etc. Since the external bus is only 24 bits wide, one XY or long external access will take two instruction cycles.

Figure 9-1 shows the port A signals divided into their three functional groups. The bus control signals can be subdivided into three additional groups: read/write control, address space selection, and bus access control. The read/write controls are self-descriptive. They can be used as decoded read and write controls, or, as seen in Figure 9-2, Figure 9-3, Figure 9-4, and Figure 9-6, the write signal can be used as the read/write control, and the read signal can be used as an output enable (or data enable) control for the memory. Decoding in this fashion simplifies connection to high-speed random-access memories (RAMs). The program memory select, data memory select, and X/Y select can be considered additional address signals, which extend the addressable memory from 64K words to 192K words

Since external logic delay is large relative to RAM timing margins, timing becomes more difficult as faster DSPs are introduced. The separate read and write strobes used by the DSP56000/DSP56001 are mutually exclusive, with a guard time between them to avoid two data buffers being enabled simultaneously. Other methods using external logic gates to generate the RAM control inputs require either faster RAM chips or external data buffers to avoid data bus buffer conflicts.

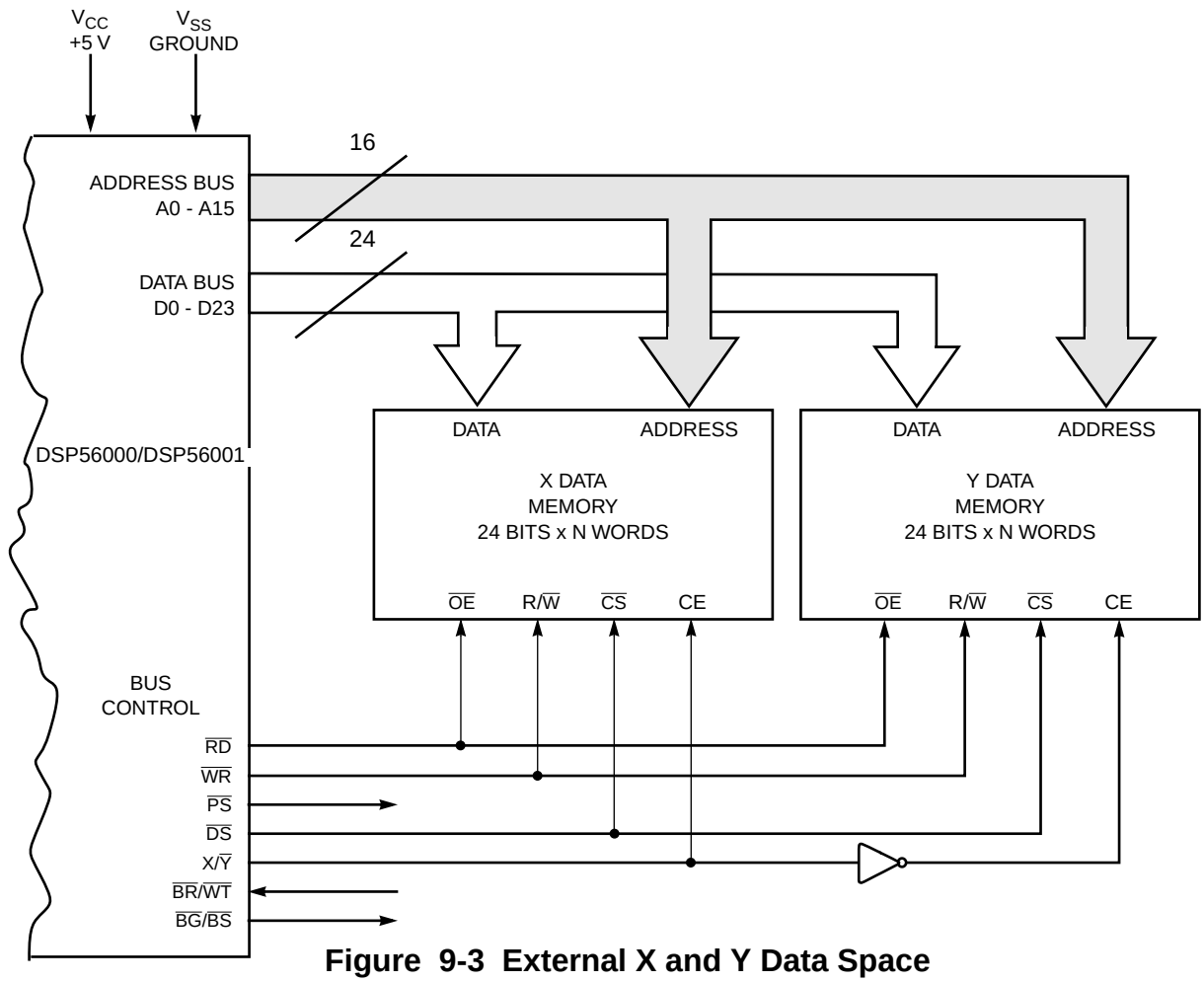


Figure 9-3 External X and Y Data Space

Additional DSP56000/DSP56001 peripherals can be memory mapped. An easy way to interface with MC6800 and MC68000 peripherals and to have an early read/write indication is to use the X/$\overline{Y}$ output pin as an early R/$\overline{W}$ indication. The peripheral chip select should be derived from the address lines and the data strobe so the peripheral registers appear in both X and Y data memory spaces at the same addresses. For a read operation, perform an X memory read:

```
MOVE X:PERIPHERAL,X0   ;X/Y signal is high.
```

For a write operation, perform a Y memory write:

```
MOVE X0,Y:PERIPHERAL   ;X/Y signal is low.
```

Since the X/$\overline{Y}$ output signal has the same timing as the address lines, it provides an early direction indication. The $\overline{RD}$ and $\overline{WR}$ signals are ANDed together to form a “data strobe” signal. The only restriction is that X and Y memory space must be external at the same address. Thus, the I/O short addressing mode and the MOVEP instruction cannot be used for this application. Otherwise, the hardware and software are trivial.

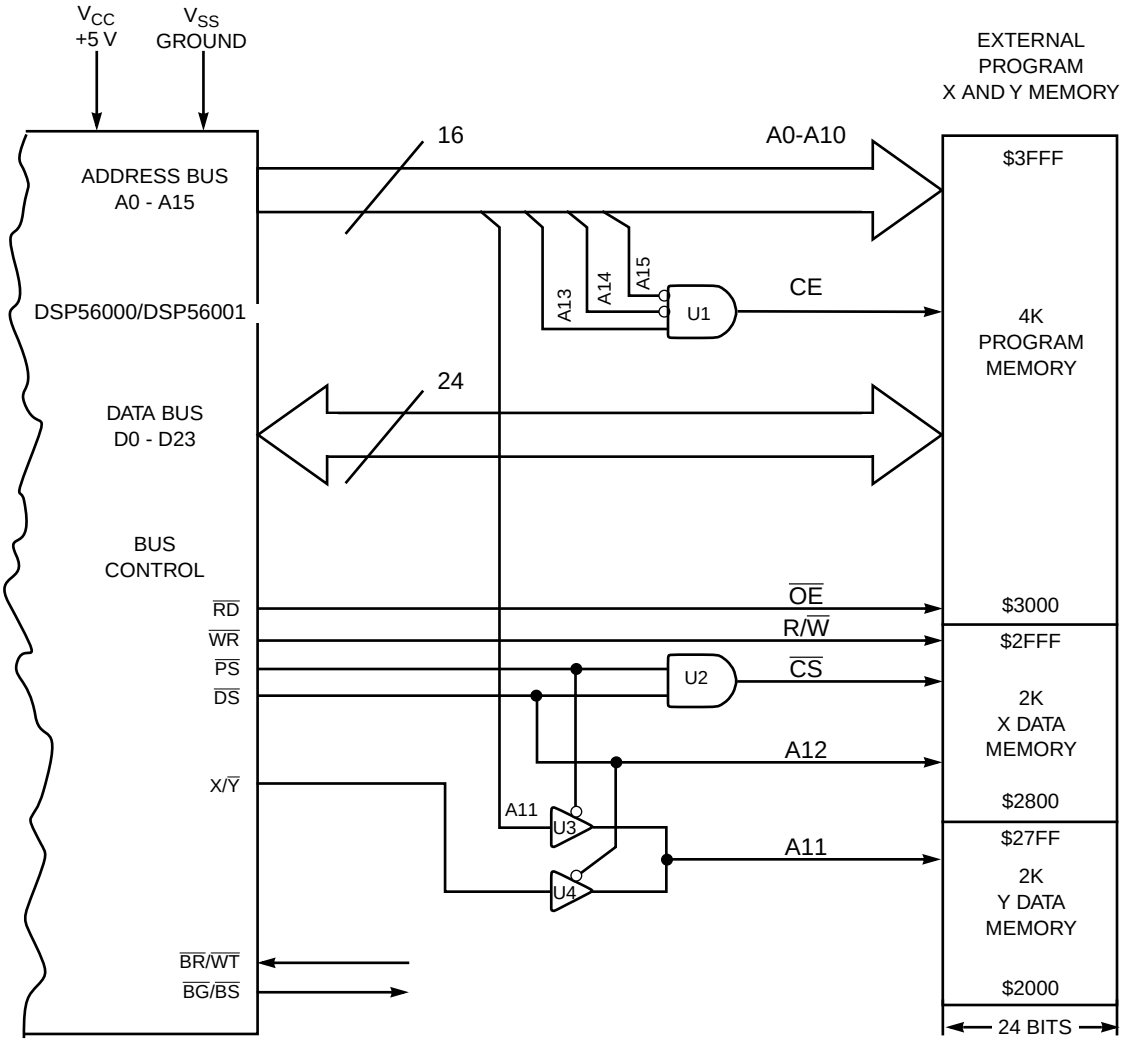
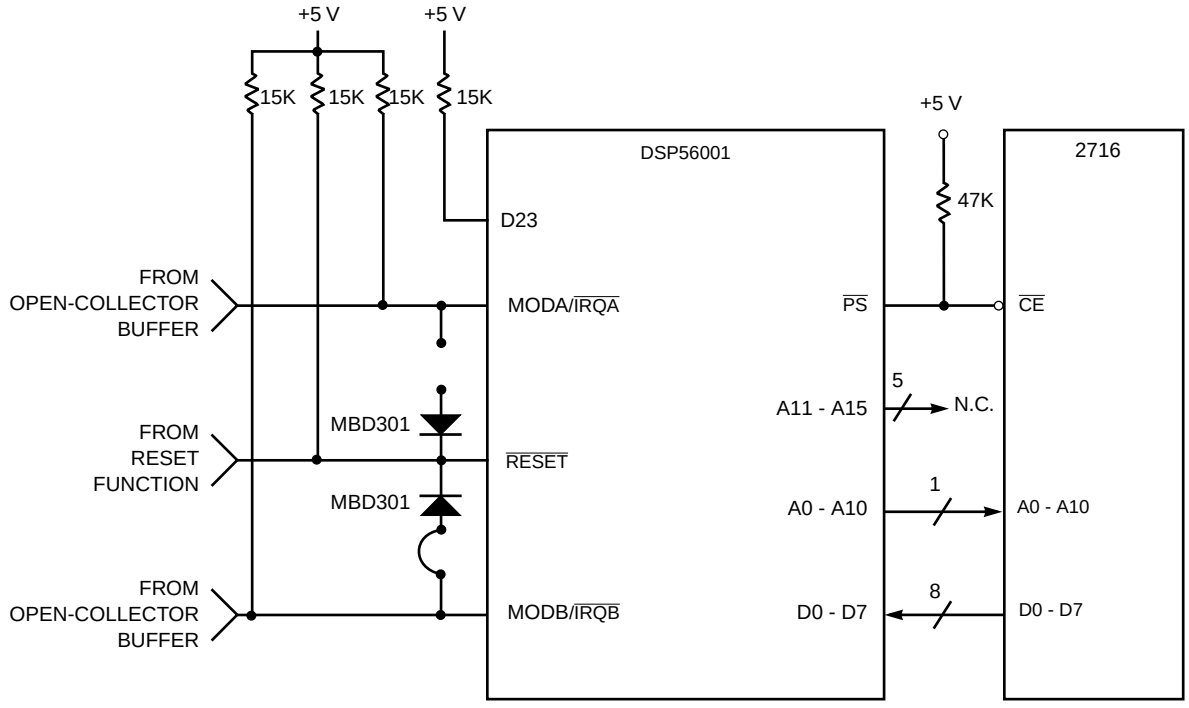


Figure 9-4 Memory Segmentation

Figure 9-2 shows an example of external program memory. A typical implementation of this circuit would use three-byte-wide static memories and would not require any additional logic. The $\overline{PS}$ signal is used as the program-memory chip-select signal to enable the program memory at the appropriate time.

Figure 9-3 shows a similar circuit using the $\overline{DS}$ signal to enable two data memories and using the X/$\overline{Y}$ signal to select between them. The three external memory spaces (program, X data, and Y data) do not have to reside in separate physical memories; a single memory can be employed by using the $\overline{PS}$, $\overline{DS}$, and X/$\overline{Y}$ signals as additional address lines to segment the memory into three spaces (see Figure 9-4). Table 9-1 shows how the $\overline{PS}$, $\overline{DS}$, and X/$\overline{Y}$ signals are decoded. If the DSP is in the development mode, an exception fetch to any interrupt vector location will cause the X/$\overline{Y}$ signal to go low when $\overline{PS}$ is asserted. This procedure is useful for debugging and for allowing external circuitry to track interrupt servicing.

Special provisions have been made to allow the DSP to load a program from an inexpensive byte-wide ROM (see Figure 9-5 and the DSP56001 Advance Information Data Sheet (ADI1290) into internal program memory during a power-on reset. On powerup, the wait-state generator adds 15 wait states to all external memory accesses so that slow memory can be used. If bit 23 of external memory is a logic one, the DSP will load the contents of an external ROM into internal program memory (if bit 23 is a logic zero, it will load from the host port). The bootstrap program uses the bytes in three consecutive memory locations in the external ROM to build a single word in internal program memory. Figure 9-6 shows a system that uses internal program memory loaded from an external ROM during powerup and that splits the data memory space of a single memory bank into X: and Y: memory spaces. Although external program memory must be 24 bits, external data memory does not. Of course, this is application specific. However, many systems use 16 or fewer bits for A/D and D/A conversion, since they only need to store 16, 12, or even eight bits of data.



ADDRESS OF EXTERNAL BYTE-WIDE P MEMORY	CONTENTS LOADED TO INTERNAL PRAM AT:
P:$C000	P:$0000 LOW BYTE
P:$C001	P:$0000 MID BYTE
P:$C002	P:$0000 HIGH BYTE
•	•
•	•
•	•
P:$C5FD	P:$01FF LOW BYTE
P:$C5FE	P:$01FF MID BYTE
P:$C5FF	P:$01FF HIGH BYTE

Figure 9-5 Port A Bootstrap Circuit

The 24/56 bits of internal precision is usually sufficient for intermediate results. Recognizing this fact can save cost by reducing the number of external memory chips.

All unused inputs should have pullup resistors for two reasons: 1) floating inputs draw excessive power, and 2) a floating input can cause erroneous operation. For example, during RESET, all signals are three-stated. Without pullup resistors, the $\overline{PS}$ and $\overline{DS}$ signals may become active, causing two or more memory chips to try to simultaneously drive the external data bus, which can damage the memory chips. A pullup resistor in the 50K-ohm range should be sufficient.

9.2 PORT A TIMING

The external bus timing is defined by the operation of the address bus, data bus, and bus control pins. The transfer of data over the external data bus is synchronous with the clock. The timing A, B, and C relative to the edges of an external clock (see Figure 9-7 and Figure 9-8) are provided in the DSP56001 Advance Information Data Sheet (ADI1290). This timing is essential for designing synchronous multiprocessor systems. Figure 9-7 shows the port A timing with no wait states (wait-state control is discussed in 9.2.1 Port A Wait States). One instruction cycle equals two clock cycles or four clock phases. The clock phases, which are numbered T0 – T3, are used for timing on the DSP. Figure 9-8 shows the same timing with two wait states added to the external X: memory access. Four TW clock phases have been added because one wait state adds two T phases and is equiv-

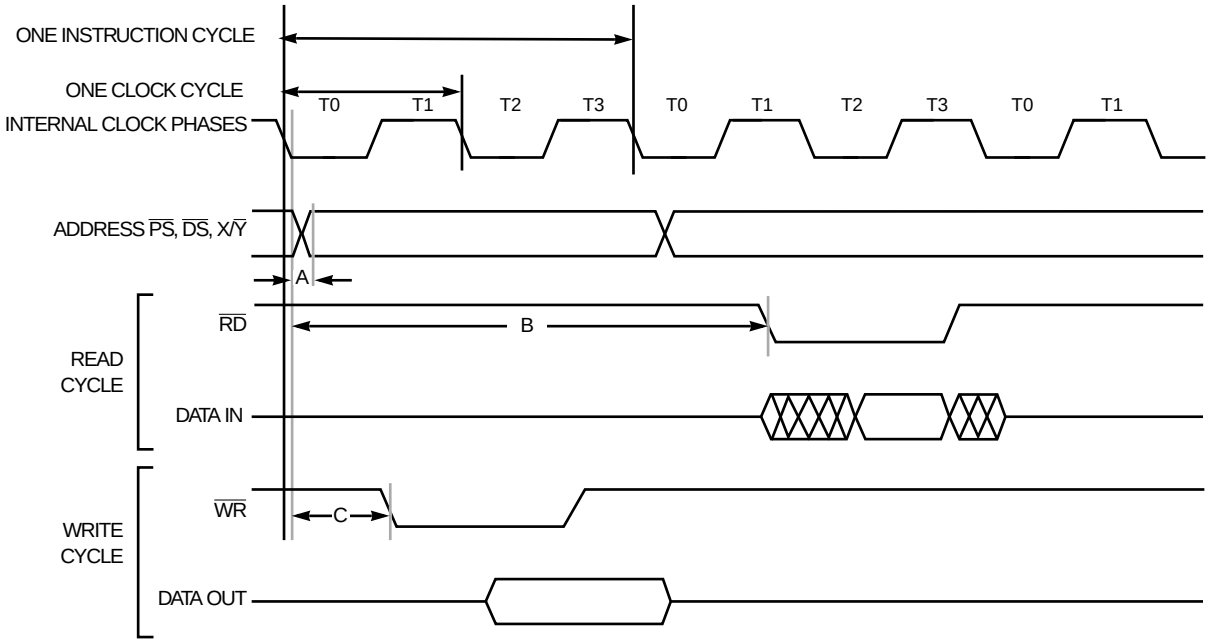


Figure 9-7 Port A Bus Operation with No Wait States

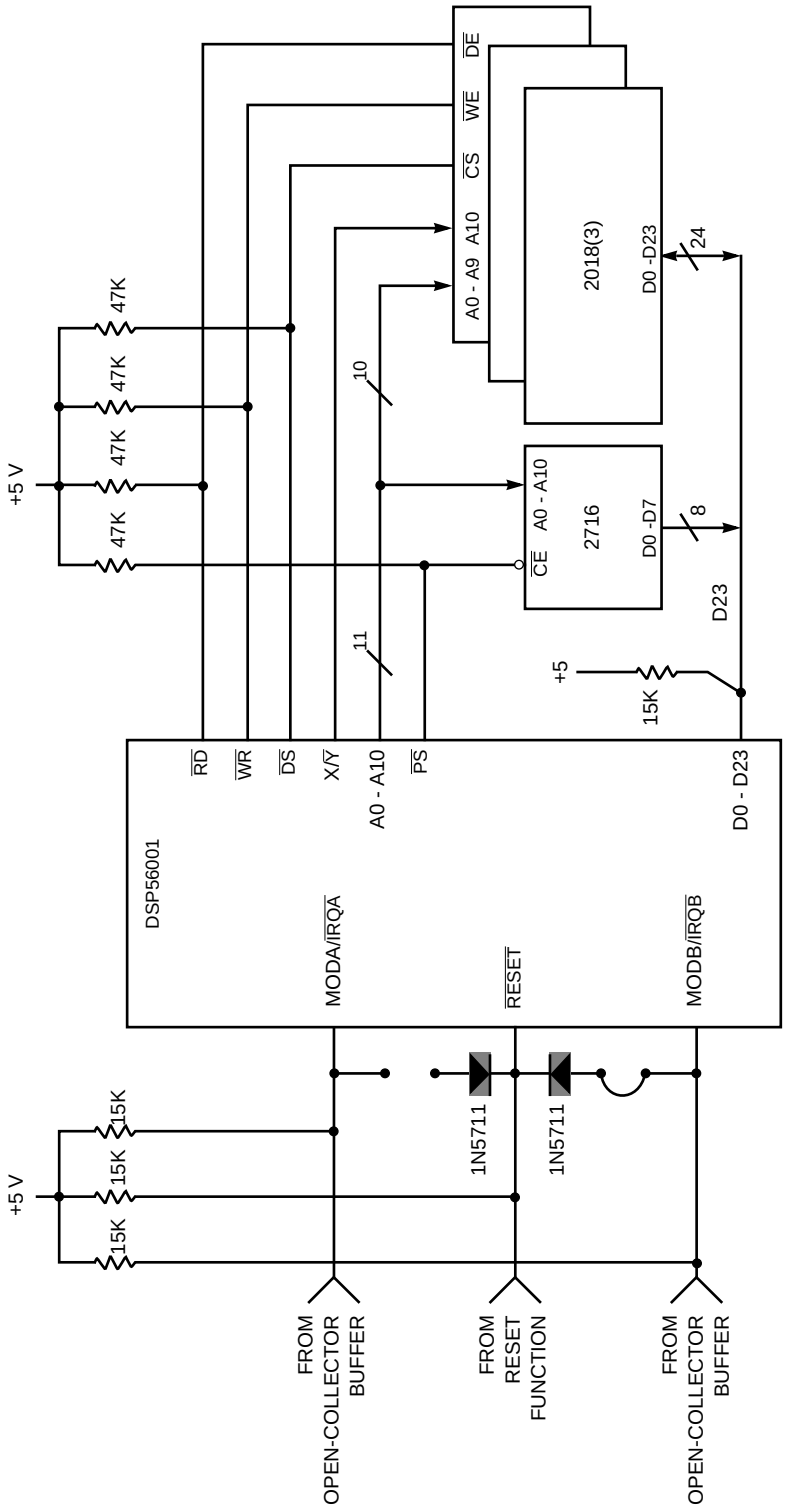


Figure 9-6 Port A Bootstrap ROM with X and Y RAM

alent to repeating the T2 and $\overline{T2}$ clock phases. The write signal is also delayed from the

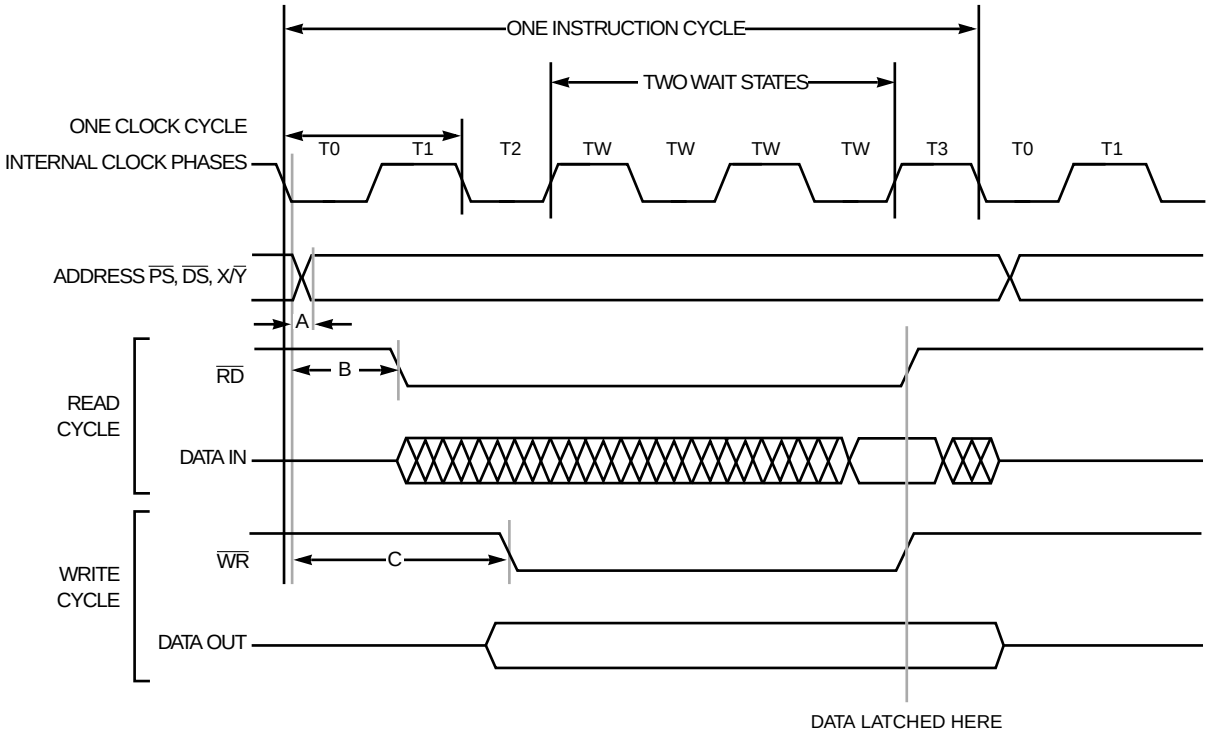


Figure 9-8 Port A Bus Operation with Two Wait States

T1 to the T2 state when one or more wait states are added to ease interfacing to the port. Each external memory access requires the following procedure:

1. The external memory address is defined by the address bus (A0–A15) and the memory reference selects ($\overline{PS}$, $\overline{DS}$, and X/$\overline{Y}$). These signals change in the first phase (T0) of the bus cycle. Since the memory reference select signals have the same timing as the address bus, they may be used as additional address lines. The address and memory reference signals are also used to generate chip-select signals for the appropriate memory chips. These chip-select signals change the memory chips from low-power standby mode to active mode and begin the read access time. This mode change allows slower memories to be used since the chip-select signals can be address based rather than read or write enable based. Read and write enable do not become active until after the address is valid. See the timing diagrams in the DSP56001 Advance Information Data Sheet (ADI1290) for detailed timing information.

2. When the address and memory reference signals are stable, the data transfer is enabled by read enable ($\overline{RD}$) or write enable ($\overline{WR}$). $\overline{RD}$ or $\overline{WR}$ is asserted to "qualify" the address and memory reference signals as stable and to perform the read or write data transfer. $\overline{RD}$ and $\overline{WR}$ are asserted in the second phase of the bus cycle (if there are no wait states). Read enable is typically con-

Table 9-1 Program and Data Memory Select Encoding

$\overline{PS}$	$\overline{DS}$	X/$\overline{Y}$	External Memory Reference
1	1	1	No Activity
1	0	1	X Data Memory on Data Bus
1	0	0	Y Data Memory on Data Bus
0	1	1	Program Memory on Data Bus (Not an Exception)
0	1	0	External Exception Fetch: Vector or Vector +1 (Development Mode Only)
0	0	X	Reserved
1	1	0	Reserved

nected to the output enable ($\overline{OE}$) of the memory chips and simply controls the output buffers of the chip-selected memory. Write enable is connected to the write enable ($\overline{WE}$) or write strobe ($\overline{WS}$) of the memory chips and is the pulse that strobes data into the selected memory. For a read operation, $\overline{RD}$ is asserted and $\overline{WR}$ remains deasserted. Since write enable remains negated, a memory read operation is performed. The DSP data bus becomes an input, and the memory data bus becomes an output. For a write operation, $\overline{WR}$ is asserted and $\overline{RD}$ remains deasserted. Since read enable remains deasserted, the memory chip outputs remain in the high-impedance state even before write strobe is asserted. This state assures that the DSP and the chip-selected memory chips are not enabled onto the bus at the same time. The DSP data bus becomes an output, and the memory data bus becomes an input.

3. Wait states are inserted into the bus cycle by a wait-state counter or by asserting $\overline{WT}$. The wait-state counter is loaded from the bus control register. If the value loaded into the wait-state counter is zero, no wait states are inserted into the bus cycle, and $\overline{RD}$ and $\overline{WR}$ are asserted as shown in Figure 9-7. If a value W=/0 is loaded into the wait state counter, W wait states are inserted into the bus cycle. When wait states are inserted into an external write cycle, $\overline{WR}$ is delayed from T1 to T2. The timing for the case of two wait states (W=2) is shown in Figure 9-8.

4. When $\overline{RD}$ or $\overline{WR}$ are deasserted at the start of T3 in a bus cycle, the data is latched in the destination device – i.e., when $\overline{RD}$ is deasserted, the DSP latches the data internally; when $\overline{WR}$ is deasserted, the external memory latches the data on the positive-going edge. The address signals remain stable until the first phase of the next external bus cycle to minimize power dissi-

pation. The memory reference signals ($\overline{PS}$, $\overline{DS}$, and X/$\overline{Y}$) are deasserted (held high) during periods of no bus activity, and the data signals are three-stated. For read-modify-write instructions such as BSET, the address and memory reference signals remain active for the complete composite (i.e., two I_{cyc}) instruction cycle.

Figure 9-9 shows an example of mixing different memory speeds and memory-mapped peripherals in different address spaces. The internal memory uses no wait states, X: memory uses two wait states, Y: memory uses four wait states, P: memory uses five wait states, and the analog converters use 14 wait states. Controlling five different devices at five different speeds requires only one additional logic package. Half the gates in that package are used to map the analog converters to the top 64 memory locations in Y: memory.

Adding wait states to external memory accesses can substantially reduce power requirements. Table 9-2 shows how the power was reduced during external memory and I/O operations by changing from zero to 15 wait states at four different clock speeds in a test circuit.

Table 9-2 Power Requirements for Minimum and Maximum External Memory Wait States

Clock	Current for 0 Wait States	Current for 15 Wait States
4.000 MHz	19.8 mA	8.6 mA
6.5536 MHz	31.0 mA	12.8 mA
10.245 MHz	46.8 mA	18.8 mA
20.000 MHz	91.0 mA	36.6 mA

9.2.1 Port A Wait States

The DSP56000/DSP56001 features two methods to allow the user to accommodate slow memory by changing the port A bus timing. The first method uses the bus control register (BCR), which allows a fixed number of wait states to be inserted in a given memory access to all locations in each of the four memory spaces: X, Y, P, and I/O. The second method uses the bus strobe/wait ($\overline{BS}$/$\overline{WT}$) facility, which allows an external device to insert an arbitrary number of wait states when accessing either a single location or multiple locations of external memory or I/O space. Wait states are executed until the external device releases the DSP to finish the external memory cycle.

9.2.2 Bus Control Register

The expansion bus timing is controlled by the BCR (see Figure 9-10) which controls the

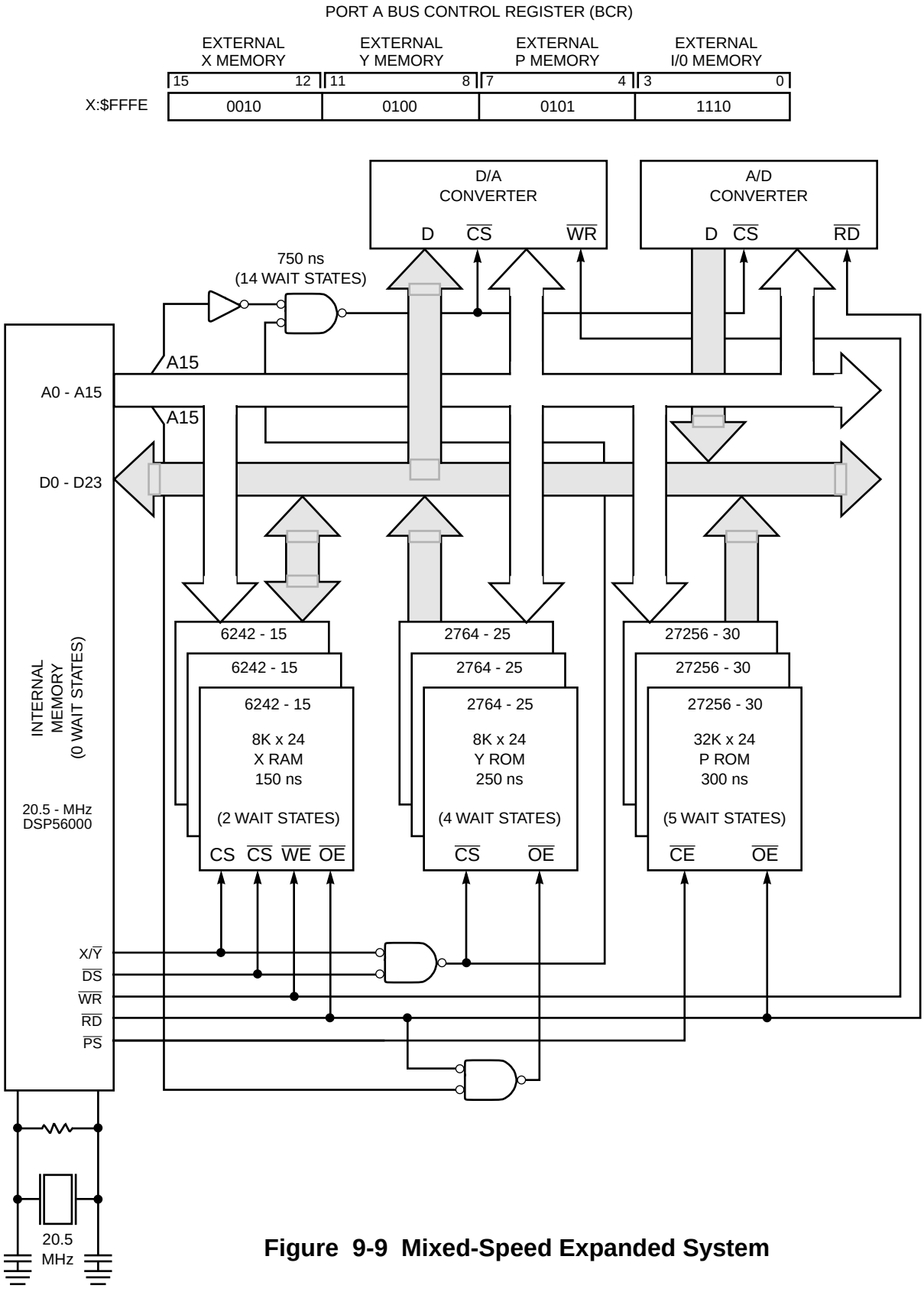
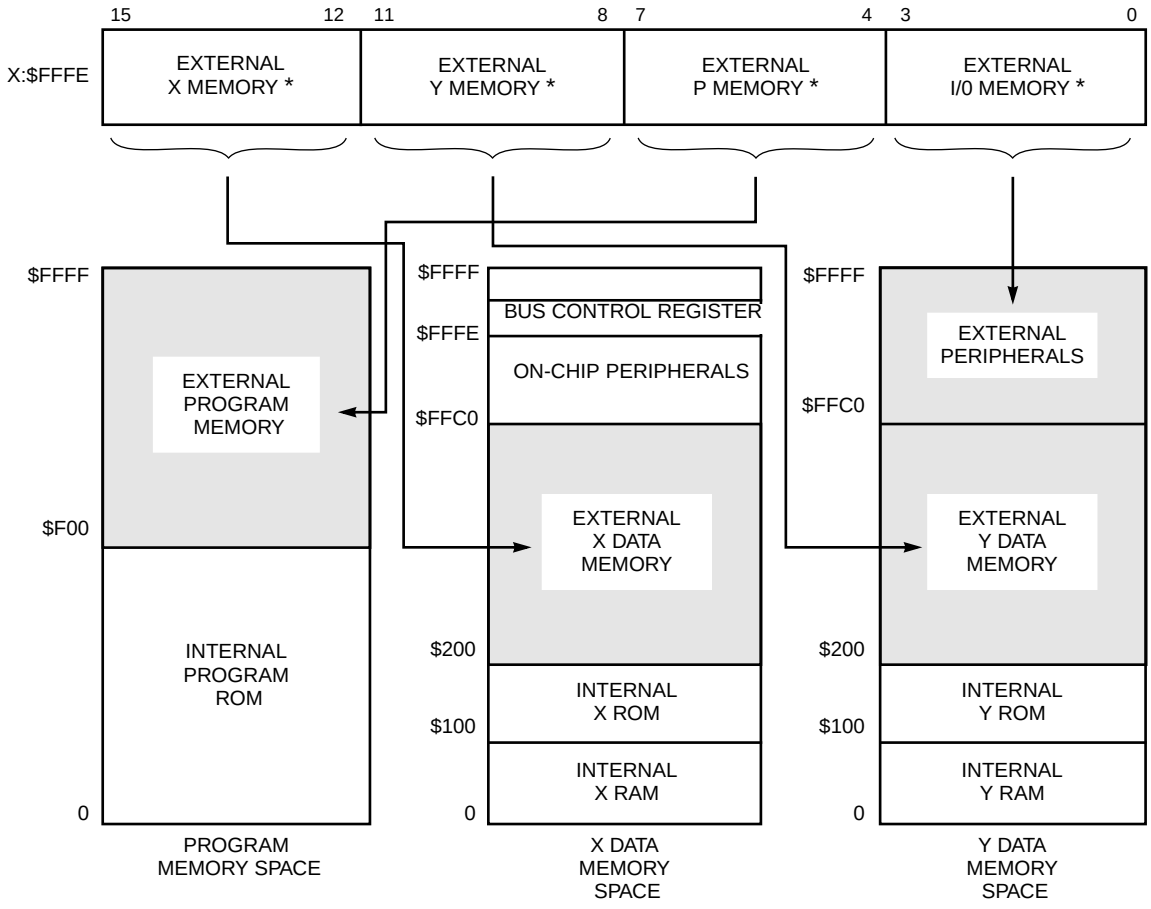


Figure 9-9 Mixed-Speed Expanded System

timing of the bus interface signals, $\overline{RD}$ and $\overline{WR}$, and the data output lines. Each of the

memory spaces, X data, Y data, program data, and I/O, has its own 4-bit BCR, which can be programmed for inserting up to 15 wait states (each wait state adds one-half instruction cycle to each memory access – i.e., 50 ns for a 20-Mhz clock). In this way, external bus timing can be tailored to match the speed requirements of the different memory spaces. On processor RESET, the BCR is preset to all ones (15 wait states).



* Zero to 15 wait states can be inserted into each external memory access.

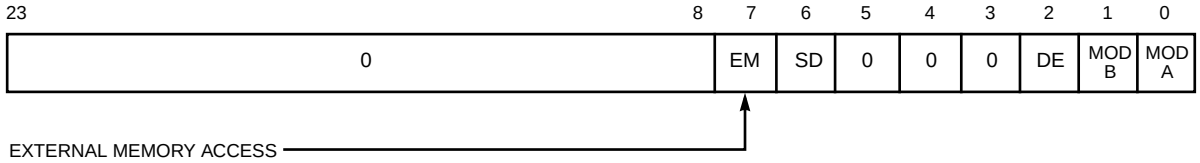
Figure 9-10 Bus Control Register

Figure 9-10 illustrates which of the four BCR subregisters affect which external memory space. The BCR is a memory-mapped register located at X:$FFFE. All the internal peripheral devices are memory mapped, and their control registers reside between X:$FC00 and X:$FFFF. Loading the BCR as shown in Figure 9-9 can be accomplished by executing a “MOVEP #$245E, X:$FFFE” instruction. Changing individual bits in one of the four subregisters can be accomplished by using the BSET and BCLR instructions.

9.2.3 Bus Strobe/Wait Pins

The DSP56000/DSP56001 has two reconfigurable pins that are used as either bus request/bus grant ($\overline{BR}/\overline{BG}$) or as bus strobe/wait ($\overline{BS}/\overline{WT}$). The ability to insert wait states using $\overline{BS}/\overline{WT}$ provides a means to connect asynchronous devices to the DSP, allows devices with differing timing requirements to reside in the same memory space, allows a bus arbiter to provide a fast multiprocessor bus access, and provides another means of halting the DSP at a known program location with a fast restart. Bus strobe in the original in-house documentation was called “memory ready strobe” and wait was called “memory ready”. The original names have been changed to be more descriptive.

RESET initializes the DSP in the $\overline{BR}/\overline{BG}$ mode for compatibility. The $\overline{BS}/\overline{WT}$ mode is selected if bit 7 in the OMR (see Figure 9-11) is set to one, which can be accomplished by executing an “ORI #80, OMR” instruction. Because the $\overline{BR}/\overline{BG}$ and $\overline{BS}/\overline{WT}$ modes are mutually exclusive, port A cannot be three-stated by an external device when in the $\overline{BS}/\overline{WT}$ mode. The BCR is still operative in the $\overline{BS}/\overline{WT}$ mode and defines the minimum number of wait states that are inserted.



0 BUS REQUEST/BUS GRANT ARBITRATION (RESET); USED BY A HOST TO THREE-STATE THE DSP56000 EXTERNAL BUS.
1 BUS STROBE/WAIT; USED BY A HOST TO DELAY OR STRETCH THE CURRENT DSP56000 BUS CYCLE

Figure 9-11 Port A Access Control

The timing of $\overline{BS}$ and $\overline{WT}$ pins is illustrated in Figure 9-12. Every external access, $\overline{BS}$ is asserted concurrently with the address lines in T0. $\overline{BS}$ can be used by external wait-state logic to establish the start of an external access. $\overline{BS}$ is deasserted in T3 of each external bus cycle, signaling that the current bus cycle will complete. Since the $\overline{WT}$ signal is internally synchronized, it can be asserted asynchronously with respect to the system clock. The $\overline{WT}$ signal should only be asserted while $\overline{BS}$ is asserted. Asserting $\overline{WT}$ while $\overline{BS}$ is deasserted will give indeterminate results. However, for the number of inserted wait states to be deterministic, $\overline{WT}$ timing must satisfy setup and hold timing with respect to the negative-going edge of EXTAL. The setup and hold times are provided in the DSP56001 Advance Information Data Sheet (ADI1290). The timing of $\overline{WR}$ is controlled by the BCR and is independent of $\overline{WT}$. The minimum number of wait states that can be inserted using the WT pin is two. Table 9-3 summarizes the effect of the BCR and $\overline{WT}$ pin on the number of wait states generated.

9.3 BUS ARBITRATION

The $\overline{BR}/\overline{BG}$ and $\overline{BS}/\overline{WT}$ pins provide bus arbitration controls. The BR/BG mode allows an

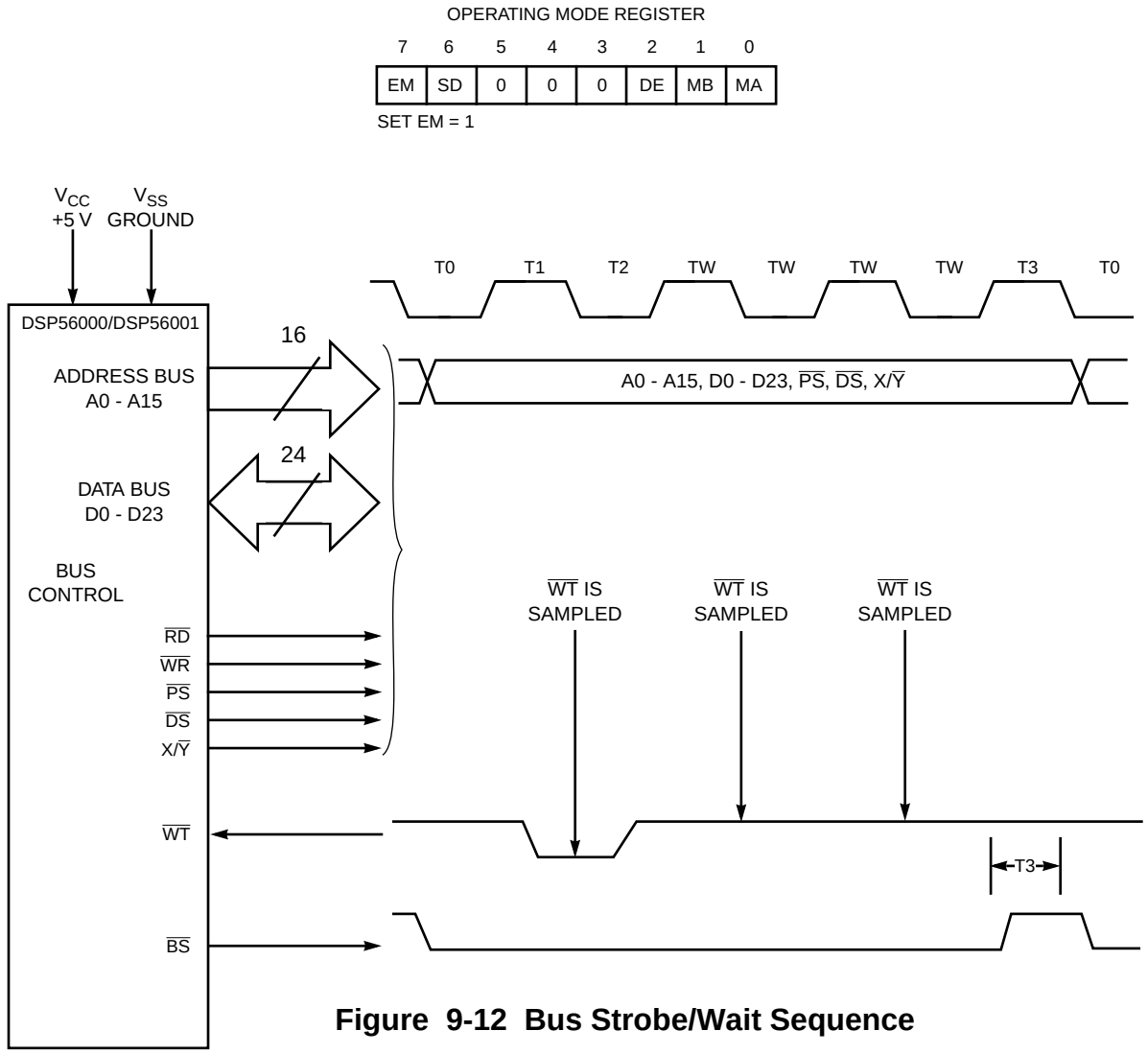


Figure 9-12 Bus Strobe/Wait Sequence

external device to request and be given control of the external memory bus (port A) while the DSP continues internal operations using internal memory spaces. This configuration allows a bus controller to arbitrate a multiple bus-master system. (A bus master can issue

Table 9-3 Wait State Control

BCR Contents	$\overline{WT}$	Number of Wait States Generated
0	Deasserted	0
0	Asserted	2 (minimum)
> 0	Deasserted	Equals value in BCR
> 0	Asserted	Minimum equals 2 or value in BCR. Maximum is determined by $\overline{WT}$.

addresses on the bus; a bus slave can respond to addresses on the bus. A single device can be both a master and a slave, but can only be one or the other at any given time.) The $\overline{BS}/\overline{BW}$ mode allows a bus arbitrator to extend the bus cycle of the DSP56000/DSP56001 to allow another bus master time to finish its bus access before allowing the DSP56000/DSP56001 access to the bus.

9.3.1 Bus Request/Bus Grant

The $\overline{BR}/\overline{BG}$ mode is selected if OMR bit 7 (see Figure 9-11) is set to zero (execute an "ANDI #7F,OMR" instruction). When $\overline{BR}$ is asserted (see Figure 9-13), the DSP will assert $\overline{BG}$ after the current external access cycle completes and will simultaneously three-state the port A signals (see the DSP56001 Advance Information Data Sheet (ADI1290) for exact timing of $\overline{BR}/\overline{BG}$). The bus is then available to be used by the bus master requesting the bus. When BR is deasserted, BG is deasserted after the current external access, and the port A signals are no longer three-stated. Reset clears bit 7 of the OMR. Information on operation of the $\overline{BR}/\overline{BG}$ pins after executing a WAIT or STOP instruction can be found in 8.4 WAIT PROCESSING STATE and 8.5 STOP PROCESSING STATE.

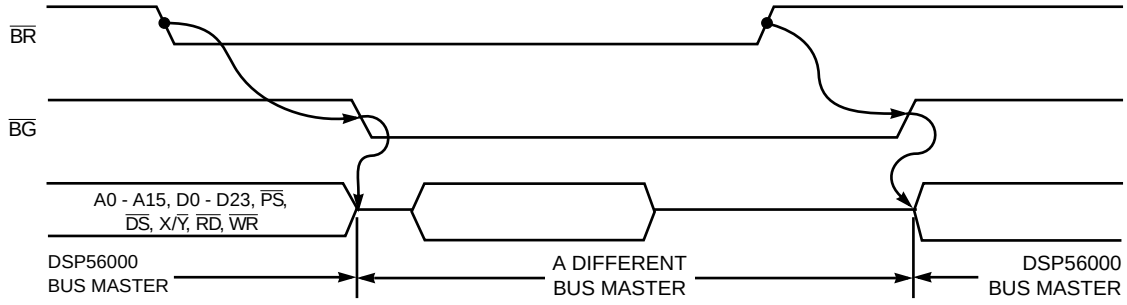


Figure 9-13 Bus Request/Bus Grant Sequence

9.3.2 Shared Memory

The bus control signals described in the previous paragraphs provide the means to connect additional bus masters (which may be additional DSPs, microprocessors, direct memory access (DMA) controllers, etc.) to the port A bus. Four arbitration examples will be described in the following paragraphs: 1) bus arbitration using only $\overline{BR}/\overline{BG}$ with internal control, 2) bus arbitration using only $\overline{BR}/\overline{BG}$ with external control, 3) bus arbitration using $\overline{BR}/\overline{BG}$ and $\overline{BS}/\overline{WT}$ with no overhead, and 4) signaling using semaphores.

9.3.2.1 Bus Arbitration Using Only BR/BG With Internal Control

Perhaps the simplest example of a shared memory system using a DSP56000/DSP56001 is shown in Figure 9-14. The bus arbitration is performed internal to the DSP#2 by using

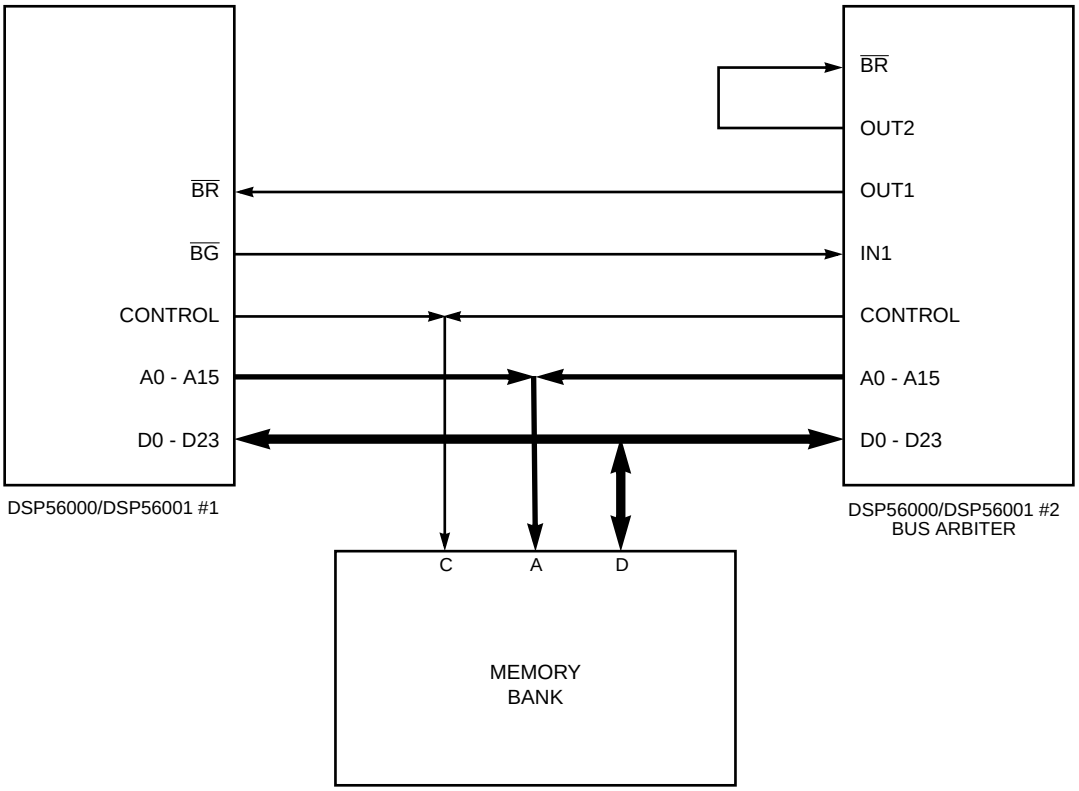


Figure 9-14 Bus Arbitration Using Only $\overline{BR}$/$\overline{BG}$ with Internal Control

software. DSP#2 controls all bus operations by using I/O pin OUT2 to three-state its own port A and by never accessing port A without first calling the subroutine that arbitrates the bus. When the DSP#2 needs to use external memory, it uses I/O pin OUT1 to request bus access and I/O pin IN1 to read bus grant. DSP#1 does not need any extra code for bus arbitration since the $\overline{BR}$/$\overline{BG}$ hardware handles its bus arbitration automatically. The protocol for bus arbitration is as follows:

At RESET: DSP#2 sets OUT2=0 ($\overline{BR}$#2=0) and OUT1=1 ($\overline{BR}$#1=1), which gives DSP#1 access to the bus and suspends DSP#2 bus access.

When DSP#2 wants control of the memory, the following steps are performed (see Figure 9-15):

1. DSP# 2 sets OUT1=0 ($\overline{BR}$#1=0).
2. DSP# 2 waits for IN1=0 ($\overline{BG}$#1=0 and DSP#1 off the bus). This takes at most 13T+4T$_*$WS+20 ns (about 400 ns at 20 MHz) where T is I_{cyc}/4 and WS is the number of wait states used by DSP# 1. If DSP#1 is not using any read/modify/write instructions in its external space, the maximum becomes only 9T+2T$_*$WS+20 ns (about 250 ns at 20 MHz).

3. DSP#2 sets OUT2=1 ($\overline{BR}$#2=1 to let DSP#2 on the bus).
4. DSP#2 accesses the bus for block transfers, etc. at full speed.
5. To release the bus, DSP#2 sets OUT2=0 ($\overline{BR}$#2=0) after the last external access.
6. DSP#2 then sets OUT1=1 ($\overline{BR}$#1=1) to return control of the bus to DSP#1.
7. DSP#1 then acknowledges mastership by deasserting $\overline{BG}$#1.

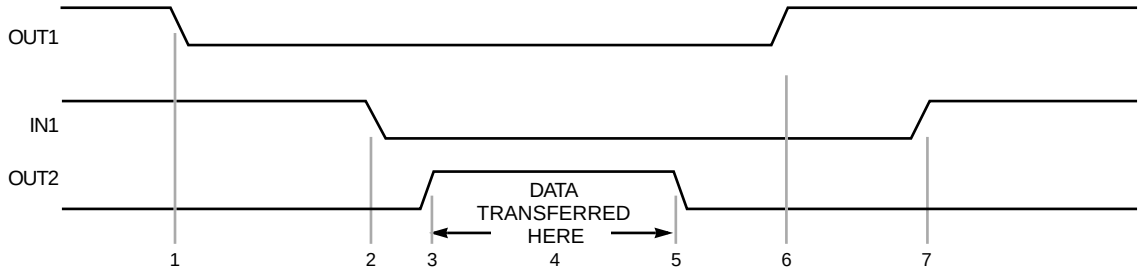


Figure 9-15 Two DSPs with External Bus Arbitration Timing

9.3.2.2 Bus Arbitration Using Only $\overline{BR}/\overline{BG}$ With External Control

Figure 9-16 can be implemented with external bus arbitration logic, which will save processing capacity on the DSPs and can make bus access much faster at a cost of additional hardware. Operation is similar to the system shown in Figure 9-14. The bus arbitration logic takes control of the external bus by deasserting an enable signal (E1, E2, and E3) to all DSPs, which will then acknowledge by granting the bus ($\overline{BG}$=0). When a DSP (DSP#1 in Figure 9-16) wants the bus, it will jump to a subroutine, which will set PC3=1. When the arbitration logic grants the bus to a DSP, it will issue a BG1 (BG2 for DSP#2; BG3 for DSP#3) to let the DSP know that it can have the bus. Arbitration logic will then enable the bus by asserting the appropriate enable (E1=1). When the DSP is ready to relinquish the bus, it deasserts PC3, and the arbiter deasserts E1 and BG1.

9.3.2.3 Bus Arbitration Using $\overline{BR}/\overline{BG}$ and $\overline{BS}/\overline{WT}$ With No Overhead

By using the circuit shown in Figure 9-17, two DSPs can share memory with hardware arbitration that requires no software on the part of the DSPs. In Figure 9-17, DSP#1 has EM=1 in its OMR, and DSP#2 has EM=0 in its OMR. The protocol for bus arbitration in Figure 9-17 is as follows:

At RESET: $\overline{BG}$ of DSP#2 is deasserted, which three-states the buffers, giving DSP#2 control of the memory. Reset causes DSP#1 to initially be in the $\overline{BR}/\overline{BG}$ mode. DSP#1 OMR bit 7 must be set by software during initialization to change $\overline{BR}/\overline{BG}$ to $\overline{BS}/\overline{WT}$.

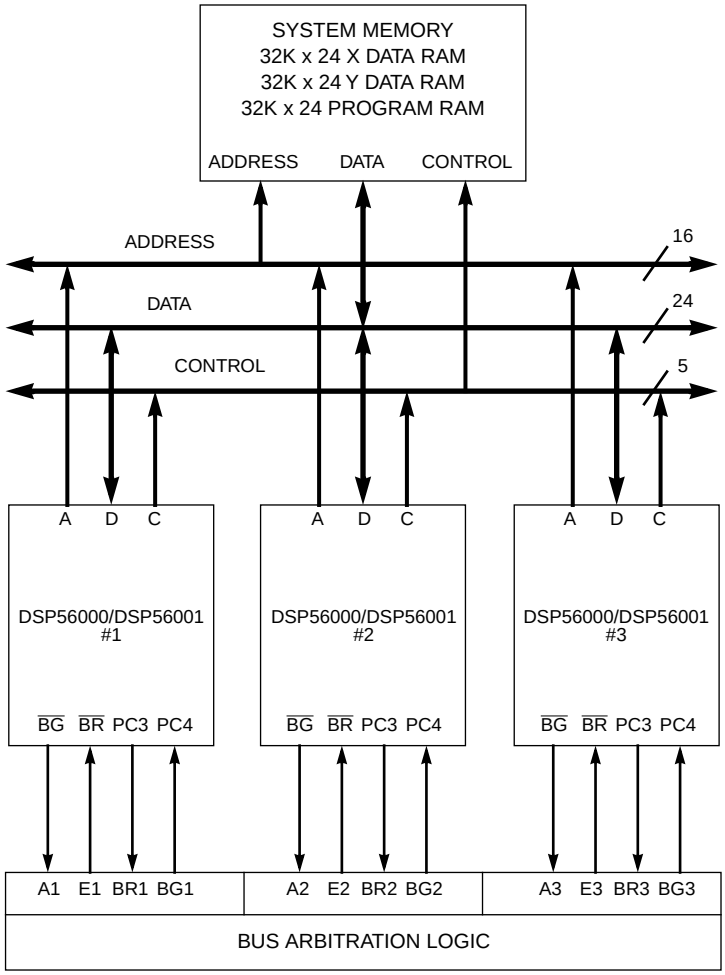


Figure 9-16 Bus Arbitration Using Only $\overline{BR}$/$\overline{BG}$ with External Control

When DSP#1 wants control of the memory the following steps are performed (see Figure 9-18):

1. DSP#1 makes an external access, thereby asserting $\overline{BS}$, which asserts $\overline{WT}$ (causing DSP#1 to execute wait states in the current cycle) and asserts DSP#2 $\overline{BR}$ (requesting that DSP#2 release the bus).
2. When DSP#2 finishes its present bus cycle, it three-states its bus drivers and asserts $\overline{BG}$. Asserting $\overline{BG}$ enables the three-state buffers, placing the DSP#1 signals on the memory bus. Asserting $\overline{BG}$ also deasserts $\overline{WT}$, which allows DSP#1 to finish its bus cycle.
3. When DSP#1's memory cycle is complete, it releases $\overline{BS}$, which deasserts $\overline{BR}$. DSP#2 then deasserts $\overline{BG}$, three-stating the buffers and allowing DSP#2 to access the memory bus.

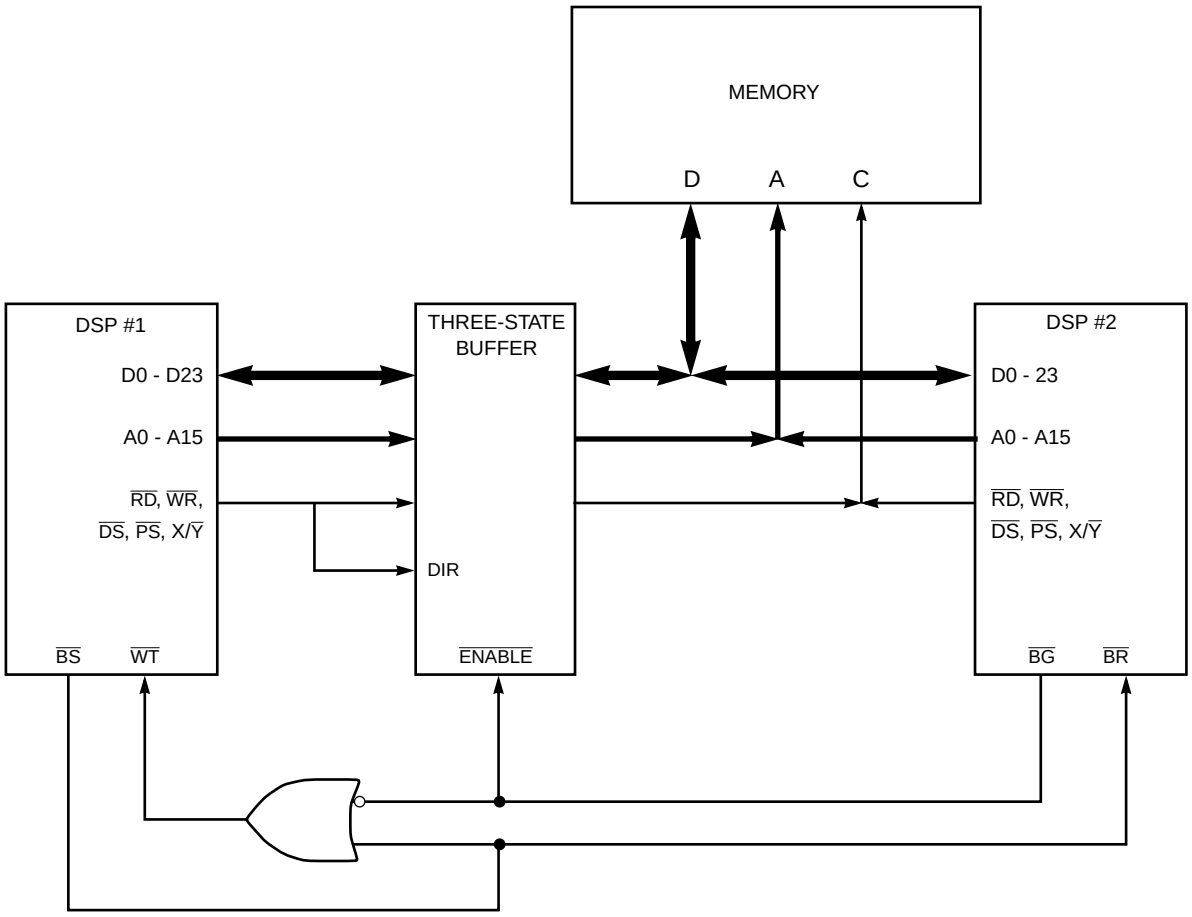


Figure 9-17 Bus Arbitration Using BR/BG and BS/WT with No Overhead

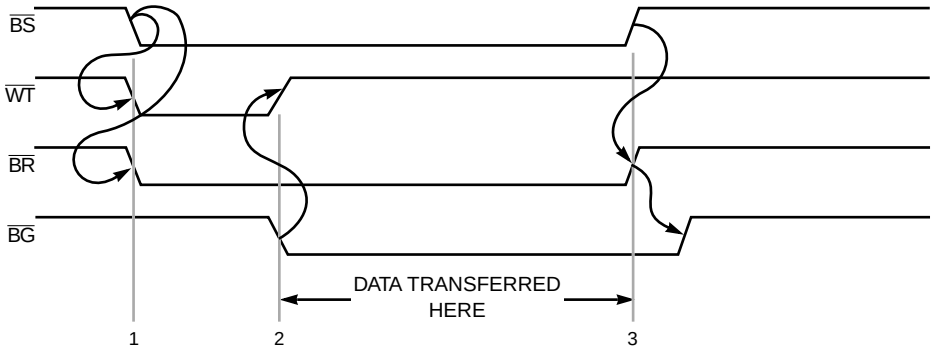


Figure 9-18 Two DSPs with External Bus Arbitration Timing

9.3.2.4 Signaling Using Semaphores

Figure 9-19 shows a more sophisticated shared memory system that uses external arbitration with both local external memory and shared memory. The four semaphores are bits in one of the words in each shared memory bank used by software to arbitrate memory use. Semaphores are commonly used to indicate that the contents of the semaphore's memory blocks are being used by one processor and are not available for use by another processor. Typically, if the semaphore is cleared, the block is not allocated to a processor; if the semaphore is set, the block is allocated to a processor.

Without semaphores, one processor may try to use data while it is being changed by another processor, which may cause errors. This problem can occur in a shared memory system when separate test and set instructions are used to "lock" a data block for use by a single processor.

The correct procedure is to test the semaphore and then set the semaphore if it was clear to lock and gain exclusive use of the data block. The problem occurs when the second processor acquires the bus and tests the semaphore after the first processor tests the semaphore but before the first processor can lock the data block. The incorrect sequence is 1) the first processor tests the semaphore and sees that the block is available; 2) the second processor then tests the bit and also sees that the block is available; 3) both processors then set the bit to lock the data; and 4) both proceed to use the data on the assumption that the data cannot be changed by another processor.

The DSP56000/DSP56001 has a group of instructions designed to prevent this problem. They perform an indivisible read-modify-write operation and do not release the bus between the read and write (specifically, A0–A15, $\overline{DS}$, $\overline{PS}$, and X/$\overline{Y}$ do not change state). Not releasing the bus allows these instructions to test the semaphore and then to set, clear, or change the semaphore without the possibility of another processor testing the semaphore before it is changed. The instructions are bit test and change (BCHG), bit test and clear (BCLR), and bit test and set (BSET). The proper way to set the semaphore to gain exclusive access to a memory block is to use BSET to test the semaphore and to set it to one. After the bit is set, the result of the test operation will reveal if the semaphore was clear before it was set by BSET and if the memory block is available. If the bit was already set and the block is in use by another processor, the DSP will wait to access the memory block.

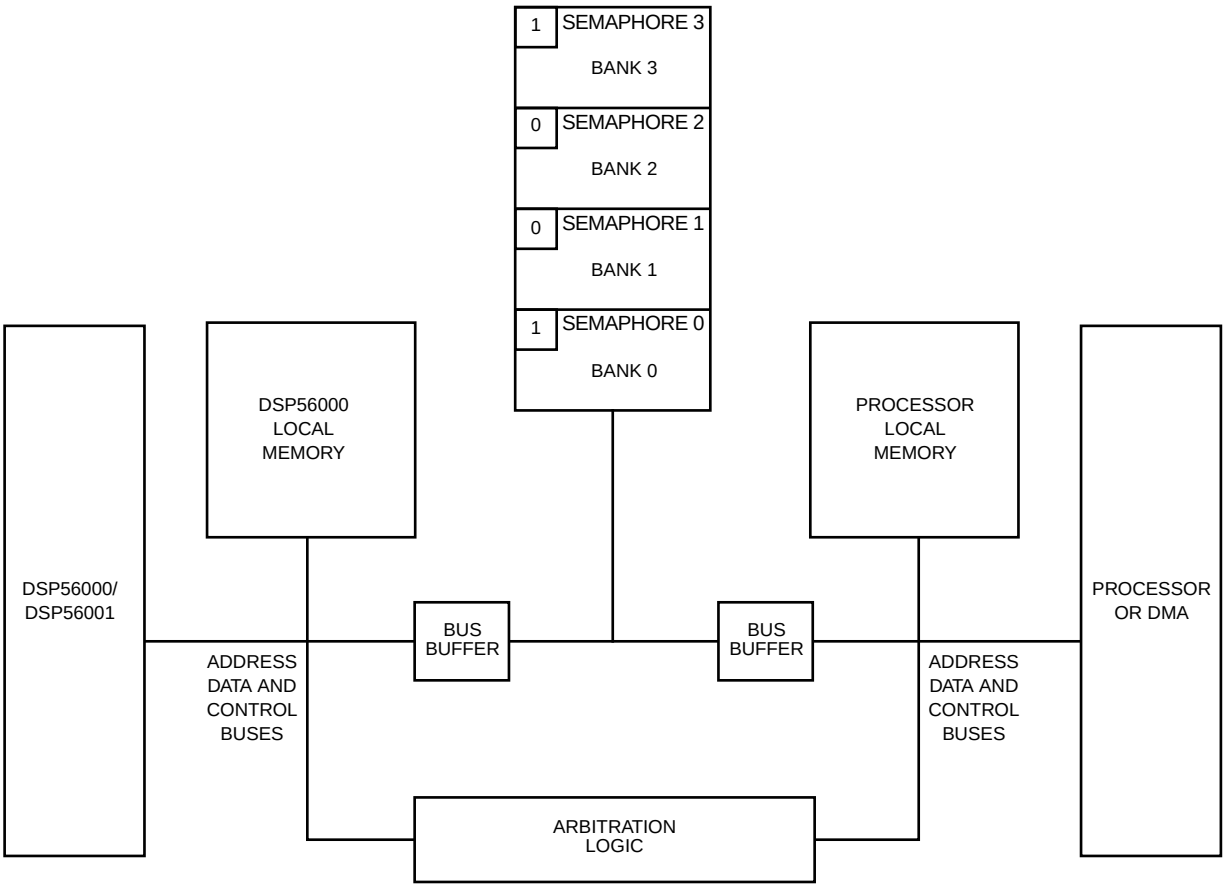


Figure 9-19 Signaling Using Semaphores

SECTION 10
PORT B

Port B is a dual-purpose I/O port that can be used as 1) 15 general-purpose pins individually configurable as either inputs or outputs or as 2) an 8-bit bidirectional host interface (HI) (see Figure 10-1). When configured as general-purpose I/O, port B can be used for device control. When configured as the HI, port B provides a convenient connection to another processor. This section describes both port B configurations, including examples of how to configure and use the port.

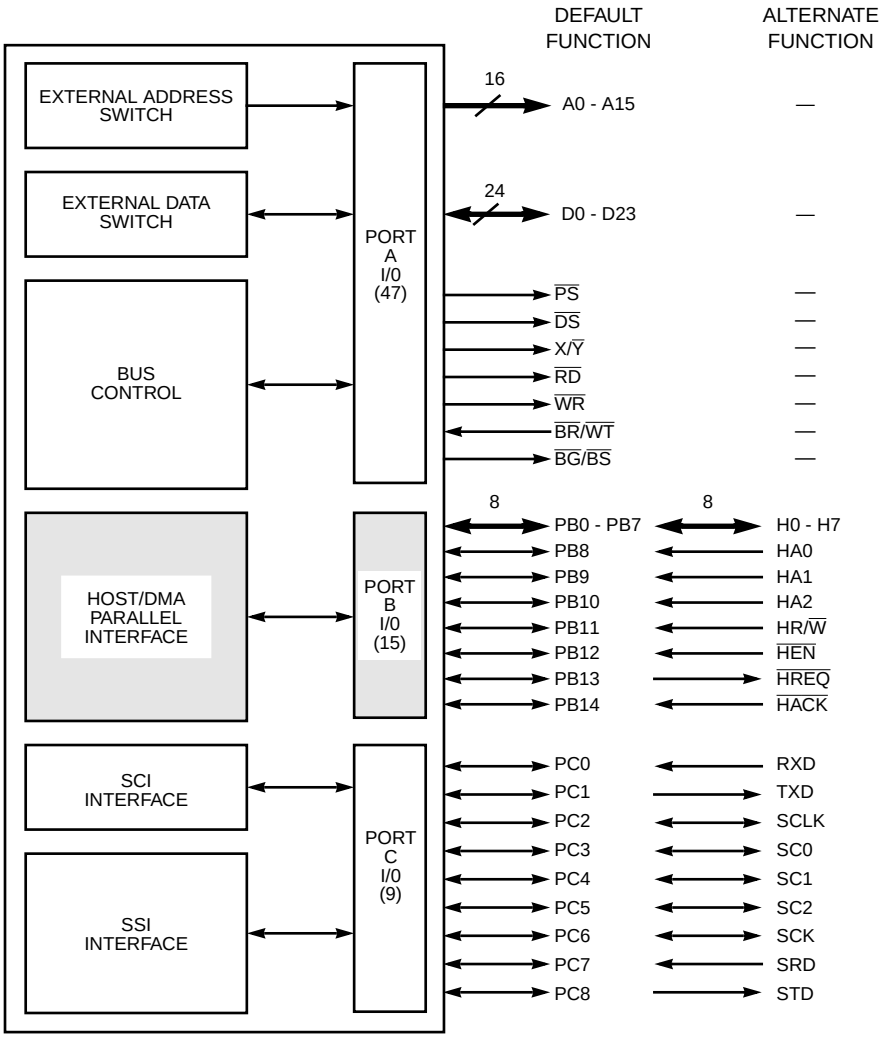


Figure 10-1 Port B Interface

10.1 GENERAL PURPOSE I/O

When it is configured as general-purpose I/O, port B can be viewed as three memory-mapped registers (see Figure 10-2) that control 15 I/O pins (see Figure 10-3). The software and hardware reset configure port B as general-purpose I/O with all 15 pins as inputs by clearing all three registers (external circuitry connected to these pins may need pullups until the pins are configured for operation). These registers are the port B control register (PBC), port B data direction register (PBDDR), and port B data register (PBD). Selection between general-purpose I/O and HI is made by setting PBC bit 0 (memory location X:$FFE0) to zero for general-purpose I/O or to one for HI. The PBDDR (memory location X:$FFE2) selects each corresponding pin in the PBD (memory location X:$FFE4), as an input pin if the PBDDR bit equals zero or as an output pin if the PBDDR bit equals one.

The port B I/O pin control logic is shown in Figure 10-4. Writing to PBD will write data to the pins designated as outputs by the PBDDR; reading the PBD will read the level on the pins designated as inputs by the PBDDR. When a pin is designated as an output and the PBD is read, the output of the output data bit latch is read, not the logic level on the pin

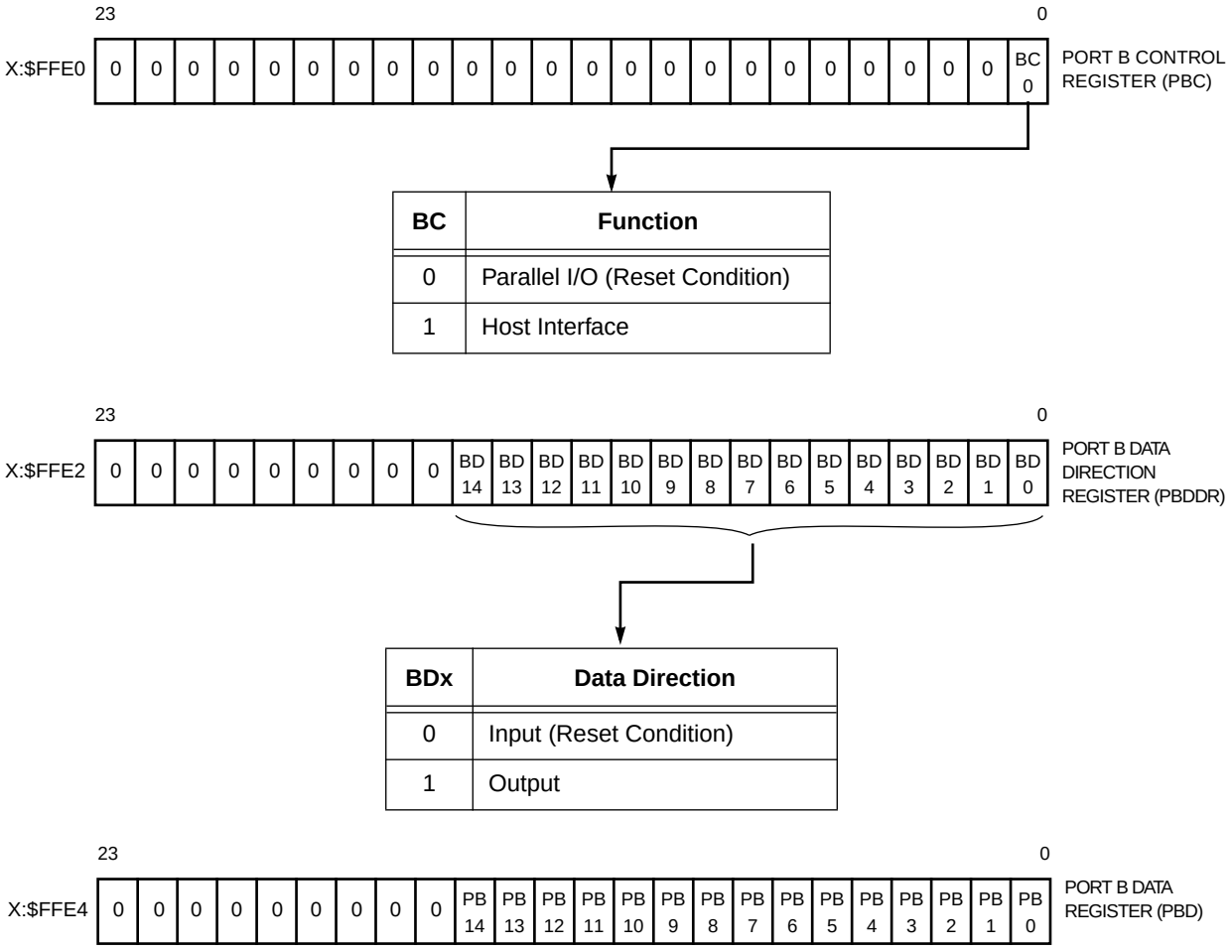


Figure 10-2 Parallel Port B Registers

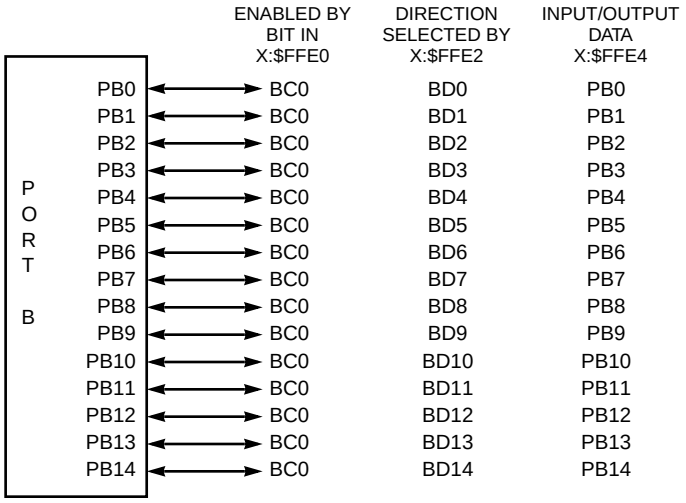


Figure 10-3 Parallel Port B Pinout

Port Control Register Bit	Data Direction Register Bit	Pin Function
0	0	Port Input Pin

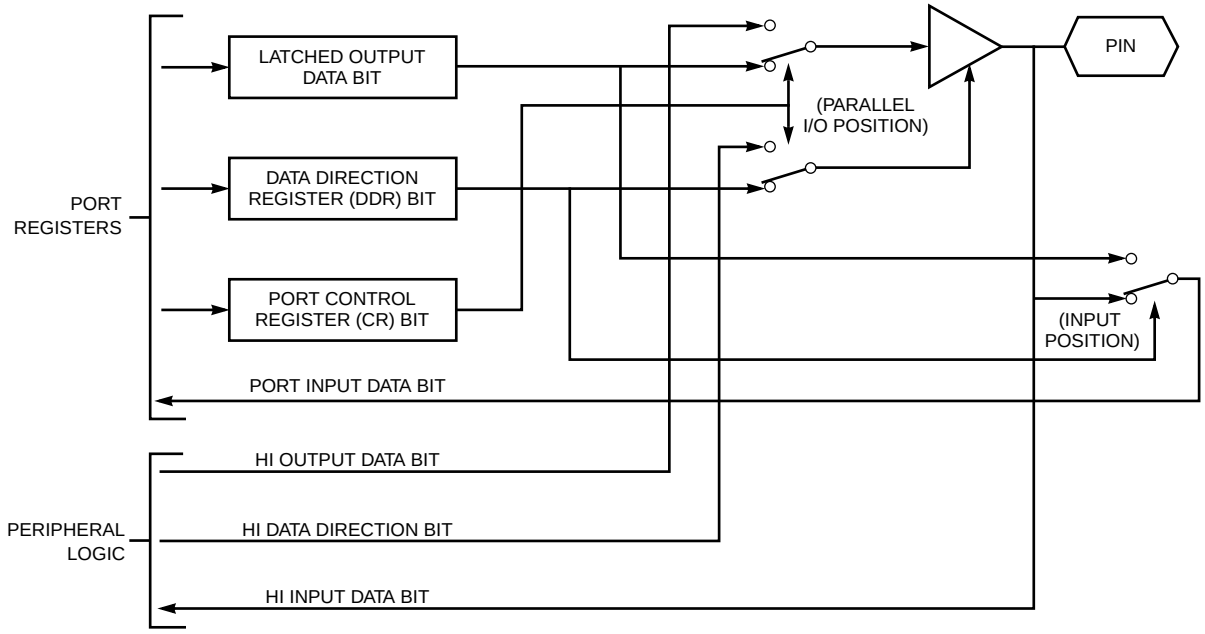


Figure 10-4 Port B I/O Pin Control Logic

itself. When the port is configured as the HI and the bit in the PBDDR is zero (input), then

```
            •
MOVEP   #$0,X:$FFE0         ;Select Port B to be general-purpose I/O
MOVE    #$7F00,X:$FFE2      ;Select pins PB0–PB7 to be inputs
            •               ;and pins PB8–PB14 to be outputs
            •
MOVEP   #data_out,X:$FFE4   ;Put bits 8–14 of ‘‘data_out’’ on pins
                            ;PB8–PB14 bits 0–7 are ignored.
MOVEP   X:$FFE4,#data_in    ;Put PB0–PB7 in bits 0–7 of ‘‘data_in’’
```

Figure 10-6 Write/Read Parallel Data with Port B

reading the PBD will show the logic level on the pin even though port B is configured as the HI. The HI function may be using the pin as an input or an output. This feature can be very useful when debugging the HI.

10.1.1 Programming Parallel I/O

Port B is a memory-mapped peripheral as are all of the DSP56000/DSP56001 peripherals (see Figure 10-5). The standard MOVE instruction transfers data between port B and a register; as a result, MOVE takes two instructions to perform a memory-to-memory data transfer and uses a temporary holding register. The MOVEP instruction is specifically designed for I/O data transfer as shown in Figure 10-6. Although the MOVEP instruction may take twice as long to execute as a MOVE instruction, only one MOVEP is required for a memory-to-memory data transfer, and MOVEP does not use a temporary register. Using the MOVEP instruction allows a fast interrupt to move data to/from a peripheral to memory and execute one other instruction or move the data to an absolute address. MOVEP is the only memory-to-memory move instruction; however, one of the operands must be in the top 64 locations of either X: or Y: memory.

The bit-oriented instructions that use I/O short addressing (BCHG, BCLR, BSET, BTST, JCLR, JSCLR, JSET, and JSSET) can also be used to address individual bits for faster I/O processing. The digital signal processor (DSP) does not have a hardware data strobe to strobe data out of the parallel I/O port. If a strobe is needed, it can be implemented using software to toggle one of the parallel I/O pins. The process of programming port B as general-purpose I/O is shown in Figure 10-7 and detailed in Figure 10-8. Normally, it is not good programming practice to activate a peripheral before programming it. However, reset activates the port B general-purpose I/O as all inputs; the alternative is to configure port B as an HI, which may not be desirable. In this case, it is probably better to insure that port B is initially configured for general-purpose I/O, and then configure the data direction and data registers. It may be better in some situations to program the data direction or the data registers first to prevent two devices from driving one signal. The order of steps 1, 2, and 3 in Figure 10-7 is optional and can be changed as needed.

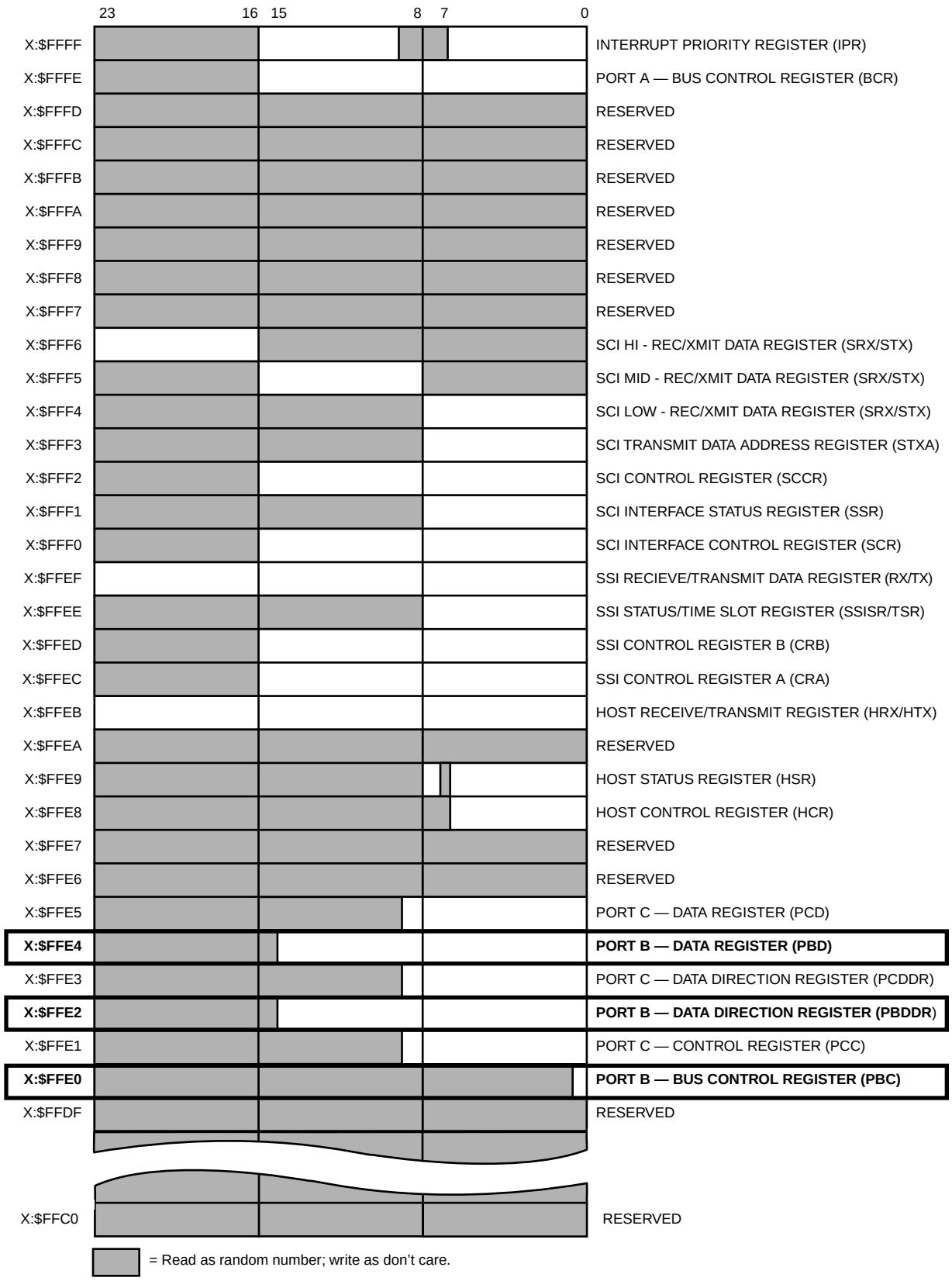


Figure 10-5 On-Chip Peripheral Memory Map

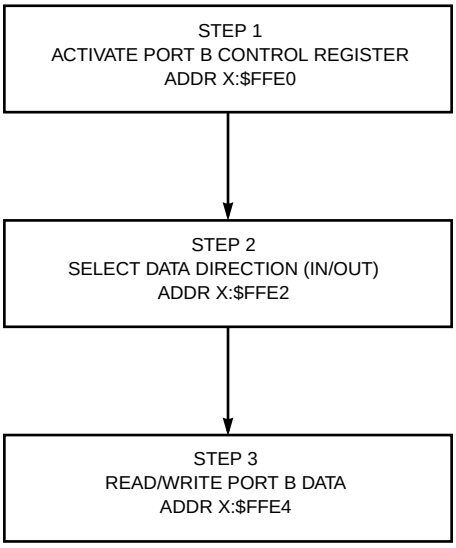


Figure 10-7 Port B Configuration Flowchart

10.1.2 Port B Parallel I/O Timing

Parallel data written to port B is synchronized to the central processing unit (CPU) but

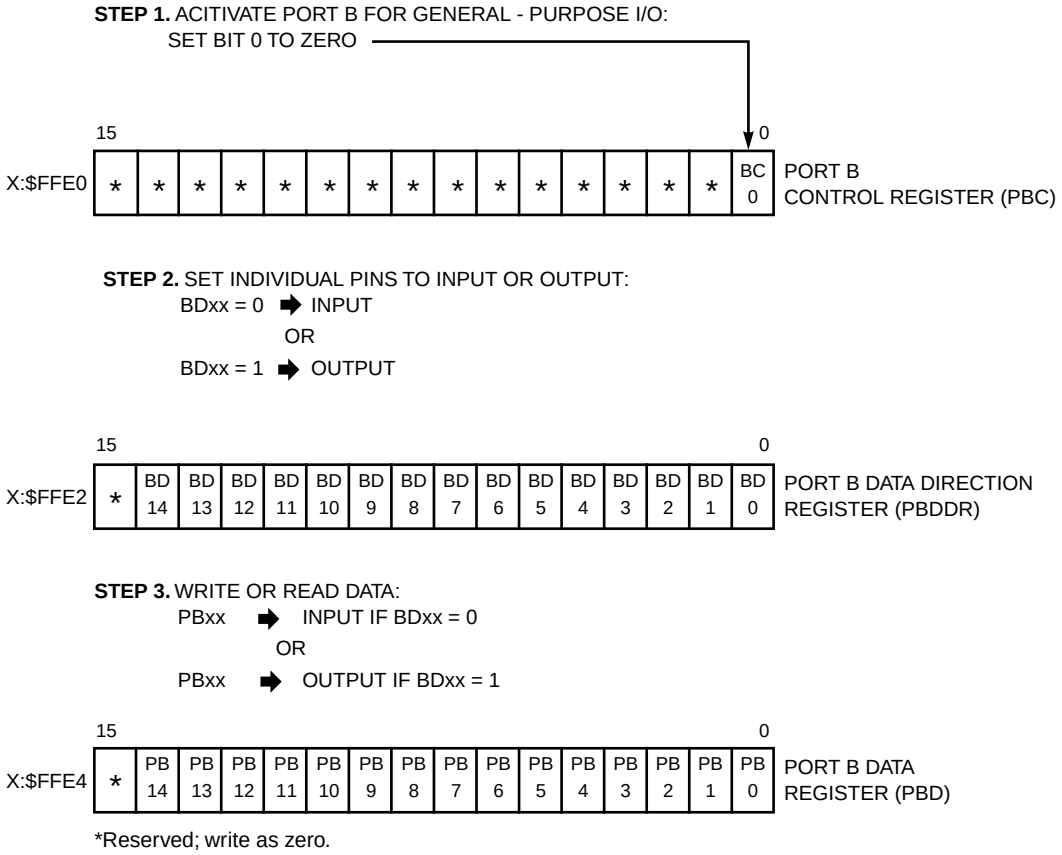


Figure 10-8 I/O Port B Configuration

delayed by one instruction cycle. For example, the instruction

```
MOVE    DATA15,X:PORTB    DATA24,Y:EXTERN
```

1) writes15 bits of data to the port B register, but the output pins do not change until the following instruction cycle, and 2) writes 24 bits of data to the external Y memory, which appears on port A during T2 and T3 of the current instruction.

As a result, if it is desirable to synchronize port A and port B outputs, two instructions must be used:

```
MOVE    DATA15,X:PORTB
NOP                       DATA24,Y:EXTERN
```

The NOP can be replaced by any instruction that allows parallel moves. Inserting one or more ‘‘MOVE DATA15,X:PORTB DATA24,Y:EXTERN’’ instructions between the first and second instruction effectively produces an external 39-bit write each instruction cycle with only one instruction cycle lost in setup time:

```
MOVE    DATA15,X:PORTB
MOVE    DATA15,X:PORTB    DATA24,Y:EXTERN
MOVE    DATA15,X:PORTB    DATA24,Y:EXTERN
:
:
MOVE    DATA15,X:PORTB    DATA24,Y:EXTERN
NOP                       DATA24,Y:EXTERN
```

One application of this technique is to create an extended address for port A by concatenating the port A address bits (instead of data bits) to the port B general-purpose output bits. The port B general-purpose I/O register would then work as a base address register, allowing the address space to be extended from 64K words (16 bits) to two billion words (16 bits+15 bits=31 bits).

Port B uses the DSP CPU four-phase clock for its operation. Therefore, if wait states are inserted in the DSP CPU timing, they also affect Port B timing. The result is that ports A and B in the previous synchronization example will always stay synchronized, regardless of how many wait states are used.

10.2 HOST INTERFACE (HI)

The HI is a byte-wide, full-duplex, double-buffered, parallel port which may be connected directly to the data bus of a host processor. The host processor may be any of a number of industry standard microcomputers or microprocessors, another DSP, or DMA hardware because this interface looks like static memory. The HI is asynchronous and consists of two banks of registers – one bank accessible to the host processor and a second bank accessible to the DSP CPU (see Figure 10-9). A brief description of the HI features is presented in the following listing:

Speed

8 Mbyte/Sec Burst Data Transfer Rate

1.71 Million Word/Sec Interrupt Driven Data Transfer Rate (This is the maximum interrupt rate for theDSP56000/DSP56001 running at 20.5 MHz – i.e., one interrupt every six instruction cycles.)

Signals (15 Pins)

H0–H7	Host Data Bus
HA0-HA2	Host Address Select
HR/$\overline{W}$	Host Read/Write Control
$\overline{HEN}$	Host Transfer Enable
$\overline{HREQ}$	Host Request
$\overline{HACK}$	Host Acknowledge

Interface – DSP CPU Side

Mapping: Three X: Memory Locations
Data Word: 24 Bits

Transfer Modes:
- DSP to Host
- Host to DSP
- Host Command

Handshaking Protocols:
- Software Polled
- Interrupt Driven (Fast or Long)
- Direct Memory Access

Instructions:
- Memory-mapped registers allow the standard MOVE instruction to be used.
- Special MOVEP instruction provides for I/O service capability using fast interrupts.
- Bit addressing instructions (BCHG, BCLR, BSET, BTST, JCLR, JSCLR, JSET, JSSET) simplify I/O service routines.
- I/O short addressing provides faster execution with fewer instruction words.

Interface – Host Side

Mapping:
- Eight Consecutive Memory Locations
- Memory-Mapped Peripheral for Microprocessors, DMA Controllers, etc.

Data Word: Eight Bits

Transfer Modes:
- DSP to Host
- Host to DSP
- Host Command
- Mixed 8-, 16-, and 24-Bit Data Transfers

Handshaking Protocols:
- Software Polled
- Interrupt Driven and Compatible with MC68000
- Cycle Stealing DMA with Initialization

Dedicated Interrupts:
- Separate Interrupt Vectors for Each Interrupt Source
- Special host commands force DSP CPU interrupts under host processor control, which are useful for:
  - Real-Time Production Diagnostics
  - Debugging Window for Program Development
  - Host Control Protocols and DMA Setup

Figure 10-9 is a block diagram showing the registers in the HI. These registers can be divided vertically down the middle into registers visible to the host processor on the left and registers visible to the DSP on the right. They can also be divided horizontally into control at the top, DSP-to-host data transfer in the middle (HTX, RXH, RXM, and RXL), and host-to-DSP data transfer at the bottom (THX, TXM, TXL, and HRX).

10.2.1 Host Interface – DSP CPU Viewpoint

The DSP CPU views the HI as a memory-mapped peripheral occupying three 24-bit words in data memory space. The DSP may use the HI as a normal memory-mapped peripheral, using either standard polled or interrupt programming techniques. Separate transmit and receive data registers are double buffered to allow the DSP and host processor to efficiently transfer data at high speed. Memory mapping allows DSP CPU communication with the HI registers to be accomplished using standard instructions and addressing modes. In addition, the MOVEP instruction allows HI-to-memory and memory-to-HI data transfers without going through an intermediate register. Both hardware and software reset disable the HI and change port B to general-purpose I/O with all pins designated as inputs.

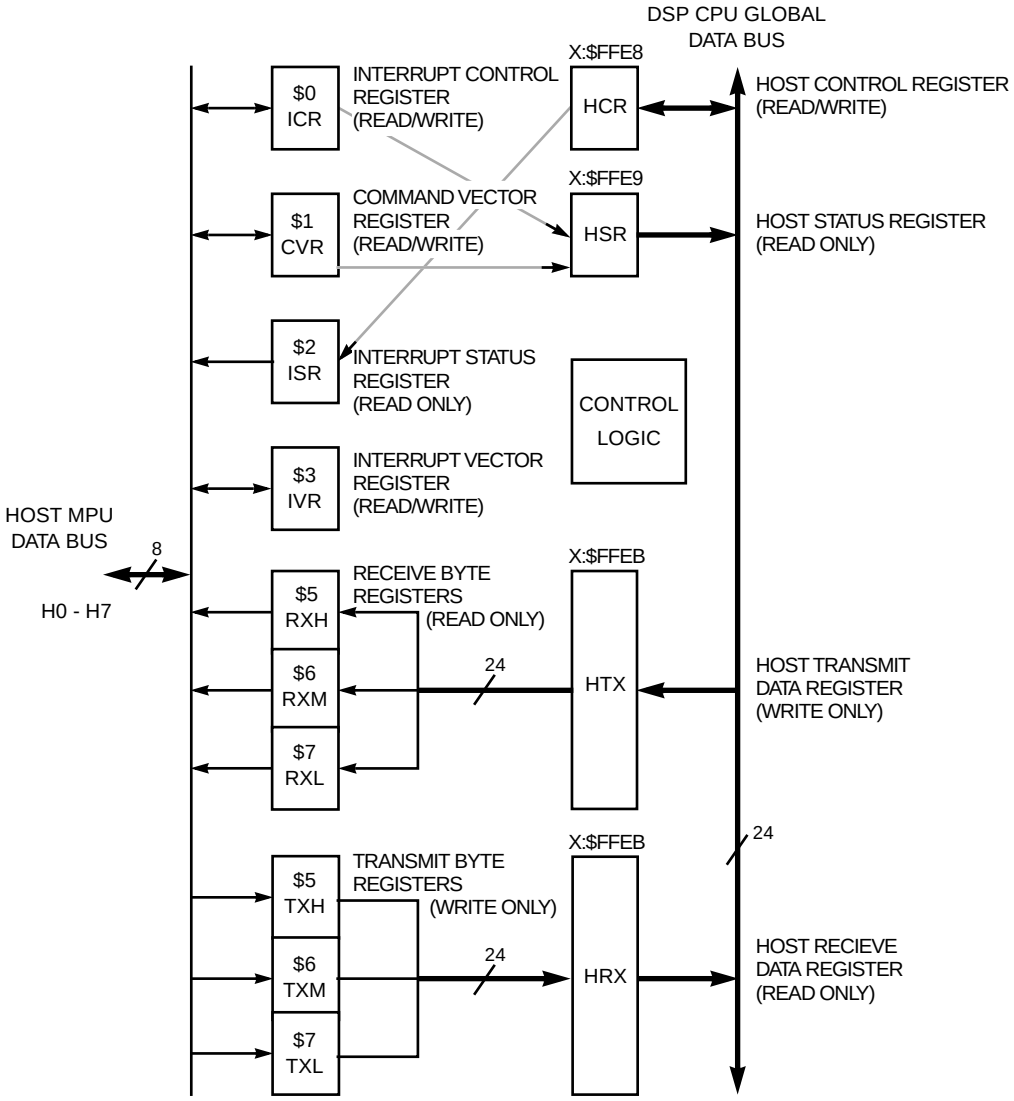
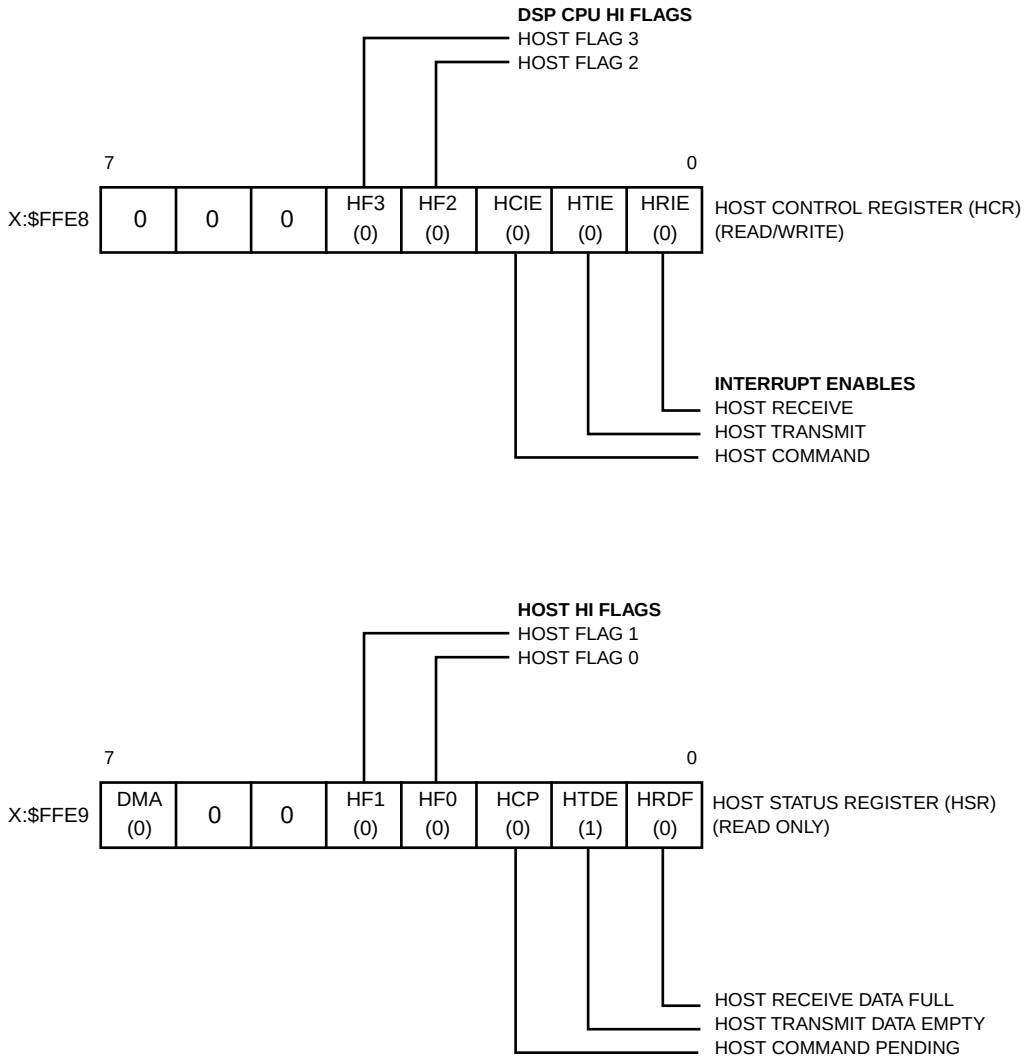


Figure 10-9 HI Block Diagram

10.2.2 Programming Model – DSP CPU Viewpoint

The HI has two programming models: one for the DSP programmer and one for the host processor programmer. In most cases, the notation used reflects the DSP perspective. The HI – DSP programming model is shown in Figure 10-10. There are three registers: a control register (HCR), a status register (HSR), and a data transmit/receive register (HTX/HRX). These registers can only be accessed by the DSP56000/DSP56001; they can not be accessed by the host processor. The HI host processor programming model is shown in Figure 10-13.

The following paragraphs describe the purpose and operation of each bit in each register of the HI visible to the DSP CPU. The effects of the different types of reset on these registers are shown. A brief discussion of interrupts and operation of the DSP side of the HI complete the programming model from the DSP viewpoint. The programming model from the host viewpoint



NOTE: The numbers in parentheses are reset values.

Figure 10-10 Host Interface Programming Model – DSP Viewpoint

begins at 10.2.3.1 Programming Model – Host Processor Viewpoint.

10.2.2.1 Host Control Register (HCR)

The HCR is an 8-bit read/write control register used by the DSP to control the HI interrupts and flags. The HCR cannot be accessed by the host processor. The HCR occupies the low-order byte of the internal data bus; the high-order portion is zero filled. HCR is a read/write register to allow individual control register bits to be set or cleared. Any reserved bits are read as zeros and should be programmed as zeros for future compatibility. The bit manipulation instructions are useful for accessing the individual bits. The contents of HCR are cleared on hardware or software reset. The control bits are described in the following paragraphs.

10.2.2.1.1 HCR Host Receive Interrupt Enable (HRIE) Bit 0

The HRIE bit is used to enable a DSP interrupt when the host receive data full (HRDF) status bit in the host status register (HSR) is set. When HRIE is cleared, HRDF interrupts are disabled. When HRIE is set, a host receive data interrupt request will occur if HRDF is set. Hardware and software resets clear HRIE.

10.2.2.1.2 HCR Host Transmit Interrupt Enable (HTIE) Bit 1

The HTIE bit is used to enable a DSP interrupt when the host transmit data empty (HTDE) status bit in the HSR is set. When HTIE is cleared, HTDE interrupts are disabled. When HTIE is set, a host transmit data interrupt request will occur if HTDE is set. Hardware and software resets clear the HTIE.

10.2.2.1.3 HCR Host Command Interrupt Enable (HCIE) Bit 2

The HCIE bit is used to enable a vectored DSP interrupt when the host command pending (HCP) status bit in the HSR is set. When HCIE is cleared, HCP interrupts are disabled. When HCIE is set, a host command interrupt request will occur if HCP is set. The starting address of this interrupt is determined by the host vector (HV). Hardware and software resets clear the HCIE.

10.2.2.1.4 HCR Host Flag 2 (HF2) Bit 3

The HF2 bit is used as a general-purpose flag for DSP-to-host communication. HF2 may be set or cleared by the DSP. HF2 is visible in the interrupt status register (ISR) on the host processor side (see Figure 10-11). Hardware and software resets clear HF2.

There are four host flags: two used by the host to signal the DSP (HF0 and HF1) and two used by the DSP to signal the host processor (HF2 and HF3). These flags are not designated for any specific purpose but are general-purpose flags. The host flags do not cause interrupts; they must be polled to see if they have changed. These flags can be used individually or as encoded pairs. See 10.2.2.7 Host Port Usage Considerations – DSP Side for additional information. An example of the usage of host flags is the bootstrap loader,which is listed in the DSP56001 Advance Information Data Sheet (ADI1290). Host flags are used to tell the bootstrap program whether or not to terminate early.

10.2.2.1.5 HCR Host Flag 3 (HF3) Bit 4

The HF3 bit is used as a general-purpose flag for DSP-to-host communication. HF3 may be set or cleared by the DSP. HF3 is visible in the ISR on the host processor side (see Figure 10-11). Hardware and software resets clear HF3.

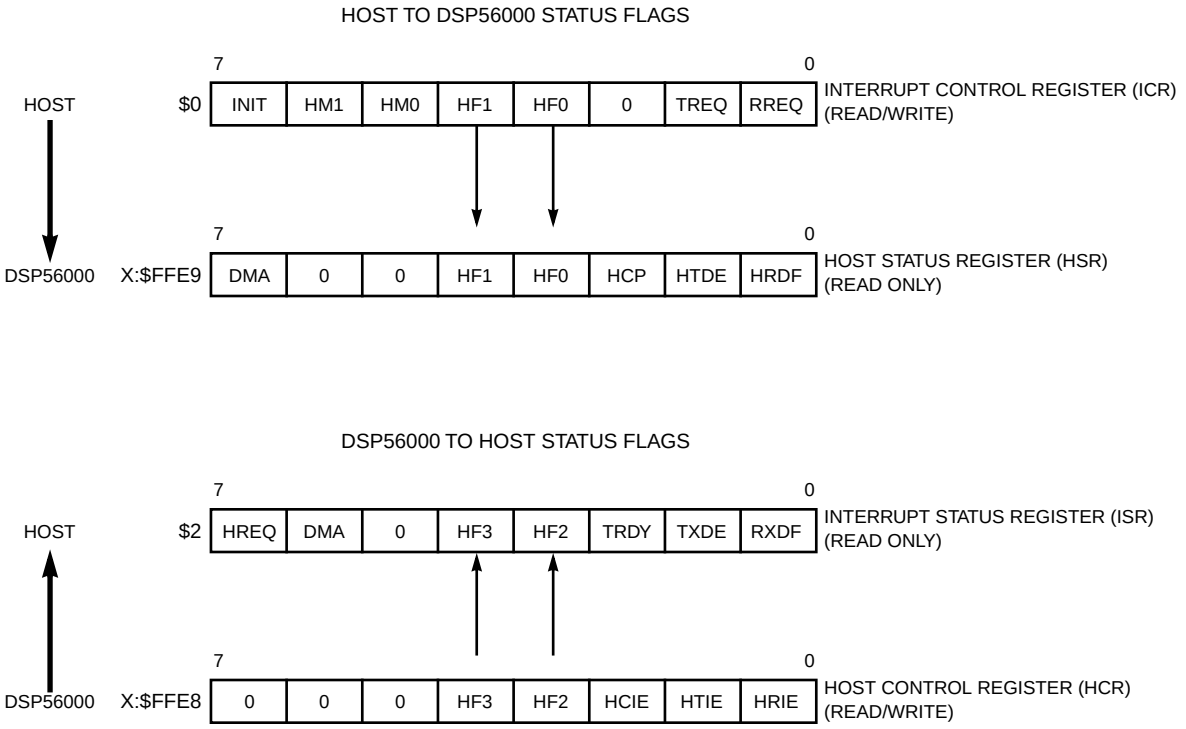


Figure 10-11 Host Flag Operation

10.2.2.1.6 HCR Reserved Control (Bits 5, 6, and 7). These unused bits are reserved for future expansion and should be written with zeros for upward compatibility.

10.2.2.2 Host Status Register (HSR). The HSR is an 8-bit read-only status register used by the DSP to interrogate status and flags of the HI. It can not be directly accessed by the host processor. When the HSR is read to the internal data bus, the register contents occupy the low-order byte of the data bus; the high-order portion is zero filled. The status bits are described in the following paragraphs.

10.2.2.2.1 HSR Host Receive Data Full (HRDF) Bit 0. The HRDF bit indicates that the host receive data register (HRX) contains data from the host processor. HRDF is set when data is transferred from the TXH:TXM:TXL registers to the HRX register. HRDF is cleared when HRX is read by the DSP. HRDF can also be cleared by the host processor using the initialize function. Hardware, software, individual, and STOP resets clear HRDF.

10.2.2.2.2 HSR Host Transmit Data Empty (HTDE) Bit 1. The HTDE bit indicates that the host transmit data register (HTX) is empty and can be written by the DSP. HTDE is set when the HTX register is transferred to the RXH:RXM:RXL registers. HTDE is cleared when HTX is written by the DSP. HTDE can also be set by the host processor using the initialize function. Hardware, software, individual, and STOP sets HTDE.

10.2.2.2.3 HSR Host Command Pending (HCP) Bit 2

The HCP bit indicates that the host has set the HC bit and that a host command interrupt is pending. The HCP bit reflects the status of the HC bit in the command vector register (CVR). HC and HCP are cleared by the DSP exception hardware when the exception is taken. The host can clear HC, which also clears HCP. Hardware, software, individual, and STOP resets clear HCP.

10.2.2.2.4 HSR Host Flag 0 (HF0) Bit 3

The HF0 bit in the HSR indicates the state of host flag 0 in the ICR on the host processor side. HF0 can only be changed by the host processor (see Figure 10-11). Hardware, software, individual, and STOP resets clear HF0.

10.2.2.2.5 HSR Host Flag 1 (HF1) Bit 4

The HF1 bit in the HSR indicates the state of host flag 1 in the ICR on the host processor side. HF1 can only be changed by the host processor (see Figure 10-11). Hardware, software, individual, and STOP resets clear HF1.

10.2.2.2.6 HSR Reserved Status (Bits 5 and 6)

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

10.2.2.2.7 HSR DMA Status (DMA) Bit 7

The DMA bit indicates that the host processor has enabled the DMA mode of the HI by setting HM1 or HM0 to one. When DMA bit is zero, it indicates that the DMA mode is disabled by the HM0 and HM1 bits in the ICR and that no DMA operations are pending. When DMA bit is set, the DMA mode has been enabled by one or more of the host mode bits being set to one. The channel not in use can be used for polled or interrupt operation by the DSP. Hardware, software, individual, and STOP resets clear the DMA bit.

10.2.2.3 Host Receive Data Register (HRX)

The HRX register is used for host-to-DSP data transfers. The HRX register is viewed as a 24-bit read-only register by the DSP CPU. The HRX register is loaded with 24-bit data from the transmit data registers (TXH:TXM:TXL) on the host processor side when both the transmit data register empty TXDE (host processor side) and DSP host receive data full (HRDF) bits are cleared. This transfer operation sets TXDE and HRDF. The HRX register contains valid data when the HRDF bit is set. Reading HRX clears HRDF. The DSP may program the HRIE bit to cause a host receive data interrupt when HRDF is set. Resets do not affect HRX.

10.2.2.4 Host Transmit Data Register (HTX)

The HTX register is used for DSP-to-host data transfers. The HTX register is viewed as a 24-bit write-only register by the DSP CPU. Writing the HTX register clears HTDE. The DSP may program the HTIE bit to cause a host transmit data interrupt when HTDE is set. The HTX register is transferred as 24-bit data to the receive byte registers (RXH:RXM:RXL) if both the HTDE bit (DSP CPU side) and receive data full (RXDF) status bits (host processor side) are cleared. This transfer operation sets RXDF and HTDE. Data should not be written to the HTX until HTDE is set to prevent the previous data from being overwritten. Resets do not affect HTX.

10.2.2.5 Register Contents After Reset

Table 10-1 shows the results of four reset types on bits in each of the HI registers seen by the DSP CPU. The hardware reset (HW) is caused by the $\overline{\text{RESET}}$ signal; the software reset (SW) is caused by executing the RESET instruction; the individual reset (IR) is caused by clearing the PBC register bit 0, and the stop reset (ST) is caused by executing the STOP instruction.

Table 10-1 Host Registers after Reset–DSP CPU Side

Register Name	Register Data	Reset Type			
		HW Reset	SW Reset	IR Reset	ST Reset
HCR	HF(3 - 2)	0	0	—	—
	HCIE	0	0	—	—
	HTIE	0	0	—	—
	HRIE	0	0	—	—
HSR	DMA	0	0	0	0
	HF(1 - 0)	0	0	0	0
	HCP	0	0	0	0
	HTDE	1	1	1	1
	HRDF	0	0	0	0
HRX	HRX (23 - 0)	—	—	—	—
HTX	HTX (23 - 0)	—	—	—	—

10.2.2.6 Host Interface DSP CPU Interrupts

The HI may request interrupt service from either the DSP or the host processor. The DSP CPU interrupts are internal and do not require the use of an external interrupt pin (see Figure 10-12). When the appropriate mask bit in the HCR is set, an interrupt condition caused

by the host processor sets the appropriate bit in the HSR, which generates an interrupt request to the DSP CPU. The DSP acknowledges interrupts caused by the host processor by jumping to the appropriate interrupt service routine. The three possible interrupts are 1) receive data register full, 2) transmit data register empty, and 3) host command. The host command can access any interrupt vector in the interrupt vector table although it has a set of vectors reserved for host command use. The DSP interrupt service routine must read or write the appropriate HI register (clearing HRDF or HTDE, for example) to clear the interrupt. In the case of host command interrupts, the interrupt acknowledge from the program controller will clear the pending interrupt condition.

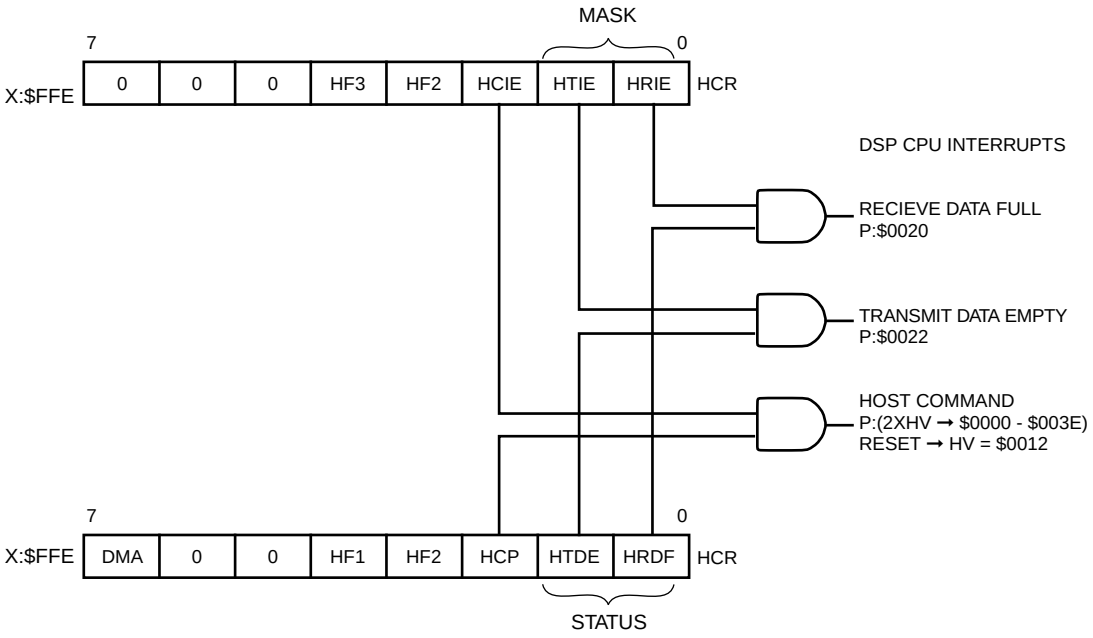


Figure 10-12 HSR–HCR Operation

10.2.2.7 Host Port Usage Considerations – DSP Side

Careful synchronization is required when reading multibit registers that are written by another asynchronous system. This is a common problem when two asynchronous systems are connected. The situation exists in the HI. However, if the HI is used in the way it was designed, proper operation is guaranteed. The considerations for proper operation on the DSP CPU side are discussed in the following paragraphs, and considerations for the host processor side are discussed in 10.2.6.5 Host Port Usage Considerations–Host Side. Careful synchronization is required when reading multi-bit registers that are written by another asynchronous system. Synchronization is a common problem when two asynchronous systems are connected. The situation exists in the host port. However, if the port is used in the way it was designed, proper operation is guaranteed. The considerations for proper operation are discussed below..

DMA, HF1, HF0, HCP, HTDE, and HRDF status bits are set or cleared by the host processor side of the interface. These bits are individually synchronized to the DSP clock.

The only system problem with reading status occurs with HF1 and HF0 if they are encoded as a pair because the four combinations (00, 01, 10, and 11) each have significance. This problem occurs because there is a very small probability that the DSP will read the status bits during the transition. The solution to this potential problem is to read the bits twice for consensus (See 10.2.6.5 Host Port Usage Considerations–Host Side. Careful synchronization is required when reading multi-bit registers that are written by another asynchronous system. Synchronization is a common problem when two asynchronous systems are connected. The situation exists in the host port. However, if the port is used in the way it was designed, proper operation is guaranteed. The considerations for proper operation are discussed below. for additional information).

10.2.3 Host Interface – Host Processor Viewpoint

The HI appears to the host processor as eight words of byte-wide static memory. The host may access the HI asynchronously by using polling techniques or interrupt-based techniques. Separate transmit and receive data registers are double buffered to allow the DSP CPU and host processor to transfer data efficiently at high speed. The HI contains a rudimentary DMA controller, which makes generating addresses (HA0–HA2) for the TX/RX registers in the HI unnecessary.

10.2.3.1 Programming Model – Host Processor Viewpoint

The HI appears to the host processor as a memory-apped peripheral occupying eight bytes in the host processor address space (see Figures 10-13 and 10-14). These registers can be viewed as one control register (ICR), one status register (ISR), three data registers (RXH/TXH, RXM/TXM, and RXL/TXL), and two vector registers (IVR and CVR). The CVR is a special command register that is used by the host processor to issue commands to the DSP. These registers can be accessed only by the host processor; they can not be accessed by the DSP CPU. Host processors may use standard host processor instructions (e.g., byte move) and addressing modes to communicate with the HI registers. The HI registers are addressed so that 8-bit MC6801-type host processors can use 16-bit load (LDD) and store (STD) instructions for data transfers. The 16-bit MC68000/MC68010 host processor can address the HI using the special MOVEP instruction for word (16-bit) or long-word (32-bit) transfers. The 32-bit MC68020 host processor can use its dynamic bus sizing feature to address the HI using standard MOVE word (16-bit), long-word (32-bit) or quad-word (64-bit) instructions. The $\overline{\text{HREQ}}$ and $\overline{\text{HACK}}$ handshake flags are provided for polled or interrupt-driven data transfers with the host processor. Because the DSP interrupt response is sufficiently fast, most host microprocessors can load or store data at their maximum programmed I/O (non-DMA) instruction rate without testing the handshake flags for each transfer. If the full handshake is not

needed, the host processor can treat the DSP as fast memory, and data can be transferred between the host processor and the DSP at the fastest host processor data rate. DMA hardware may be used with the handshake flags to transfer data without host processor intervention.

One of the most innovative features of the host interface is the host command feature. With this feature, the host processor can issue vectored exception requests to the DSP56000/DSP56001. The host may select any one of 32 DSP56000/DSP56001 exception routines to be executed by writing a vector address register in the HI. This flexibility allows the host programmer to execute up to 32 preprogrammed functions inside the DSP56000/DSP56001. For example, host exceptions can allow the host processor to read or write DSP56000/DSP56001 registers (X, Y, or program memory locations), force exception handlers (e.g., SSI, SCI, $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$ exception routines), and perform control and debugging operations if exception routines are implemented in the DSP56000/DSP56001 to perform these tasks.

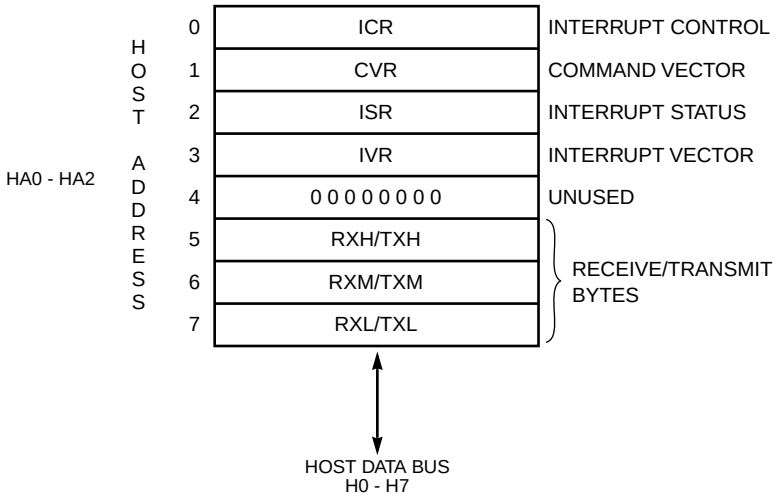


Figure 10-14 HI Register Map

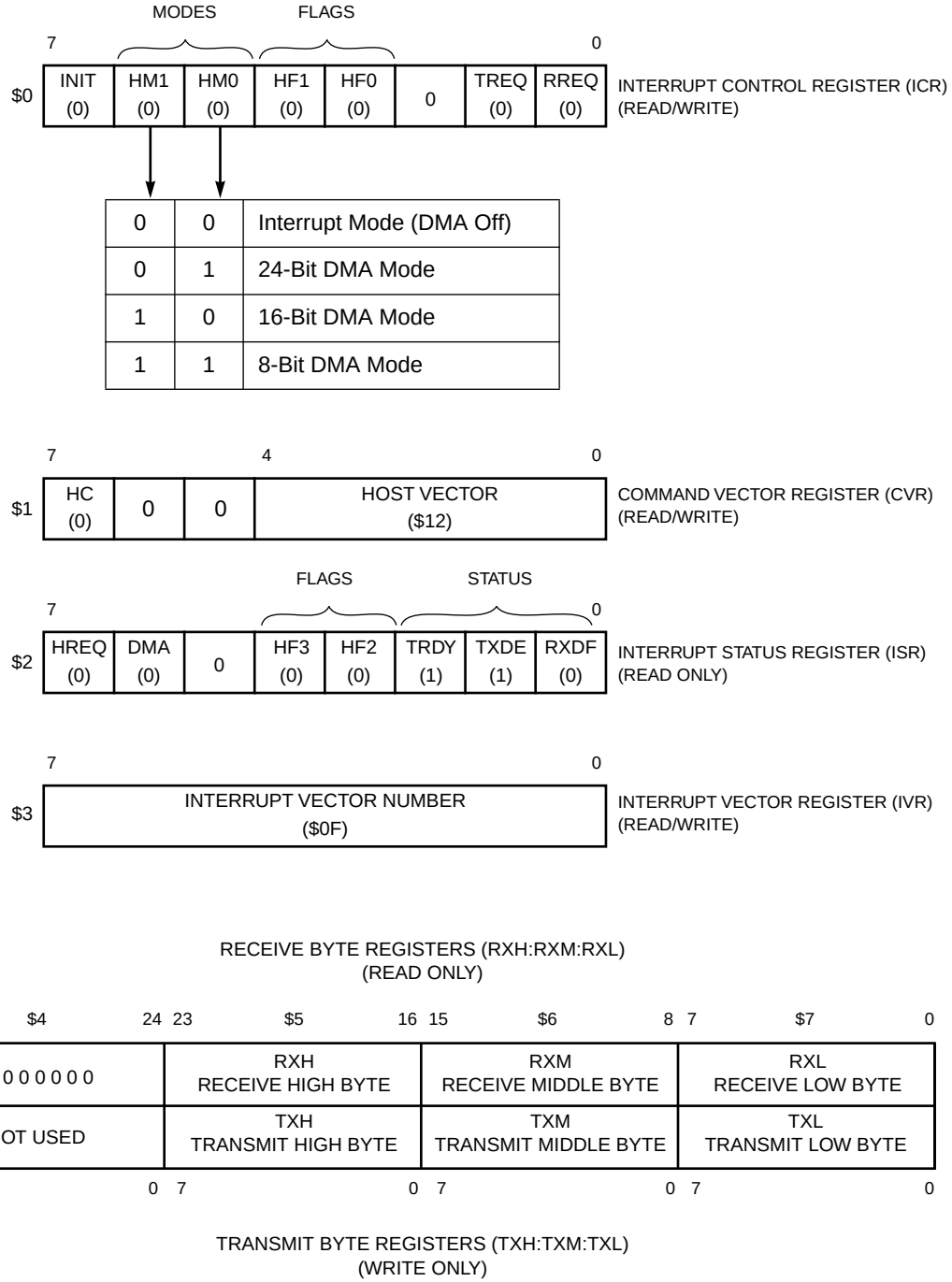
10.2.3.2 Interrupt Control Register (ICR)

The ICR is an 8-bit read/write control register used by the host processor to control the HI interrupts and flags. ICR cannot be accessed by the DSP CPU. ICR is a read/write register, which allows the use of bit manipulation instructions on control register bits. The control bits are described in the following paragraphs.

10.2.3.2.1 ICR Receive Request Enable (RREQ) Bit 0

The RREQ bit is used to control the $\overline{\text{HREQ}}$ pin for host receive data transfers.

In interrupt mode (DMA off), RREQ is used to enable interrupt requests via the external



NOTE: The numbers in parentheses are reset values.

Figure 10-13 Host Processor Programming Model–Host Side

host request ($\overline{\text{HREQ}}$) pin when the receive data register full (RXDF) status bit in the ISR is set. When RREQ is cleared, RXDF interrupts are disabled. When RREQ is set, the external $\overline{\text{HREQ}}$ pin will be asserted if RXDF is set.

In DMA modes, RREQ must be set or cleared by software to select the direction of DMA transfers. Setting RREQ sets the direction of DMA transfer to be DSP to host and enables the $\overline{\text{HREQ}}$ pin to request data transfer. Hardware, software, individual, and STOP resets clear RREQ.

10.2.3.2.2 ICR Transmit Request Enable (TREQ) Bit 1

The TREQ bit is used to control the $\overline{\text{HREQ}}$ pin for host transmit data transfers.

In interrupt mode (DMA off), TREQ is used to enable interrupt requests via the external $\overline{\text{HREQ}}$ pin when the transmit data register empty (TXDE) status bit in the ISR is set. When TREQ is cleared, TXDE interrupts are disabled. When TREQ is set, the external $\overline{\text{HREQ}}$ pin will be asserted if TXDE is set.

In DMA modes, TREQ must be set or cleared by software to select the direction of DMA transfers. Setting TREQ sets the direction of DMA transfer to be host to DSP and enables the $\overline{\text{HREQ}}$ pin to request data transfer. Hardware, software, individual, and STOP resets clear TREQ.

Table 10-2 summarizes the effect of RREQ and TREQ on the $\overline{\text{HREQ}}$ pin.

Table 10-2 $\overline{\text{HREQ}}$ Pin Definition

TREQ	RREQ	$\overline{\text{HREQ}}$ Pin
Interrupt Mode		
0	0	No Interrupts (Polling)
0	1	RXDF Request (Interrupt)
1	0	TXDE Request (Interrupt)
1	1	RXDF and TXDE Request (Interrupts)
DMA Mode		
0	0	No DMA
0	1	DSP to Host Request (RX)
1	0	Host to DSP Request (TX)
1	1	Undefined (Illegal)

10.2.3.2.3 ICR Reserved Bit (Bit 2)

This bit, which is reserved and unused, reads as a logic zero.

10.2.3.2.4 ICR Host Flag 0 (HF0) Bit 3

The HF0 bit is used as a general-purpose flag for host-to-DSP communication. HF0 may be set or cleared by the host processor and cannot be changed by the DSP. HF0 is visible in the HSR on the DSP CPU side of the HI (see Figure 10-11). Hardware, software, individual, and STOP resets clear HF0.

10.2.3.2.5 ICR Host Flag 1 (HF1) Bit 4

The HF1 bit is used as a general-purpose flag for host-to-DSP communication. HF1 may be set or cleared by the host processor and cannot be changed by the DSP. Hardware, software, individual, and STOP resets clear HF1.

10.2.3.2.6 ICR Host Mode Control (HM1 and HM0 bits) Bits 5 and 6

The HM0 and HM1 bits select the transfer mode of the HI (see Table 10-3). HM1 and HM0 enable the DMA mode of operation or interrupt (non-DMA) mode of operation.

When both HM1 and HM0 are cleared, the DMA mode is disabled, and the TREQ and RREQ control bits are used for host processor interrupt control via the external $\overline{\text{HREQ}}$ output pin. Also, in the non-DMA mode, the $\overline{\text{HACK}}$ input pin is used for the MC68000 Family vectored interrupt acknowledge input.

Table 10-3 Host Mode Bit Definition

HM1	HM0	Mode
0	0	Interrupt Mode (DMA Off)
0	1	DMA Mode (24 Bit)
1	0	DMA Mode (16 Bit)
1	1	DMA Mode (8 Bit)

When HM1 or HM0 are set, the DMA mode is enabled, and the $\overline{\text{HREQ}}$ pin is used to request DMA transfers. When the DMA mode is enabled, the TREQ and RREQ bits select the direction of DMA transfers. The $\overline{\text{HACK}}$ input pin is used as a DMA transfer acknowledge input. If the DMA direction is from DSP to host, the contents of the selected register are enabled onto the host data bus when $\overline{\text{HACK}}$ is asserted. If the DMA direction is from host to DSP, the selected register is written from the host data bus when $\overline{\text{HACK}}$ is asserted.

The size of the DMA word to be transferred is determined by the DMA control bits, HM0 and HM1. The HI register selected during a DMA transfer is determined by a 2-bit address counter, which is preloaded with the value in HM1 and HM0. The address counter substitutes for the HA1 and HA0 bits of the HI during a DMA transfer. The host address bit (HA2) is forced to one during each DMA transfer. The address counter can be initialized with the INIT bit feature. After each DMA transfer on the host data bus, the address counter is incremented to the next register. When the address counter reaches the highest register (RXL or TXL), the address counter is not incremented but is loaded with the value in HM1 and HM0. This allows 8-, 16- or 24-bit data to be transferred in a circular fashion and eliminates the need for the DMA controller to supply the HA2, HA1, and HA0 pins. For 16- or 24-bit data transfers, the DSP CPU interrupt rate is reduced by a factor of 2 or 3, respectively, from the host request rate – i.e., for every two or three host processor data transfers of one byte each, there is only one 24-bit DSP CPU interrupt.

Hardware, software, individual, and STOP resets clear HM1 and HM0.

10.2.3.2.7 ICR Initialize Bit (INIT) Bit 7

The INIT bit is used by the host processor to force initialization of the HI hardware. Initialization consists of configuring the HI transmit and receive control bits and loading HM1 and HM0 into the internal DMA address counter. Loading HM1 and HM0 into the DMA address counter causes the HI to begin transferring data on a word boundary rather than transferring only part of the first data word.

There are two methods of initialization: 1) allowing the DMA address counter to be automatically set after transferring a word, and 2) setting the INIT bit, which sets the DMA address counter. Using the INIT bit to initialize the HI hardware may or may not be necessary, depending on the software design of the interface.

The type of initialization done when the INIT bit is set depends on the state of TREQ and RREQ in the HI. The INIT command, which is local to the HI, is designed to conveniently configure the HI into the desired data transfer mode. The commands are described in the following paragraphs and in Table 10-4. The host sets the INIT bit, which causes the HI hardware to execute the INIT command. The interface hardware clears the INIT bit when the command has been executed. Hardware, software, individual, and STOP resets clear INIT.

INIT execution always loads the DMA address counter and clears the channel according to TREQ and RREQ. INIT execution is not affected by HM1 and HM0.

Table 10-4 $\overline{\text{HREQ}}$ Pin Definition

TREQ	RREQ	After INIT Execution	Transfer Direction Initialized
Interrupt Mode (HM1 = 0, HM0 = 0) INIT Execution			
0	0	INIT = 0; Address Counter = 00	None
0	1	INIT = 0; RXDF = 0; HTDE = 1; Address Counter = 00	DSP to Host
1	0	INIT = 0; TXDE = 1; HRDF = 0; Address Counter = 00	Host to DSP
1	1	INIT = 0; RXDF = 0; HTDE = 1; TXDE = 1; HRDF = 0; Address Counter = 00	Host to/from DSP
DMA Mode (HM1 or HM0 = 1) INIT Execution			
0	0	INIT = 0; Address Counter = HM1, HM0	None
0	1	INIT = 0; RXDF = 0; HTDE = 1; Address Counter = HM1, HM0	DSP to Host
1	0	INIT = 0; TXDE = 1; HRDF = 0; Address Counter = HM1, HM0	Host to DSP
1	1	Undefined (Illegal)	Undefined

The internal DMA counter is incremented with each DMA transfer (each $\overline{\text{HACK}}$ pulse) until it reaches the last data register (RXL or TXL). When the DMA transfer is completed, the counter is loaded with the value of the HM1 and HM0 bits. When changing the size of the DMA word (changing HM0 and HM1 in the ICR), the DMA counter is not automatically updated, and, as a result, the DMA counter will point to the wrong data register immediately after HM1 and HM0 are changed. The INIT function must be used to preset the internal DMA counter correctly. Always set INIT after changing HM0 and HM1. However, the DMA counter can not be initialized in the middle of a DMA transfer. Even though the INIT bit is set, the internal DMA controller will wait until after completing the data transfer in progress before executing the initialization.

10.2.3.3 Command Vector Register (CVR). The CVR is used by the host processor to cause the DSP to execute a vectored interrupt. The host command feature is independent of any of the data transfer mechanisms in the HI. It can be used to cause any of the 32 possible interrupt routines in the DSP CPU to be executed.

10.2.3.3.1 CVR Host Vector (HV) Bits 0–4. The five HV bits select the host command exception address to be used by the host command exception logic. When the host command exception is recognized by the DSP interrupt control logic, the starting address of the exception taken is 2×HV. The host can write HC and HV in the same write cycle, if desired.

The host processor can select any of the 32 possible exception routine starting addresses in the DSP by writing the exception routine starting address divided by 2 into HV. This means that the host processor can force any of the existing exception handlers (SSI, SCI, IRQA, IRQB, etc.) and can use any of the reserved or otherwise unused starting addresses provided they have been preprogrammed in the DSP. HV is set to $12 (vector location $0024) by hardware, software, individual, and STOP resets. Vector location $0024 is the first of thirteen special host command vectors.

CAUTION

The HV should not be used with a value of zero because the reset location is normally programmed with a JMP instruction. Doing so will cause an improper fast interrupt.

10.2.3.3.2 CVR Reserved Bits (Bits 5 and 6). Reserved bits are unused and are read by the host processor as zeros.

10.2.3.3.3 CVR Host Command Bit (HC) Bit 7. The HC bit is used by the host processor to handshake the execution of host command exceptions. Normally, the host processor sets HC=1 to request the host command exception from the DSP. When the host command exception is acknowledged by the DSP, the HC bit is cleared by the HI hardware. The host processor can read the state of HC to determine when the host command has been accepted. The host processor may elect to clear the HC bit, canceling the host command exception request at any time

before it is accepted by the DSP CPU.

CAUTION

The command exception might be recognized by the DSP and executed before it can be canceled by the host, even if the host clears the HC bit.

Setting HC causes host command pending (HCP) to be set in the HSR. The host can write HC and HV in the same write cycle if desired. Hardware, software, individual, and STOP resets clear HC.

10.2.3.4 Interrupt Status Register (ISR)

The ISR is an 8-bit read-only status register used by the host processor to interrogate the status and flags of the HI. The host processor can write this address without affecting the internal state of the HI, which is useful if the user desires to access all of the HI registers by stepping through the HI addresses. The ISR can not be accessed by the DSP. The status bits are described in the following paragraphs.

10.2.3.4.1 ISR Receive Data Register Full (RXDF) Bit 0

The RXDF bit indicates that the receive byte registers (RXH, RXM, and RXL) contain data from the DSP CPU and may be read by the host processor. RXDF is set when the HTX is transferred to the receive byte registers. RXDF is cleared when the receive data low (RXL) register is read by the host processor. RXL is normally the last byte of the receive byte registers to be read by the host processor. RXDF can be cleared by the host processor using the initialize function. RXDF may be used to assert the external $\overline{\text{HREQ}}$ pin if the RREQ bit is set. Regardless of whether the RXDF interrupt is enabled, RXDF provides valid status so that polling techniques may be used by the host processor. Hardware, software, individual, and STOP resets clear RXDF.

10.2.3.4.2 ISR Transmit Data Register Empty (TXDE) Bit 1

The TXDE bit indicates that the transmit byte registers (TXH, TXM, and TXL) are empty and can be written by the host processor. TXDE is set when the transmit byte registers are transferred to the HRX register. TXDE is cleared when the transmit byte low (TXL) register is written by the host processor. TXL is normally the last byte of the transmit byte registers to be written by the host processor. TXDE can be set by the host processor using the initialize feature. TXDE may be used to assert the external $\overline{\text{HREQ}}$ pin if the TREQ bit is set. Regardless of whether the TXDE interrupt is enabled, TXDE provides valid status so that polling techniques may be used by the host processor. Hardware, software, individual, and STOP resets set TXDE.

10.2.3.4.3 ISR Transmitter Ready (TRDY) Bit 2

The TRDY status bit indicates that **both** the TXH,TXM,TXL and the HRX registers are

empty.

$$TRDY = TXDE \bullet \overline{HRDF}$$

When TRDY is set to one, the data that the host processor writes to TXH,TXM, and TXL will be immediately transferred to the DSP CPU side of the HI. This has many applications. For example, if the host processor issues a host command which causes the DSP CPU to read the HRX, the host processor can be guaranteed that the data it just transferred to the HI is what is being received by the DSP CPU.

Hardware, software, individual, and STOP resets set TRDY.

10.2.3.4.4 ISR Host Flag 2 (HF2) Bit 3

The HF2 bit in the ISR indicates the state of host flag 2 in the HCR on the CPU side. HF2 can only be changed by the DSP (see Figure 10-11). HF2 is cleared by a hardware or software reset.

10.2.3.4.5 ISR Host Flag 3 (HF3) Bit 4

The HF3 bit in the ISR indicates the state of host flag 3 in the HCR on the CPU side. HF3 can only be changed by the DSP (see Figure 10-11). HF3 is cleared by a hardware or software reset.

10.2.3.4.6 ISR Reserved Bit (Bit 5)

This status bit is reserved for future expansion and will read as zero during host processor read operations.

10.2.3.4.7 ISR DMA Status (DMA) Bit 6

The DMA status bit indicates that the host processor has enabled the DMA mode of the HI (HM1 or HM0=1). When the DMA status bit is clear, it indicates that the DMA mode is disabled (HM0=HM1=0) and no DMA operations are pending. When DMA is set, it indicates that the DMA mode is enabled and the host processor should not use the active DMA channel (RXH, RXM, RXL or TXH, TXM, TXL depending on DMA direction) to avoid conflicts with the DMA data transfers. The channel not in use can be used for polled operation by the host and operates in the interrupt mode for internal DSP exceptions or polling. Hardware, software, individual, and STOP resets clear the DMA status bit.

10.2.3.4.8 ISR Host Request (HREQ) Bit 7

The HREQ bit indicates the status of the external host request output pin ($\overline{HREQ}$). When the HREQ status bit is cleared, it indicates that the external $\overline{HREQ}$ pin is deasserted and no host processor interrupts or DMA transfers are being requested. When the HREQ status bit is set, it indicates that the external $\overline{HREQ}$ pin is asserted, indicating that the DSP

is interrupting the host processor or that a DMA transfer request is occurring. The HREQ interrupt request may originate from either or both of two sources – the receive byte registers are full or the transmit byte registers are empty. These conditions are indicated by the ISR RXDF and TXDE status bits, respectively. If the interrupt source has been enabled by the associated request enable bit in the ICR, HREQ will be set if one or more of the two enabled interrupt sources is set. Hardware, software, individual, and STOP resets clear HREQ.

10.2.3.5 Interrupt Vector Register (IVR)

The IVR is an 8-bit read/write register which typically contains the exception vector number used with MC68000 Family processor vectored interrupts. Only the host processor can read and write this register. The contents of IVR are placed on the host data bus (H0–H7) when both the $\overline{\text{HREQ}}$ and $\overline{\text{HACK}}$ pins are asserted and the DMA mode is disabled. The contents of this register are initialized to $0F by a hardware or software reset, which corresponds to the uninitialized exception vector in the MC68000 Family.

10.2.3.6 Receive Byte Registers (RXH, RXM, RXL)

The receive byte registers are viewed as three 8-bit read-only registers by the host processor. These registers are called receive high (RXH), receive middle (RXM), and receive low (RXL). These three registers receive data from the high byte, middle byte, and low byte, respectively, of the HTX register and are selected by three external host address inputs (HA2, HA1, and HA0) during a host processor read operation or by an on-chip address counter in DMA operations. The receive byte registers (at least RXL) contain valid data when the receive data register full (RXDF) bit is set. The host processor may program the RREQ bit to assert the external $\overline{\text{HREQ}}$ pin when RXDF is set. This informs the host processor or DMA controller that the receive byte registers are full. These registers may be read in any order to transfer 8-, 16-, or 24-bit data. However, reading RXL clears the receive data full RXDF bit. Because reading RXL clears the RXDF status bit, it is normally the last register read during a 16- or 24-bit data transfer. Reset does not affect RXH, RXM, or RXL.

10.2.3.7 Transmit Byte Registers (TXH, TXM, TXL)

The transmit byte registers are viewed as three 8-bit write-only registers by the host processor. These registers are called transmit high (TXH), transmit middle (TXM), and transmit low (TXL). These three registers send data to the high byte, middle byte and low byte, respectively, of the HRX register and are selected by three external host address inputs (HA2, HA1, and HA0) during a host processor write operation. Data may be written into the transmit byte registers when the transmit data register empty (TXDE) bit is set. The host processor may program the TREQ bit to assert the external $\overline{\text{HREQ}}$ pin when TXDE is set. This informs the host processor or DMA controller that the transmit byte reg-

isters are empty. These registers may be written in any order to transfer 8-, 16-, or 24-bit data. However, writing TXL clears the TXDE bit. Because writing the TXL register clears the TXDE status bit, TXL is normally the last register written during a 16- or 24-bit data transfer. The transmit byte registers are transferred as 24-bit data to the HRX register when both TXDE and the HRDF bit are cleared. This transfer operation sets TXDE and HRDF. Reset does not affect TXH, TXM, or TXL.

10.2.3.8 Registers After Reset

Table 10-5 shows the result of four kinds of reset on bits in each of the HI registers seen by the host processor. The hardware reset is caused by asserting the $\overline{\text{RESET}}$ pin; the software reset is caused by executing the RESET instruction; the individual reset is caused by clearing the PBC register bit 0; and the stop reset is caused by executing the STOP instruction.

10.2.4 Host Interface Pins

The 15 HI pins are described here for convenience. Additional information, including timing, is given in the DSP56001 Advance Information Data Sheet (ADI1290).

10.2.4.1 Host Data Bus (H0-H7)

This bidirectional data bus is used to transfer data between the host processor and the DSP56000/DSP56001. This bus is an input unless enabled by a host processor read. H0-H7 may be programmed as general purpose parallel I/O pins called PB0-PB7 when the HI is not being used.

10.2.4.2 Host Address (HA0–HA2)

These inputs provide the address selection for each HI register. These inputs are stable when $\overline{\text{HEN}}$ is asserted. HA0-HA2 may be programmed as general purpose parallel I/O pins called PB8-PB10 when the HI is not being used.

10.2.4.3 Host Read/Write (HR/$\overline{\text{W}}$)

This input selects the direction of data transfer for each host processor access. If HR/$\overline{\text{W}}$ is high and $\overline{\text{HEN}}$ is asserted, H0-H7 are outputs and DSP data is transferred to the host processor. If HR/$\overline{\text{W}}$ is low and $\overline{\text{HEN}}$ is asserted, H0-H7 are inputs and host data is transferred to the DSP. HR/$\overline{\text{W}}$ is stable when $\overline{\text{HEN}}$ is asserted. HR/$\overline{\text{W}}$ may be programmed as a general purpose I/O pin called PB11 when the HI is not being used.

10.2.4.4 Host Enable ($\overline{\text{HEN}}$)

This input enables a data transfer on the host data bus. When $\overline{\text{HEN}}$ is asserted and HR/$\overline{\text{W}}$ is high, H0-H7 become outputs and DSP data may be latched by the host processor. When $\overline{\text{HEN}}$ is asserted and HR/$\overline{\text{W}}$ is low, H0-H7 become inputs and host data is latched

Table 10-5 Host Registers after Reset (Host Side)

Register Name	Register Data	Reset Type			
		HW Reset	SW Reset	IR Reset	ST Reset
ICR	INIT	0	0	0	0
	HM (1 - 0)	0	0	0	0
	TREQ	0	0	0	0
	RREQ	0	0	0	0
	HF (1 - 0)	0	0	0	0
CVR	HC	0	0	0	0
	HV (4 - 0)	$12	$12	$12	$12
ISR	HREQ	0	0	0	0
	DMA	0	0	0	0
	HF (3 - 2)	0	0	—	—
	TRDY	1	1	1	1
	TXDE	1	1	1	1
	RXDF	0	0	0	0
IVR	IV (7 - 0)	$0F	$0F	—	—
RX	RXH (23 - 16)	—	—	—	—
	RXM (15 - 8)	—	—	—	—
	RXL (7 - 0)	—	—	—	—
TX	TXH (23 - 21)	—	—	—	—
	TXM (15 - 8)	—	—	—	—
	TXL (7 - 0)	—	—	—	—

inside the DSP when $\overline{\text{HEN}}$ is deasserted. When $\overline{\text{HEN}}$ is deasserted, H0-H7 are three-stated. Normally a chip select signal derived from host address decoding and an enable clock, is used to generate $\overline{\text{HEN}}$. $\overline{\text{HEN}}$ may be programmed as a general purpose I/O pin called PB12 when the HI is not being used.

10.2.4.5 Host Request ($\overline{\text{HREQ}}$)

This open-drain output signal is used by the DSP56000/DSP56001 HI to request service from the host processor, DMA controller, or a simple external controller. $\overline{\text{HREQ}}$ may be

connected to an interrupt request pin of a host processor, a transfer request of a DMA controller or a control input of external circuitry. $\overline{\text{HREQ}}$ is asserted when an enabled request occurs in the host interface. $\overline{\text{HREQ}}$ is deasserted when the enabled request is cleared or masked, DMA $\overline{\text{HACK}}$ is asserted, or the DSP is reset. $\overline{\text{HREQ}}$ may be programmed as a general purpose I/O pin (not open-drain) called PB13 when the HI is not being used.

10.2.4.6 Host Acknowledge ($\overline{\text{HACK}}$)

This input has two functions: 1) to provide a Host Acknowledge handshake signal for DMA transfers and, 2) to receive a Host Interrupt Acknowledge compatible with MC68000 Family processors. If programmed as a host acknowledge signal, $\overline{\text{HACK}}$ may be used as a data strobe for HI DMA data transfers. If programmed as a MC68000 host interrupt acknowledge, $\overline{\text{HACK}}$ is used to enable the HI interrupt vector register (IVR) onto the host data bus H0-H7 if $\overline{\text{HREQ}}$ is asserted. In this case, all other HI control pins are ignored and the state of the HI is not affected. $\overline{\text{HACK}}$ may be programmed as a general purpose I/O pin called PB14 when the HI is not being used.

10.2.5 Servicing the Host Interface

The HI can be serviced by using one of the following protocols:

1. Polling, or
2. Interrupts, which can be either
   a. non-DMA or
   b. DMA.

From the host processor viewpoint, the service consists of making a data transfer since this is the only way to reset the appropriate status bits.

10.2.5.1 HI Host Processor Data Transfer

The HI looks like static RAM to the host processor. Accordingly, in order to transfer data with the HI, the host processor

1. asserts the HI address (HA0, HA1, HA2) to select the register to be read or written;
2. asserts HR/$\overline{\text{W}}$ to select the direction of the data transfer;
3. strobes the data transfer using $\overline{\text{HEN}}$. When data is being written to the HI by the host processor, the positive-going edge of $\overline{\text{HEN}}$ latches the data in the HI register selected. When data is being read by the host processor, the negative-going edge of $\overline{\text{HEN}}$ strobes the data onto the data bus H0-H7.

This process is illustrated inFigure 10-15. The specified timing relationships are given in the DSP56001 Advance Information Data Sheet (ADI1290).

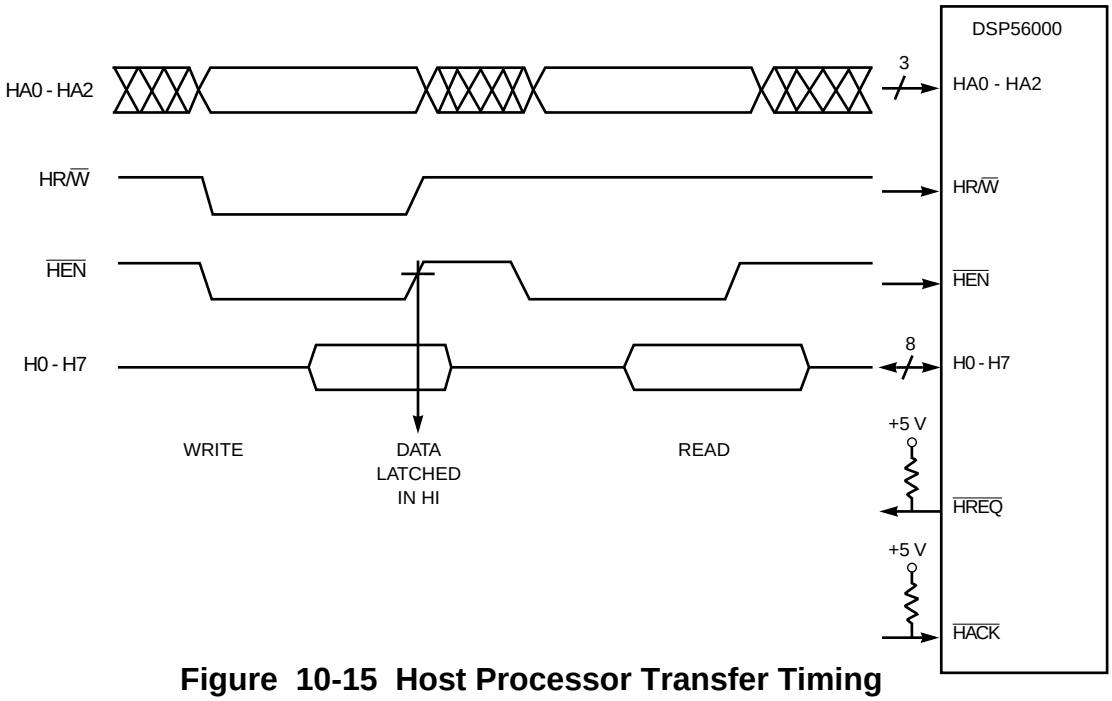


Figure 10-15 Host Processor Transfer Timing

10.2.5.2 HI Interrupts Host Request ($\overline{\text{HREQ}}$)

The host processor interrupts are external and use the $\overline{\text{HREQ}}$ pin. $\overline{\text{HREQ}}$ is normally connected to the host processor maskable interrupt (IPL0 or IPL1 or IPL2 in Figure 10-16) input. The host processor acknowledges host interrupts by executing an interrupt service routine. The most significant bit ($\overline{\text{HREQ}}$) of the ISR may be tested by the host processor to determine if the DSP is the interrupting device and the two least significant bits (RXDF and TXDE) may be tested to determine the interrupt source (see Figure 10-17). The host processor interrupt service routine must read or write the appropriate HI register to clear the interrupt. $\overline{\text{HREQ}}$ is deasserted when1) the enabled request is cleared or masked, 2) DMA $\overline{\text{HACK}}$ is asserted, or 3) the DSP is reset.

10.2.5.3 Polling

In the polling mode of operation, the $\overline{\text{HREQ}}$ pin is not connected to the host processor and $\overline{\text{HACK}}$ must be deasserted to insure DMA data or IVR data is not being output on H0-H7 when other registers are being polled.

The host processor first performs a data read transfer to read the ISR (see Figure 10-17) to determine, whether:

1. RXDF=1, signifying the receive data register is full and therefore a data read should be performed.
2. TXDE=1, signifying the transmit data register is empty so that a data write can

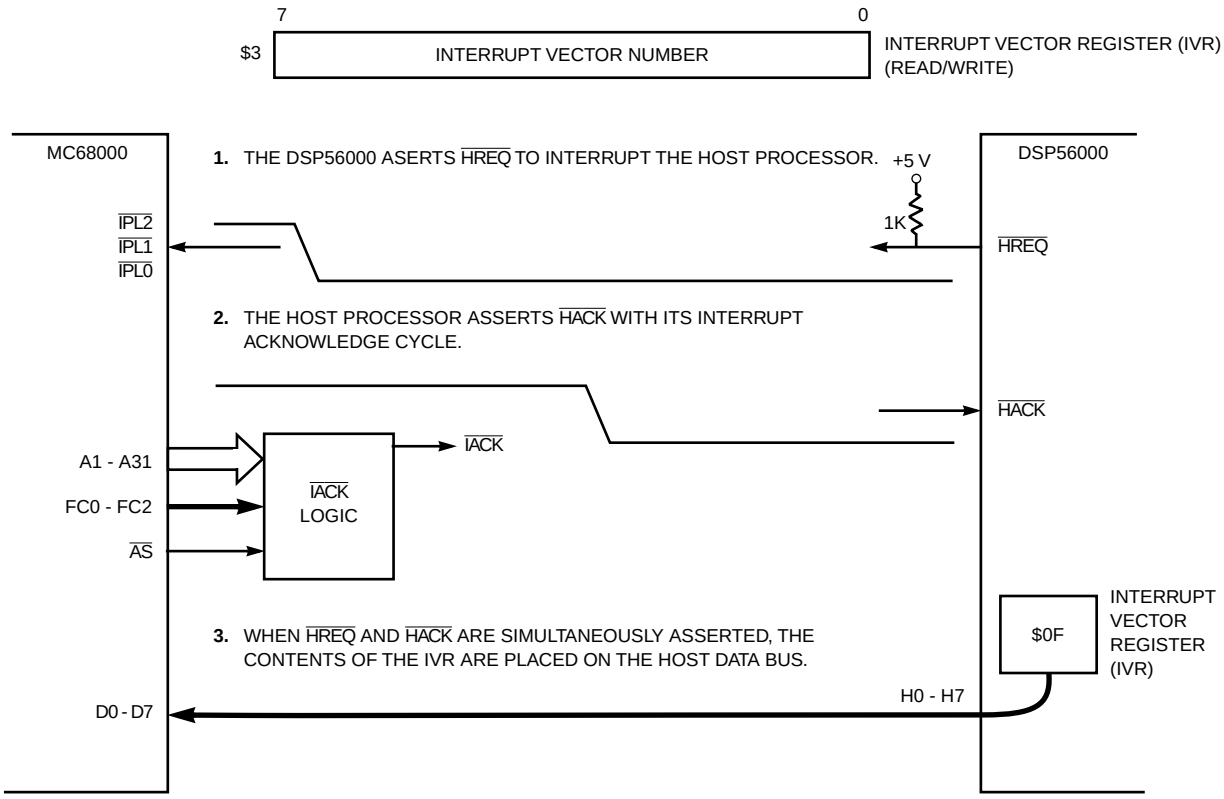


Figure 10-16 Interrupt Vector Register Read Timing

be performed.

3. TRDY=1, signifying the transmit data register is empty and that the receive data register on the DSP CPU side is also empty so that the data written by the host processor will be transferred directly to the DSP side.
4. HF2 • HF3 ≠ 0, signifying an application-specific state within the DSP CPU has been reached, which requires action on the part of the host processor.
5. DMA=1, signifying the HI is currently being used for DMA transfers. If DMA transfers are possible in the system, care must be exercised to deactivate HACK prior to reading the ISR so both DMA data and the contents of ISR are not simultaneously output on H0- H7.
6. If HREQ=1, the HREQ pin has been asserted, and one of the previous five conditions exists.

Generally, after the appropriate data transfer has been made, the corresponding status bit will toggle.

If the host processor has issued a command to the DSP by writing the CVR and setting the HC bit, it can read the HC bit in the CVR to determine when the command has been accepted by the interrupt controller in the DSP CPU. When the command has been accepted for execution, the HC bit will be reset to zero by the interrupt controller in the

DSP CPU.

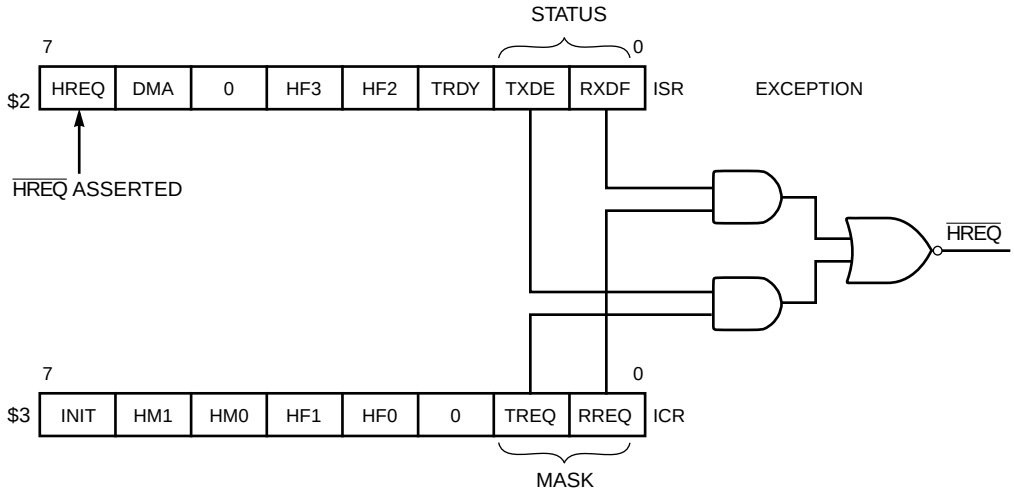


Figure 10-17 HI Interrupt Structure

10.2.5.4 Servicing Non-DMA Interrupts

When HM0=HM1=0 (i.e., non-DMA) and $\overline{\text{HREQ}}$ is connected to the host processor interrupt input, the HI can request service from the host processor by asserting $\overline{\text{HREQ}}$. In the non-DMA mode, $\overline{\text{HREQ}}$ will be asserted when TXDE=1 and/or RXDF=1 and the corresponding mask bit (TREQ or RREQ, respectively) is set. This is depicted in Figure 10-17.

Generally, servicing the interrupt starts with reading the ISR, as described in the previous section on polling, to determine which DSP has generated the interrupt and why. When multiple DSPs occur in a system, the $\overline{\text{HREQ}}$ bit in the ISR will normally be read first to determine the interrupting device. The host processor interrupt service routine must read or write the appropriate HI register to clear the interrupt. $\overline{\text{HREQ}}$ is deasserted when the enabled request is cleared or masked.

In the case where the host processor is a member of the MC680XX Family, servicing the interrupt will start by asserting $\overline{\text{HREQ}}$ to interrupt the processor (see Figure 10-17). The host processor then acknowledges the interrupt by asserting $\overline{\text{HACK}}$. While $\overline{\text{HREQ}}$ and $\overline{\text{HACK}}$ are simultaneously asserted, the contents of the IVR are placed on the host data bus. This vector will tell the host processor which routine to use to service the $\overline{\text{HREQ}}$ interrupt.

The $\overline{\text{HREQ}}$ pin is an open-drain output pin so that it can be wire-ORed with the $\overline{\text{HREQ}}$ pins from other DSP56000/DSP56001 processors in the system. When one of the DSP56000/DSP56001 processors generates an interrupt request the host processor can poll the HREQ bit in each of the ISRs to determine which device generated the interrupt.

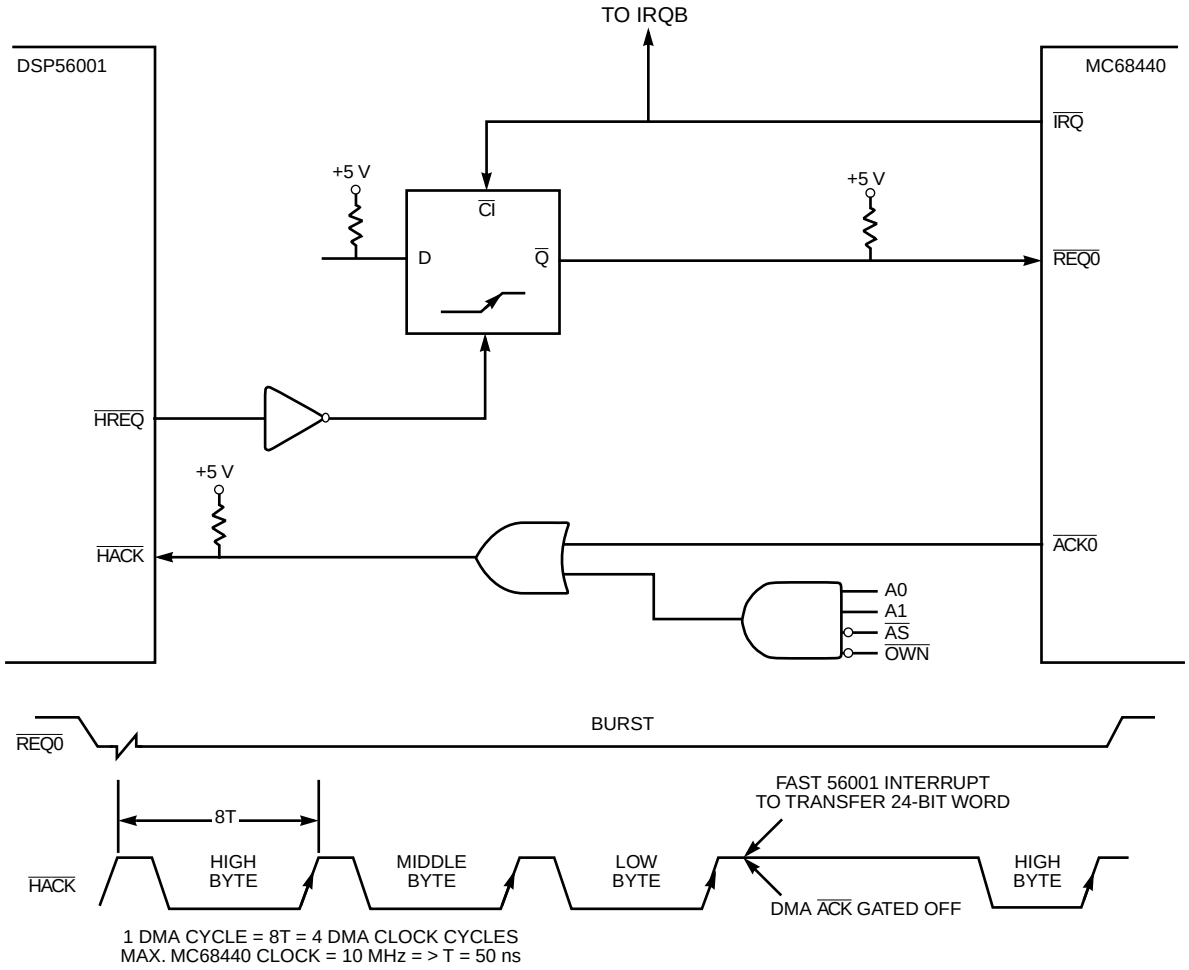


Figure 10-18 DMA Transfer Logic and Timing

10.2.5.5 Servicing DMA Interrupts

When HM0≠0 and/or HM1≠0, $\overline{\text{HREQ}}$ will be asserted to request a DMA transfer. Generally the $\overline{\text{HREQ}}$ pin will be connected to the $\overline{\text{REQ}}$ input of a DMA controller. The HA0-2, $\overline{\text{HEN}}$, and HR/$\overline{\text{W}}$ pins are not used during DMA transfers; DMA transfers only use the $\overline{\text{HREQ}}$ and $\overline{\text{HACK}}$ pins after the DMA channel has been initialized. $\overline{\text{HACK}}$ is used to strobe the data transfer as shown in Figure 10-20 where an MC68440 is used as the DMA controller. DMA transfers to and from the HI are considered in more detail in 10.2.6 HI Application Examples.

10.2.6 HI Application Examples

In the sections that follow, examples of initializing the HI, transferring data with the HI, bootstrapping via the HI, and performing DMA transfers through the HI are described.

10.2.6.1 HI Initialization

Initializing the HI takes two steps (see Figure 10-19). The first step is to initialize the DSP

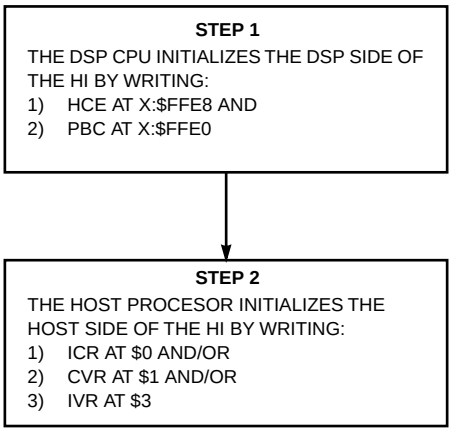


Figure 10-19 HI Initialization Flowchart

side of the HI, which requires that the options for interrupts and flags be selected and then the HI be selected (see Figure 10-20). The second step is for the host processor to clear the HC bit by writing the CVR, select the data transfer method - polling, interrupts, or DMA (see figures 10-21 and10-23), and write the IVR in the case of a MC680XX Family host processor.Figures 10-19 through 10-22 provide a genera description of how to initialize the HI. Later paragraphs in this section provide more detailed descriptions for specific examples.These subsections include some code fragments illustrating how to initialize and transfer data using the HI.

10.2.6.2 Polling/Interrupt Controlled Data Transfer

Handshake flags are provided lfor polled or interrupt-driven data transfers. Because the DSP interrupt response is sufficiently fast, most host microprocessors can load or store data at their maximum programmed I/O (non-DMA) instruction rate without testing the handshake flags for each transfer. If the full handshake is not needed, the host processor can treat the DSP as fast memory, and data can be transferred between the host and DSP at the fastest host processor rate. DMA hardware may be used with the external host request and host acknowledge pins to transfer data at the maximum DSP interrupt rate.

The basic data transfer process from the host processor's view (see Figure 10-15) is for the host to

1. Assert $\overline{\text{HREQ}}$ when the HI is ready to transfer data.
2. Assert $\overline{\text{HACK}}$ If the interface is using $\overline{\text{HACK}}$.
3. Assert HR/$\overline{\text{W}}$ to select whether this operation will read or write a register.
4. Assert the HI address (HA2, HA1, and HA0) to select the register to be read or

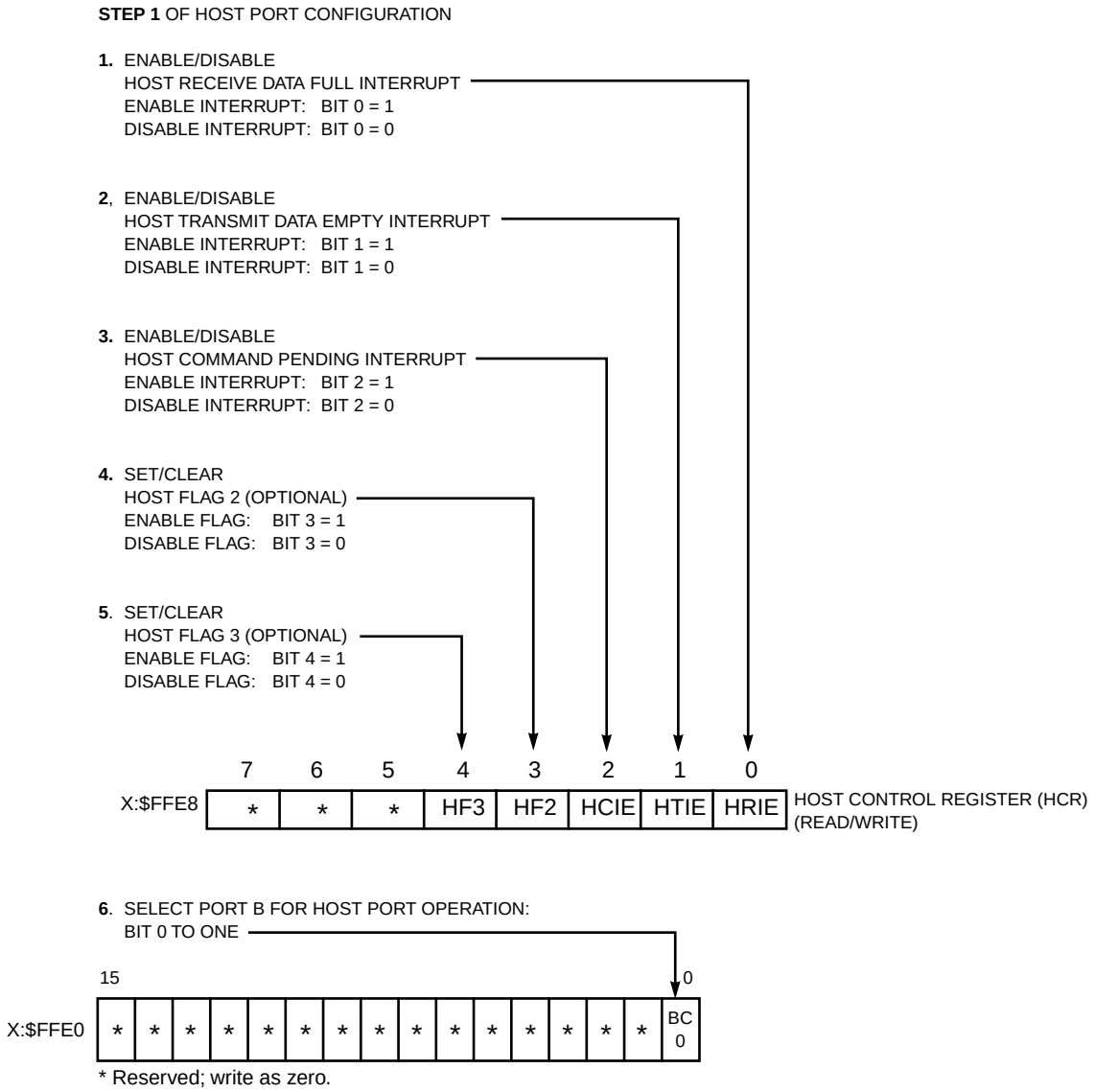


Figure 10-20 HI Initialization–DSP Side

written.

5. Assert $\overline{\text{HEN}}$ to enable the HI.
6. When $\overline{\text{HEN}}$ is deasserted, the data can be latched or read as appropriate if the timing requirements have been observed.
7. $\overline{\text{HREQ}}$ will be deasserted if the operation is complete.

The previous transfer description is an overview. Specific and exact information for the HI

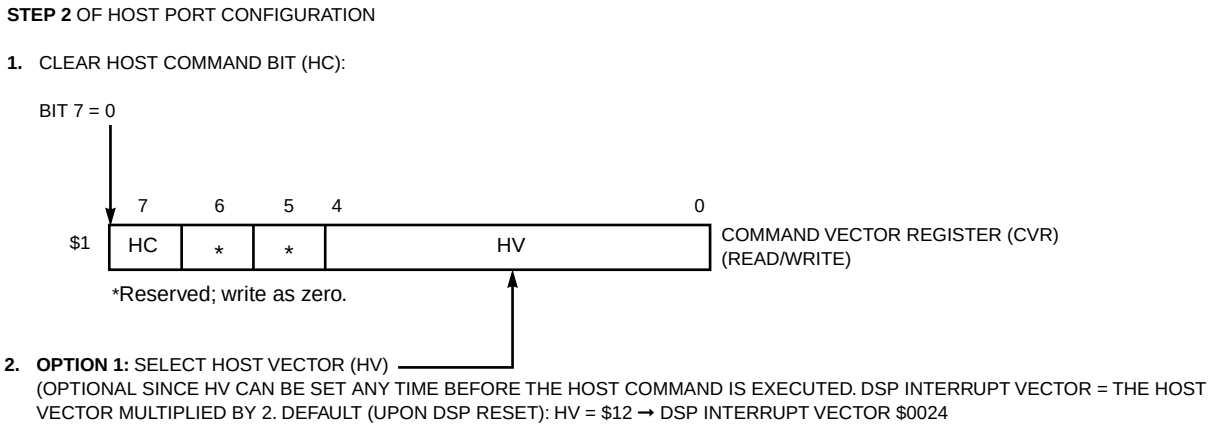


Figure 10-21(a) HI Configuration–Host Side

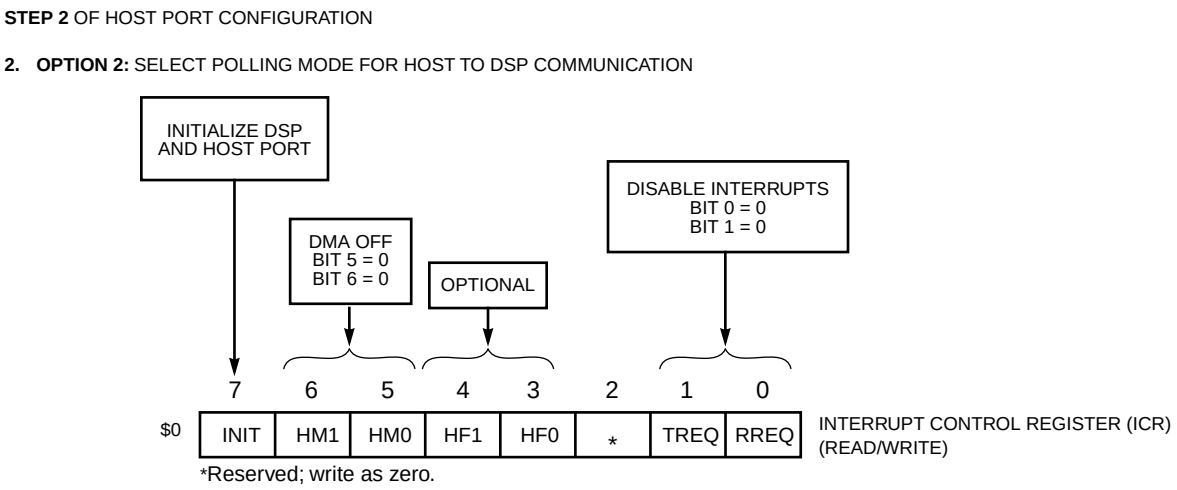


Figure 10-21(b) HI Initialization–Host Side, Polling Mode

data transfers and their timing can be found in 10.2.6.3 DMA Data Transfer and in the DSP5600l Advance Information Data Sheet (ADI1290).

10.2.6.2.1 Host to DSP - Data Transfer

Figure 10-23 shows the bits in the ISR and ICR registers used by the host processor and the bits in the HSR and HCR registers used by the DSP to transfer data from the host processor to the DSP. The registers shown are the status register and control register seen by the host processor and status register and control register seen by the DSP. Only the registers used to transmit data from the host processor to the DSP are described. Figure 10-24 illustrates the process of that data transfer. The steps in Figure 10-24 can be summarized as follows:

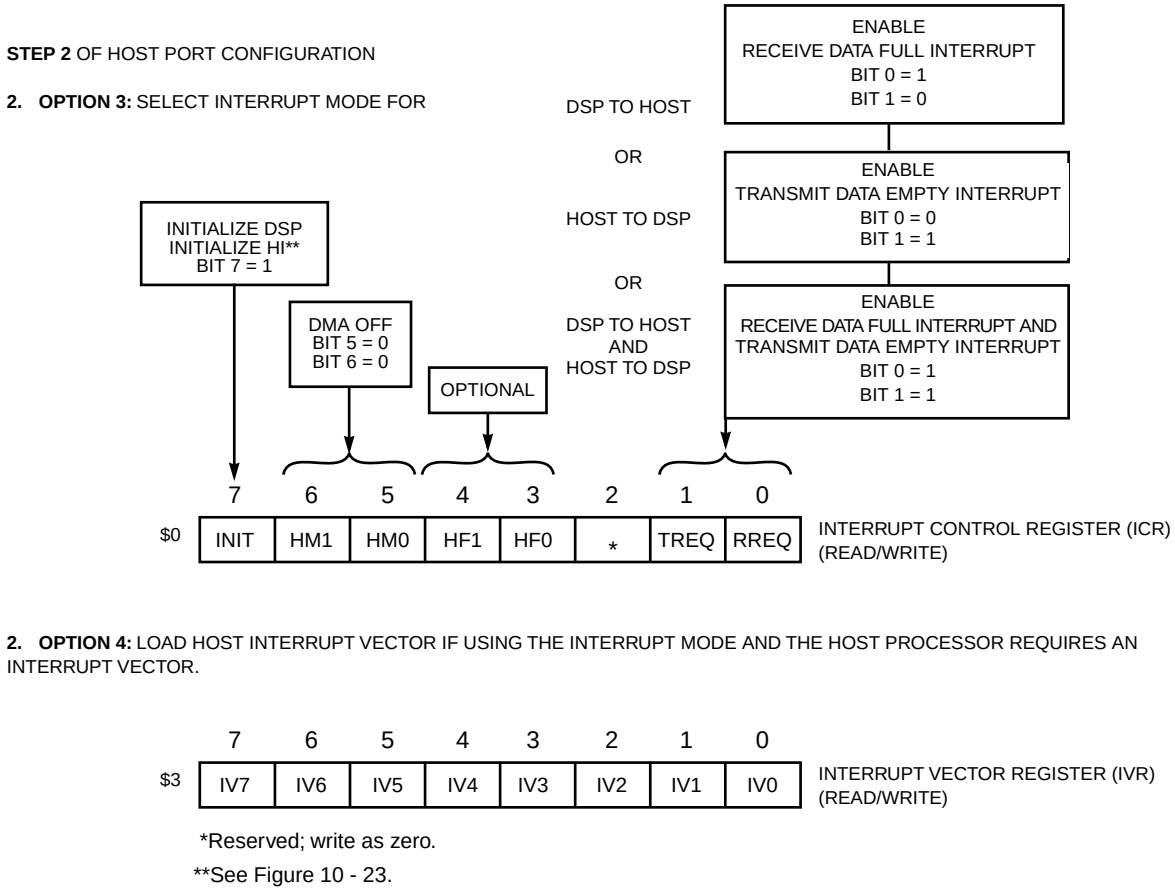


Figure 10-21(c) HI Initialization–Host Side, Interrupt Mode

1. When the TXDE bit in the ISR is set, it indicates that the HI is ready to receive a data byte from the host processor because the transmit byte registers (TXH, TXM, TXL) are empty.
2. The host processor can either poll or
3. use interrupts to determine the status of this bit. Setting the TREQ bit in the ICR causes the $\overline{\text{HREQ}}$ pin to interrupt the host processor when TXDE is set.
4. Once the TXDE bit is set, the host can write data to the HI. It does this by writing three bytes to TXH, TXM, and TXL, respectively, or two bytes to TXM and TXL, respectively, or one byte to TXL.
5. Writing data to TXL clears TXDE in the ISR.
6. From the DSP's viewpoint, the HRDF bit (when set) in the HSR indicates that data is waiting in the HI for the DSP.
7. When the DSP reads the HRX, the HRDF bit is automatically cleared and TXDE in the ISR is set.
8. When TXDE=0 and HRDF=0, data is automatically transferred from TBR to HRX which sets HRDF.

STEP 2 OF HOST PORT CONFIGURATION

2. OPTION 5: SELECT DMA MODE FOR

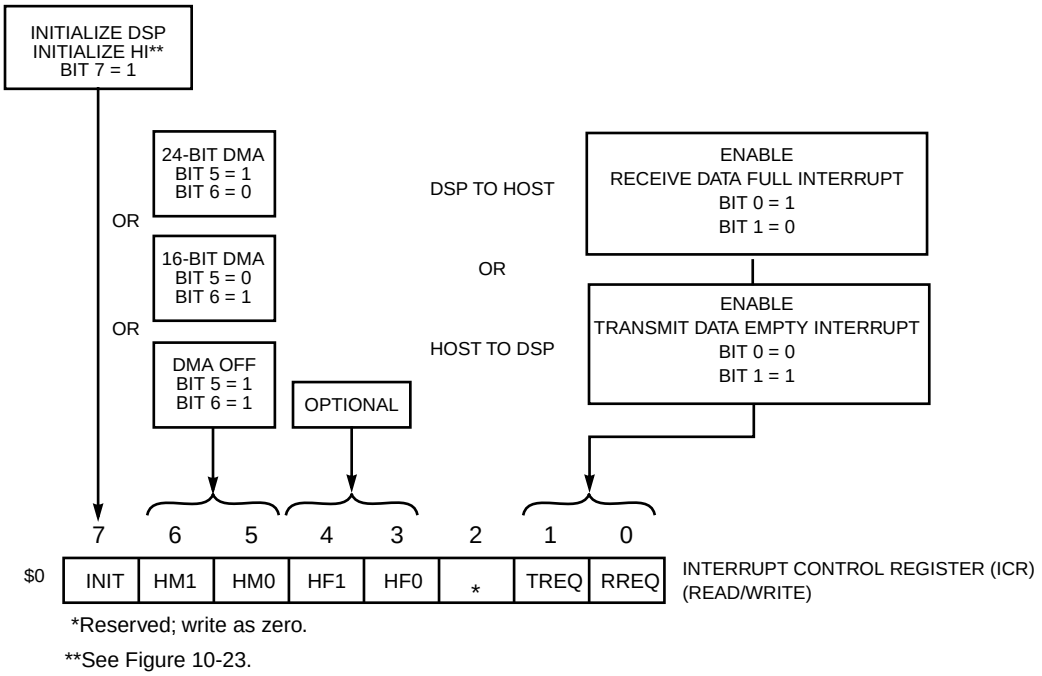
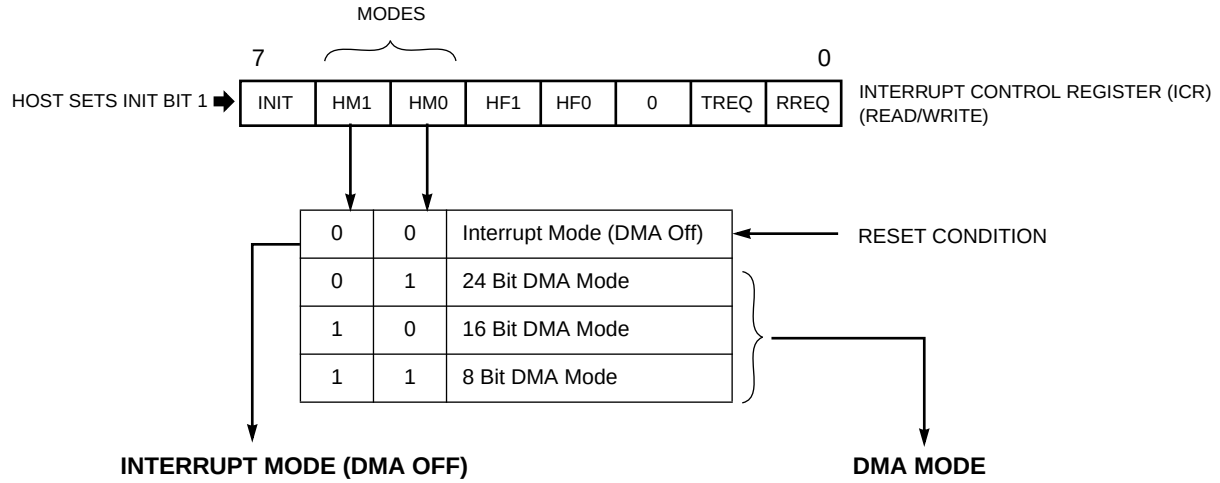


Figure 10-21(d) HI Initialization–Host Side, DMA Mode

9. The DSP can poll HRDF to see when data has arrived, or it can use interrupts.
10. If HRIE (in the HCR) and HRDF are set, exception processing is started using interrupt vector P:$0020.

The code shown in Figure 10-25 is an excerpt from the Host I/O Port Technical Bulletin (in-house document). The MAIN PROGRAM initializes the HI and then hangs in a wait loop and allows interrupts to transfer data from the host processor to the DSP. The first three MOVEP instructions enable the HI and configure the interrupts. The following two moves enable the interrupts (this should always be done after the interrupt programs and hardware are completely initialized) and prepare the DSP CPU to look for the host flag, HF0=1. LOOP is a polling loop that looks for HF0=1, which indicates that the host processor is ready. When the host processor is ready to transfer data to the DSP, the DSP enables HRIE in the HCR, which allows the interrupt routine to receive data from the host processor. The jump-to-self instruction that follows is for test purposes only, it can be replaced by any other code in normal operation.

The receive routine in Figure 10-26 was implemented as a long interrupt (the instruction at the interrupt vector location, which is not shown, is a JSR). Since there is only one instruction, this could have been implemented as a fast interrupt. The MOVEP instruction moves data from the HI to a buffer area in memory and increments the buffer pointer so



TREQ	RREQ	INIT Execution
0	0	INIT = 0; Address Counter = 00
0	1	INIT = 0; RXDF = 0; HTDE = 1; Address Counter = 00
1	0	INIT = 0; TXDE = 1; HRDF = 0; Address Counter = 00

TREQ	RREQ	INIT Execution
0	0	INIT = 0; Address Counter = HM1, HM0
0	1	INIT = 0; RXDF = 0; HTDE = 1; Address Counter = HM1, HM0
1	0	INIT = 0; TXDE = 1; HRDF = 0; Address Counter = HM1, HM0
1	1	Undefined (Illegal)

INIT is used by the HOST to force initialization of the HI hardware.
The HI hardware automatically clears INIT when the command is executed.
INIT is cleared by DSP RESET.

Figure 10-22 Host Mode and INIT Bits

that the next word received will be put in the next sequential location.

10.2.6.2.2 Host to DSP – Command Vector

The host processor can cause three types of interrupts in the DSP (see Figure 10-26). These are host receive data (P:$0020), host transmit data (P:$0022), and host command (P:$0024 - P:$003C). The host command (HC) can be used to control the DSP by forcing it to execute any of thirteen subroutines that can be used to run tests, transfer data, process data, etc. In addition, the HC can cause any of the other 19 interrupt routines in the DSP to be executed. The process to execute a HC (see Figure 10-28) is as follows:

1. The host processor writes the CVR with the desired HV (the HV is the DSP's interrupt vector (IV) location divided by two - i.e. if HV=$12, IV=$24).
2. The HC is then set.
3. The HCP bit in the HSR is set when HC is set.
4. If the HCIE bit in the HCR has been set by the DSP, the HC exception processing will start. The HV is multiplied by 2 and the result is used by the DSP

as the interrupt vector.

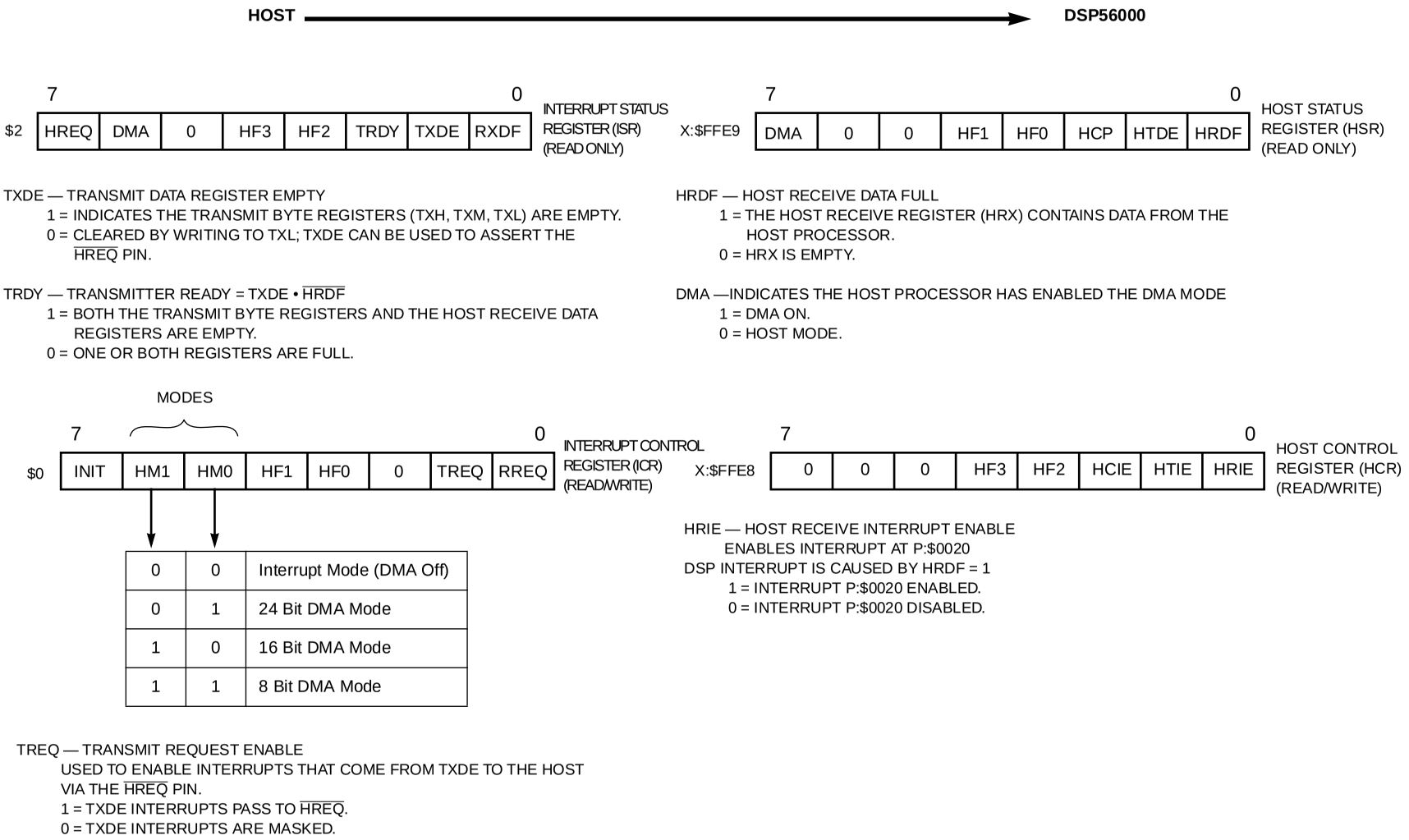


Figure 10-23 Bits Used for Host-to-DSP Transfer

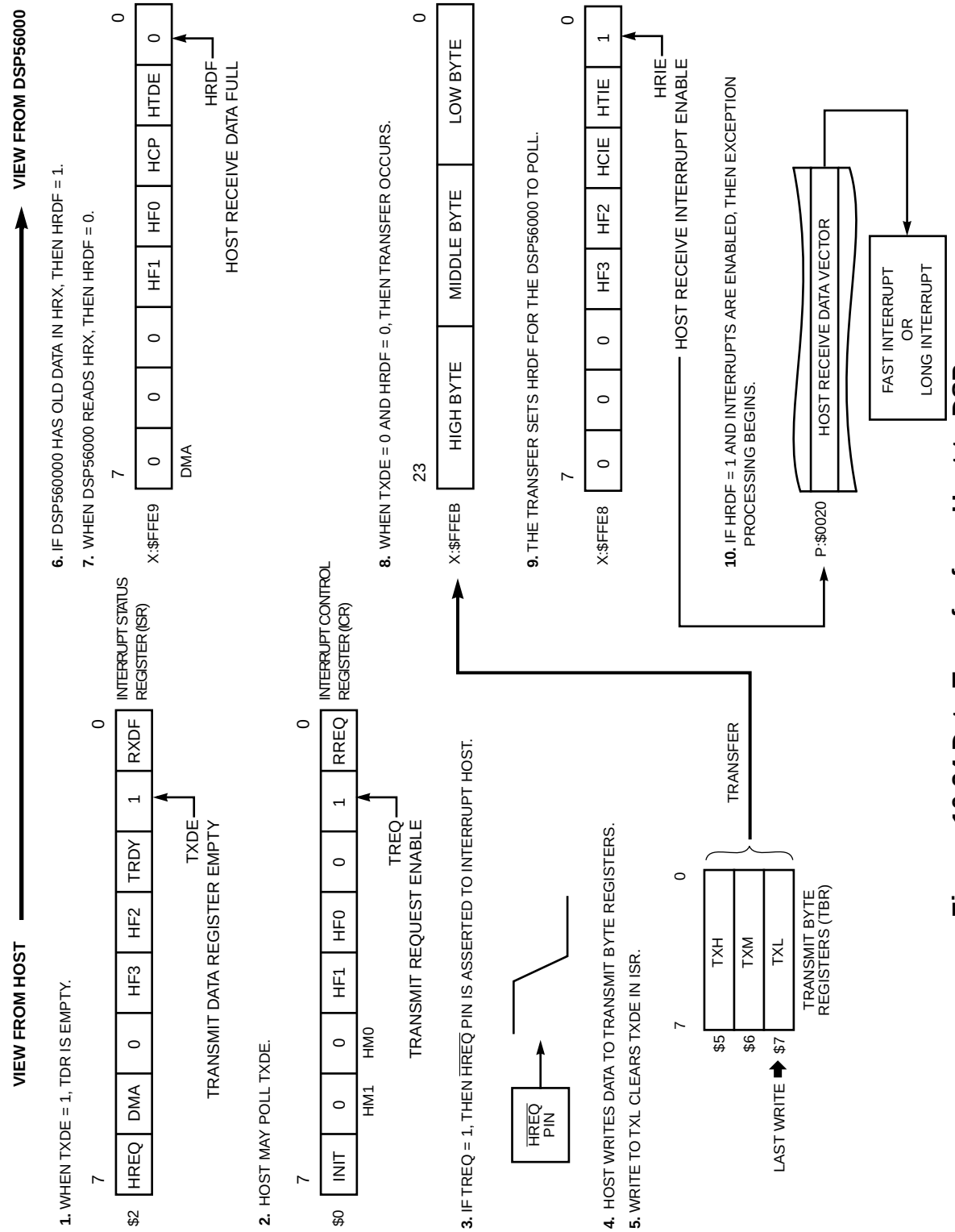


Figure 10-24 Data Transfer from Host to DSP

5. When the HC exception is acknowledged, the HC bit (and therefore the HCP

```
;***************************************
; MAIN PROGRAM ... receive data from host
;***************************************
                ORG     P:$40
                MOVE    #0,R0
                MOVE    #3,M0

                MOVEP   #1,X:PBC        ;Turn on Host Port
                MOVEP   #0,X:HCR        ;Turn off XMT and RCV interrupts
                MOVEP   #$0C00,X:IPR    ;Turn on host interrupt

                MOVE    #0,SR           ;Unmask interrupts
                MOVE    #>$8,X0         ;Host flag mask for HF0

LOOP            MOVEP   X:HSR,A         ;Wait for HF0 (from host) set to 1
                AND     X0,A
                JEQ     LOOP

                MOVEP   #$1,X:HGR       ;Enable host receive interrupt

                JMP     *               ;Now wait for interrupt
```

Figure 10-25 Receive Data from Host–Main Program

bit) is cleared by the HC logic. HC can be read by the host processor as a status bit to determine when the command is accepted. Similarly, the HCP bit can be read by the DSP CPU to determine if an HCis pending.

To guarantee a stable interrupt vector, write HV only when HC is clear. The HC bit and HV can be written simultaneously. The host processor can clear the HC bit to cancel a host command at any time before the DSP exception is accepted. Although the HV can be programmed to any exception vector, it is **not** recommended that HV=0 (RESET) be used because it does not reset the DSP hardware. DMA must be disabled to use the host exception .

```
;***********************************
; Receive from Host Interrupt Routine
;***********************************

RCV             MOVEP   X:HRX,X:(R0)+   ;Receive data.
                RTI

                END
```

Figure 10-26 Receive Data from Host Interrupt Routine

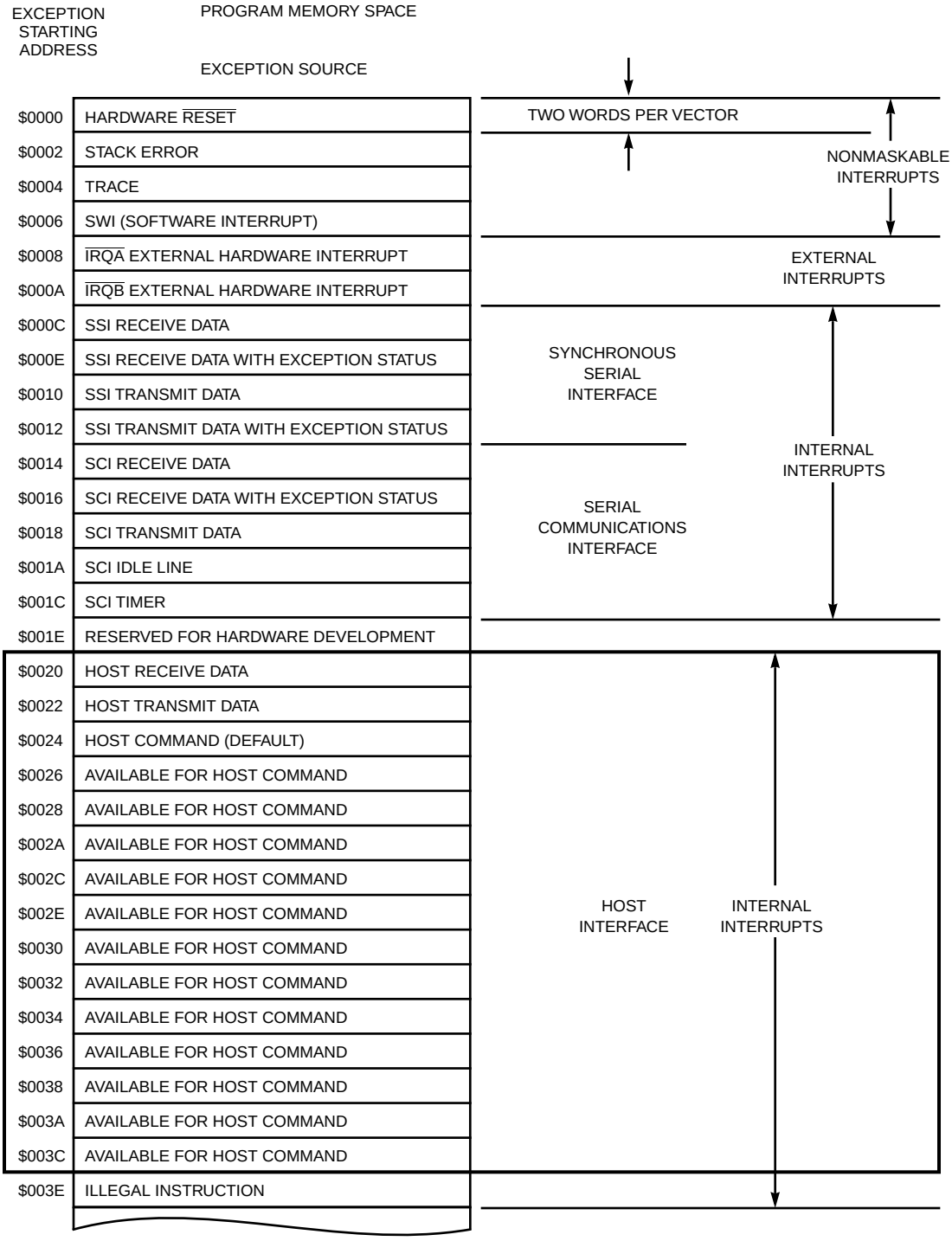


Figure 10-26 Vector Table of Exception Sources

10.2.6.2.3 Host to DSP - Bootstrap Loading Using the HI

The circuit shown in Figure 10-27 will cause the DSP to boot through the HI on power up. During the bootstrap program, the DSP looks at P:$C000 data bit 23. If D23 is high, it

will boot from an external memory location. If it is low, as shown in Figure 10-27, it will

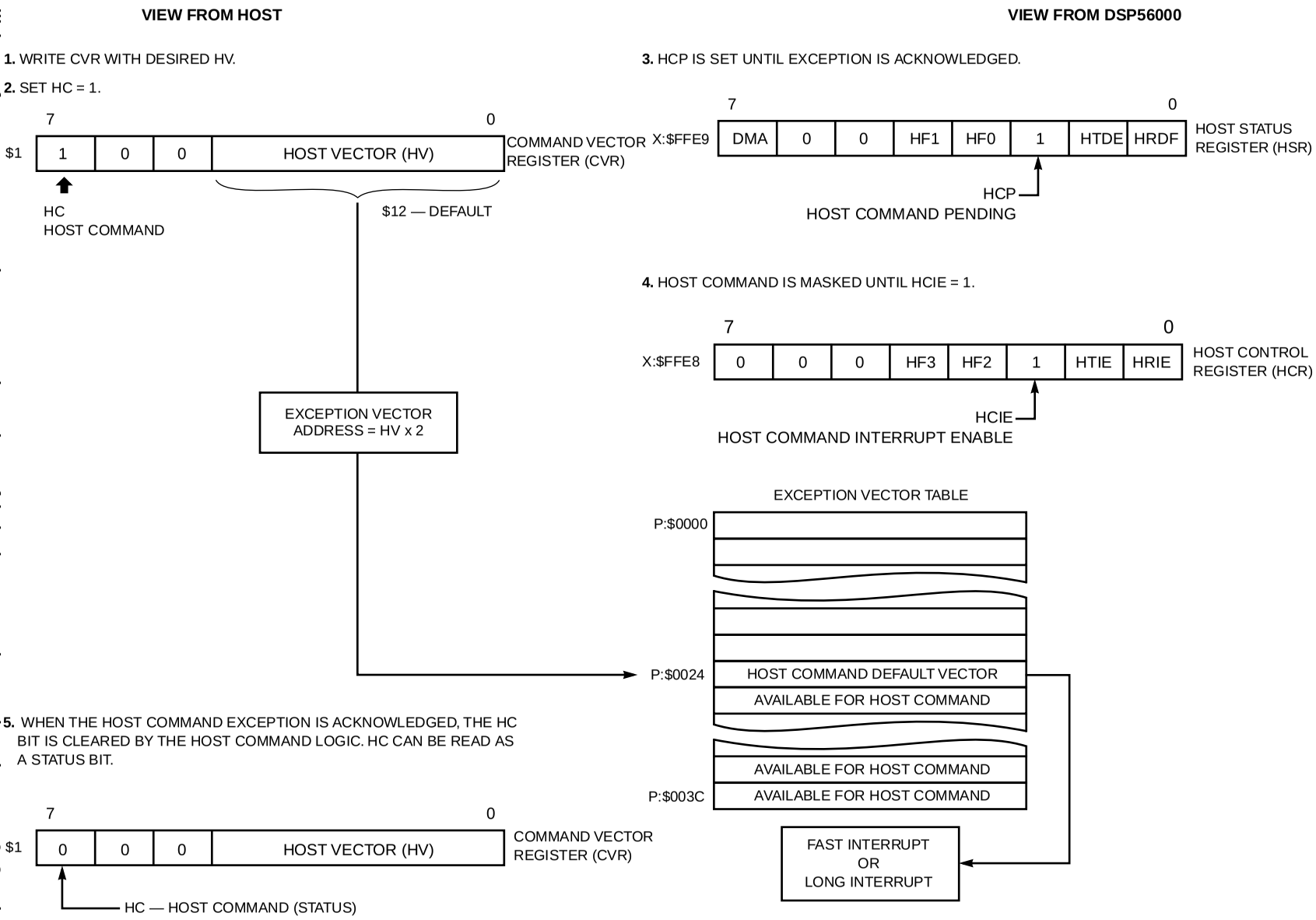
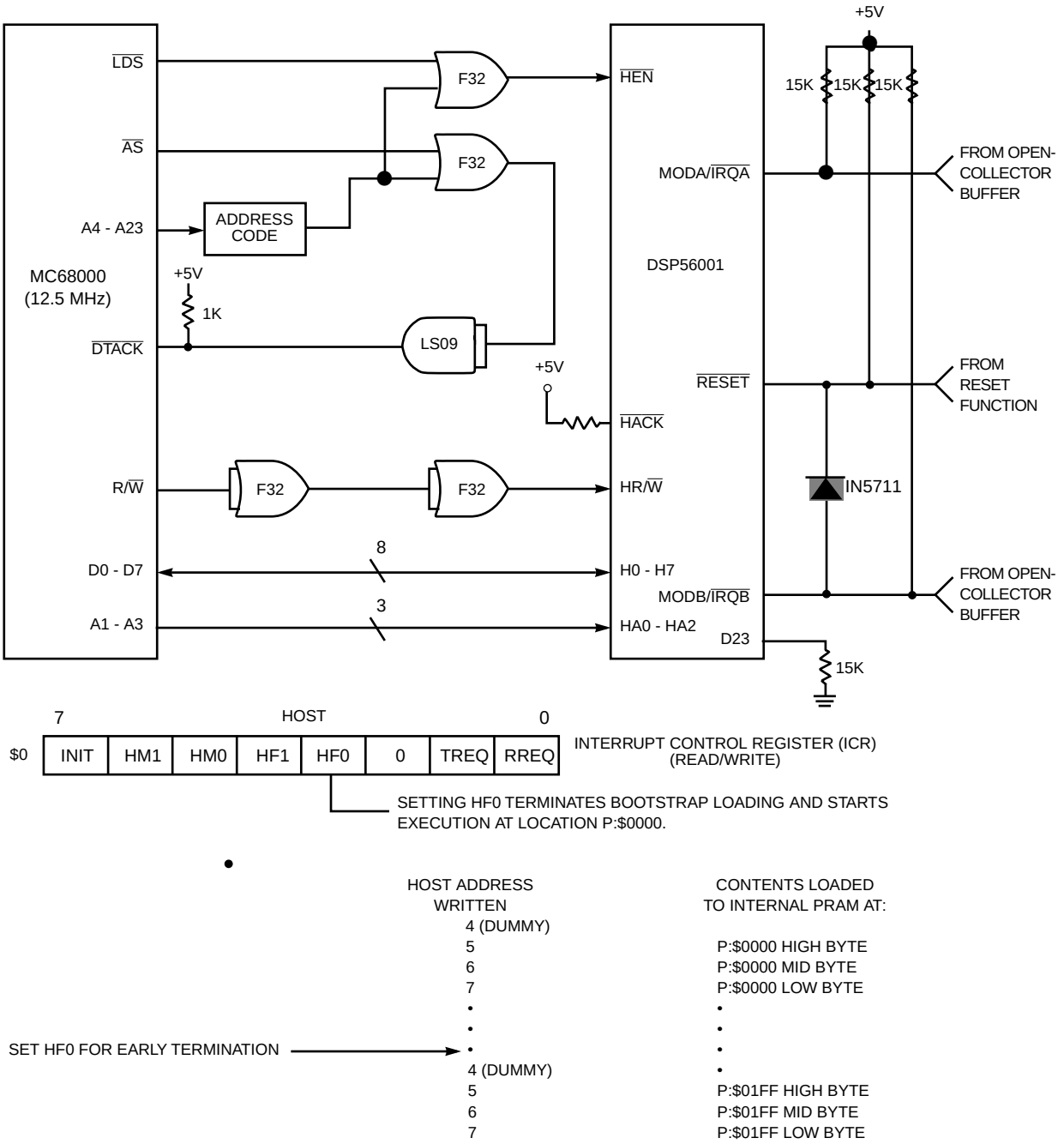


Figure 10-28 Host Command

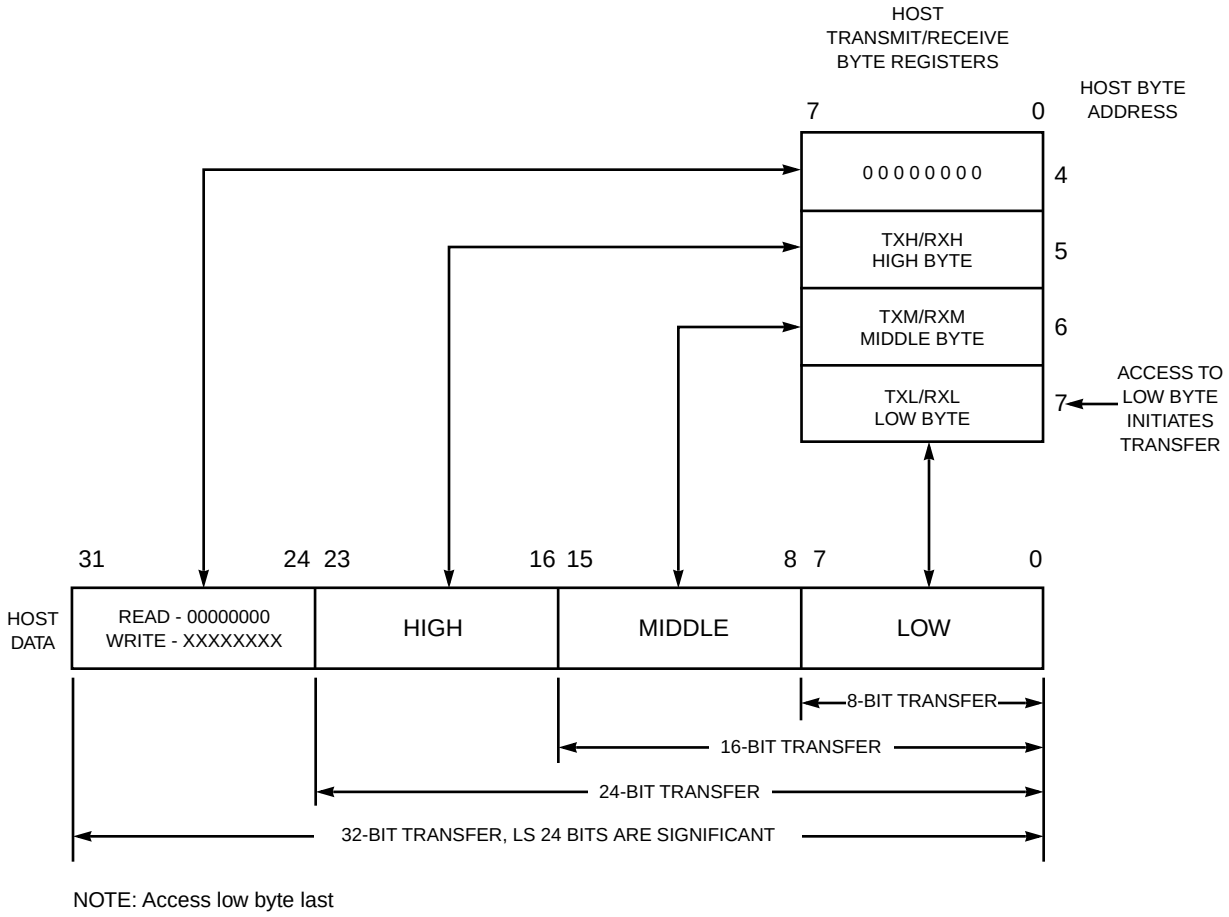


• Because the DSP56000 is so fast, host handshaking is generally not required.

Figure 10-27 Bootstrap Using the HI

load from the HI. Data is written by the host processor in a pattern of four bytes, with the high byte being a dummy and the low byte being the low byte of the DSP word (see Figure 10-27 and Figure 10-30. Figure 10-30 shows how an 8-,16-, 24-, or 32-bit word in the host processor maps into the HI registers. The HI register at address $4 is not used and will read as zero. It is not necessary to use address $4, but since many host processors

are 16- or 32-bit processors, address $4 will often be used as part of the 16- or 32-bit word. The low order byte (at $7) should always be written last since writing to it causes the HI to initiate the transfer of the word to the HRX. Data is then transferred from the HRX to the DSP program memory. If the host processor needs to terminate the bootstrap loading before 512 words have been down loaded, it can set the HF0 bit in the ICR. The DSP will then terminate the down load and start executing at location P:$0000. Since the DSP56000/DSP56001 is typically faster than the host processor, hand shaking during the data transfer is normally not required.



NOTE: Access low byte last

Figure 10-30 Transmit/Receive Byte Registers

The actual code used in the bootstrap program is given in the DSP56001 Advance Information Data Sheet (ADI1290). The portion of the code that loads from the HI is shown in Figure 10-31. The BSET instruction configures Port B as the HI and the first JCLR looks for a flag (HF0) to indicate an early termination of the download. The second JCLR instruction causes the DSP to wait for a complete word to be received and then two MOVEs are used to move the data from the HI to memory through an intermediate register, A1.

```
INLOOP      DO          #512,_LOOP1         ;Load 512 instruction words.
            •
            •
            •
_HOSTLD     BSET        #0,X:$FFE0          ;Configure Port B as Host Interface
_LBLA       JCLR        #3,X:$FFE9,_LBLB    ;If HF0=1, stop loading data.
            ENDDO                           ;Must terminate the DO loop
            JMP         <_BOOTEND           ;Boot complete, go to exit handler
_LBLB       JCLR        #0,X:(R2),_LBLA     ;Wait for HRDF to go high
                                            ;(meaning 24-bit data is present)
            MOVE        X:$FFEB,A1          ;Put 24-bit host data in A1
_STORE      MOVE        A1,P:(R0)+          ;Store 24-bit result in PRAM
_LOOP1                                      ;Return for another 24-bit word
```

Figure 10-31 Bootstrap Code Fragment

10.2.6.2.4 DSP-to-Host Data Transfer

Data is transferred from the DSP to the host processor in a similar manner as from the host processor to the DSP. Figure 10-32 shows the bits in the status registers (ISR and HSR) and control registers (ICR and HCR) used by the host processor and DSP CPU, respectively. The DSP CPU (see Figure 10-33) can poll the HTDE bit in the HSR (1) to see when it can send data to the host, or it can use interrupts enabled by the HTIE bit in the HCR (2). If HTIE=1 and interrupts are enabled, exception processing begins at interrupt vector P:$0022 (3). The interrupt routine should write data to the HTX (4), which will clear HTDE in the HSR. From the host's viewpoint, (5) reading the RXL clears RXDF in the ISR. When RXDF=0 and HTDE=0 (6) the contents of the HTX will be transferred to the receive byte registers (RXH:RXM:RXL). This transfer sets RXDF in the ISR (7), which the host processor can poll to see if data is available or, if the RREQ bit in the ICR is set, the HI will interrupt the host processor with $\overline{\text{HREQ}}$ (8).

The code shown in Figure 10-34 is essentially the same as the MAIN PROGRAM in Figure 10-25 except that, since this code will transmit instead of receive data, the HTIE bit is set in the HCR instead of the HRIE bit.

The transmit routine used by the code in Figure 10-34 is shown in Figure 10-35. The interrupt vector contains a JSR, which makes it a long interrupt. The code (shown in Figure 10-38) sends a fixed test pattern ($123456) and then resets the HI for the next interrupt.

10.2.6.3 DMA Data Transfer

The DMA mode allows the transfer of 8-, 16- or 24-bit data through the DSP HI under the control of an external DMA controller. The HI provides the pipeline data registers and the

Figure 10-32 Bits Used for DSP to Host Transfer

synchronization logic between the two asynchronous processor systems. The DSP host

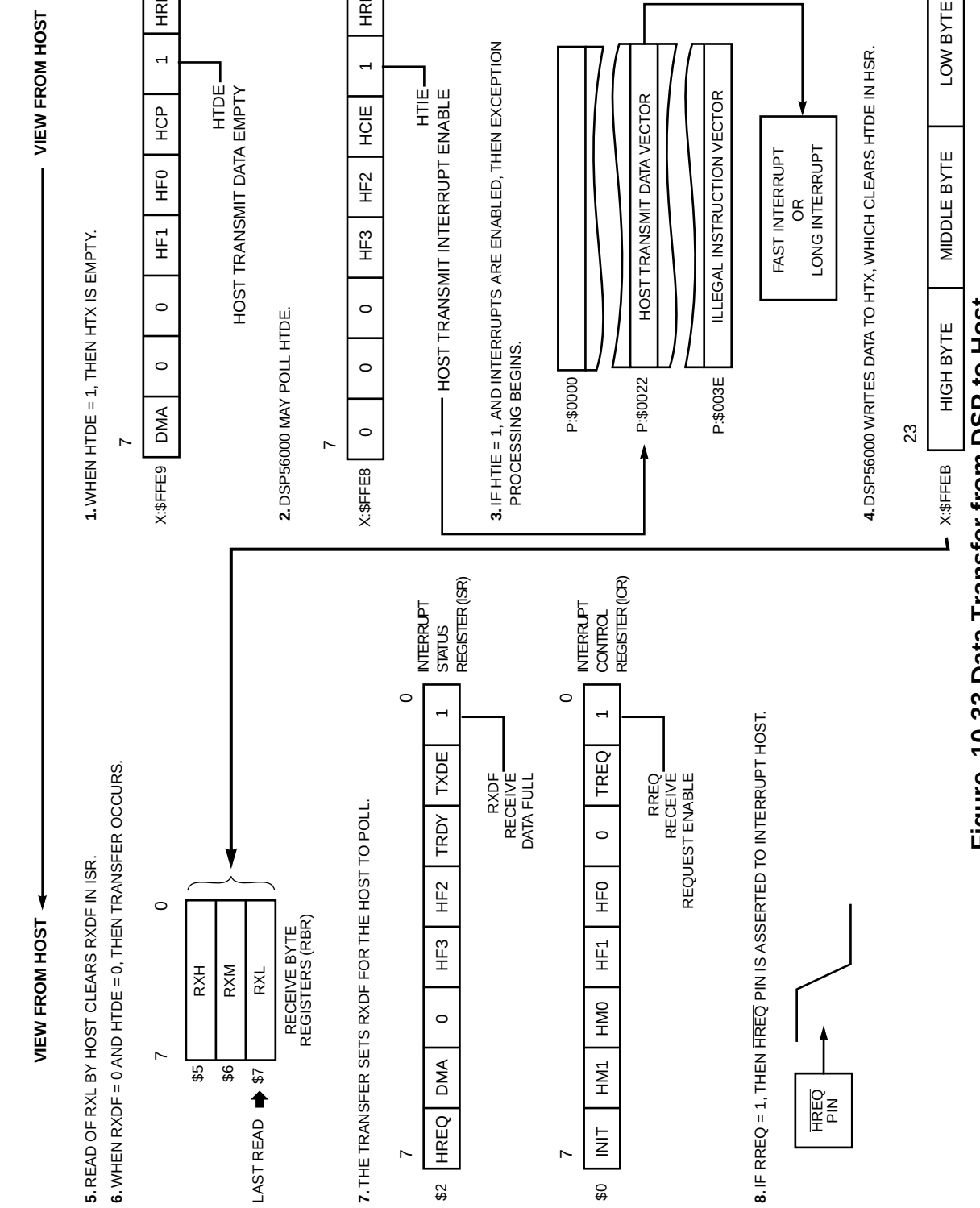


Figure 10-33 Data Transfer from DSP to Host

exceptions provide cycle-stealing data transfers with the DSP internal or external mem-

```
;***************************************
; MAIN PROGRAM ... transmit 24-bit data to host
;***************************************
                ORG     P:$40

                MOVEP   #1,X:PBC        ;Turn on Host Port
                MOVEP   #$0C00,X:IPR    ;Turn on host interrupt
                MOVEP   #0,X:HCR        ;Turn off XMT and RCV interrupts

                MOVE    #0,SR           ;Unmask interrupts
                MOVE    #>$8,X0         ;Host flag mask for HF0

   LOOP         MOVEP   X:HSR,A         ;Wait for HF0 (from host) set to 1
                AND     X0,A
                JEQ     LOOP

                MOVEP   #$2,X:HCR       ;Enable host transmit interrupt

                JMP     *               ;Now wait for interrupt
```

Figure 10-34 Main Program - Transmit 24-Bit Data to Host

```
;***********************************
;TRANSMIT to Host Interrupt Routine
;***********************************

   XMT          MOVEP   #$123456,X:HTX  ;Test value to transmit
                MOVEP   #0,X:HCR        ;Turn off XMT Interrupt
                RTI

                END
```

Figure 10-35 Transmit to HI Routine

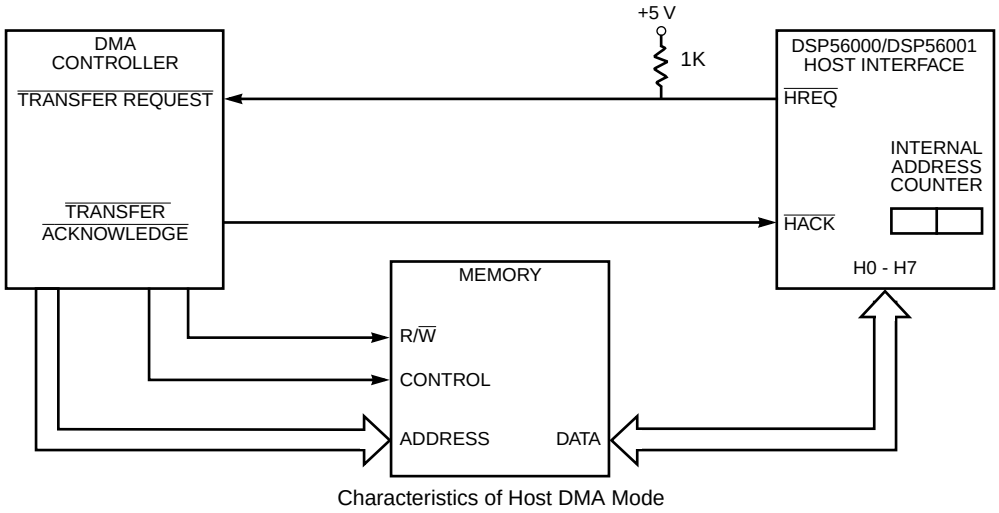
ory. This technique allows the DSP memory address to be generated using any of the DSP addressing modes and modifiers. Queues and circular sample buffers are easily created for DMA transfer regions. The host exceptions can be programmed as high priority fast or long exception service routines. The external DMA controller provides the transfers between the DSP HI registers and the external DMA memory. The external DMA controller must provide the address to the external DMA memory; however, the address of the selected HI register is provided by a DMA address counter in the HI.

DMA transfers can only be in one direction at a time; however, the host processor can access any of the registers not in use during the DMA transfer by deasserting $\overline{\text{HACK}}$ and

using $\overline{\text{HEN}}$ and HA0-HA2 to transfer data. The host can therefore transfer data in the other direction during the DMA operation using polling techniques.

10.2.6.3.1 Host To DSP Internal Processing

The following procedure outlines the steps that the HI hardware takes to transfer DMA data from the host data bus to DSP memory (see Figures 10-36 and 10-37).



Characteristics of Host DMA Mode

- The $\overline{\text{HREQ}}$ pin is **NOT** available for host processor interrupts.
- TREQ and RREQ select the direction of DMA transfer.
  - — DMA to DSP56000
  - — DSP56000 to DMA
  - — Simultaneous bidirectional DMA transfers are not permitted.
- Host processor software polled transfers are permitted in the opposite direction of the DMA transfer.
- 8-, 16-, or 24-bit transfers are supported.
  - — 16-, or 24-bit transfers reduce the DSP interrupt rate by a factor of 2 or 3, respectively.

Figure 10-36 HI Hardware–DMA Mode

11. HI asserts the $\overline{\text{HREQ}}$ pin (see Figure 10-36 and Figure 10-37) when TXDE=1.
12. DMA controller enables data on H0-H7 and asserts $\overline{\text{HACK}}$.
13. When $\overline{\text{HACK}}$ is asserted, the HI deasserts $\overline{\text{HREQ}}$.
14. When the DMA controller deasserts $\overline{\text{HACK}}$, the data on H0-H7 is latched into the TXH, TXM, TXL registers.
15. If the byte register written was not TXL (i.e., not $7) the DMA address counter internal to the HI increments and $\overline{\text{HREQ}}$ is again asserted. Steps 2-5 are then repeated.
16. If TXL ($7) was written, TXDE will be set to zero and the address counter in the HI will be loaded with the contents of HM1 and HM0. When TXDE=0, the contents of TXH:TXM:TXL are transferred to HRX provided HRDF=0. After the

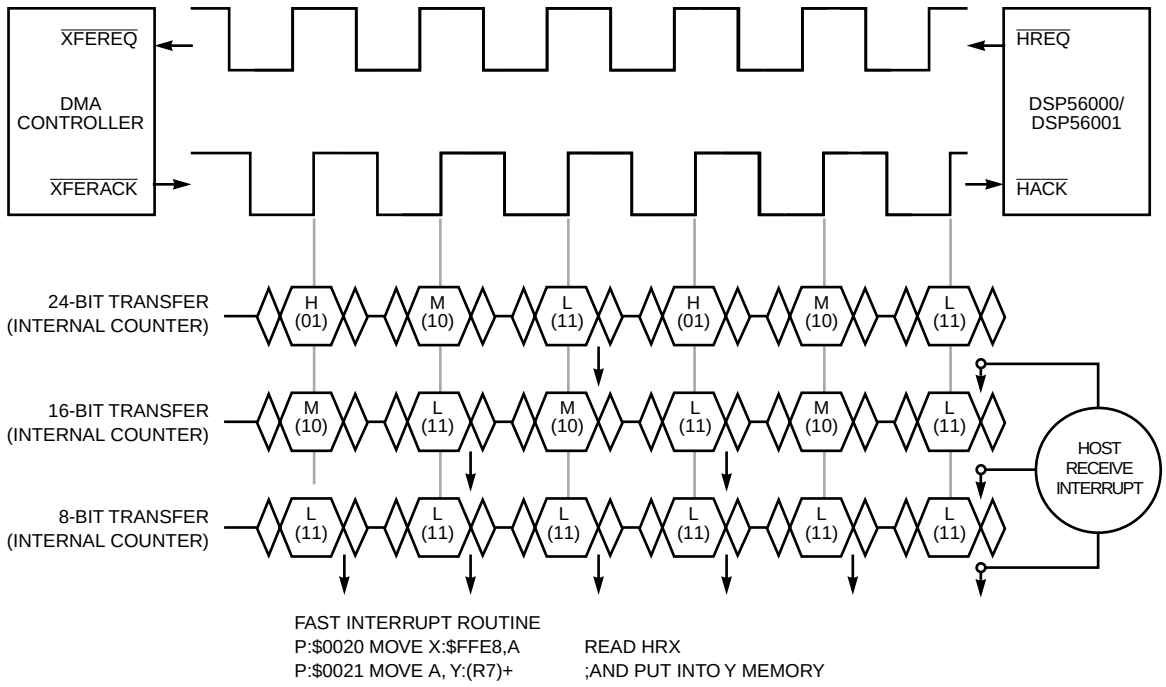


Figure 10-37 DMA Transfer and Host Interrupts

transfer to HRX, TXDE will be set to one, and $\overline{\text{HREQ}}$ will be asserted to start the transfer of another word from external memory to the HI.

17. When the transfer to HRX occurs within the HI, HRDF is set to one. Assuming HRIE=1, a host receive exception will be generated. The exception routine must read the HRX to clear HRDF.

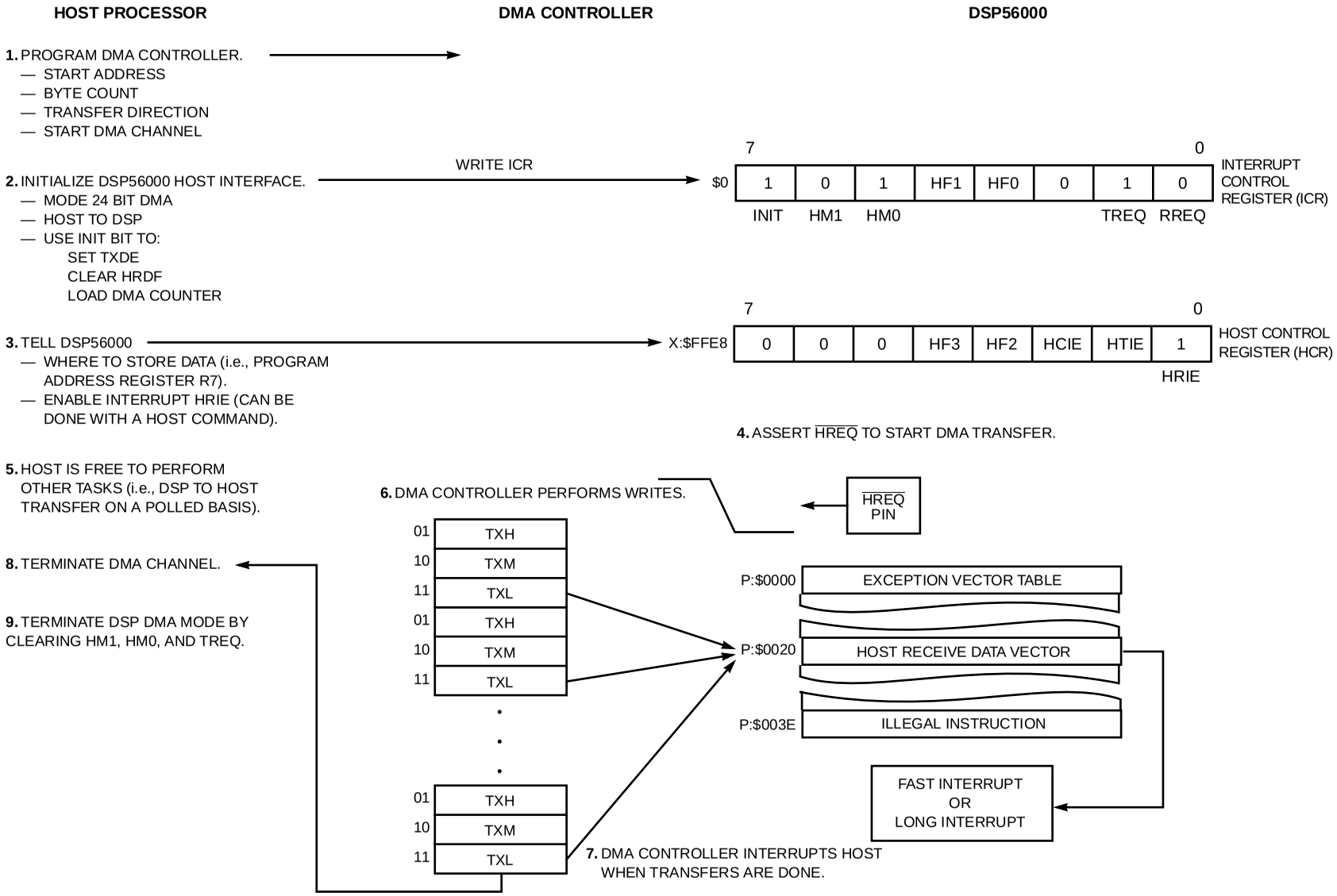
Note: The transfer of data from the TXH, TXM, TXL registers to the HRX register automatically loads the DMA address counter from the HM1 and HM0 bits in the DMA host to DSP mode. This DMA address is used with the HI to place the received byte in the correct register (TXH, TXM, or TXL).

Figure 10-37 shows the differences between 24-, 16-, and 8-bit DMA data transfers. The interrupt rate is three times faster for 8-bit data transfers than for 24-bit transfers. TXL is always loaded last.

10.2.6.3.2 Host-to-DSP DMA Procedure

The following procedure outlines the typical steps that the host processor must take to setup and terminate a host-to-DSP DMA transfer (see Figure 10-38).

1. Set up the external DMA controller (1) source address, byte count, direction, and other control registers. Enable the DMA controller channel.
2. Initialize the HI (2) by writing the ICR to select the word size (HM0 and HM1), to select the direction (TREQ=1, RREQ=0), and to initialize the channel setting



INIT=1 (see Figure 10-39).

Figure 10-38 Host-to-DSP DMA Procedure

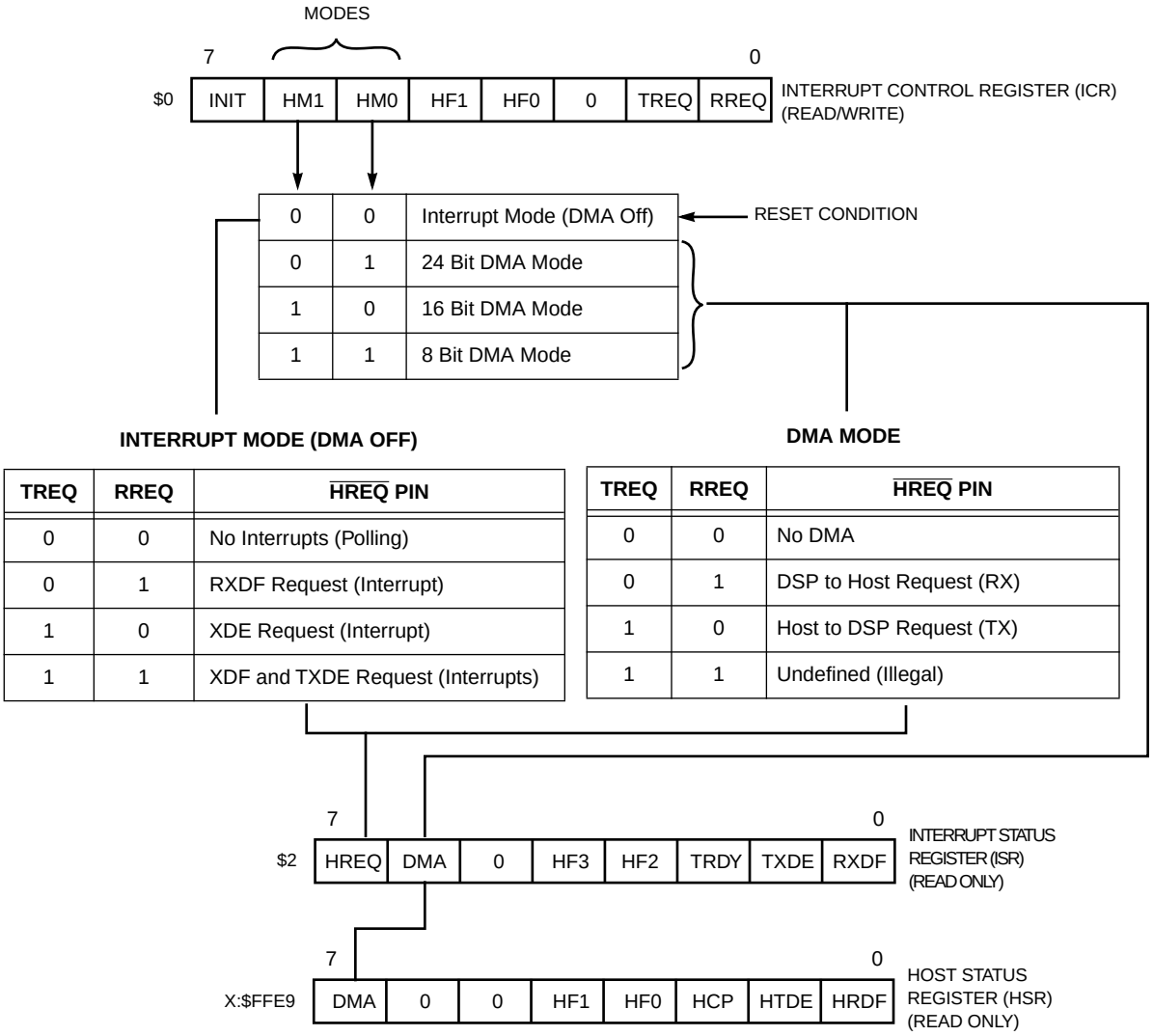


Figure 10-39 Host Bits with TREQ and RREQ

3. The DSP's destination pointer (3) used in the DMA exception handler (an address register, for example) must be initialized and HRIE must be set to enable the HRDF interrupt to the DSP CPU. This procedure can be done with a separate host command exception routine in the DSP. $\overline{\text{HREQ}}$ will be asserted (4) immediately by the HI to begin the DMA transfer.
4. Perform other tasks (5) while the DMA controller transfers data (6) until interrupted by the DMA controller DMA transfer complete interrupt (7). The DSP interrupt control register (ICR), the interrupt status register (ISR), and RXH, RXM, and RXL registers may be accessed at any time by the host processor but the TXH, TXM and TXL registers may not be accessed until the DMA mode is disabled.

5. Terminate the DMA controller channel (8) to disable DMA transfers.
6. Terminate the DSP HI DMA mode (9) in the ICR by clearing the HM1 and HM0 bits and clearing TREQ.

The $\overline{\text{HREQ}}$ will be active immediately after initialization is completed (depending on hardware) because the data direction is host to DSP and TXH, TXM, and TXL registers are empty. When the host writes data to TXH, TXM, and TXL, this data will be immediately transferred to HRX. If the DSP is due to work in interrupt mode, HRIE must be enabled.

10.2.6.3.3 DSP-to-Host Internal Processing

The following procedure outlines the steps that the HI hardware takes to transfer DMA data from DSP memory to the host data bus.

1. On the DSP side of the HI, a host transmit exception will be generated when HTDE=1 and HTIE=1. The exception routine must write HTX, thereby setting HTDE=0.
2. If RXDF=0 and HTDE=0, the contents of HTX will be automatically transferred to RXH:RXM:RXL, thereby setting RXDF=1 and HTDE=1. Since HTDE=1 again on the initial transfer, a second host transmit exception will be generated immediately, and HTX will be written, which will clear HTDE again.
3. When RXDF is set to one, the HI's internal DMA address counter is loaded (from HM1 and HM0) and $\overline{\text{HREQ}}$ is asserted.
4. The DMA controller enables the data from the appropriate byte register onto H0-H7 by asserting $\overline{\text{HACK}}$. When $\overline{\text{HACK}}$ is asserted, $\overline{\text{HREQ}}$ is deasserted by the HI.
5. The DMA controller latches the data presented on H0-H7 and deasserts $\overline{\text{HACK}}$. If the byte register read was not RXL (i.e., not $7), the HI's internal DMA counter increments, and $\overline{\text{HREQ}}$ is again asserted. Steps 3, 4, and 5 are repeated until RXL is read.
6. If RXL was read, RXDF will be set to zero and, since HTDE=0, the contents of HTX will be automatically transferred to RXH:RXM:RXL, and RXFD will be set to one. Steps 3, 4, and 5 are repeated until RXL is read again.

Note: The transfer of data from the HTX register to the RXH:RXM:RXL registers automatically loads the DMA address counter from the HM1 and HM0 bits when in the DMA DSP--HOST mode. This DMA address is used within the HI to place the appropriate byte on H0-H7.

10.2.6.3.4 DSP-to-Host DMA Procedure

The following procedure outlines the typical steps that the host processor must take to setup and terminate a DSP-to-host DMA transfer (see Figure 10-40).

1. Set up the DMA controller (1) destination address, byte count, direction, and other control registers. Enable the DMA controller channel.
2. Initialize the HI (2) by writing the ICR to select the word size (HM0 and HM1), the direction (TREQ=0, RREQ=1), and setting INIT=1 (see Figure10-40 for additional information on these bits).
3. The DSP's source pointer (3) used in the DMA exception handler (an address register, for example) must be initialized, and HTIE must be set to enable the DSP host transmit interrupt. This could be done by the host processor with a host command exception routine.

   The DSP host transmit exception will be activated immediately after HTIE is set. The DSP CPU will move data to HTX. The HI circuitry will transfer the contents of HTX to RXH:RXM:RXL, setting RXDF which asserts $\overline{\text{HREQ}}$. Asserting $\overline{\text{HREQ}}$ (4) starts the DMA transfer from RXH, RXM, and RXL to the host processor.

4. Perform other tasks (5) while the DMA controller transfers data (6) until interrupted by the DMA controller DMA complete interrupt (7). The DSP interrupt control register (ICR), the interrupt status register (ISR), and TXH, TXM, and TXL may be accessed at any time by the host processor but the RXH, RXM and RXL registers may not be accessed until the DMA mode is disabled.
5. Terminate the DMA controller channel (8) to disable DMA transfers.
6. Terminate the DSP HI DMA mode (9) in the Interrupt Control Register (ICR) by clearing the HM1 and HM0 bits and clearing RREQ.

10.2.6.4 Example Circuits

Figure 10-41, Figure 10-43, and Figure 10-42 illustrate the simplicity of the HI. The MC68HC11 in Figure 10-41 has a multiplexed address and data bus which

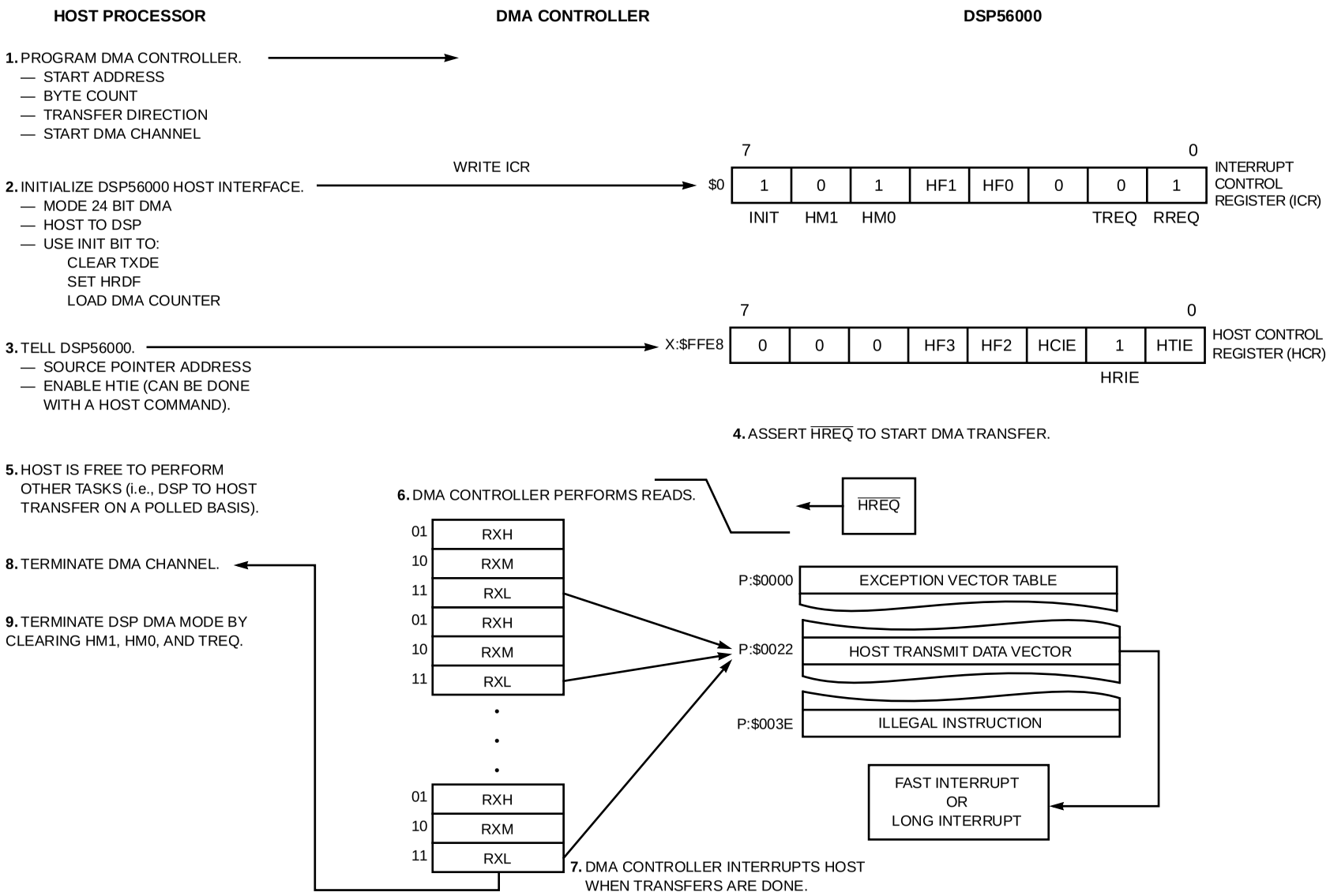


Figure 10-40 DSP to Host DMA Procedure

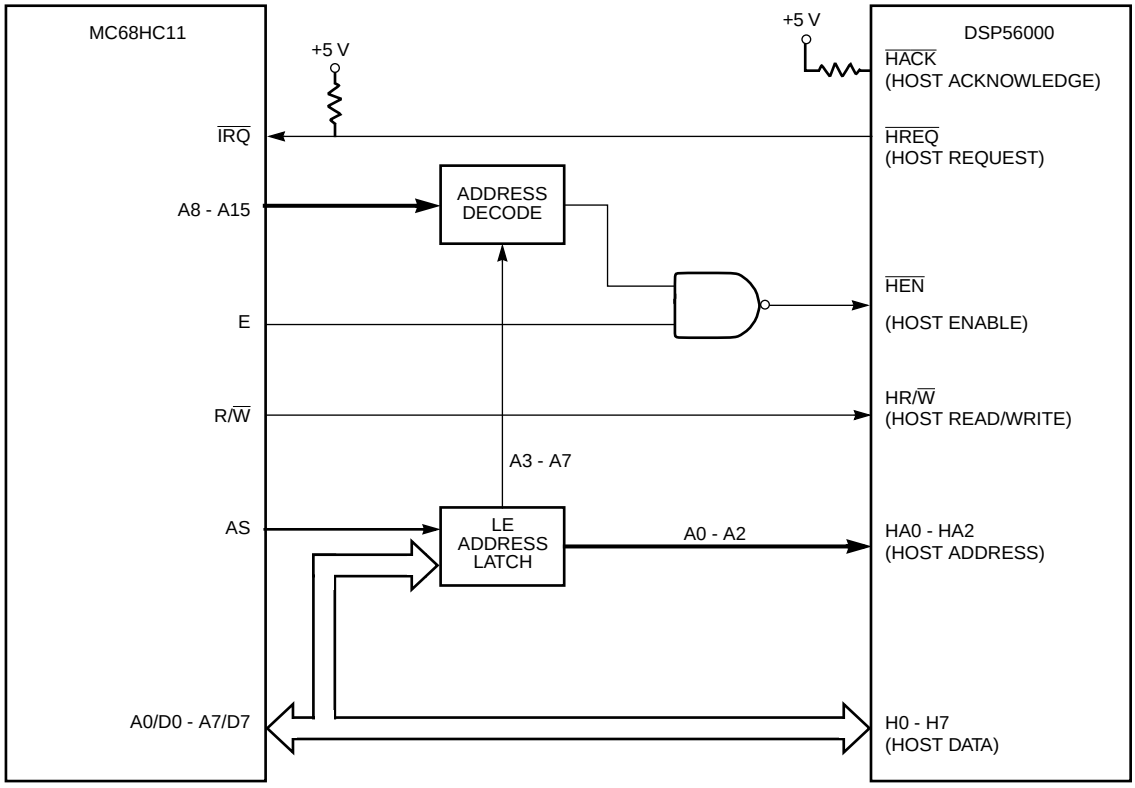


Figure 10-41 MC68HC11 to DSP56000 Host Interface

requires that the address be latched. Although the HACK is not used in this circuit, it is pulled up. All unused input pins should be terminated to prevent erroneous signals. When determining whether a pin is an input, keep in mind that it may change during reset or while changing port B between general purpose I/O and HI functions.

The MC68000 (see Figure 10-43) can use a MOVEP instruction with word and long-word data size to transfer multiple bytes. If an MC68020 or MC68030 is used, dynamic bus sizing can be used to transfer multiple bytes with any instruction.

Figure 10-42 is a high level block diagram of a system using a single host to control multiple DSPs. In addition, the DSPs use the SSI to network together the DSPs and multiple codecs. This system, as shown with four DSPs, can process 41 million instructions per second and can be easily expanded if more processing power is needed.

10.2.6.5 Host Port Usage Considerations–Host Side. Careful synchronization is required when reading multi-bit registers that are written by another asynchronous system. Synchronization is a common problem when two asynchronous systems are connected. The

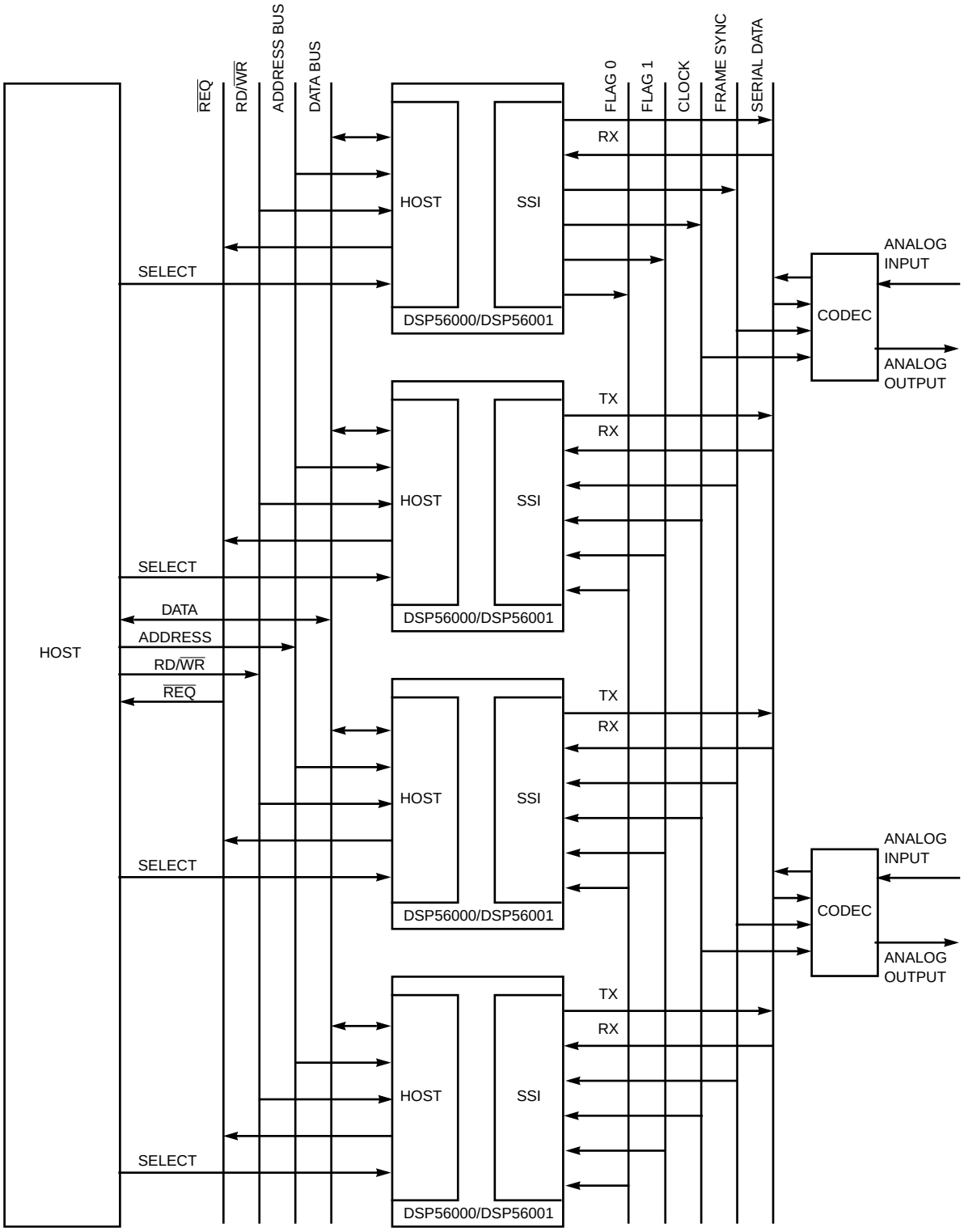
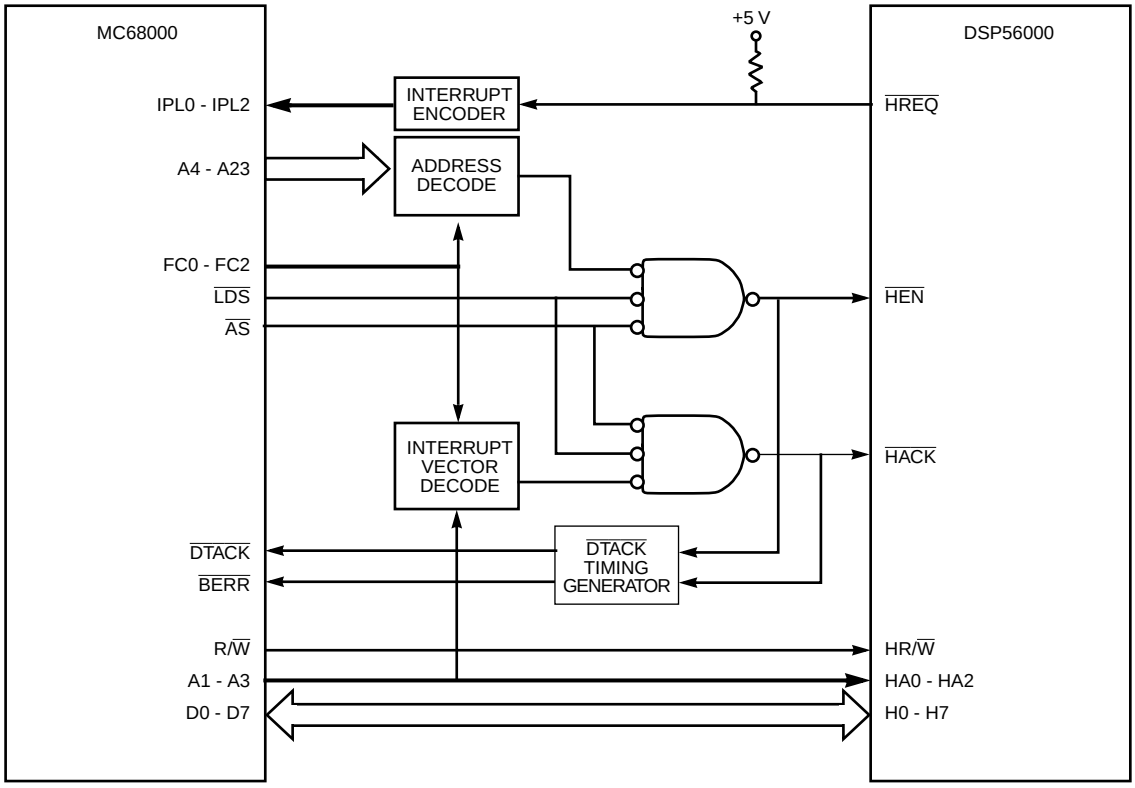


Figure 10-42 Multi-DSP Network Example

situation exists in the host port. However, if the port is used in the way it was designed, proper



MC68000 — USE MOVEP for multiple byte transfers.
MC68020 or MC68030 — Any Memory references will work due to dynamic bus sizing.

Figure 10-43 MC68000 to DSP56000 Host Interface

operation is guaranteed. The considerations for proper operation are discussed below.

1. Unsynchronized Reading of Receive Byte Registers:
   When reading receive byte registers, RXH, RXM, or RXL, the host processor should use interrupts or poll the RXDF flag which indicates that data is available. This guarantees that the data in the receive byte registers will be stable.

2. Overwriting Transmit Byte Registers:
   The host processor should not write to the transmit byte registers, TXH, TXM, or TXL, unless the TXDE bit is set, indicating that the transmit byte registers are empty. This guarantees that the DSP will read stable data when it reads the HRX register.

3. Synchronization of Status Bits from DSP to Host:
   HC, HREQ, DMA, HF3, HF2, TRDY, TXDE, and RXDF status bits are set or cleared from inside the HI and read by the host processor. The host can read these status bits very quickly without regard to the clock rate used by the DSP, but there is a chance that the state of the bit could be changing during the read

operation. This possible change is generally not a system problem, since the bit will be read correctly in the next pass of any host polling routine.

However, if the host holds the $\overline{\text{HEN}}$ for the minimum assert time plus 1.5 clock cycle, the status data is guaranteed to be stable. The 1.5 clock cycle is used to synchronize the $\overline{\text{HEN}}$ signal and block internal updates of the status bits.There is no other minimum $\overline{\text{HEN}}$ assert time relationship to DSP clocks.

There is a minimum $\overline{\text{HEN}}$ deassert time of 1.5 clock cycle so that the blocking latch can be updated if host is in a tight polling loop. This minimum time only applies to reading status bits.

The only potential problem with the host processor reading status bits is reading HF3 and HF2 as an encoded pair. For example, if the DSP changes HF3 and HF2 from ‘‘00’’ to ‘‘11’’ there is a very small probability that the host could read the bits during the transition and receive ‘‘01’’ or ‘‘10’’ instead of ‘‘11’’. If the combination of HF3 and HF2 has significance, the host processor would potentially read the wrong combination. Two solutions would be to 1) read the bits twice and check for consensus, and 2) hold $\overline{\text{HEN}}$ access for $\overline{\text{HEN}}$ +1.5 clock cycle so that status bit transitions are stabilized

4. Overwriting the Host Vector:
   The host programmer should change the host vector register only when the HC bit is clear. This will guarantee that the DSP interrupt control logic will receive a stable vector.

5. Cancelling a Pending Host Command Exception:
   The host processor may elect to clear the HC bit to cancel the host command exception request at any time before it is recognized by the DSP. The DSP CPU may execute the host exception after the HC bit is cleared because the host processor does not know exactly when the exception will be recognized. This uncertainty in timing is due to differences in synchronization between the host processor and DSP CPU and the uncertainties of pipelined exception processing. For this reason, the HV should not be changed at the same time the HC bit is cleared. However, the HV can be changed when the HC bit is set.

6. When using the $\overline{\text{HREQ}}$ pin for handshaking, wait until $\overline{\text{HREQ}}$ is asserted and then start writing/reading data using the $\overline{\text{HEN}}$ pin or the $\overline{\text{HACK}}$ pin.

   When not using $\overline{\text{HREQ}}$ for handshaking, poll the INIT bit in the ICR to make sure it is cleared by the hardware (which means the INIT execution is completed). Then, start writing/reading data.

   If using neither $\overline{\text{HREQ}}$ for handshaking, nor polling the INIT bit, wait at least 6T

after negation of $\overline{\text{HEN}}$ that wrote ICR, before writing/reading data. This wait ensures that the INIT is completed, because it needs 3T for synchronization (worst case) plus 3T for executing the INIT.

7. All unused input pins should be terminated. Also, any pin that is temporarily not driven by an output1) during reset, 2) when reprogramming a port or pin, 3) when a bus is not driven, or 4) at any other time, should be pulled up or down with a resistor. For example, the $\overline{\text{HEN}}$ is capable of reacting to 2-ns noise spikes when it is not terminated. Allowing $\overline{\text{HACK}}$ to float may cause problems even though it is not needed in the circuit.

SECTION 11
PORT C

Port C is a triple-function I/O port with nine pins (see Figure 11-1). Three of the nine pins can be configured as general-purpose I/O or as the serial communications interface (SCI) pins, and the other six pins can be configured as general-purpose I/O or as the synchronous serial interface (SSI) pins. When configured as general-purpose I/O, port C can be used for device control. When the pins are configured as serial interfaces, port C provides a convenient connection to other DSPs, processors, codecs, digital-to-analog and analog-to-digital converters, and any of several transducers. This section describes all three port C functions as well as examples of how to configure and use each function.

11.1 GENERAL-PURPOSE I/O (PORT C)

When configured as general-purpose I/O, port C can be viewed as nine I/O pins (see Figure 11-2), which are controlled by three memory-mapped registers (see Figure 11-3). RESET configures port C as general-purpose I/O with all nine pins as inputs by clearing all three registers (external circuitry connected to these pins may need pullups until the pins are configured for operation). These registers are the port C control register (PCC), port C data direction register (PCDDR), and port C data register (PCD). Each port C pin may be individually programmed as a general-purpose I/O pin or as a dedicated on-chip peripheral pin under software control. Pin selection between general-purpose I/O and SCI or SSI is made by setting the appropriate PCC bit (memory location X:$FFE1) to zero for general-purpose I/O or to one for serial interface. The PCDDR (memory location X:$FFE3) programs each pin corresponding to a bit in the PCD (memory location X:$FFE5) as an input pin (if PCDDR=0) or as an output pin (if PCDDR=1). Writing to the PCD will write data to the pins designated as outputs by the PCDDR; reading the PCD will read the pins designated as inputs by the PCDDR.

The port C I/O pin control logic is shown in Figure 11-4. When a pin is designated as an output and the PCD is read, the output of the output data bit latch is read, not the logic level on the pin itself. When a port pin is configured as an SCI or SSI pin and the bit in the PCDDR is zero (input), then reading the PCD will show the logic level on the pin even though the pin is configured as a peripheral pin. The SCI or SSI function may be using the pin as an input or an output, which can be very useful when debugging the SCI or SSI.

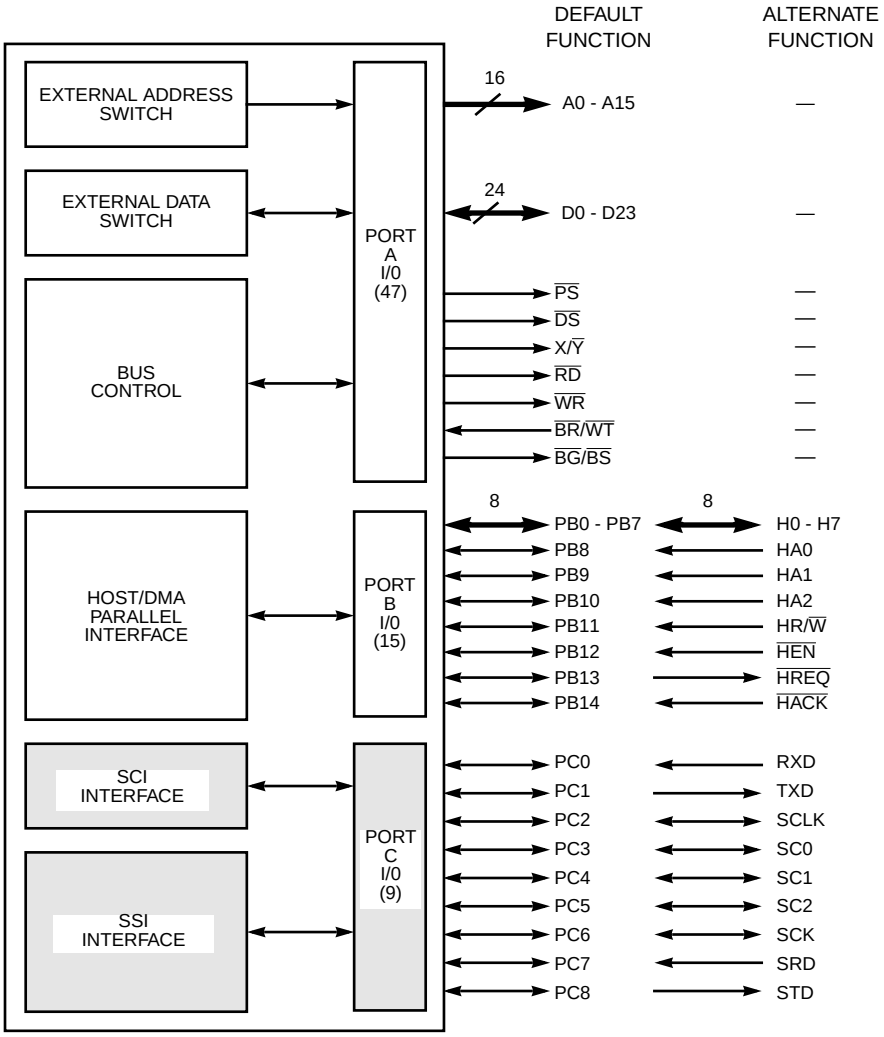


Figure 11-1 Port C Interface

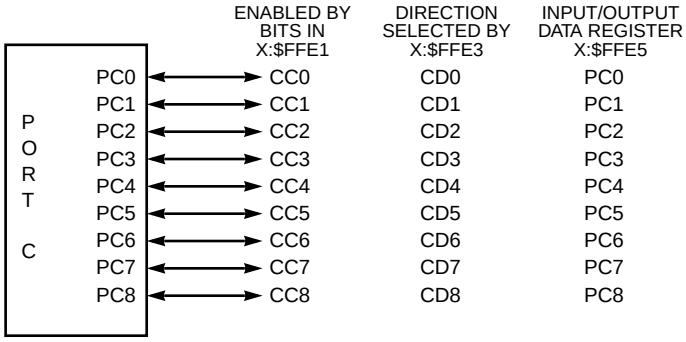
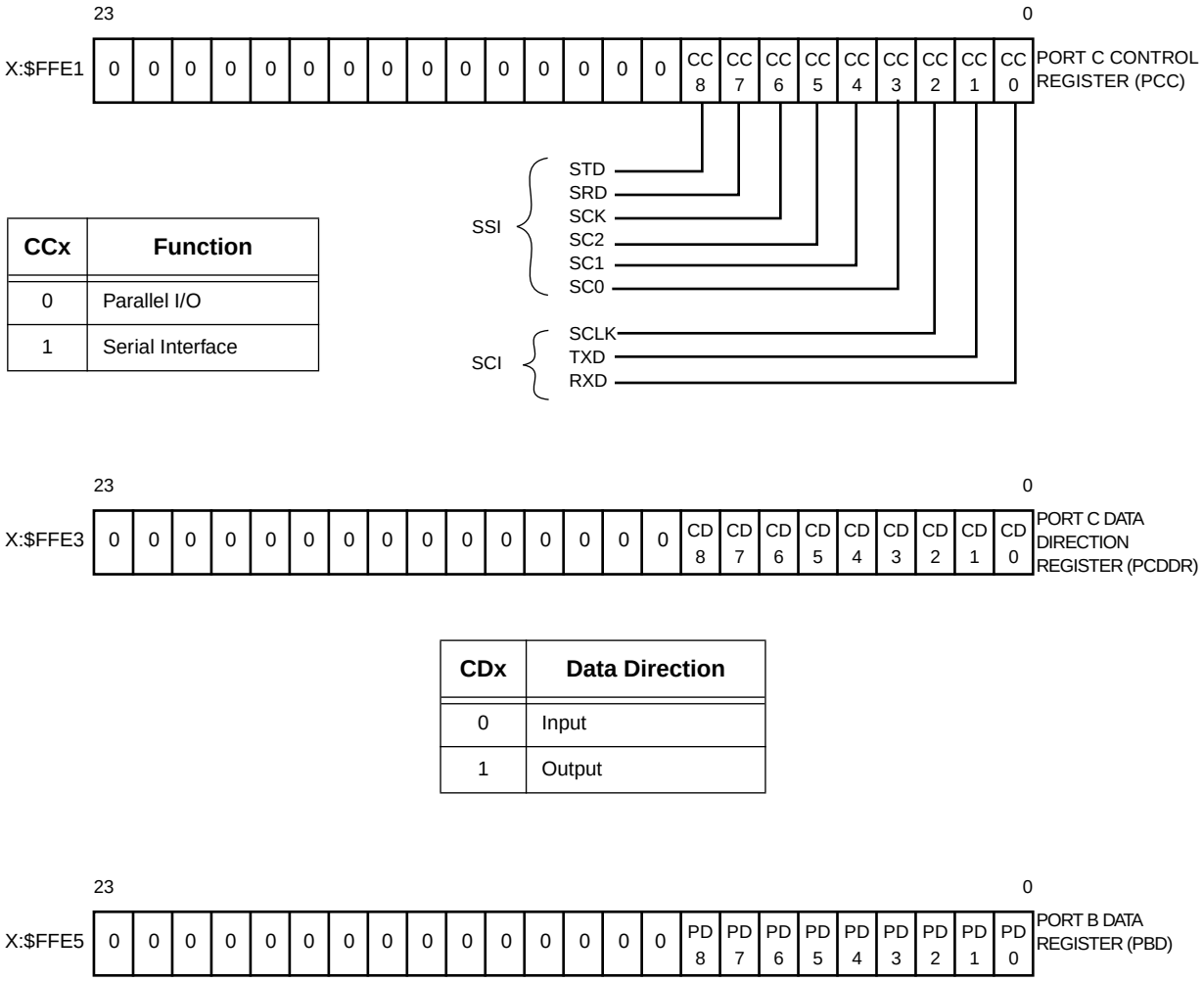


Figure 11-2 Parallel Port C Pinout



NOTE: Hardware and software reset clears PCC and PCDDR.

Figure 11-3 Parallel Port C Registers

Programming Parallel I/O

Port C and all the DSP56000/DSP56001 peripherals are memory mapped (see Figure 11-5). The standard MOVE instruction transfers data between port C and a register; as a result, performing a memory-to-memory data transfer takes two MOVE instructions and a register. The MOVEP instruction is specifically designed for I/O data transfer as shown in Figure 11-6. Although the MOVEP instruction may take twice as long to execute as a MOVE instruction, only one MOVEP is required for a memory-to-memory data transfer, and MOVEP does not use a temporary register. Using the MOVEP instruction allows a fast interrupt to move data to/from a peripheral to memory and execute one other instruction or to move the data to an absolute address. MOVEP is the only memory-to-memory move instruction; however, one of the operands must be in the top 64 locations of either

Port Control Register Bit	Data Direction Register Bit	Pin Function
0	0	Port Input Pin

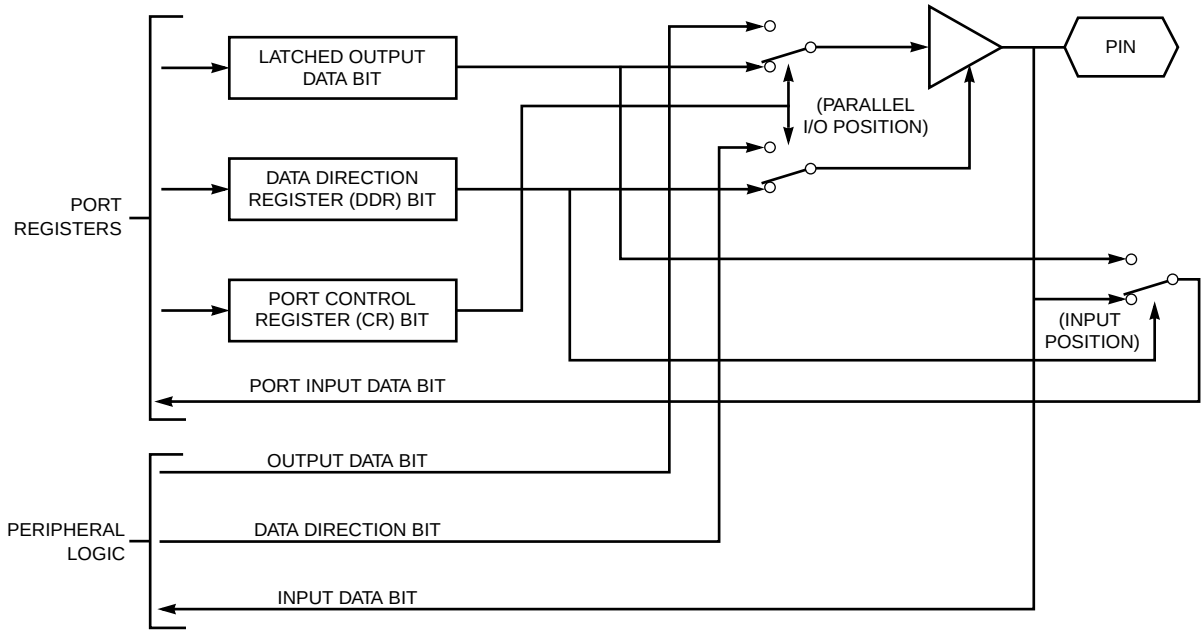


Figure 11-4 Port C I/O Pin Control Logic

X: or Y: memory. The bit-oriented instructions using I/O short addressing (BCHG, BCLR, BSET, BTST, JCLR, JSCLR, JSET, and JSSET) can also be used to address individual bits for faster I/O processing. The DSP does not have a hardware data strobe to strobe data out of the parallel I/O port. If a data strobe is needed, it can be implemented using software to toggle one of the parallel I/O pins. The process of programming port C as general-purpose I/O is shown as a flowchart in Figure 11-7 and detailed in Figure 11-8. Normally, it is not good programming practice to activate a peripheral before programming it. However, reset activates the port C general-purpose I/O as all inputs, and the alternative is to configure the port as an SCI and/or SSI, which may not be desirable. In this case, it is probably better to insure that port C is initially configured for general-purpose I/O and then configure the data direction and data registers. It may be better in some situations to program the data direction or the data registers first to prevent two devices from driving one signal. The order of steps 1, 2, and 3 in Figure 11-7 is optional and can be changed as needed.

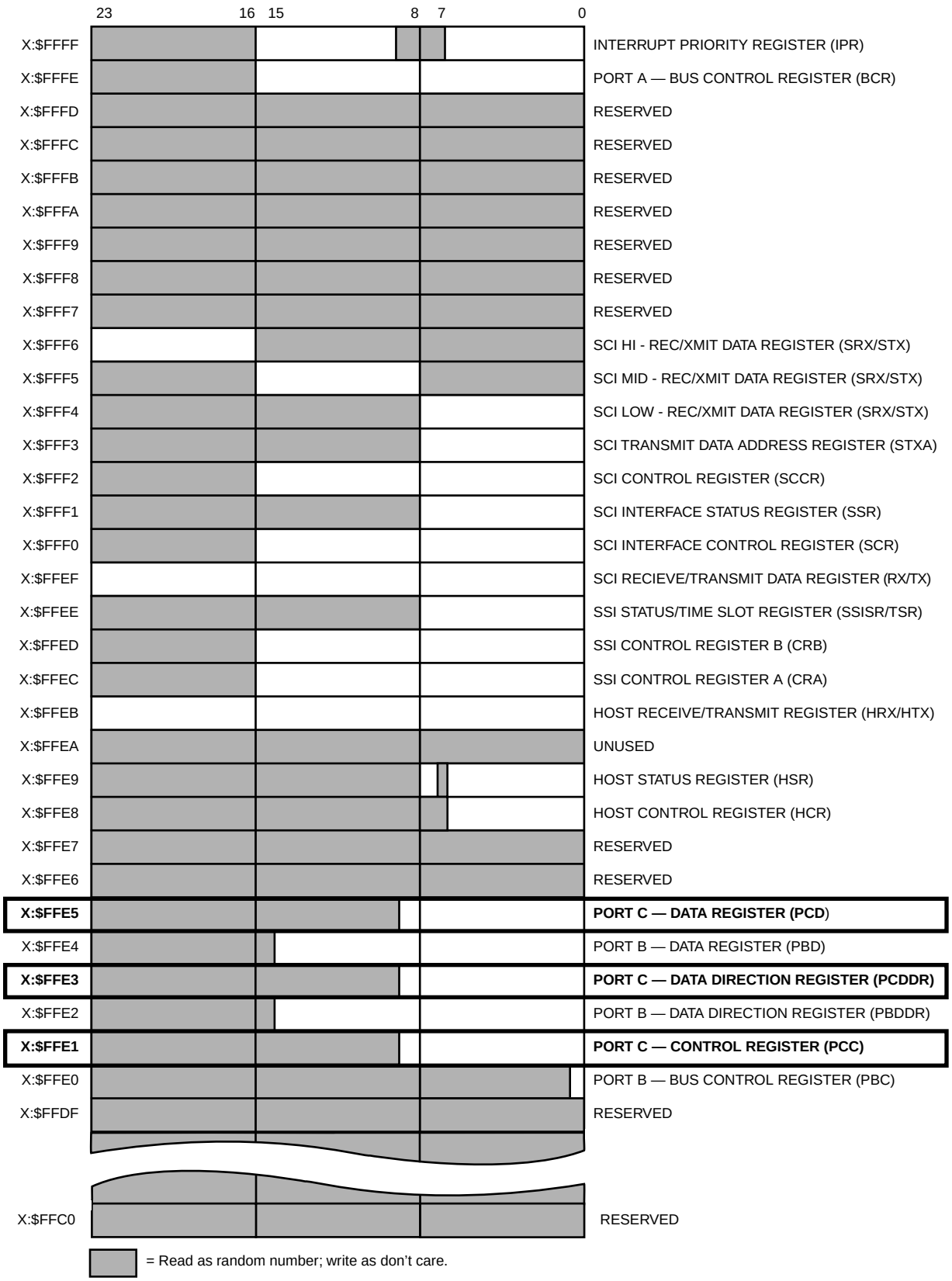


Figure 11-5 On-Chip Peripheral Memory Map

```
    :
    :
MOVEP     #$0,X:$FFE1          ;Select port C to be general-purpose I/O
MOVEP     #$01F0,X:$FFE3       ;Select pins PC0–PC3 to be inputs
                               ;and pins PC4–PC8 to be outputs
    :
    :
MOVEP     #data_out,X:$FFE5    ;Put bits 4–8 of “data_out” on pins
                               ;PB4–PB8 bits 0–3 are ignored.
MOVEP     X:$FFE0,#data_in     ;Put PB0–PB3 in bits 0–3 of “data_in”
```

Figure 11-6 Write/Read Parallel Data with Port C

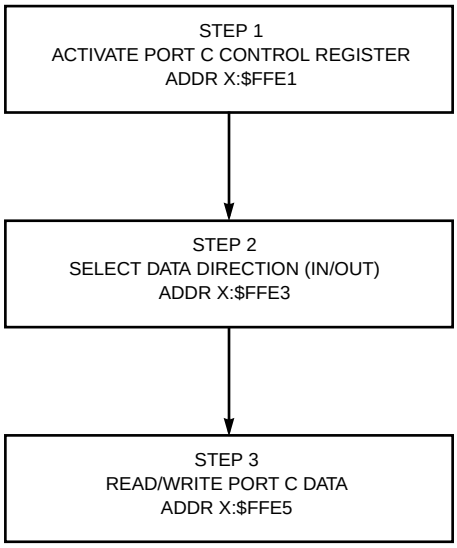


Figure 11-7 Port C Configuration Flowchart

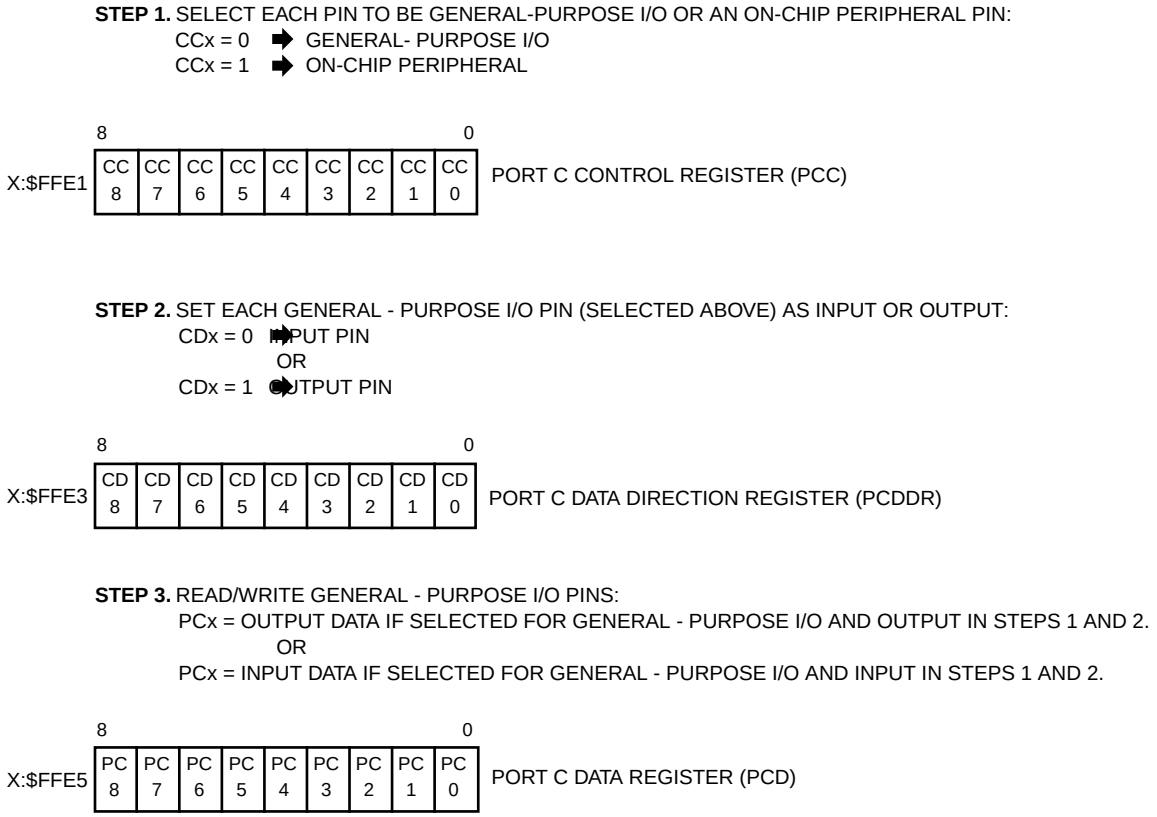


Figure 11-8 I/O Port C Configuration

11.1.1 Port C Parallel I/O Timing

Parallel data written to port C is delayed by one instruction cycle – i.e., the following instruction

```
MOVE    DATA9,X:PORTC    DATA24,Y:EXTERN
```

1. writes nine bits of data to the port C register, but the output pins do not change until the following instruction cycle, and
2. writes 24 bits of data to the external Y memory, which appears on port A during T2 and T3 of the current instruction.

As a result, if it is desirable to synchronize the port A and port C outputs, two instructions must be used:

```
MOVE    DATA9,X:PORTC
NOP                      DATA24,Y:EXTERN
```

The NOP can be replaced by any instruction that allows parallel moves. Inserting one or

more “MOVE DATA15,X:PORTC DATA24,Y:EXTERN” instructions between the first and second instruction produces an external 33-bit write each instruction cycle with only one instruction cycle lost in setup time:

```
MOVE    DATA15,X:PORTC
MOVE    DATA15,X:PORTC    DATA24,Y:EXTERN
MOVE    DATA15,X:PORTC    DATA24,Y:EXTERN
   :
   :
MOVE    DATA15,X:PORTC    DATA24,Y:EXTERN
NOP                       DATA24,Y:EXTERN
```

One application of this technique is to create an extended address for port A by concatenating the port A address bits (instead of data bits) to the port C general-purpose output bits. The port C general-purpose I/O register would then work as a base address register, allowing the address space to be extended from 64K words (16 bits) to 33.5 million words (16 bits+ 9 bits=25 bits).

Port C uses the DSP central processing unit (CPU) four-phase clock for its operation. Therefore, if wait states are inserted in the DSP CPU timing, they also affect port C timing. The result is that port A and port C in the previous synchronization example will always stay synchronized, regardless of how many wait states are used.

11.2 SERIAL COMMUNICATION INTERFACE (SCI)

The SCI provides a full-duplex port for serial communication to other DSPs, microprocessors, or peripherals such as modems. The communication can be TTL-level signals or, with additional logic, RS232C, RS422, etc. This interface uses three dedicated pins: transmit data (TXD), receive data (RXD), and SCI serial clock (SCLK). It supports industry-standard asynchronous bit rates and protocols as well as high-speed (up to 3.375 Mbps for a 27-MHz clock) synchronous data transmission. The asynchronous protocols include a multidrop mode for master/slave operation with wakeup on idle line and wakeup on address bit capability. The SCI consists of separate transmit and receive sections whose operations can be asynchronous with respect to each other. A programmable baud-rate generator is included to generate the transmit and receive clocks. An enable vector and an interrupt vector have been included so that the baud-rate generator can function as a general-purpose timer when it is not being used by the SCI peripheral or when the interrupt timing is the same as that used by the SCI. The following is a short list of SCI features:

- Three-Pin Interface:
  - TXD – Transmit Data
  - RXD – Receive Data
  - SCLK – Serial Clock
- 422 Kbps NRZ Asynchronous Communications Interface (27-MHz System Clock)
- 3.375 Mbps Synchronous Serial Mode (27-MHz System Clock)
- Multidrop Mode for Multiprocessor Systems:
  - Two Wakeup Modes: Idle Line and Address Bit
  - Wired-OR Mode
- On-Chip or External Baud Rate Generation/Interrupt Timer
- Four Interrupt Priority Levels
- Fast or Long Interrupts

11.2.1 SCI I/O Pins

The SCI has three I/O pins, which can be configured as either general-purpose I/O or as a specific SCI pin. Each pin is independent of the other two, which means that if only TXD is needed, RXD and SCLK can be programmed for general-purpose I/O. At least one of the three pins must be selected as an SCI pin to release the SCI from SCI reset.

However, the SCI interrupts may be enabled by programming the SCI control registers before any of the SCI pins are programmed as SCI functions. In this case, only one transmit interrupt can be generated because the transmit data register is empty. The timer and timer interrupt do not require that any SCI pins be configured for SCI use to operate.

11.2.1.1 Receive Data (RXD). This input receives byte-oriented serial data and transfers the data to the SCI receive shift register. Asynchronous input data is sampled on the positive edge of the receive clock (1 × SCLK) if SCKP equals zero. See the DSP56001 Advance Information Data Sheet (DSP56001/D) for detailed timing information. RXD may be programmed as a general-purpose I/O pin (PC0) when the SCI RXD function is not being used.

11.2.1.2 Transmit Data (TXD). This output transmits serial data from the SCI transmit shift register. Data changes on the negative edge of the asynchronous transmit clock (SCLK) if SCKP equals zero. This output is stable on the positive edge of the transmit clock. See the DSP56001 Advance Information Data Sheet (ADI1290) for detailed timing information. TXD may be programmed as a general-purpose I/O pin (PC1) when the SCI TXD function is not being used.

11.2.1.3 SCI Serial Clock (SCLK). This bidirectional pin provides an input or output clock from which the transmit and/or receive baud rate is derived in the asynchronous mode and from which data is transferred in the synchronous mode. SCLK may be programmed as a general-purpose I/O pin (PC2) when the SCI SCLK function is not being used. This pin may be programmed as PC2 when data is being transmitted on TXD since, in the asynchronous mode, the clock need not be transmitted. There is no connection between programming the PC2 pin as SCLK and data coming out the TXD pin because SCLK is independent of SCI data I/O.

11.2.2 Programming Model

The resources available in the SCI are described before discussing specific examples of how the SCI is used. The registers comprising the SCI are shown in Figure 11-9 and Figure 11-10. These registers are the SCI control register (SCR), SCI status register (SSR), SCI clock control register (SCCR), SCI receive data registers (SRX), SCI transmit data registers (STX), and the SCI transmit data address register (STXA). The SCI programming model

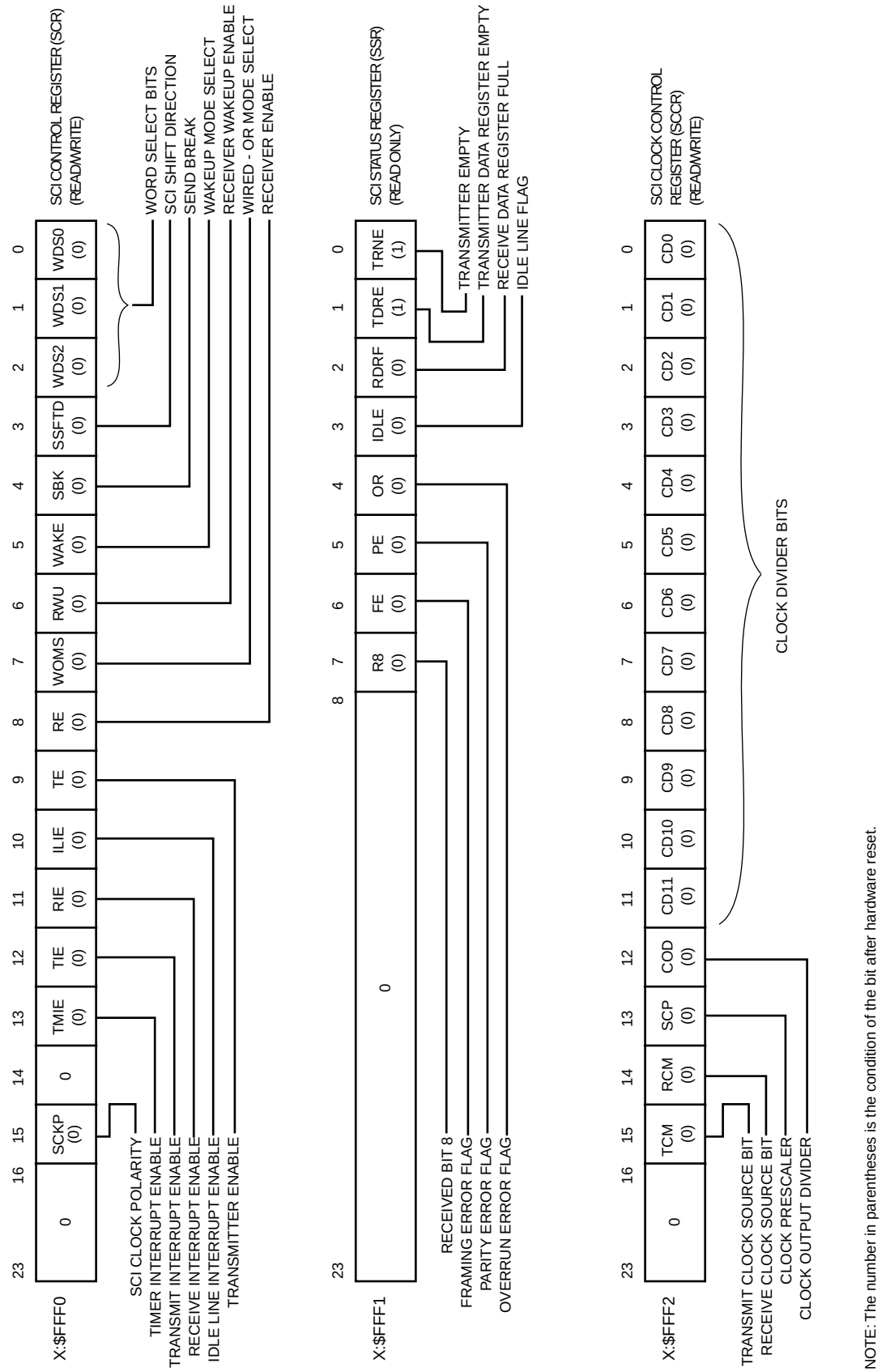


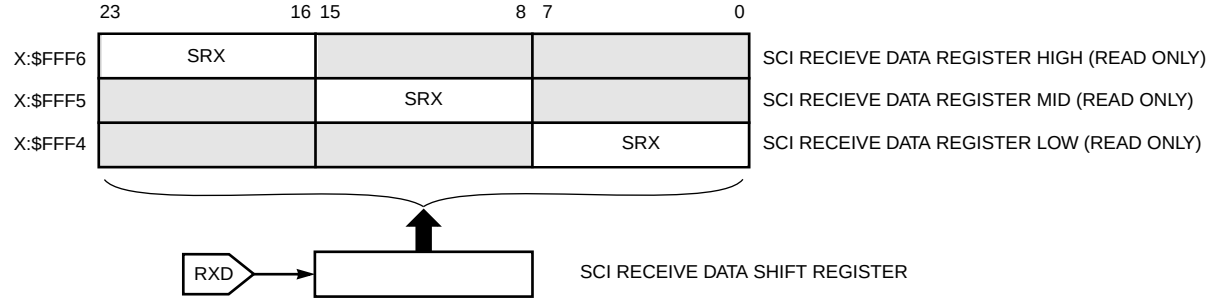
Figure 11-9 SCI Programming Model – Control and Status Registers

can be viewed as three types of registers: 1) control – SCR and SCCR in Figure 11-9; 2) status

– SSR in Figure 11-9; and 3) data transfer – SRX, STX, and STXA in Figure 11-10. The following paragraphs describe each bit in the programming model.

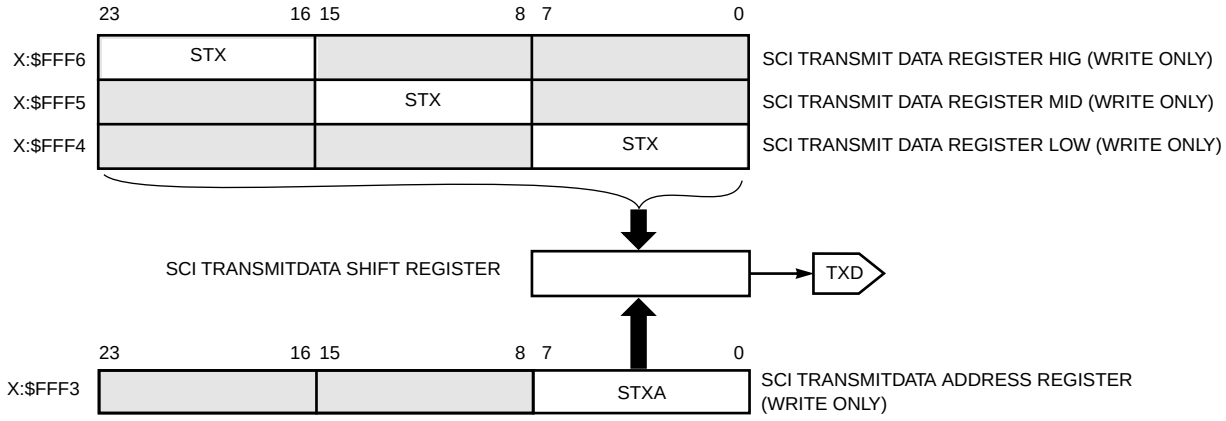
11.2.2.1 SCI Control Register (SCR)

The SCR is a 16-bit read/write register that controls the serial interface operation. Fifteen of the 16 bits are currently defined. Each bit is described in the following paragraphs.



NOTE: SRX is the same register decoded at three different addresses.

(a) Receive Data Register



NOTES:
1. Bytes are masked on the fly.
2. STX is the same register decoded at three different addresses.

(b) Transmit Data Register

Figure 11-10 SCI Programming Model

11.2.2.1.1 SCR Word Select (WDS0, WDS1, WDS2) Bits 0, 1, and 2

The three word-select bits (WDS0, WDS1, WDS2) select the format of the transmit and receive data. The formats include three asynchronous and one multidrop asynchronous

mode as well as an 8-bit synchronous (shift register) mode. The asynchronous modes are compatible with most UART-type serial devices. Standard RS232C communication links are supported by these modes.

The multidrop asynchronous modes are compatible with the MC68681 DUART, the M68HC11 SCI interface, and the Intel 8051 serial interface.

The synchronous data mode is essentially a high-speed shift register used for I/O expansion and stream-mode channel interfaces. Data synchronization is accomplished by the use of a gated transmit and receive clock that is compatible with the Intel 8051 serial interface mode 0. These formats are indicated below (also see Figure 11-11).

WDS2	WDS1	WDS0	Word Formats
0	0	0	8-Bit Synchronous Data (shift register mode)
0	0	1	Reserved
0	1	0	10-Bit Asynchronous (1 start, 8 data, 1 stop)
0	1	1	Reserved
1	0	0	11-Bit Asynchronous (1 start, 8 data, 1 even parity, 1 stop)
1	0	1	11-Bit Asynchronous (1 start, 8 data, 1 odd parity, 1 stop)
1	1	0	11-Bit Multidrop (1 start, 8 data, 1 data type, 1 stop)
1	1	1	Reserved

The word-select bits are cleared by hardware reset.

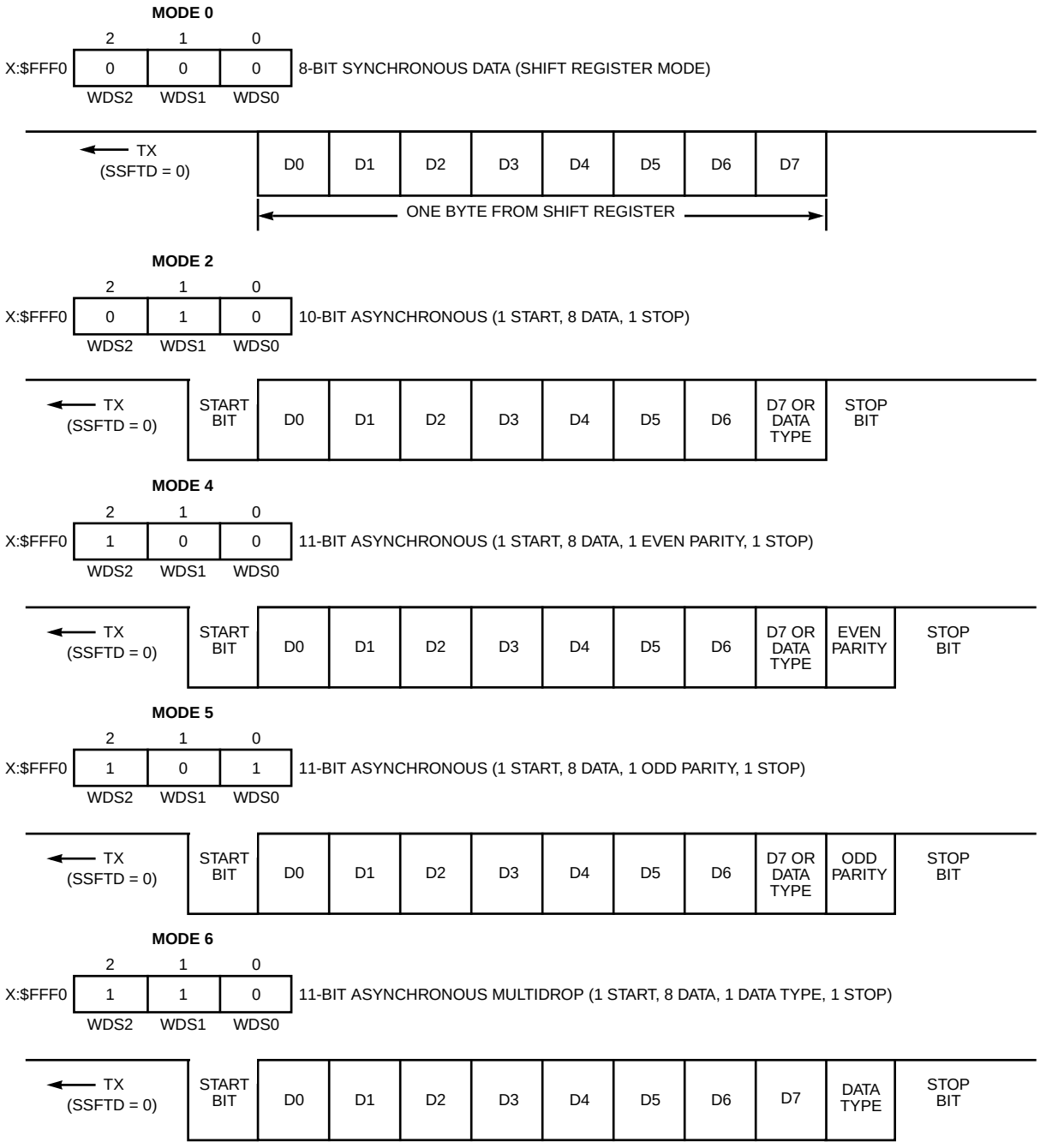
When odd parity is selected, the transmitter will count the number of bits in the data word; if the total is not an odd number, the parity bit is made equal to one and thus produces an odd number. If the receiver counts an even number of ones, an error in transmission has occurred. When even parity is selected, an even number must result from the calculation performed at both ends of the line or an error in transmission has occurred. The three word-select bits are cleared by hardware and software reset.

11.2.2.1.2 SCR SCI Shift Direction (SSFTD) Bit 3

The SCI data shift registers can be programmed to shift data in/out either LSB first if SSFTD equals zero or MSB first if SSFTD equals one. The parity and data type bits do not change position and remain adjacent to the stop bit. SSFTD should be cleared for compatibility with early versions of the DSP56000/DSP56001. SSFTD is cleared by hardware and software reset.

11.2.2.1.3 SCR Send Break (SBK) Bit 4

A break is an all-zero word frame – a start bit zero, a character of all zeros (including any



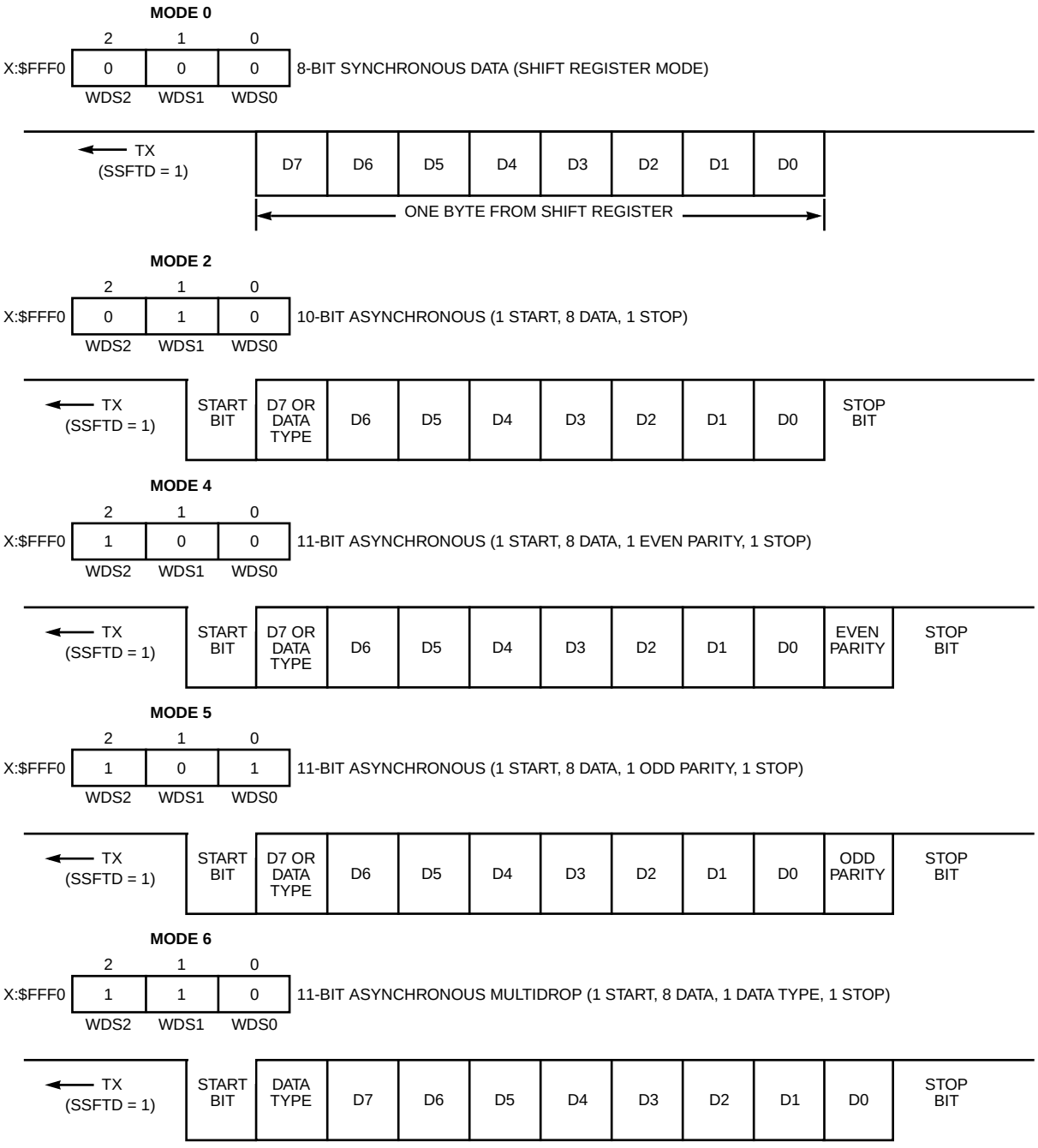
Data Type: 1 = Address Byte
0 = Data Byte

NOTES:
1. Modes1, 3, and 7 are reserved.
2. D0 =LDS;D7 = MSB
3. Data is transmitted and received LSB first if SSFTD = 0 or MSB first if SSFTD = 1.

(a) SSFTD = 0

Figure 11-11 Serial Formats (Sheet 1 of 2)

parity), and a stop bit zero: i.e., 10 or 11 zeros depending on the WDS mode selected. If



Data Type: 1 = Address Byte
0 = Data Byte

NOTES:
1. Modes 1, 3, and 7 are reserved.
2. D0 = LSB;D7 = MSB
3. Data is transmitted and received LSB first if SSFTD = 0 or MSB first if SSFTD = 1.

(b) SSFTD = 1

Figure 11-11 Serial Formats (Sheet 2 of 2)

SBK is set and then cleared, the transmitter completes transmission of any data, sends

10 or 11 zeros, and reverts to idle or sending data. If SBK remains set, the transmitter will continually send whole frames of zeros (10 or 11 bits with no stop bit). At the completion of the break code, the transmitter sends at least one high bit before transmitting any data to guarantee recognition of a valid start bit. Break can be used to signal an unusual condition, message, etc. by forcing a frame error, which is caused by a missing stop bit. Hardware and software reset clear SBK.

11.2.2.1.4 SCR Wakeup Mode Select (WAKE) Bit 5

When WAKE equals zero, an idle line wakeup is selected. In the idle line wakeup mode, the SCI receiver is re-enabled by an idle string of at least 10 or 11 (depending on WDS mode) consecutive ones. The transmitter's software must provide this idle string between consecutive messages. The idle string cannot occur within a valid message because each word frame contains a start bit that is a zero.

When WAKE equals one, an address bit wakeup is selected. In the address bit wakeup mode, the SCI receiver is re-enabled when the last (eighth or ninth) data bit received in a character (frame) is one. The ninth data bit is the address bit (R8) in the 11-bit multidrop mode; the eighth data bit is the address bit in the 10-bit asynchronous and 11-bit asynchronous with parity modes. Thus, the received character is an address that has to be processed by all sleeping processors – i.e., each processor has to compare the received character with its own address and decide whether to receive or ignore all following characters. WAKE is cleared by hardware and software reset.

11.2.2.1.5 SCR Receiver Wakeup Enable (RWU) Bit 6

When RWU equals one and the SCI is in an asynchronous mode, the wakeup function is enabled – i.e., the SCI is put to sleep waiting for a reason (defined by the WAKE bit) to wakeup. In the sleeping state, all receive flags, except IDLE, and interrupts are disabled. When the receiver wakes up, this bit is cleared by the wakeup hardware. The programmer may also clear the RWU bit to wake up the receiver.

RWU can be used by the programmer to ignore messages that are for other devices on a multidrop serial network. Wakeup on idle line (WAKE=0) or wakeup on address bit (WAKE=1) must be chosen.

1. When WAKE equals zero and RWU equals one, the receiver will not respond to data on the data line until an idle line is detected.
2. When WAKE equals one and RWU equals one, the receiver will not respond to data on the data line until a data byte with bit 9 equal to one is detected.

When the receiver wakes up, the RWU bit is cleared, and the first byte of data is received. If interrupts are enabled, the CPU will be interrupted, and the interrupt routine will read the message header to determine if the message is intended for this DSP.

1. If the message is for this DSP, the message will be received, and RWU will again be set to one to wait for the next message.
2. If the message is not for this DSP, the DSP will immediately set RWU to one. Setting RWU to one causes the DSP to ignore the remainder of the message and wait for the next message.

RWU is cleared by hardware and software reset. RWU is a don't care in the synchronous mode.

11.2.2.1.6 SCR Wired-OR Mode Select (WOMS) Bit 7

When the WOMS bit is set, the SCI TXD driver is programmed to function as an open-drain output and may be wired together with other TXD pins in an appropriate bus configuration such as a master-slave multidrop configuration. An external pullup resistor is required on the bus. When the WOMS is cleared, the TXD pin uses an active internal pullup. This bit is cleared by hardware and software reset.

11.2.2.1.7 SCR Receiver Enable (RE) Bit 8

When RE is set, the receiver is enabled. When RE is cleared, the receiver is disabled, and data transfer is inhibited to the receive data register (SRX) from the receive shift register. If RE is cleared while a character is being received, the reception of the character will be completed before the receiver is disabled. RE does not inhibit RDRF or receive interrupts. RE is cleared by a hardware and software reset.

11.2.2.1.8 SCR Transmitter Enable (TE) Bit 9

When TE is set, the transmitter is enabled. When TE is cleared, the transmitter will complete transmission of data in the SCI transmit data shift register; then the serial output is forced high (idle). Data present in the SCI transmit data register (STX) will not be transmitted. STX may be written and TDRE will be cleared, but the data will not be transferred into the shift register. TE does not inhibit TDRE or transmit interrupts. TE is cleared by a hardware and software reset.

Setting TE will cause the transmitter to send a preamble of 10 or 11 consecutive ones (depending on WDS). This procedure gives the programmer a convenient way to ensure that the line goes idle before starting a new message. To force this separation of messages by the minimum idle line time, the following sequence is recommended:

1. Write the last byte of the first message to STX.
2. Wait for TDRE to go high, indicating the last byte has been transferred to the transmit shift register.
3. Clear TE and set TE back to one. This queues an idle line preamble to imme-

diately follow the transmission of the last character of the message (including the stop bit).

4. Write the first byte of the second message to STX.

In this sequence, if the first byte of the second message is not transferred to the STX prior to the finish of the preamble transmission, then the transmit data line will simply mark idle until STX is finally written.

11.2.2.1.9 SCR Idle Line Interrupt Enable (ILIE) Bit 10

When ILIE is set, the SCI interrupt occurs when IDLE is set. When ILIE is clear, the IDLE interrupt is disabled. ILIE is cleared by hardware and software reset.

An internal flag, the shift register idle interrupt (SRIINT) flag, is the interrupt request to the interrupt controller. SRIINT is not directly accessible to the user.

When a valid start bit has been received, an idle interrupt will be generated if both IDLE (SCI Status Register bit 3) and ILIE equals one. The idle interrupt acknowledge from the interrupt controller clears this interrupt request. The idle interrupt will not be asserted again until at least one character has been received. The result is as follows:

1. The IDLE bit shows the real status of the receive line at all times.
2. Idle interrupt is generated once for each idle state, no matter how long the idle state lasts.

11.2.2.1.10 SCR SCI Receive Interrupt Enable (RIE) Bit 11

The RIE bit is used to enable the SCI receive data interrupt. If RIE is cleared, receive interrupts are disabled, and the RDRF bit in the SCI status register must be polled to determine if the receive data register is full. If both RIE and RDRF are set, the SCI will request an SCI receive data interrupt from the interrupt controller.

One of two possible receive data interrupts will be requested:

1. Receive without exception will be requested if PE, FE, and OR are all clear (i.e., a normal received character).
2. Receive with exception will be requested if PE, FE, and OR are not all clear (i.e., a received character with an error condition).

RIE is cleared by hardware and software reset.

11.2.2.1.11 SCR SCI Transmit Interrupt Enable (TIE) Bit 12

The TIE bit is used to enable the SCI transmit data interrupt. If TIE is cleared, transmit data interrupts are disabled, and the transmit data register empty (TDRE) bit in the SCI

status register must be polled to determine if the transmit data register is empty. If both TIE and TDRE are set, the SCI will request an SCI transmit data interrupt from the interrupt controller. TIE is cleared by hardware and software reset.

11.2.2.1.12 SCR Timer Interrupt Enable (TMIE) Bit 13

The TMIE bit is used to enable the SCI timer interrupt. If TMIE is set (enabled), the timer interrupt requests will be made to the interrupt controller at the rate set by the SCI clock register. The timer interrupt is automatically cleared by the timer interrupt acknowledge from the interrupt controller. This feature allows DSP programmers to use the SCI baud clock generator as a simple periodic interrupt generator if the SCI is not in use, if external clocks are used for the SCI, or if periodic interrupts are needed at the SCI baud rate. The SCI internal clock is divided by 16 (to match the $1 \times$ SCI baud rate) for timer interrupt generation. This timer does not require that any SCI pins be configured for SCI use to operate. TMIE is cleared by hardware and software reset.

11.2.2.1.13 SCR Reserved (Bit 14)

This unused bit is reserved and should be written with a zero for upward compatibility. It is read as a zero.

11.2.2.1.14 SCR SCI Clock Polarity (SCKP) Bit 15

The clock polarity, sourced or received on the clock pin (SCLK), can be inverted using this bit, eliminating the need for an external inverter. When bit 15 equals zero, the clock polarity is positive; when bit 15 equals one, the clock polarity is negative. In the synchronous mode, positive polarity means that the clock is normally positive and transitions negative during data valid; whereas, negative polarity means that the clock is normally negative and transitions positive during valid data. In the asynchronous mode, positive polarity means that the rising edge of the clock occurs in the center of the period that data is valid; negative polarity means that the falling edge of the clock occurs during the center of the period that data is valid. This bit should be cleared for compatibility with early versions of the DSP56000/DSP56001. SCKP is cleared on hardware and software reset.

11.2.2.2 SCI Status Register (SSR)

The SSR is an 8-bit read-only register used by the DSP CPU to determine the status of the SCI. When the SSR is read onto the internal data bus, the register contents occupy the low-order byte of the data bus and all high-order portions are zero filled. The status bits are described in the following paragraphs.

11.2.2.2.1 SSR Transmitter Empty (TRNE) Bit 0

The TRNE flag is set when both the transmit shift register and data register are empty to indicate that there is no data in the transmitter. When TRNE is set, data written to one of the three STX locations or to the STXA will be transferred to the transmit shift register and be the first data transmitted. TRNE is cleared when TDRE is cleared by writing data into the transmit data register (STX) or the transmit data address register (STXA), or when an idle, preamble, or break is transmitted. The purpose of this bit is to indicate that the transmitter is empty; therefore, the data written to STX or STXA will be transmitted next – i.e., there is not a word in the transmit shift register presently being transmitted. This procedure is useful when initiating the transfer of a message (i.e., a string of characters).

TRNE is set by the hardware, software, SCI individual, and stop reset.

11.2.2.2.2 SSR Transmit Data Register Empty (TDRE) Bit 1

The TDRE bit is set when the SCI transmit data register is empty. When TDRE is set, new data may be written to one of the SCI transmit data registers (STX) or transmit data address register (STXA). TDRE is cleared when the SCI transmit data register is written. TDRE is set by the hardware, software, SCI individual, and stop reset.

In the SCI synchronous mode, when using the internal SCI clock, there is a delay of up to 5.5 serial clock cycles between the time that STX is written until TDRE is set, indicating the data has been transferred from the STX to the transmit shift register. There is a two to four serial clock cycle delay between writing STX and loading the transmit shift register; in addition, TDRE is set in the middle of transmitting the second bit. When using an external serial transmit clock, if the clock stops, the SCI transmitter stops. TDRE will not be set until the middle of the second bit transmitted after the external clock starts. Gating the external clock off after the first bit has been transmitted will delay TDRE indefinitely.

In the SCI asynchronous mode, the TDRE flag is not set immediately after a word is transferred from the STX or STXA to the transmit shift register nor when the word first begins to be shifted out. TDRE is set two cycles of the 16 $\times$ clock after the start bit – i.e., two 16 $\times$ clock cycles into to transmission time of the first data bit.

11.2.2.2.3 SSR Receive Data Register Full (RDRF) Bit 2

The RDRF bit is set when a valid character is transferred to the SCI receive data register from the SCI receive shift register. RDRF is cleared when the SCI receive data register is read or by the hardware, software, SCI individual, and stop reset.

11.2.2.2.4 SSR Idle Line Flag (IDLE) Bit 3

IDLE is set when 10 (or 11) consecutive ones are received. IDLE is cleared by a start-bit detection. The IDLE status bit represents the status of the receive line. The transition of IDLE from zero to one can cause an IDLE interrupt (ILIE). IDLE is cleared by the hardware, software, SCI individual, and stop reset.

11.2.2.2.5 SSR Overrun Error Flag (OR) Bit 4

The OR flag is set when a byte is ready to be transferred from the receive shift register to the receive data register (SRX) that is already full (RDRF=1). The receive shift register data is not transferred to the SRX. The OR flag indicates that character(s) in the receive data stream may have been lost. The only valid data is located in the SRX. OR is cleared when the SCI status register is read, followed by a read of SRX. The OR bit clears the FE and PE bits – i.e., overrun error has higher priority than FE or PE. OR is cleared by the hardware, software, SCI individual, and stop reset.

11.2.2.2.6 SSR Parity Error (PE) Bit 5

In the 11-bit asynchronous modes, the PE bit is set when an incorrect parity bit has been detected in the received character. It is set simultaneously with RDRF for the byte which contains the parity error – i.e., when the received word is transferred to the SRX. If PE is set, it does not inhibit further data transfer into the SRX. PE is cleared when the SCI status register is read, followed by a read of SRX. PE is also cleared by the hardware, software, SCI individual, or stop reset. In the 10-bit asynchronous mode, the 11-bit multidrop mode, and the 8-bit synchronous mode, the PE bit is always cleared since there is no parity bit in these modes. If the byte received causes both parity and overrun errors, the SCI receiver will only recognize the overrun error.

11.2.2.2.7 SSR Framing Error Flag (FE) Bit 6

The FE bit is set in the asynchronous modes when no stop bit is detected in the data string received. FE and RDRE are set simultaneously – i.e., when the received word is transferred to the SRX. However, the FE flag inhibits further transfer of data into the SRX

until it is cleared. FE is cleared when the SCI status register is read followed by reading the SRX. The hardware, software, SCI individual, and stop reset also clear FE. In the 8-bit synchronous mode, FE is always cleared. If the byte received causes both framing and overrun errors, the SCI receiver will only recognize the overrun error.

11.2.2.2.8 SSR Received Bit 8 (R8) Address Bit 7

In the 11-bit asynchronous multidrop mode, the R8 bit is used to indicate whether the received byte is an address or data. R8 is not affected by reading the SRX or status register. The hardware, software, SCI individual, and stop reset clear R8.

11.2.2.3 SCI Clock Control Register (SCCR)

The SCCR is a 16-bit read/write register, which controls the selection of the clock modes and baud rates for the transmit and receive sections of the SCI interface. The control bits are described in the following paragraphs. The SCCR is cleared by hardware reset.

The basic points of the clock generator are as follows:

1. The SCI core always uses a 16 × internal clock in the asynchronous modes and always uses a 2 × internal clock in the synchronous mode. The maximum internal clock available to the SCI peripheral block is the oscillator frequency divided by 4. With a 20-MHz crystal, this gives a maximum data rate of 312.5 Kbps for asynchonous data and 2.5 Mbps for synchronous data. These maximum rates are the same for internally or externally supplied clocks.
2. The 16 × clock is necessary for the asynchronous modes to synchronize the SCI to the incoming data (see Figure 11-12).
3. For the asynchronous modes, the user must provide a 16 × clock if he wishes to use an external baud rate generator (i.e., SCLK input).
4. For the asynchronous modes, the user may select either 1 × or 16 × for the output clock when using internal TX and RX clocks (TCM=0 and RCM=0).

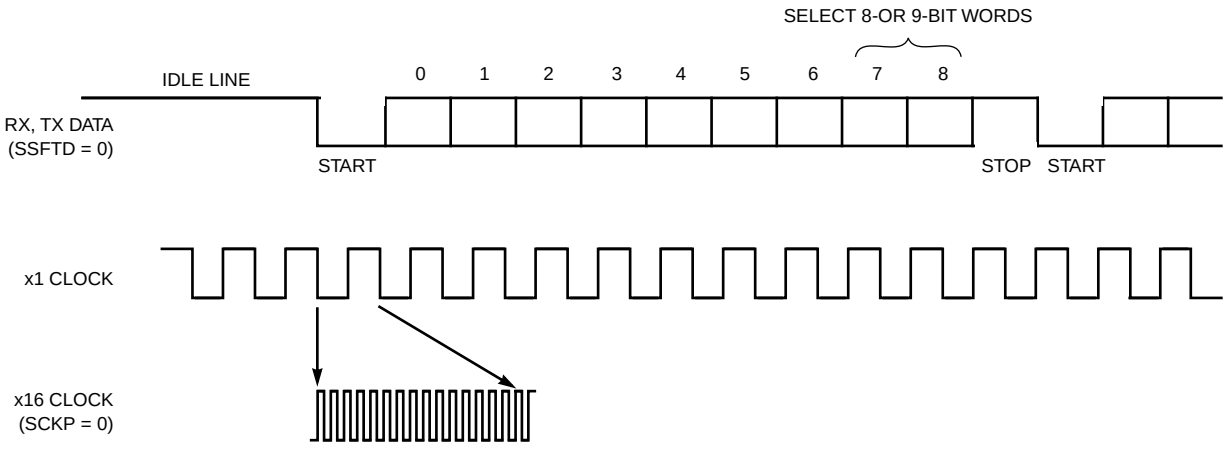


Figure 11-12 16 x Serial Clock

5. The transmit data on the TXD pin changes on the negative edge of the 1 × serial clock and is stable on the positive edge (SCKP=0). For SCKP equals one, the data changes on the positive edge and is stable on the negative edge.
6. The receive data on the RXD pin is sampled on the positive edge (if SCKP=0) or on the negative edge (if SCKP=1) of the 1 × serial clock.
7. For the asynchronous mode, the output clock is continuous.
8. For the synchronous mode, a 1 × clock is used for the output or input baud rate. The maximum 1 × clock is the crystal frequency divided by 8.
9. For the synchronous mode, the clock is gated.
10. For both the asynchronous and synchronous modes, the transmitter and receiver are synchronous with each other.

11.2.2.3.1 SCCR Clock Divider (CD11–CD0) Bits 11–0

The clock divider bits (CD11–CD0) are used to preset a 12-bit counter, which is decremented at the I_{cyc} rate (crystal frequency divided by 2). The counter is not accessible to the user. When the counter reaches zero, it is reloaded from the clock divider bits. Thus, a value of 0000 0000 0000 in CD11–CD0 produces the maximum rate of I_{cyc}, and a value of 0000 0000 0001 produces a rate of $I_{cyc}/2$. The lowest rate available is $I_{cyc}/4096$. Figure 11-13 and Figure 11-36 show the clock dividers. Bits CD11–CD0 are cleared by hardware and software reset.

11.2.2.3.2 SCCR Clock Out Divider (COD) Bit 12

Figure 11-13 and Figure 11-36 show the clock divider circuit. The output divider is controlled by COD and the SCI mode. If the SCI mode is synchronous, the output divider is fixed at divide by 2; if the SCI mode is asynchronous, and

1. If COD equals zero and SCLK is an output (i.e., TCM and RCM=0), the SCI clock is divided by 16 before being output to the SCLK pin; thus, the SCLK output is a 1 × clock.
2. If COD equals one and SCLK is an output, the SCI clock is fed directly out to the SCLK pin; thus, the SCLK output is a 16 × baud clock.

The COD bit is cleared by hardware and software reset.

11.2.2.3.3 SCCR SCI Clock Prescaler (SCP) Bit 13

The SCI SCP bit selects a divide by 1 (SCP=0) or divide by 8 (SCP=1) prescaler for the clock divider. The output of the prescaler is further divided by 2 to form the SCI clock. Hardware and software reset clear SCP. Figure 11-13 and Figure 11-36 show the clock divider diagram.

11.2.2.3.4 SCCR Receive Clock Mode Source Bit (RCM) Bit 14

RCM selects internal or external clock for the receiver (see Figure 11-36). RCM equals zero selects the internal clock; RCM equals one selects the external clock from the SCLK pin. Hardware and software reset clear RCM.

11.2.2.3.5 SCCR Transmit Clock Source Bit (TCM) Bit 15

The TCM bit selects internal or external clock for the transmitter (see Figure 11-36). TCM equals zero selects the internal clock; TCM equals one selects the external clock from the SCLK pin. Hardware and software reset clear TCM.

11.2.2.4 SCI Data Registers

The SCI data registers are divided into two groups: receive and transmit. There are two receive registers – a receive data register (SRX) and a serial-to-parallel receive shift register. There are also two transmit registers – a transmit data register (called either STX or STXA) and a parallel-to-serial transmit shift register.

11.2.2.4.1 SCI Receive Registers

Data words received on the RXD pin are shifted into the SCI receive shift register. When the complete word has been received, the data portion of the word is transferred to the byte-wide SRX. This process converts the serial data to parallel data and provides double buffering. Double buffering provides flexibility to the programmer and increased throughput since the programmer can save the previous word while the current word is being received.

The SRX can be read at three locations: X:$FFF4, X:$FFF5, and X:$FFF6 (see Figure 11-14). When location X:$FFF4 is read, the contents of the SRX are placed in the lower byte of the data bus and the remaining bits on the data bus are written as zeros. Similarly, when X:$FFF5 is read, the contents of SRX are placed in the middle byte of the bus, and when X:$FFF6 is read, the contents of SRX are placed in the high byte with the remaining bits zeroed. Mapping SRX as described allows three bytes to be efficiently packed into one 24-bit word by ORing three data bytes read from the three addresses. The following code fragment requires that R0 initially points to X:$FFF4, register A is initially cleared, and R3 points to a data buffer. The only programming trick is using BCLR to test bit 1 of the packing pointer to see if it is pointing to X:$FFF6 and clearing bit 1 to point to X:$FFF4 if it had been pointing to X:$FFF6. This procedure resets the packing pointer after receiving three bytes.

TCM	RCM	TX Clock	RX Clock	SCLK Pin	Mode
0	0	Internal	Internal	Output	Synchronous/Asynchronous
0	1	Internal	External	Input	Asynchronous Only
1	0	External	Internal	Input	Asynchronous Only
1	1	External	External	Input	Synchronous/Asynchronous

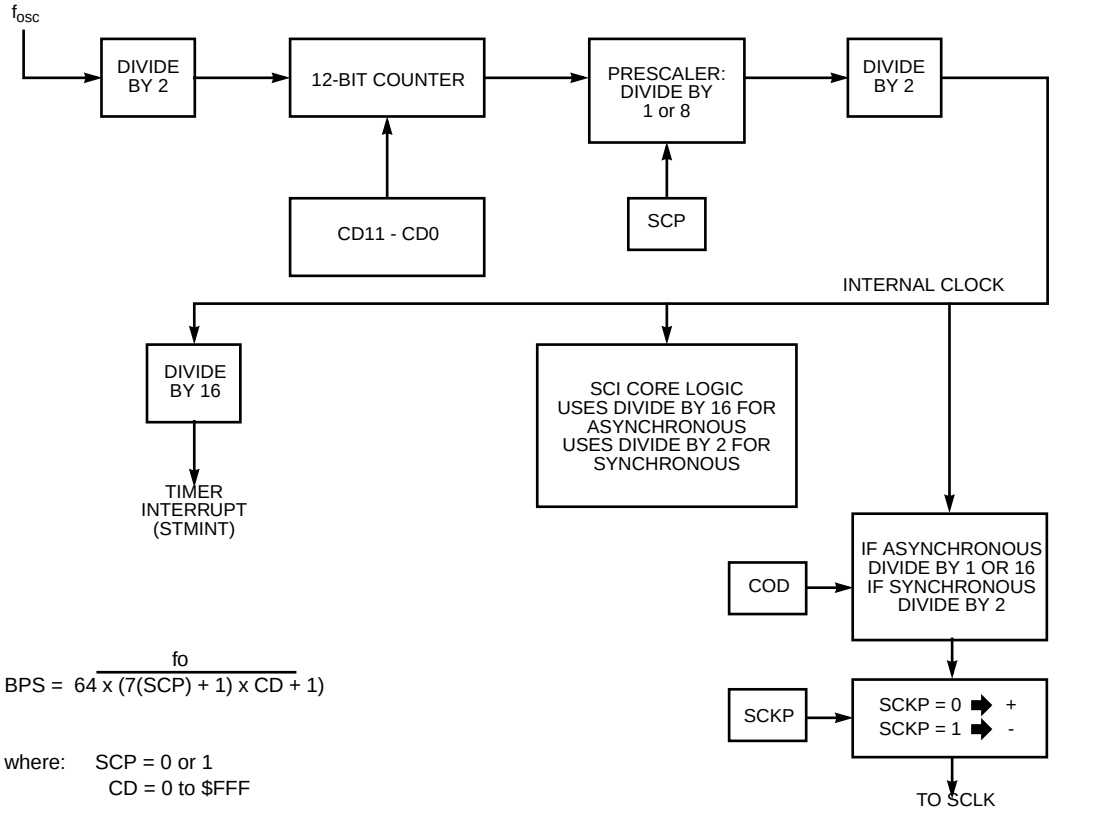
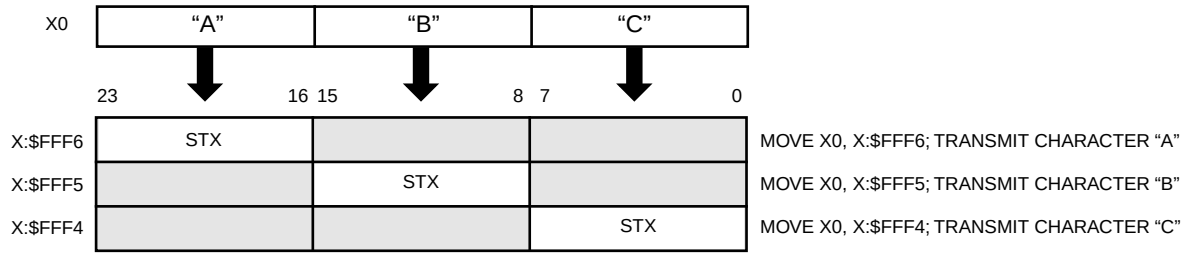


Figure 11-13 SCI Baud Rate Generator

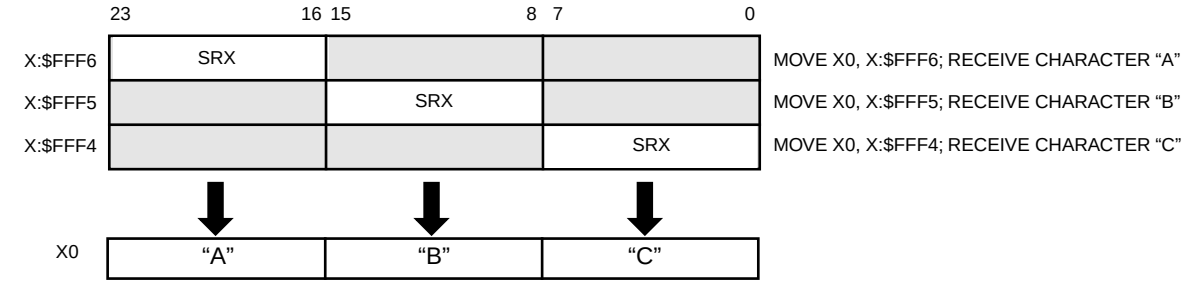
```
        MOVE    X:(R0),X0       ;Copy received data to temporary register
        BCLR    #$1,R0          ;Test for last byte
                                ;reset pointer if it is the last byte
        OR      X0,A            ;Pack the data into register A
        MOVE    (R0)+           ;and increment the packing pointer
        JCS     FLAG            ;Jump to clean up routine if last byte
        RTI                     ;Else return until next byte is received
FLAG    MOVE    A,(R3)+         ;Move the packed data to memory
        CLR     A               ;Prepare A for packing next three bytes
        RTI                     ;Return until the next byte is received
```

The length and format of the serial word is defined by the WDS0, WDS1, and WDS2 control bits in the SCI control register. In the synchronous modes, the start bit, the eight data bits with LSB first, the address/data indicator bit and/or the parity bit, and the stop bit are



NOTE: STX is the same register decoded at three different addresses.

(a) Unpacking



NOTE: SRX is the same register decoded at three different addresses.

(b) Packing

Figure 11-14 Data Packing and Unpacking

received in that order for SSFTD equals zero (see Figure 11-11 (a)). For SSFTD equals one, the data bits are transmitted MSB first (see Figure 11-11(b)). The clock source is defined by the receive clock mode (RCM) select bit in the SCR. In the synchronous mode, the synchronization is provided by gating the clock. In either mode, when a complete word has been clocked in, the contents of the shift register can be transferred to the SRX and the flags; RDRF, FE, PE, and OR are changed appropriately. Because the operation of the SCI receive shift register is transparent to the DSP, the contents of this register are not directly accessible to the programmer.

11.2.2.4.2 SCI Transmit Registers

The transmit data register is one byte-wide register mapped into four addresses: X:$FFF3, X:$FFF4, X:$FFF5, and X:$FFF6. In the asynchronous mode, when data is to be transmitted, X:$FFF4, X:$FFF5, and X:$FFF6 are used, and the register is called STX. When

X:$FFF4 is written, the low byte on the data bus is transferred to the STX; when X:$FFF5 is written, the middle byte is transferred to the STX; and when X:$FFF6 is written, the high byte is transferred to the STX. This structure (see Figure 11-10) makes it easy for the programmer to unpack the bytes in a 24-bit word for transmission. Location X:$FFF3 should be written in the 11-bit asynchronous multidrop mode when the data is an address and it is desired that the ninth bit (the address bit) be set. When X:$FFF3 is written, the transmit data register is called STXA, and data from the low byte on the data bus is stored in STXA. The address data bit will be cleared in the 11-bit asynchronous multidrop mode when any of X:$FFF4, X:$FFF5, or X:$FFF6 is written. When either STX or STXA is written, TDRE is cleared.

The transfer from either STX or STXA to the transmit shift register occurs automatically, but not immediately, when the last bit from the previous word has been shifted out – i.e., the transmit shift register is empty. Like the receiver, the transmitter is double buffered. However, there will be a two to four serial clock cycle delay between when the data is transferred from either STX or STXA to the transmit shift register and when the first bit appears on the TXD pin. (A serial clock cycle is the time required to transmit one data bit). The transmit shift register is not directly addressable, and a dedicated flag for this register does not exist. Because of this fact and the two to four cycle delay, two bytes cannot be written consecutively to STX or STXA without polling. The second byte will overwrite the first byte. The TDRE flag should always be polled prior to writing STX or STXA to prevent overruns unless transmit interrupts have been enabled. Either STX or STXA is usually written as part of the interrupt service routine. Of course, the interrupt will only be generated if TDRE equals one. The transmit shift register is indirectly visible via the TRNE bit in the SSR.

In the synchronous modes, data is clocked synchronously with the transmit clock, which may have either an internal or external source as defined by the TCM bit in the SCCR. The length and format of the serial word is defined by the WDS0, WDS1, and WDS2 control bits in the SCR. In the asynchronous modes, the start bit, the eight data bits (with the LSB first if SSFTD=0 and the MSB first if SSFTD=1), the address/data indicator bit or parity bit, and the stop bit are transmitted in that order (see Figure 11-11).

The data to be transmitted can be written to any one of the three STX addresses. If SCKP equals one and SSHTD equals one, the SCI synchronous mode is equivalent to the SSI operation in the 8-bit data on-demand mode.

11.2.2.5 PREAMBLE, BREAK, AND DATA TRANSMISSION PRIORITY. It is possible that two or three transmission commands are set simultaneously:

1. A preamble (TE was toggled).
2. A break (SBK was set or was toggled).
3. There is data for transmission (TDRE=0).

After the current character transmission, if two or more of these commands are set, the transmitter will execute them in the following priority:

1. Preamble
2. Break
3. Data

11.2.3 Register Contents After Reset

Four different methods of resetting the SCI exist. Hardware or software reset clears the port control register bits, which configure all I/O as general-purpose input. The SCI will remain in the reset state while all SCI pins are programmed as general-purpose I/O (CC2, CC1, and CC0=0); the SCI will become active only when at least one of the SCI I/O pins is programmed as not general-purpose I/O.

During program execution, the CC2, CC1, and CC0 bits may be cleared (individual reset), which will cause the SCI to stop serial activity and enter the reset state. All SCI status bits will be set to their reset state; however, the contents of the interface control register are not affected, allowing the DSP program to reset the SCI separately from the other internal peripherals.

Executing the STOP instruction halts operation of the SCI until the DSP is restarted, causing the SSR to be reset. No other SCI registers are affected by the STOP instruction. Table 11-1 illustrates how each type of reset affects each register in the SCI.

11.2.4 SCI Initialization

The correct way to initialize the SCI is as follows:

1. Hardware or software reset.
2. Program SCI control registers.
3. Configure SCI pins (at least one) as not general-purpose I/O.

Figure 11-15 and Figure 11-16 show how to configure the bits in the SCI registers. Figure 11-15 is the basic initialization procedure showing which registers must be configured. (1) A hardware or software reset should be used to reset the SCI and prevent it from doing anything unexpected while it is being programmed. (2) Both the SCI interface control register and the clock control register must be configured for any operation using the SCI. (3) The pins to be used must then be selected to release the SCI from reset and (4) begin operation. If interrupts are to be used, the pins must be selected, and interrupts must be enabled and unmasked before the SCI will operate. The order does not matter; any one of these three requirements for interrupts can be used to finally enable the SCI. Figure 11-16 shows the meaning of the individual bits in the SCR and SCCR. The figures below do not assume that interrupts will be used; they recommend selecting the appropriate pins to enable the SCI. Programs shown in Figures 11-21, 11-22, 11-29, 11-35, and 11-37 use interrupts and control the SCI by enabling and disabling interrupts. Either method is acceptable.

Table 11-2(a) and Table 11-3(a) provide the settings for common baud rates for the SCI. The asynchronous SCI baud rates show a baud rate error for the fixed oscillator frequency (see Table 11-2(a)(a)). These small-percentage baud rate errors should allow most UARTs to synchronize. The synchronous applications usually require exact frequencies, which require that the crystal frequency be chosen carefully (see Table 11-3(a)(a) and Table 11-3(b)(b)). An alternative to selecting the system clock to accommodate the SCI requirements is to provide an external clock to the SCI.

Table 11-1 SCI Registers after Reset

Register Bit	Bit Mnemonic	Bit Number	Reset Type			
			HW Reset	SW Reset	IR Reset	ST Reset
SCR	SCKP	15	0	0	–	–
	TMIE	13	0	0	–	–
	TIE	12	0	0	–	–
	RIE	11	0	0	–	–
	ILIE	10	0	0	–	–
	TE	9	0	0	–	–
	RE	8	0	0	–	–
	WOMS	7	0	0	–	–
	RWU	6	0	0	–	–
	WAKE	5	0	0	–	–
	SBK	4	0	0	–	–
	SSFTD	3	0	0	–	–
	WDS (2–0)	2–0	0	0	–	–
SSR	R8	7	0	0	0	0
	FE	6	0	0	0	0
	PE	5	0	0	0	0
	OR	4	0	0	0	0
	IDLE	3	0	0	0	0
	RDRF	2	0	0	0	0
	TDRE	1	1	1	1	1
	TRNE	0	1	1	1	1
SCCR	TCM	15	0	0	–	–
	RCM	14	0	0	–	–
	SCP	13	0	0	–	–
	COD	12	0	0	–	–
	CD (11–0)	11–0	0	0	–	–
SRX	SRX (23–0)	23–16, 15–8, 7–0	–	–	–	–
STX	STX (23–0)	23–0	–	–	–	–
SRSH	SRS (8–0)	8–0	–	–	–	–
STSH	STS (8–0)	8–0	–	–	–	–

NOTES:
SRSH – SCI receive shift register
STSH – SCI transmit shift register
HW – Hardware reset is caused by asserting the external RESET pin.
SW – Software reset is caused by executing the RESET instruction.
IR – Individual reset is caused by clearing PCC (bits 0–2) (configured for general-purpose I/O).
ST – Stop reset is caused by executing the STOP instruction.
1 – The bit is set during the xx reset.
0 – The bit is cleared during the xx reset.

11.2.5 SCI Exceptions

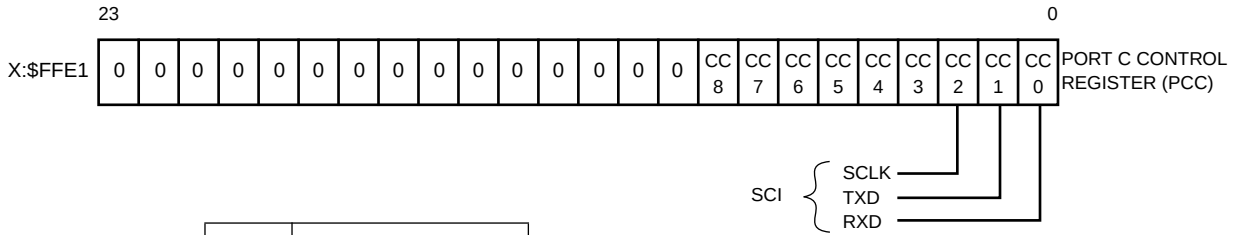
The SCI can cause five different exceptions in the DSP (see Figure 11-17). These exceptions are as follows:

1. SCI Receive Data – caused by receive data register full with no receive error conditions existing. This error-free interrupt may use a fast interrupt service routine for minimum overhead. This interrupt is enabled by SCR bit 11 (RIE).
2. SCI Receive Data with Exception Status – caused by receive data register full with a receiver error (parity, framing, or overrun error). The SCI status register must be read to clear the receiver error flag. A long interrupt service routine should be used to handle the error condition. This interrupt is enabled by SCR bit 11 (RIE).
3. SCI Transmit Data – caused by transmit data register empty. This error-free interrupt may use a fast interrupt service routine for minimum overhead. This interrupt is enabled by SCR bit 12 (TIE).
4. SCI Idle Line – caused by the receive line entering the idle state (10 or 11 bits of ones). This interrupt is latched and then automatically reset when the interrupt is accepted. This interrupt is enabled by SCR bit 10 (ILIE).
5. SCI Timer – caused by the baud rate counter underflowing. This interrupt is automatically reset when the interrupt is accepted. This interrupt is enabled by SCR bit 13 (TMIE).

1. PERFORM HARDWARE OR SOFTWARE RESET.

2. PROGRAM SCI CONTROL REGISTERS:
a) SCI INTERFACE CONTROL REGISTER — X:$FFF0
b) SCI CLOCK CONTROL REGISTER — X:$FFF2

3. CONFIGURE AT LEAST ONE PORT C CONTROL BIT AS SCI.



CCx	Function
0	Parallel I/O
1	Serial Interface

4. SCI IS NOW ACTIVE.

Figure 11-15 SCI Initialization Procedure

Table 11-2(a) Asynchronous SCI Baud Rates for a 20.48-MHz Crystal

Baud Rate(BPS)	SCP Bit	Divider Bits (CD0–CD11)	Baud Rate Error, Percent
320.0K	0	$000	0
56.0K	0	$005	4.762
38.4K	0	$007	4.167
19.2K	0	$010	1.961
9600	0	$020	1.010
8000	0	$027	0
4800	0	$042	0.498
2400	0	$084	0.251
1200	1	$020	1.010
600	1	$042	0.498
300	1	$084	0.251

BPS= $f_0 \div (64 \times (7(SCP) + 1) \times (CD + 1))$; f_0=20.48 MHz
SCP=0 or 1
CD=0 to $FFF

Table 11-2(b) Frequencies for Exact Asynchronous SCI Baud Rates

Baud Rate (BPS)	SCP Bit	Divider Bits (CD0–CD11)	Crystal Frequency
9600	0	$021	20,500,000
4800	0	$042	20,275,200
2400	0	$084	20,275,200
1200	0	$108	20,275,200
300	0	$420	20,275,200
9600	1	$004	19,660,800
4800	1	$008	19,660,800
2400	1	$010	19,660,800
1200	1	$020	19,660,800
300	1	$080	19,660,800

f_0=BPS $\times$ 64 $\times$ (7(SCP) + 1) $\times$ (CD + 1)
SCP=0 or 1
CD=0 to $FFF

Table 11-3(a) Synchronous SCI Baud Rates for a 20.48-MHz Crystal

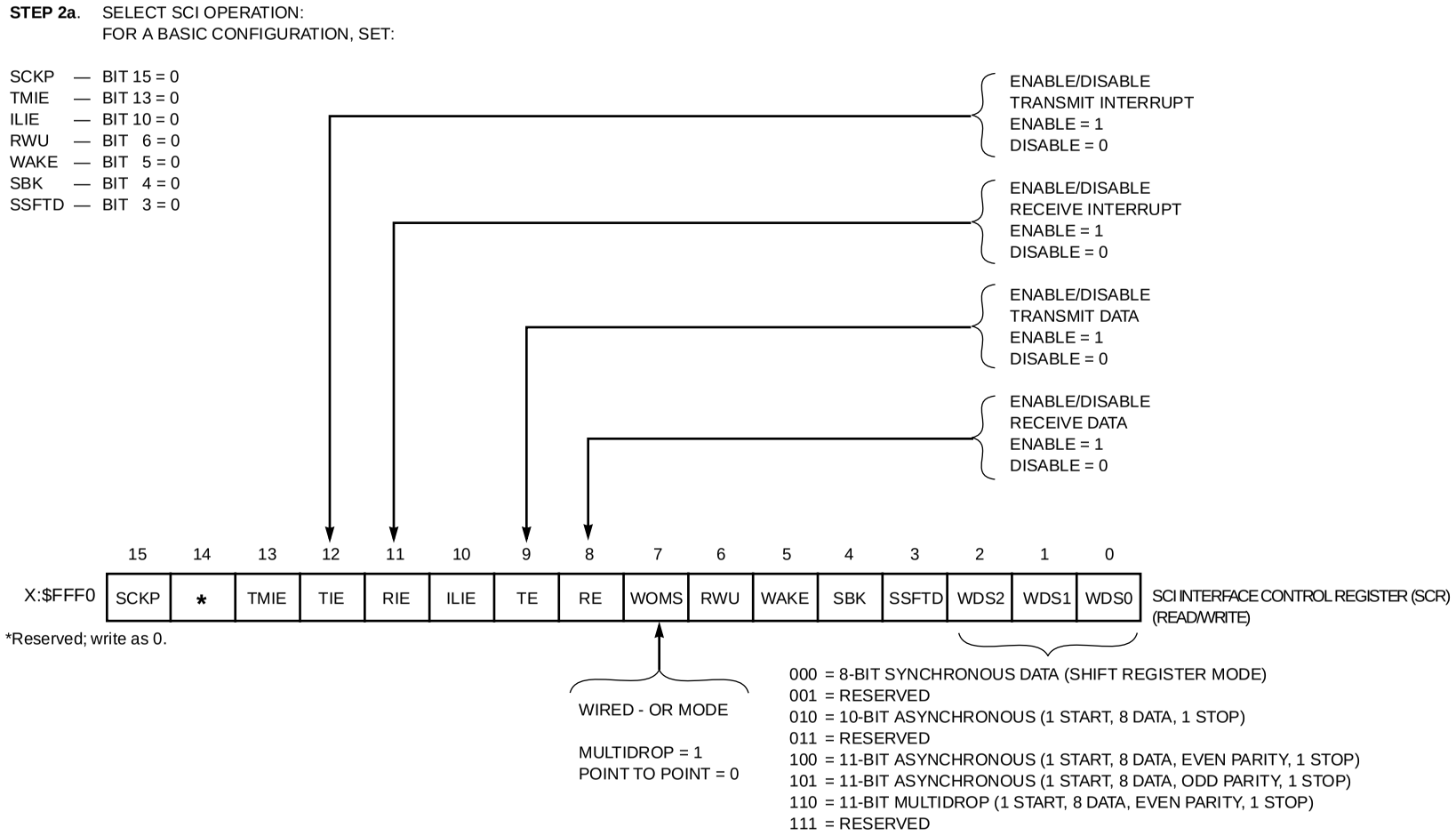
Baud Rate (BPS)	SCP Bit	Divider Bits (CD0–CD11)	Baud Rate Error, Percent
2.56M	0	$000	0
128K	0	$014	0
64K	0	$027	0
56K	0	$02E	0.621
32K	0	$04F	0
16K	0	$09F	0
8000	0	$140	0
4000	0	$27F	0
2000	0	$4FF	0
1000	0	$9FF	0

BPS= $f_0 \div (64 \times (7(SCP) + 1) \times (CD + 1))$; f_0=20.48 MHz
SCP=0 or 1
CD=0 to $FFF

Table 11-3(b) Frequencies for Exact Synchronous SCI Baud Rates

Baud Rate (BPS)	SCP Bit	Divider Bits (CD0–CD11)	Baud Rate Error, MHz
2.048M	0	$000	16.384
1.544M	0	$001	24.576
1.536M	0	$001	24.704

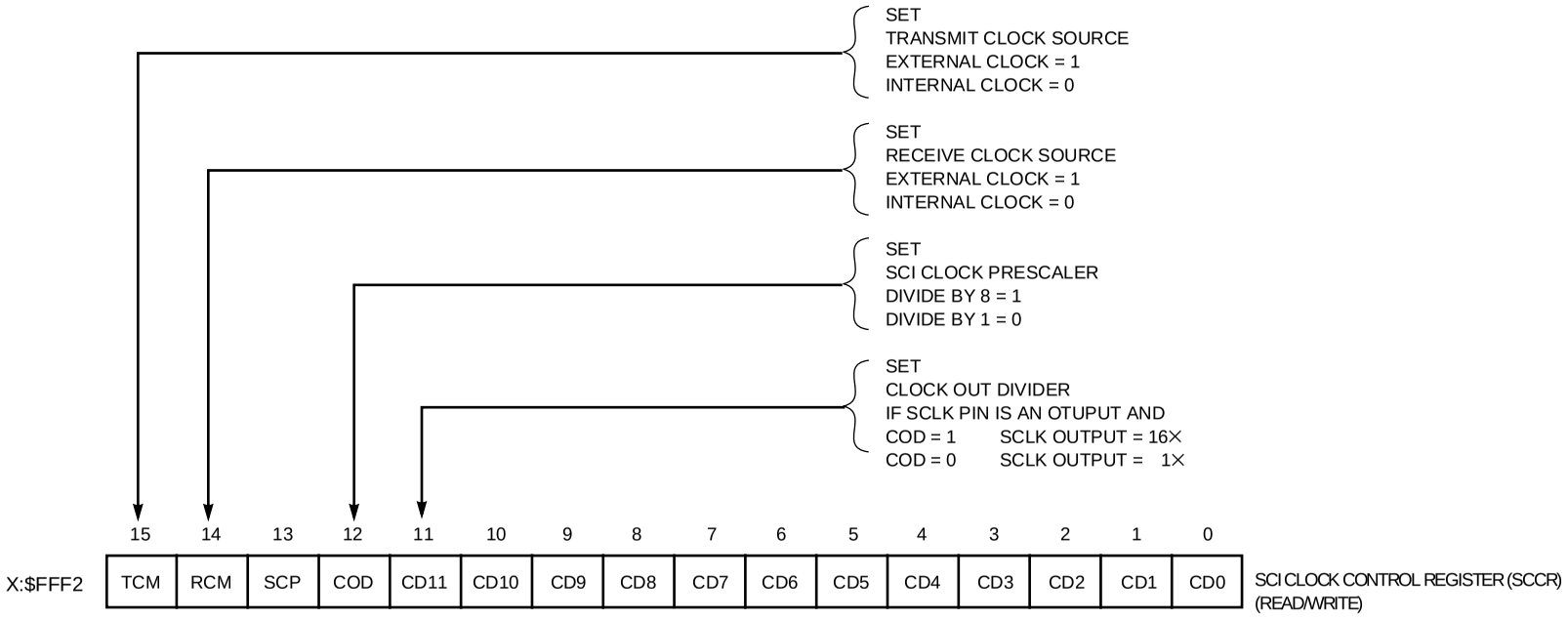
f_0=BPS $\times$ 64 $\times$ (7(SCP) + 1) $\times$ (CD + 1)
SCP=0 or 1
CD=0 to $FFF



Step 2a

Figure 11-16 SCI General Initialization Detail – Step 2 (Sheet 1 of 2)

STEP 2b. SELECT CLOCK AND DATA RATE:
SET THE CLOCK DIVIDER BITS (CD0 - CD11) ACCORDING TO TABLES 11 - 2 OR 11 - 3.
SET THE SCI CLOCK PRESCALER BIT (SCP, BIT 13) ACCORDING TO TABLES 11 - 2 OR 11 - 3.



Step 2b

Figure 11-16 SCI General Initialization Detail – Step 2 (Sheet 2 of 2)

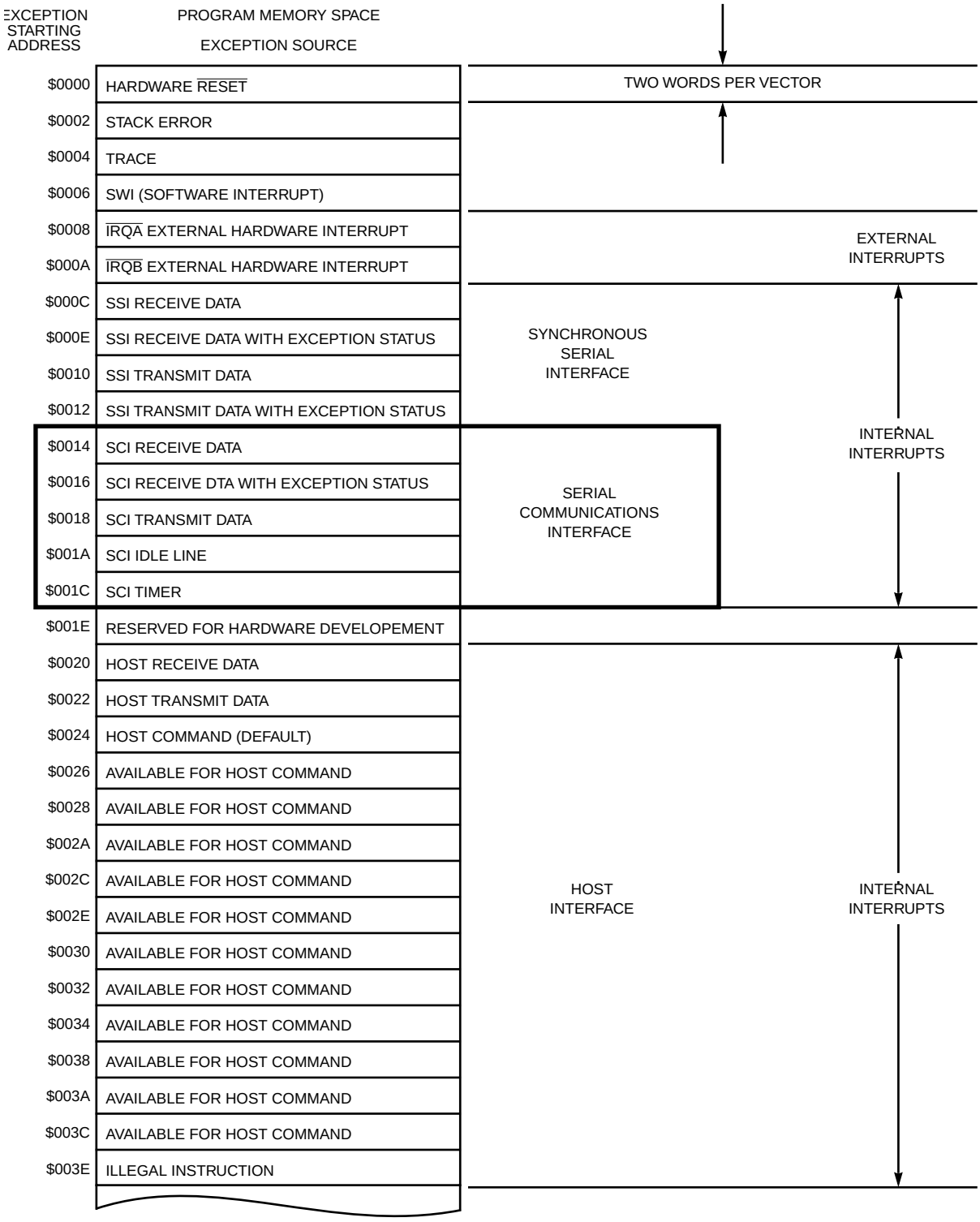


Figure 11-17 SCI Exception Vector Locations

11.2.6 Synchronous Data

The synchronous mode (WDS=0, shift register mode) is designed to implement serial-to-parallel and parallel-to-serial conversions. This mode will directly interface to 8051/8096 synchronous (mode 0) buses as both a controller (master) or a peripheral (slave) and is compatible with the SSI mode if SCKP equals one. In synchronous mode, the clock is always common to the transmit and receive shift registers.

As a controller (synchronous master) shown in Figure 11-18, the DSP outputs a clock on the SCLK pin when data is present in the transmit shift register (a gated clock mode). The master mode is selected by choosing internal transmit and receive clocks (setting TCM and RCM=0). The example shows a 74HC165 parallel-to-serial shift register and 74HC164 serial-to-parallel shift register being used to convert eight bits of serial I/O to eight bits of parallel I/O. The load pulse latches eight bits into the 74HC165 and then SCLK shifts the RXD data into the SCI (these data bits are sample bits 0-7 in the timing diagram). At the same time, TXD shifts data out (B0-B7) to the 74HC164. When using the internal clock, data is transmitted when the transmit shift register is full. Data is valid on both edges of the output clock, which is compatible with an 8051 microprocessor. Received data is sampled in the middle of the clock low time if SCKP equals zero or in the middle of the clock high time if SCKP equals one. There is a window during which STX must be written with the next byte to be transmitted to prevent a gap between words. This window is from the time TDRE goes high halfway into transmission of bit 1 until the middle of bit 6 (see Figure 11-20(a)).

As a peripheral (synchronous slave) shown in Figure 11-19, the DSP accepts an input clock from the SCLK pin. If SCKP equals zero, data is clocked in on the rising edge of SCLK, and data is clocked out on the falling edge of SCLK. If SCKP equals one, data is clocked in on the falling edge of SCLK, and data is clocked out on the rising edge of SCLK. The slave mode is selected by choosing external transmit and receive clocks (TCM and RCM=1). Since there is no frame signal, if a clock is missed due to noise or any other reason, the receiver will lose synchronization with the data without any error signal being generated. Detecting an error of this type can be done with an error detecting protocol or with external circuitry such as a watchdog timer. The simplest way to recover synchronization is to reset the SCI.

The timing diagram in Figure 11-19 shows transmit data in the normal driven mode. Bit B7 is essentially one-half SCI clock long ($T_{SCI}/2 + 1.5\ T_{EXTAL}$) The last data bit is truncated so that the pin is guaranteed to go to its reset state before the start of the next data word, thereby delimiting data words. The 1.5 crystal clock cycles provide sufficient hold time to satisfy most external logic requirements. The example diagram requires that the WOMS bit be set in the SCR to wired-OR RXD and TXD, which causes TXD to be three-stated when not transmitting. Collisions (two devices transmitting simultaneously) must be avoided with this circuit by using a protocol such as alternating transmit and receive

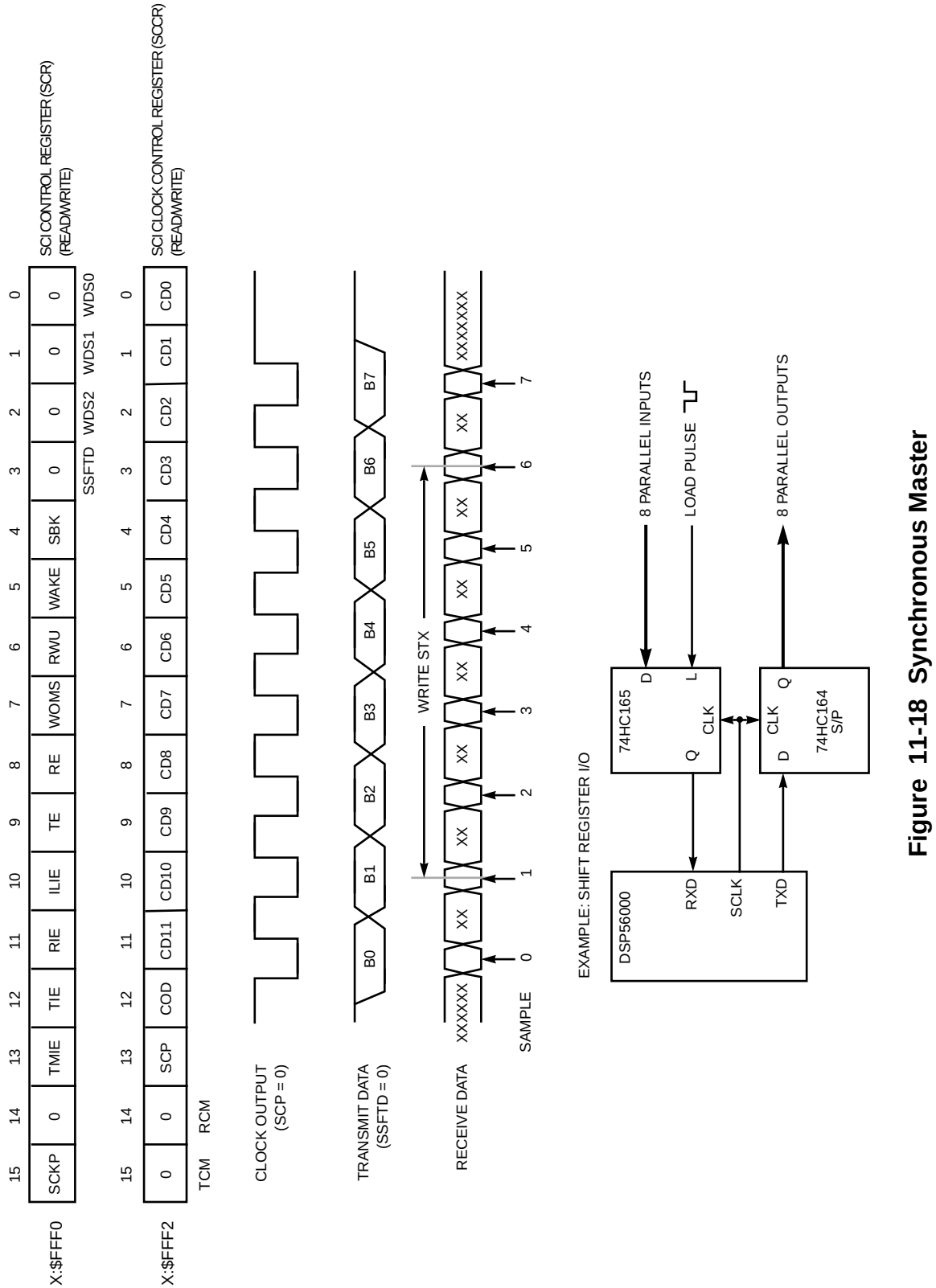


Figure 11-18 Synchronous Master

periods. In the example, the 8051 is the master device because it controls the clock.

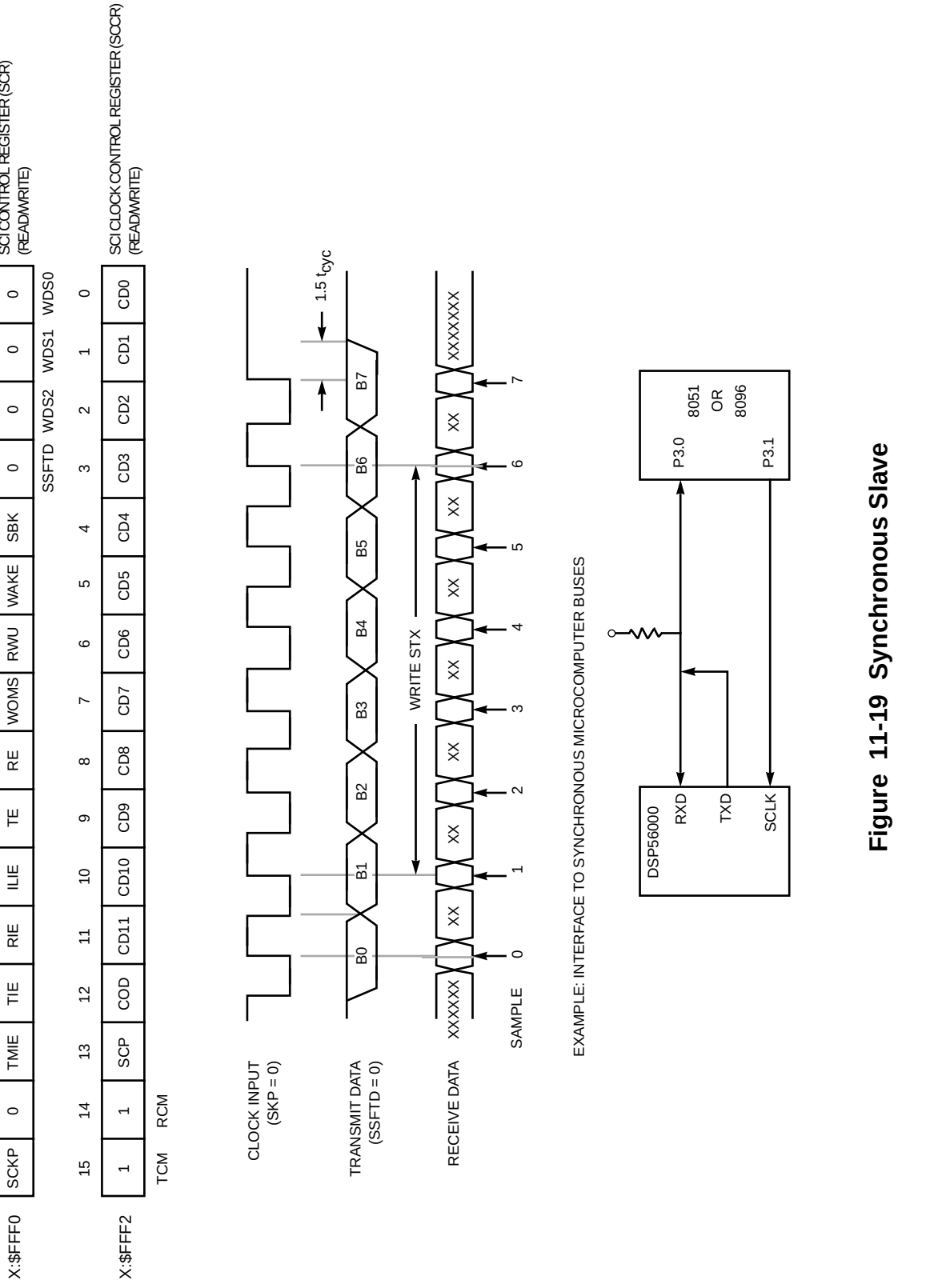


Figure 11-19 Synchronous Slave

There is a window during which STX must be written with the next byte to be transmitted

to prevent the current word from being retransmitted. This window is from the time TDRE goes high, which is halfway into the transmission of bit 1 until the middle of bit 6 (see Figure 11-20(b)). Of course, this assumes the clock remains continuous – i.e., there is a second word. If the clock stops, the SCI stops.

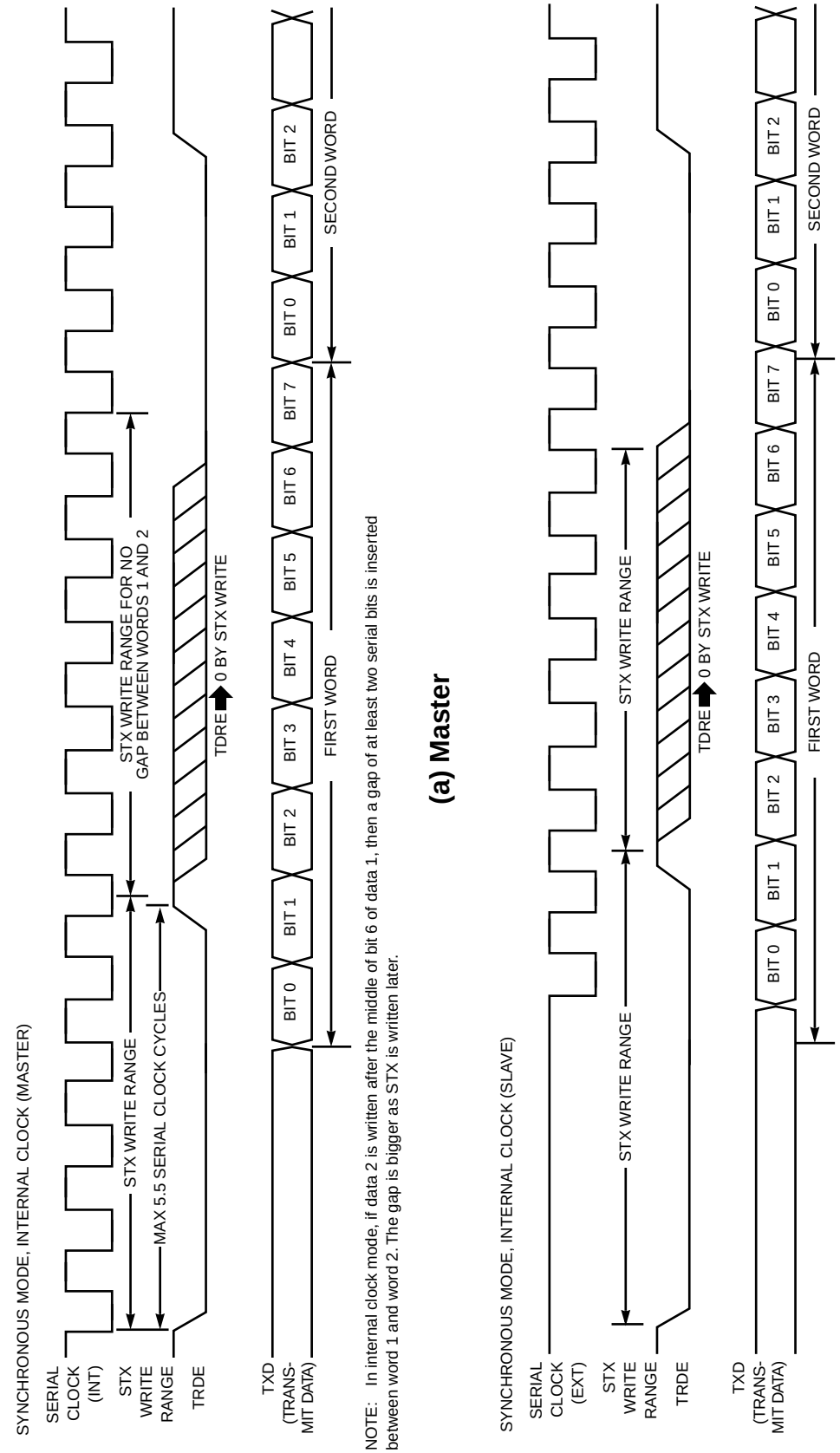
The DSP is initially configured according to the protocol to either receive data or transmit data. If the protocol determines that the next data transfer will be a DSP transmit, the DSP will configure the SCI for transmit and load STX (or STXA). When the master starts SCLK, data will be ready and waiting. If the protocol determines that the next data transfer will be a DSP receive, the DSP will configure the SCI for receive and will either poll the SCI or enable interrupts. This methodology allows multiple slave processors to use the same data line. Selection of individual slave processors can be under protocol control or by multiplexing SCLK.

Note: TCM=0, RCM=1 and TCM=1,RCM=0 are not allowed in the synchronous mode. The results are undefined.

The assembly program shown in Figure 11-21 uses the SCI synchronous mode to transmit only the low byte of the Y data ROM contents. The program sets the reset vector to run the program after a hardware reset, puts the MOVEP instruction at the SCI transmit interrupt vector location, sets the memory wait states to zero, and configures the memory pointers, operating mode register, and the IPR. The SCI is then configured and the interrupts are unmasked, which starts the data transfer. The jump-to-self instruction (LAB0 JMP LAB0) is used to wait while interrupts transfer the data.

```
        ORG     P:0                 ;Reset vector
        JMP     $40                 ;

        ORG     P:$18               ;SCI transmit interrupt vector
        MOVEP   Y:(R0)+,X:$FFF4     ;Transmit low byte of data

        ORG     P:$40
        MOVEP   #0,X:$FFFE          ;Clear BCR
        MOVE    #$100,R0            ;Data ROM start address
        MOVE    #$FF,M0             ;Size of data ROM - Wraps around at $200
        MOVEC   #6,OMR              ;Change operating mode to enable data ROM
        MOVEP   #$C000,X:$FFFF      ;Interrupt priority register
        MOVEP   #$1200,X:$FFF0      ;8-bit synchronous mode
        MOVEP   #7,X:$FFE1          ;Port C control register – enable SCI
        MOVEC   #0,SR               ;Unmask interrupts
LAB0    JMP     LAB0                ;Wait in loop for interrupts
```

Figure 11-21 SCI Synchronous Transmit



NOTE: In internal clock mode, if data 2 is written after the middle of bit 6 of data 1, then a gap of at least two serial bits is inserted between word 1 and word 2. The gap is bigger as STX is written later.

NOTE: In external clock mode, if data 2 is written after the middle of bit 6 of data 1, then the previous data is retransmitted and data 2 is transmitted after the retransmission of data 1.

Figure 11-20 Synchronous Timing

The program shown in Figure 11-22 is the program for receiving data from the program

presented in Figure 11-21. The program sets the reset vector to run the program after hardware reset, puts the MOVEP instruction to store the data in a circular buffer starting at $100 at the SCI receive interrupt vector location, puts another MOVEP instruction at the SCI receive interrupt vector location, sets the memory wait states to zero, and configures the memory pointers and IPR. The SCI is then configured and the interrupts are unmasked, which starts the data transfer. The jump-to-self instruction (LAB0 JMP LAB0) is used to wait while interrupts transfer the data.

```
        ORG     P:0                 ;Reset vector
        JMP     $40                 ;

        ORG     P:$14               ;SCI receive data vector
        MOVEP   X:$FFF4,Y:(R0)+     ;Receive low byte of data
        NOP                         ;Fast interrupt response

        MOVEP   X:$FFF1,X0          ;Receive with exception. Read status register
        MOVEP   X:$FFF4,Y:(R0)+     ;Receive low byte of data

        ORG     P:$40
        MOVEP   #0,X:$FFFE          ;Clear BCR
        MOVE    #$100,R0            ;Data ROM start address
        MOVE    #$FF,M0             ; Size of data ROM – wraps around at $200
        MOVEP   #$C000,X:$FFFF      ;Interrupt priority register
        MOVEP   #$900,X:$FFF0       ; 8-bit synchronous mode receive only
        MOVEP   #$C000,X:$FFF2      ;Clock control register external clock
        MOVEP   #7,X:$FFE1          ;Port C control register – enable SCI
        MOVEC   #0,SR               ;Unmask interrupts
LAB0    JMP     LAB0                ;Wait in loop for interrupts
```

Figure 11-22 SCI Synchronous Receive

11.2.7 Asynchronous Data

Asynchronous data uses a data format with embedded word sync, which allows an unsynchronized data clock to be synchronized with the word if the clock rate and number of bits per word is known. Thus, the clock can be generated by the receiver rather than requiring a separate clock signal. The transmitter and receiver both use an internal clock that is 16 × the data rate to allow the SCI to synchronize the data. The data format requires that each data byte have an additional start bit and stop bit. In addition, two of the word formats have a parity bit. The multidrop mode used when SCIs are on a common bus has an additional data type bit. The SCI can operate in full-duplex or half-duplex modes since the transmitter and receiver are independent. The SCI transmitter and receiver can use either the internal clock (TCM=0 and/or RCM=0) or an external clock (TCM=1 and/or RCM=1) or a combination. If a combination is used, the transmitter and

receiver can run at different data rates.

11.2.7.1 Asynchronous Data Reception

Figure 11-23 illustrates initializing the SCI data receiver for asynchronous data. The first step (1) resets the SCI to prevent the SCI from transmitting or receiving data. Step two (2) selects the desired operation by programming the SCR. As a minimum, the word format (WDS2, WDS1, and WDS0) must be selected, and (3) the receiver must be enabled (RE=1). If (4) interrupts are to be used, set RIE equals one. Use Tables 11-2 and 11-3 to set (5) the baud rate (SCP and CD0–CD11 in the SCCR). Once the SCI is completely configured, it is enabled by (6) setting the RXD bit in the PCC.

The receiver is continually sampling RDX at the 16 × clock rate to find the idle-start-bit transition edge. When that edge is detected (1) the following eight or nine bits, depending on the mode, are clocked into the receive shift register (see Figure 11-24). Once a complete byte is received, (2) the character is latched into the SRX, and RDRF is set as well as the error flags, OR, PE, and FE. If (3) interrupts are enabled, an interrupt is generated. The interrupt service routine, which can be a fast interrupt or a long interrupt, (4) reads the received character. Reading the SRX (5) automatically clears RDFR in the SSR and makes the SRX ready to receive another byte.

If (1) an FE, PE, or OR occurs while receiving data (see Figure 11-25), (2) RDRF is set because a character has been received; FE, PE, or OR is set in the SSR to indicate that an error was detected. Either (3) the SSR can be polled by software to look for errors, or (4) interrupts can be used to execute an interrupt service routine. This interrupt is different from the normal receive interrupt and is caused only by receive errors. The long interrupt service routine should (5) read the SSR to determine what error was detected and then (6) read the SRX to clear RDRF and all three error flags.

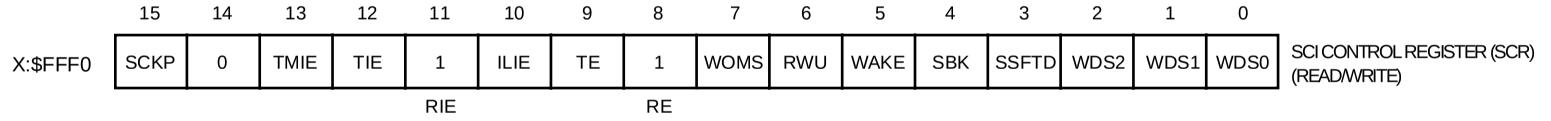
11.2.7.2 Asynchronous Data Transmission

Figure 11-26 illustrates initializing the SCI data transmitter for asynchronous data. The first step (1) resets the SCI to prevent the SCI from transmitting or receiving data. Step two (2) selects the desired operation by programming the SCR. As a minimum, the word format (WDS2, WDS1, and WDS0) must be selected, and (3) the transmitter must be enabled (TE=1). If (4) interrupts are to be used, set TIE equals one. Use Tables 11-2 and 11-3 to set (5) the baud rate (SCP and CD0–CD11 in the SCCR). Once the SCI is completely configured, it can be enabled by (6) setting the TXD bit in the PCC. Transmission begins with (7) a preamble of ones.

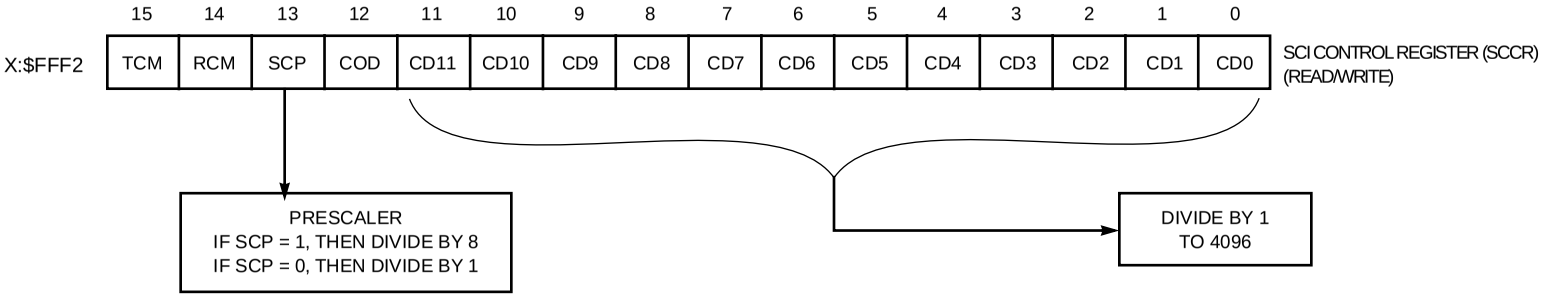
If polling is used to transmit data (see Figure 11-27), the polling routine can look at either TDRE or TRNE to determine when to load another byte into STX. If TDRE is used (1), one byte may be loaded into STX. If TRNE is used (2), two bytes may be loaded into STX if enough time is allowed for the first byte to begin transmission (see 11.2.2.4.2 SCI

Transmit Registers). If interrupts are used (3), then an interrupt is generated when STX

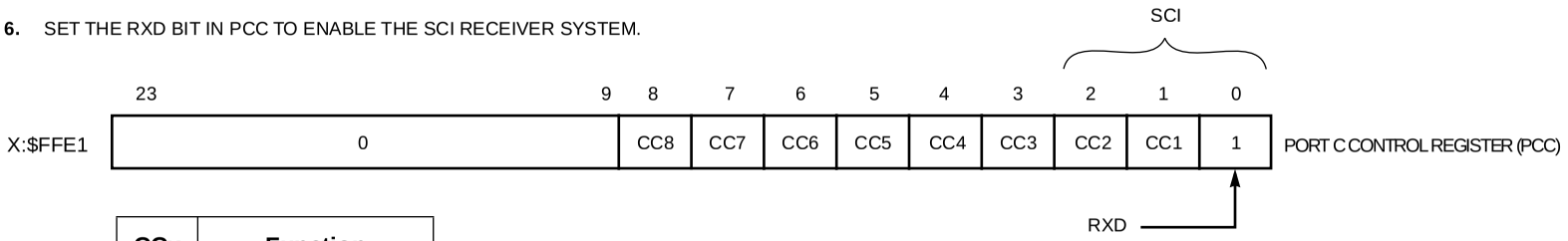
1. HARDWARE OR SOFTWARE RESET
2. PROGRAM SCR WITH DESIRED MODE AND FEATURES.
3. TURN ON RECEIVER (RE = 1).
4. OPTIONALLY ENABLE RECEIVER INTERRUPTS (RIE = 1).



5. SET THE BAUD RATE BY PROGRAMMING THE SCCR.



6. SET THE RXD BIT IN PCC TO ENABLE THE SCI RECEIVER SYSTEM.



CCx	Function
0	Parallel I/O
1	Serial Interface

NOTE: If RE is cleared while a valid character is being received, the reception of the character will be completed before the receiver is disabled.

Figure 11-23 Asynchronous SCI Receiver

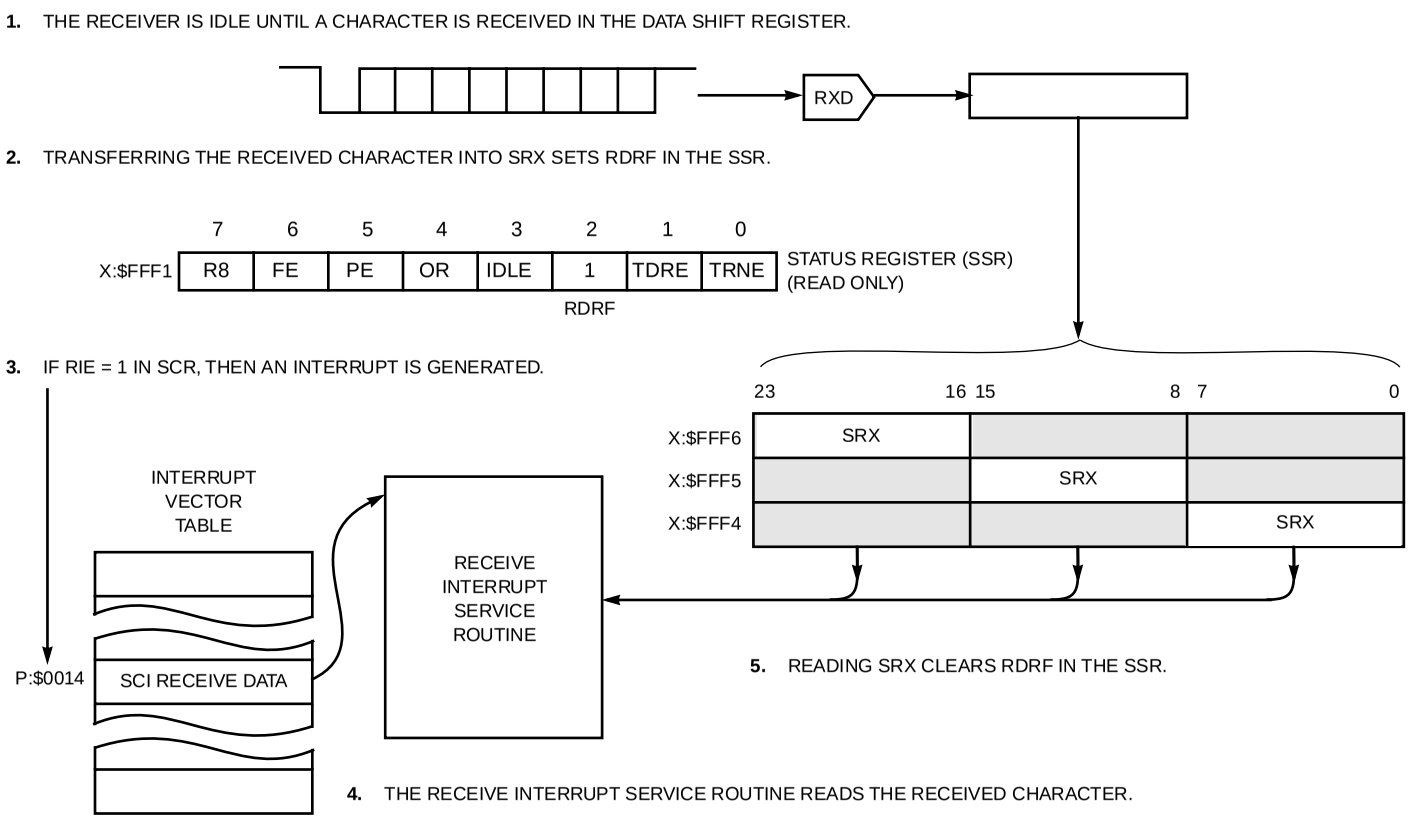


Figure 11-24 SCI Character Reception

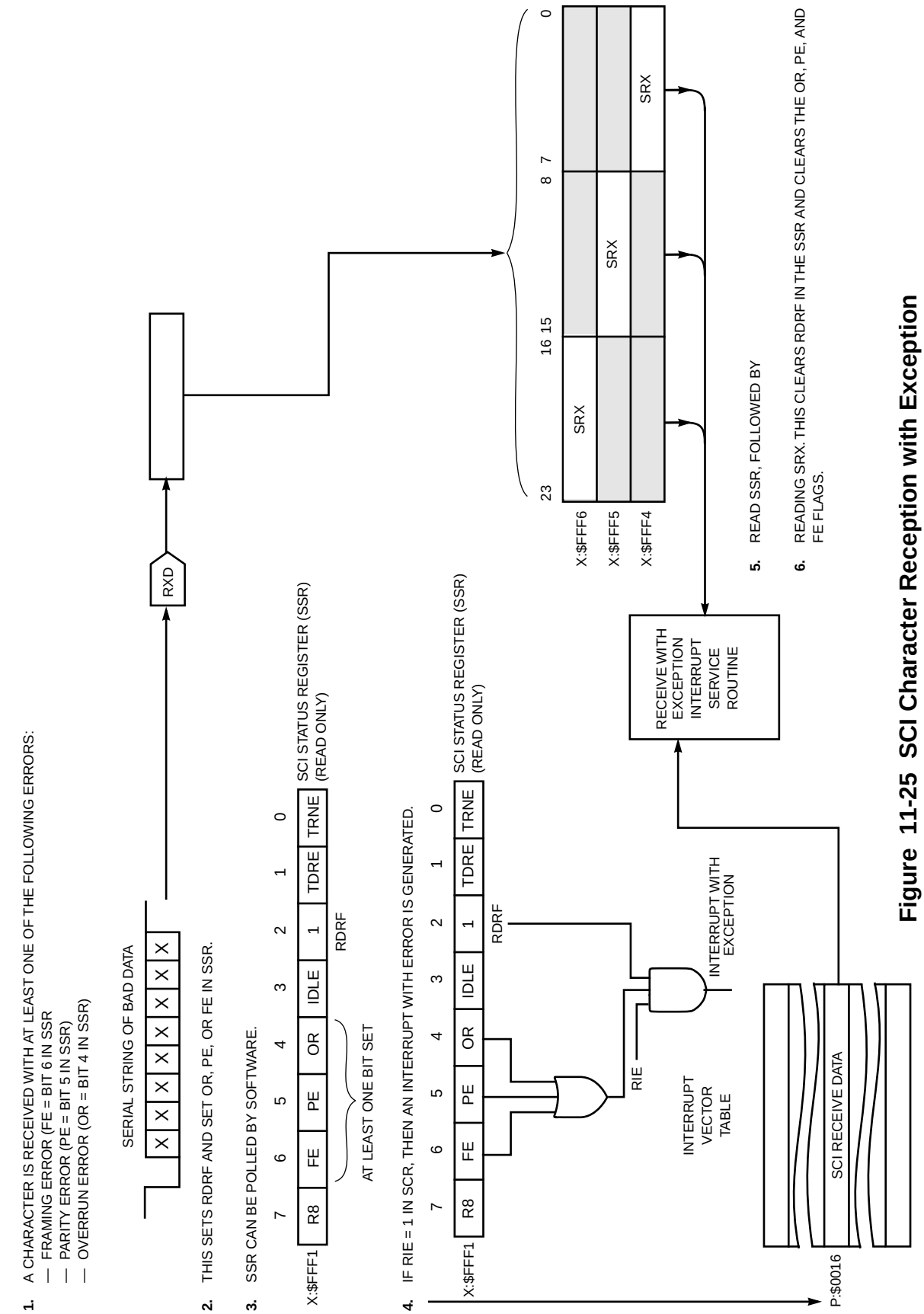
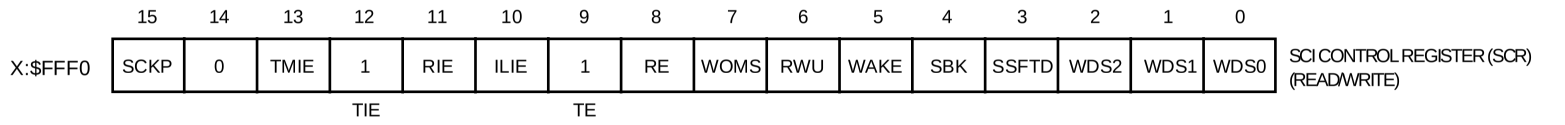


Figure 11-25 SCI Character Reception with Exception

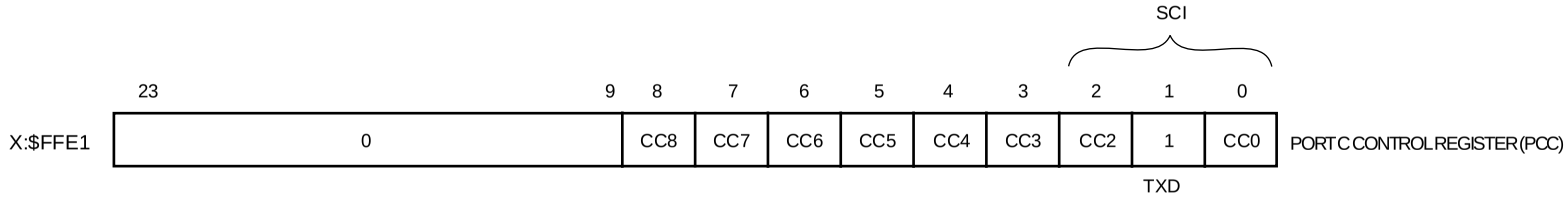
is empty. The interrupt routine, which can be a fast interrupt or a long interrupt, writes (4)

one byte into STX. If multidrop mode is being used and this byte is an address, STXA

1. HARDWARE OR SOFTWARE RESET
2. PROGRAM SCR WITH DESIRED MODE AND FEATURES.
3. TURN ON TRANSMITTER (TE = 1).
4. OPTIONALLY ENABLE TRANSMITTER INTERRUPTS (TIE = 1).



5. SET THESCI CLOCK PRESCALER BIT AND THE CLOCK DIVIDER BITS IN THE SCCR.
6. SET THE TXD BIT IN PCC TO ENABLE THE SCI TRANSMITTER SYSTEM.



CCx	Function
0	Parallel I/O
1	Serial Interface

7. THE TRANSMITTER WILL FIRST BROADCAST A PREAMBLE OF ONES BEFORE BEGINNING DATA TRANSMISSION:
   10 ONES WILL BE TRANSMITTED FOR THE 10-BIT ASYNCHRONOUS MODE.
   11 ONES WILL BE TRANSMITTED FOR THE 11-BIT ASYNCHRONOUS MODE.

NOTE: If TE is cleared while transmitting a character, the transmission of the character will be completed before the transmitter is disabled.

Figure 11-26 Asynchronous SCI Transmitter Initialization

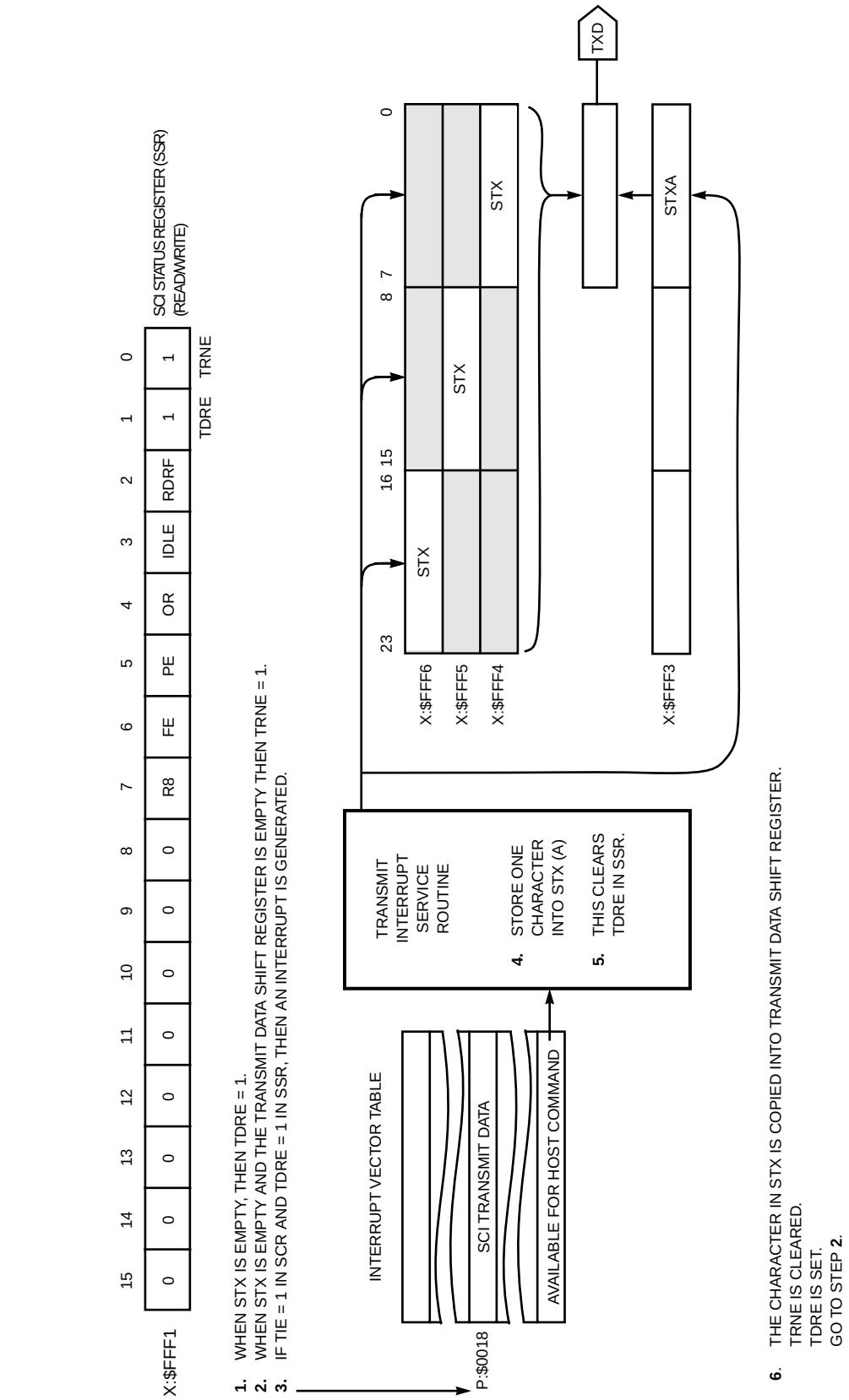


Figure 11-27 Asynchronous SCI Character Transmission

should be used instead of STX. Writing STX or STXA (5) clears TDRE in the SSR. When

the transmit data shift register is empty (6), the byte in STX (or STXA) is latched into the transmit data shift register, TRNE is cleared, and TDRE is set.

There is a provision to send a break or preamble. A break (space) consists of a period of zeros with no start or stop bits that is as long or longer than a character frame. A preamble (mark) is an inverted break. A preamble of 10 or 11 ones (depending on the word length selected by WDS2, WDS1, and WDS0) can be sent with the following procedure (see Figure 11-28). (1) Write the last byte to STX and (2) wait for TDRE equals one. This is the byte that will be transmitted immediately before the preamble. (3) Clear TE and then again set it to one. Momentarily clearing TE causes the output to go high for one character frame. If TE remains cleared for a longer period, the output will remain high for an even number of character frames until TE is set. (4) Write the first byte to follow the preamble into SRX before the preamble is complete and resume normal transmission. Sending a break follows the same procedure except that instead of clearing TE, SBK is set in the SCR to send breaks and then reset to resume normal data transmission.

The example presented in Figure 11-29 uses the SCI in the asynchronous mode to transfer data into buffers. Interrupts are used, allowing the DSP to perform other tasks while the data transfer is occurring. This program can be tested by connecting the SCI transmit and receive pins. Equates are used for convenience and readability.

The program sets the reset vector to run the program after reset, puts a MOVEP instruction at the SCI receive interrupt vector location, and puts a MOVEP and BCLR at the SCI transmit interrupt vector location so that, after transmitting a byte, the transmitter is disabled until another byte is ready for transmission. The SCI is initialized by setting the interrupt level, which configures the SCR and SCCR, and then is enabled by writing the PCC. The main program begins by enabling interrupts, which allows data to be received. Data is transmitted by moving a byte of data to the transmit register and by enabling interrupts. The jump-to-self instruction (SEND JMP SEND) is used to wait while interrupts transfer the data.

```
;*********************************************************************************
;          SCI ASYNC WITH INTERRUPTS AND SINGLE BYTE BUFFERS          *
;*********************************************************************************

;*********************************************
;          SCI and other EQUATES          *
;*********************************************

START     EQU     $0040          ;Start of program
PCC       EQU     $FFE1          ;Port C control register
SCR       EQU     $FFF0          ;SCI interface control register
SCCR      EQU     $FFF2          ;SCI clock control register
SRX       EQU     $FFF4          ;SCI receive register
```

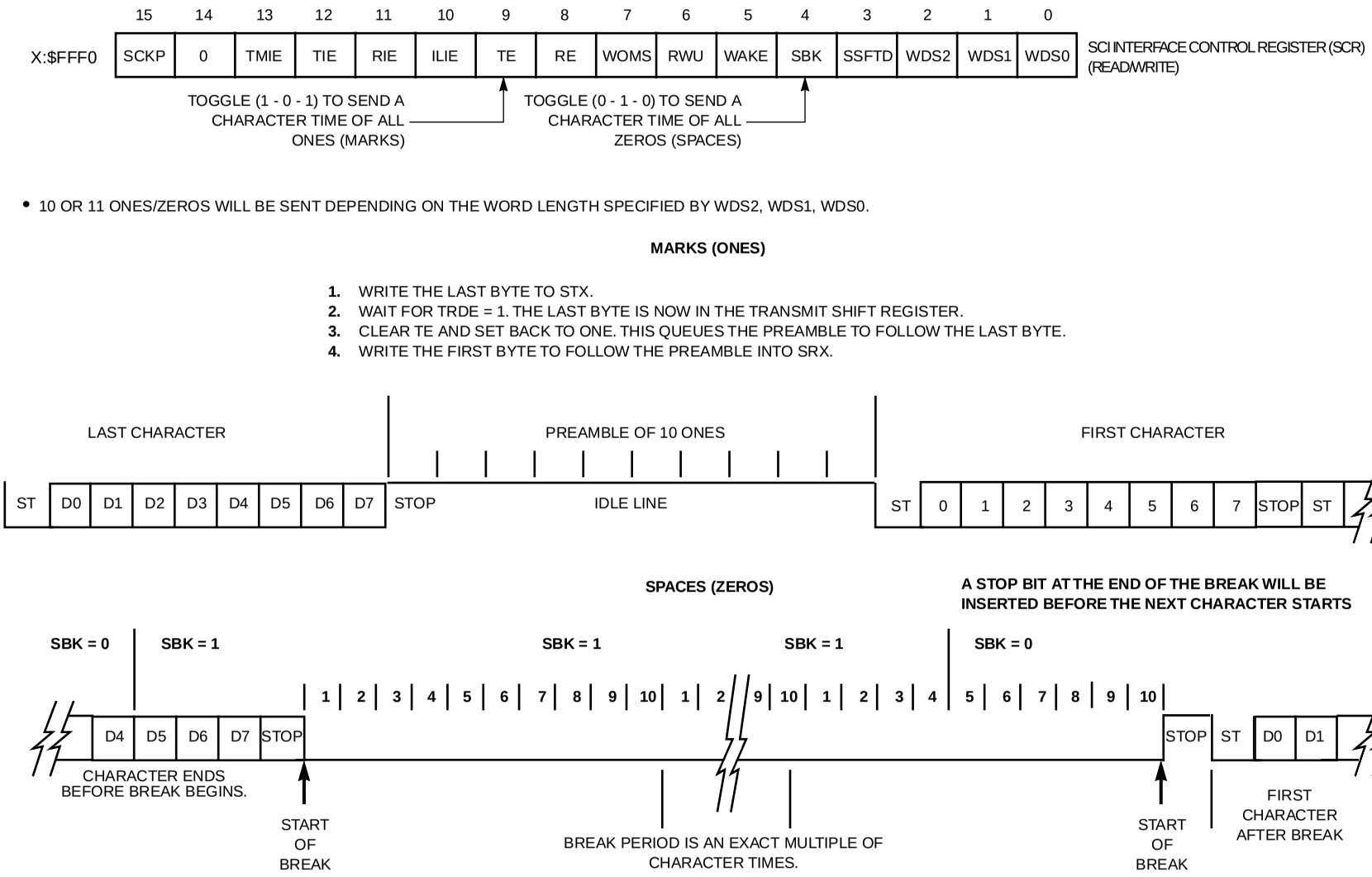


Figure 11-28 Transmitting Marks and Spaces

```
STX             EQU     $FFF4           ;SCI transmit register
BCR             EQU     $FFFE           ;Bus control register
IPR             EQU     $FFFF           ;Interrupt priority register
RXBUF           EQU     $100            ;Receive buffer
TXBUF           EQU     $200            ;Transmit buffer

;************************************************
;               RESET VECTOR                    *
;************************************************

                ORG     P:$0000
                JMP     START

;************************************************
;     SCI RECEIVE INTERRUPT VECTOR              *
;************************************************

                ORG     P:$0014         ;Load the SCI RX interrupt vectors
                MOVEP   X:SRX,Y:(R0)+   ;Put the received byte in the receive
                                        ;buffer. This receive routine is
                                        ;implemented as a fast interrupt.

;************************************************
;     SCI TRANSMIT INTERRUPT VECTOR             *
;************************************************

                ORG     P:$0018         ;Load the SCI TX interrupt vectors
                MOVEP   X:(R3)+,X:STX   ;Transmit a byte and
                                        ;increment the pointer in the
                                        ;transmit buffer.
                BCLR    #12,X:SCR       ;Disable transmit interrupts

;*********************************************************************************************
;           INITIALIZE THE SCI PORT AND RX, TX BUFFER POINTERS                                *
;*********************************************************************************************

                ORG     P:START         ;Start the program at location $40
                ORI     #$03,MR         ;Mask interrupts temporarily
                MOVEP   #$C000,X:IPR    ;Set interrupt priority to 2
                MOVEP   #$0B02,X:SCR    ;Disable TX, enable RX interrupts
                                        ;Enable transmitter, receiver
                                        ;Point to point
                                        ;10-bit asynchronous
```

```
                                        ;(1 start, 8 data, 1 stop)
            MOVEP   #$0022,X:SCCR       ;Use internal TX, RX clocks
                                        ;9600 BPS
            MOVEP   #>$03,X:PCC         ;Select pins TXD and RXD for SCI

            MOVE    RXBUF,R0            ;Initialize the receive buffer
            MOVE    TXBUF,R3            ;Initialize the transmit buffer

;*********************************************
;           MAIN PROGRAM                    *
;*********************************************

            ANDI    #$FC,MR             ;Re-enable interrupts
            MOVE    #>$41,X:(R3)        ;Move a byte to the transmit buffer
            MOVE    R0,X:(R3)
            BSET    #12,X:SCR           ;and enable interrupts so it
                                        ;will be transmitted
SEND        JMP     SEND                ;Normally something more useful
                                        ;would be put here.
            END                         ;=+End of example.
```

Figure 11-29 SCI Asynchronous Transmit/Receive Example

11.2.8 Multidrop

Multidrop is a special case of asynchronous data transfer. The key difference is that a protocol is used to allow networking transmitters and receivers on a single data-transmission line. Interprocessor messages in a multidrop network typically begin with a destination address. All receivers check for an address match at the start of each message. Receivers with no address match can ignore the remainder of the message and use a wakeup mode to enable the receiver at the start of the next message. Receivers with an address match can receive the message and optionally transmit an acknowledgment to the sender. The particular message format and protocol used are determined by the user's software. These message formats include point-to-point, bus, token-ring, and custom configurations. The SCI multidrop network is compatible with other leading microprocessors.

Figure 11-30 shows a multidrop system with one master and N slaves. The multidrop mode is selected by setting WDS2 equals one, WDS1 equals one, and WDS0 equals zero. One possible protocol is to have a preamble or idle line between messages, followed by an address and then a message. The idle line causes the slaves to wake up and compare the address with their own address. If the addresses match, the slave receives the message. If the addresses do not match, the slave ignores the message

gure 11-30 11-Bit Multidrop Mode

and goes back to sleep. It is also possible to generate an interrupt when an address is received, eliminating the need for idle time between consecutive messages and addresses. It is also possible for each slave to look for more than one address, which allows each slave to respond to individual messages as well as broadcast messages (e.g., a global reset).

11.2.8.1 Transmitting Data and Address Characters

Transmitting data and address when the multidrop mode is selected is shown in Figure 11-31. The output sequence shown is idle line, data/address, and the next character. In both cases, an “A” is being transmitted. To send data, TE must be toggled to send the idle line, and then “A” must be sent to STX. Sending the “A” to the STX sets the ninth bit in the frame to zero, which indicates that this frame contains data. If the “A” is sent to STXA instead, the ninth bit in the frame is set to a one, which indicates that this frame contains an address.

11.2.8.2 Wired-OR Mode

Building a multidrop bus network requires connecting multiple transmitters to a common wire. The wired-OR mode allows this to be done without damaging the transmitters when the transmitters are not in use. A protocol is still needed to prevent two transmitters from simultaneously driving the bus. The SCI multidrop word format provides an address field to support this protocol. Figure 11-32 shows a multidrop configuration using wired-OR (set bit 7 of the SCR). The protocol shown consists of an idle line between messages; each message begins with an address character. The message can be any length, depending on the protocol. Each processor in this system has one address that it responds to although each processor can be programmed to respond to more than one address.

11.2.8.3 Idle Line Wakeup

The purpose of a wakeup mode is to free a DSP from reading messages intended for other processors. The usual operational procedure is for each DSP to suspend SCI reception (the DSP can continue processing) until the beginning of a message. Each DSP compares the address in the message header with the DSPs address. If the addresses do not match, the SCI again suspends reception until the next address. If the address matches, the DSP will read and process the message and then suspend reception until the next address.

The idle line wakeup mode wakes up the SCI to read a message before the first character arrives. This mode allows the message to be in any format.

Figure 11-33 shows how to configure the SCI to detect and respond to an idle line. The word format chosen (WDS2, WDS1, and WDS0 in the SCR) must be asynchronous. The

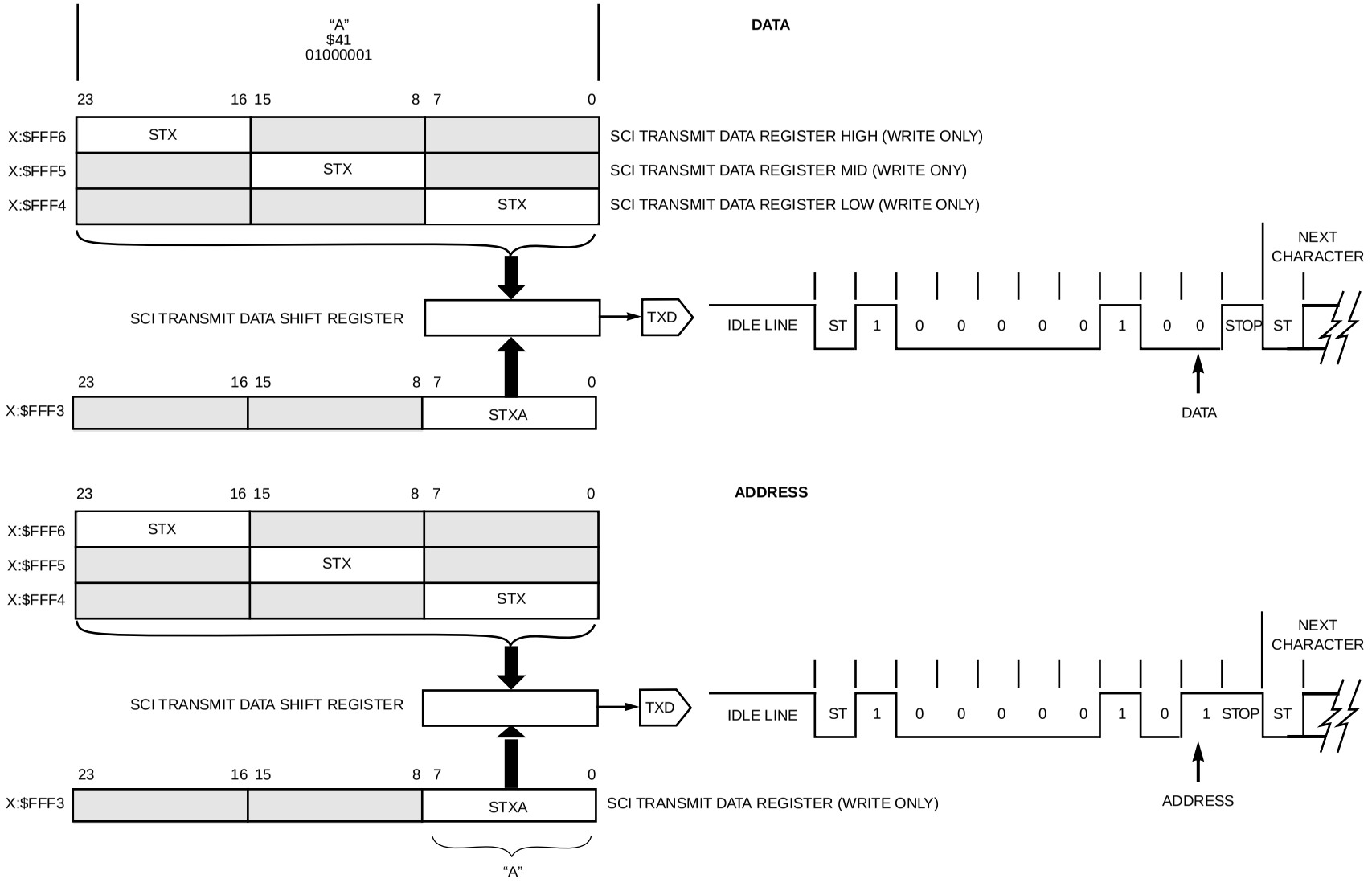


Figure 11-31 Transmitting Data and Address

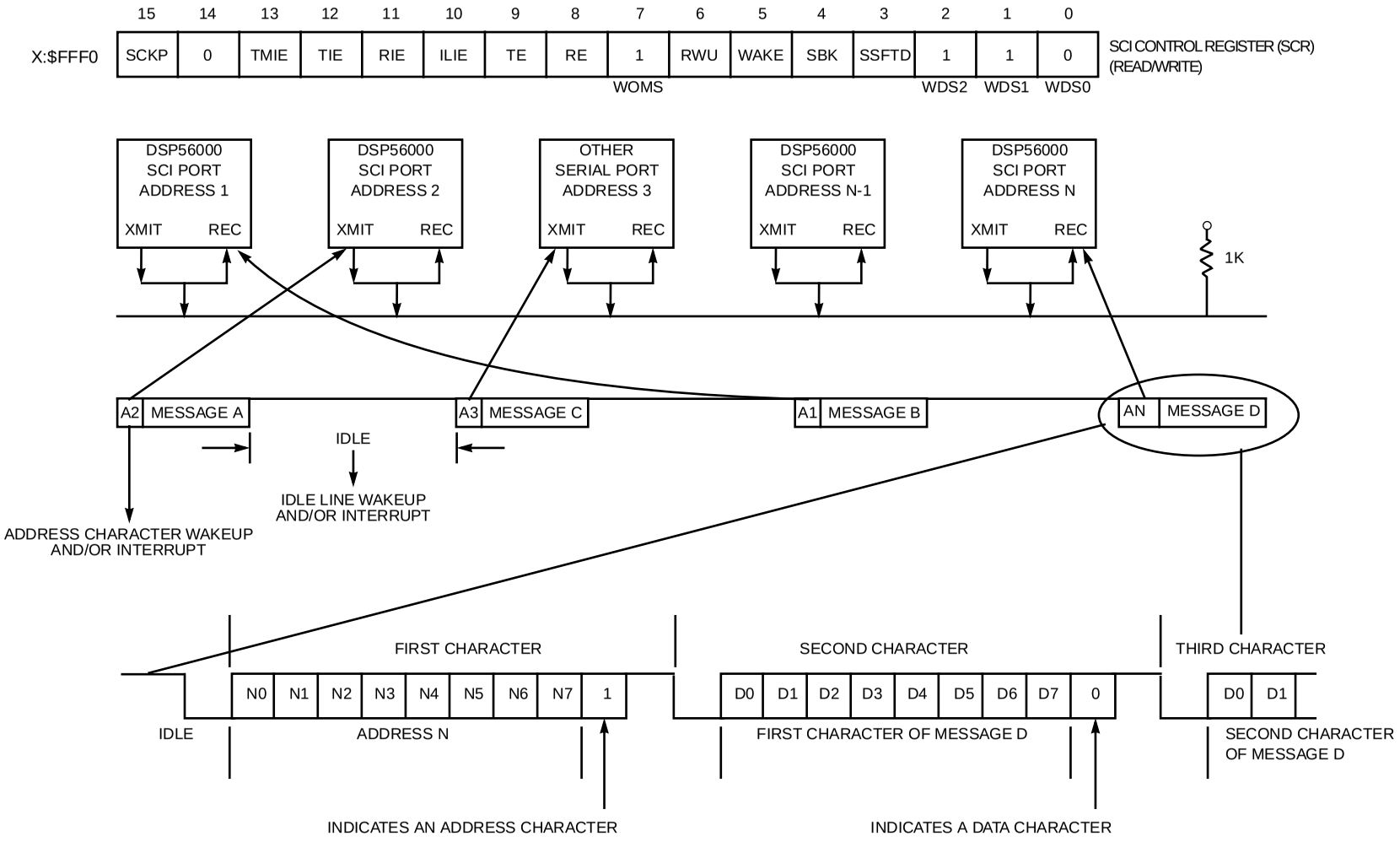


Figure 11-32 Wired-OR Mode

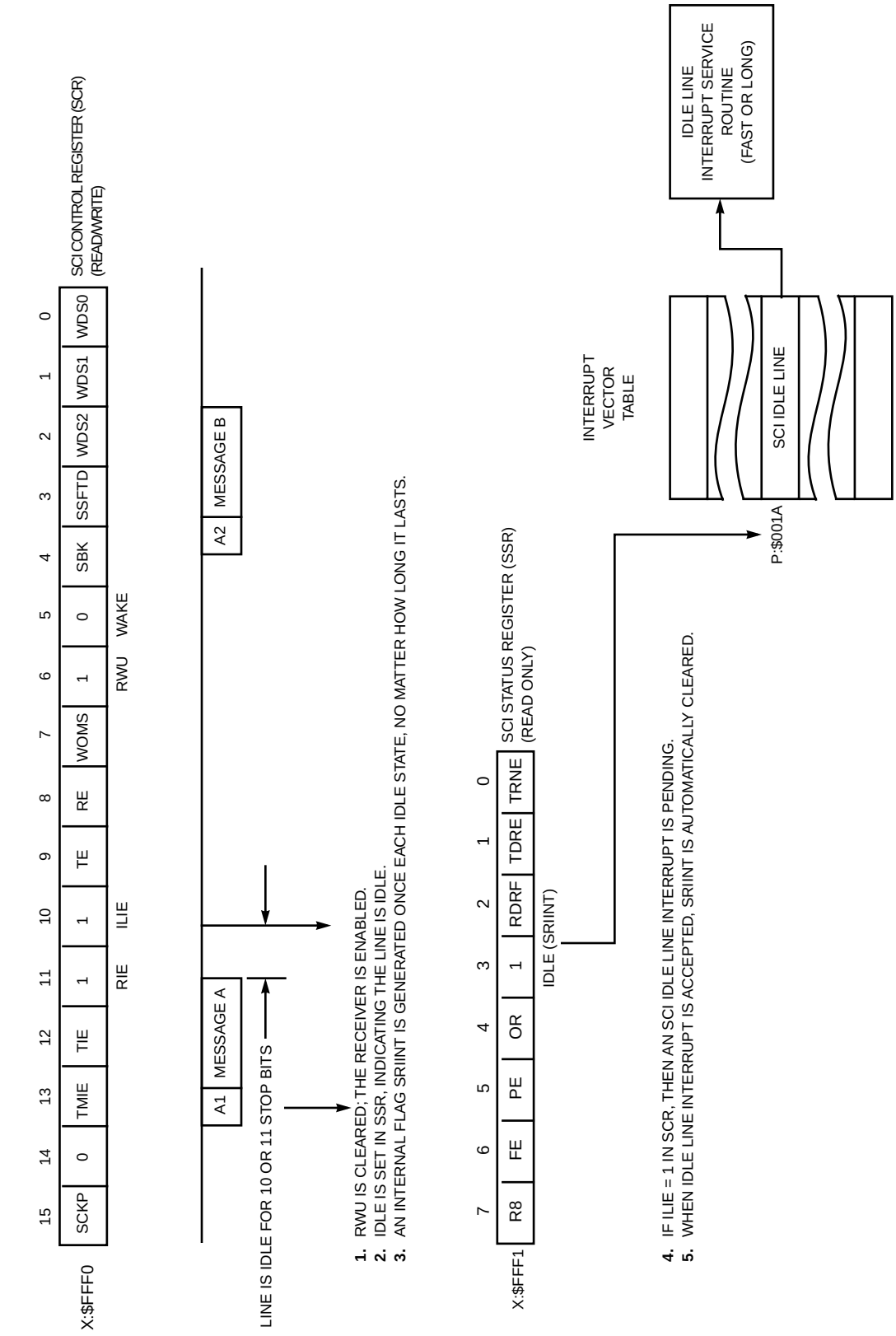


Figure 11-33 Idle Line Wakeup

WAKE bit must be clear to select idle line wakeup, and RWU must be set to put the SCI

to “sleep” and enable the wakeup function. RIE should be set if interrupts are to be used to receive data. If processing must occur when the idle line is first detected, ILIE should be set. The current message is followed by one or more data frames of ones (10 or 11 bits each, depending on which word format is used), which are detected as an idle line. If the word format is multidrop (an 11-bit code), after the 11 ones, the receiver determines the line is idle and (1) clears the RWU, enabling the receiver. The IDLE bit (2) and an internal flag SRIINT (3) are set, indicating the line is idle. The SCI is now ready to receive messages; however, nothing more will happen until the next start bit unless (4) ILIE is set. If ILIE is set, an SCI idle line interrupt will be recognized as pending. When the idle line interrupt is recognized (5), SRIINT is automatically cleared, and the SCI waits for the first start bit of the next character. Since RIE was set, when the first character is received, an SCI receive data interrupt (or SCI receive data with exception status interrupt if an error is detected) will be recognized as pending. When the receiver has processed the message and is ready to wait for another idle line, RWU must be set to one again.

11.2.8.4 Address Mode Wakeup

The purpose and basic operational procedure for address mode wakeup is the same as idle line wakeup. The difference is that address mode wakeup re-enables the SCI when the ninth bit in a character is set to one (if cleared, this bit marks a character as data; if set, an address). As a result, an idle line is not needed, which eliminates the dead time between messages. If the protocol is such that the address byte is not needed or is not wanted in the first byte of the message, a data byte can be written to STXA at the beginning of each message. It is not essential that the first byte of the message contain an address; it is essential that the start of a new message is indicated by setting the ninth bit to one using STXA.

Figure 11-34 shows how to configure the SCI to detect and respond to an address character. The word format chosen (WDS2, WDS1, and WDS0 in the SCR) must be an asynchronous word format. The WAKE bit must be set to select address mode wakeup and RWU must be set to put the SCI to “sleep” and enable the wakeup function. RIE should be set if interrupts are to be used to receive data. (1) When an address character (ninth bit=1) is received, then R8 is set to one in the SSR, and RWU is cleared. Clearing RWU re-enables the SCI receiver. Since (2) RIE was set in this example, when the first character is received, an SCI receive data interrupt (or SCI receive data with exception status interrupt if an error is detected) will be recognized as pending. When the receiver is ready to wait for another address character, RWU must be set to one again.

11.2.8.5 Multidrop Example

The program shown in Figure 11-35 configures the SCI as a multidrop master transmitter and slave receiver (using wakeup on address bit) that uses interrupts to transmit data

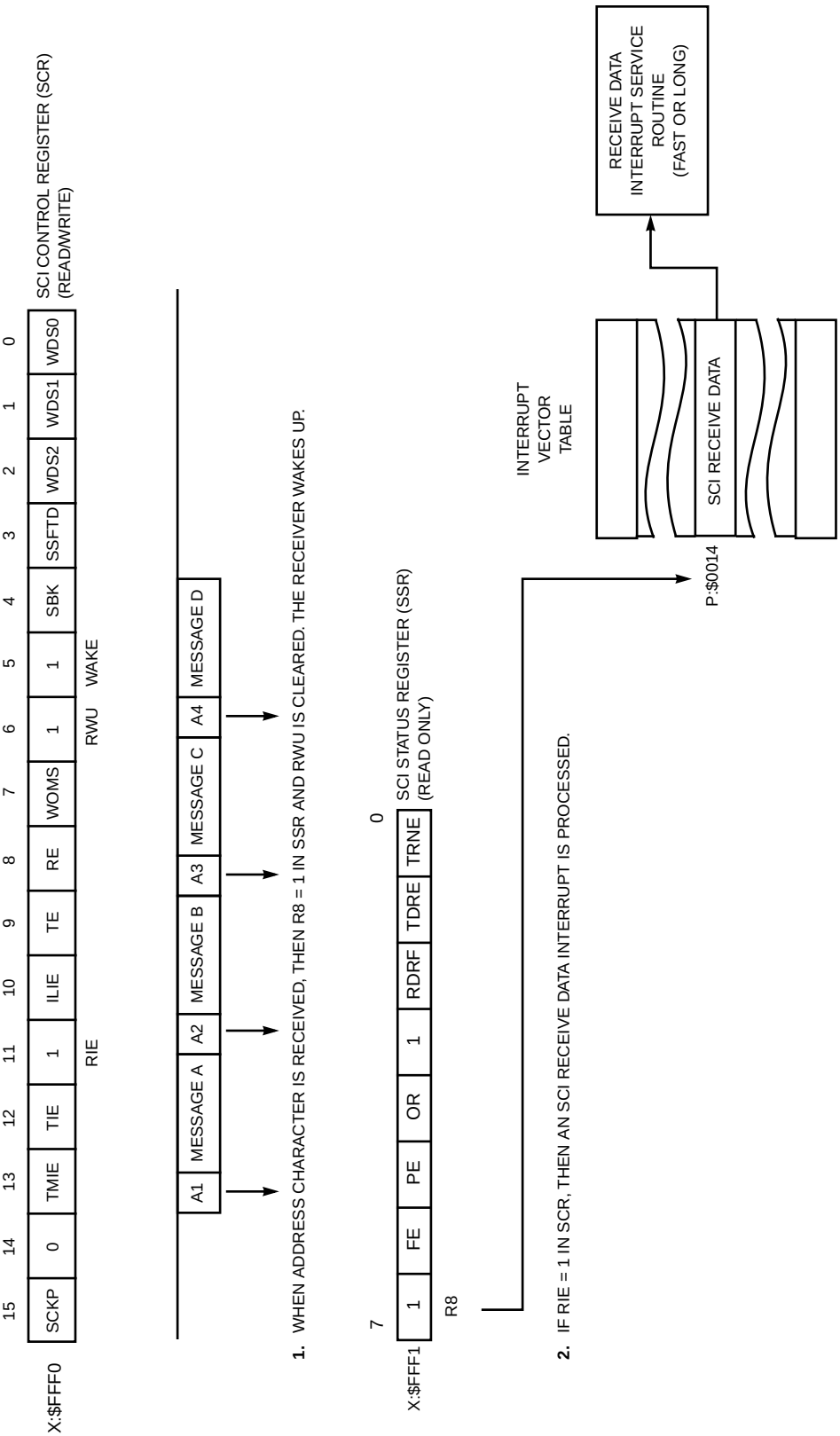


Figure 11-34 Address Mode Wakeup

from a circular buffer and to receive data into a different circular buffer. This program can

be run with the I/O pins (RXD and TXD) connected and with a pullup resistor for test purposes.

The program starts by setting equates for convenience and clarity and then points the reset vector to the start of the program. The receive and transmit interrupt vector locations have JSRs forming long interrupts because the multidrop protocol and circular buffers require more than two instructions for maintenance. Byte packing and unpacking are not used in this example. The SRX and STX registers are equated to $FFF4, causing only the LSB of the 24-bit DSP word to be used for SCI data. The SCI is then initialized as wired-OR, multidrop, and using interrupts. The SCI is enabled but the interrupts are masked, which prevents the SCI from transmitting or receiving data at this time.

The circular buffers used have two pointers. The first points to the first data byte; the second points to the last data byte. This configuration allows the transmit buffer to act as a first-in first-out (FIFO) memory. The FIFO can be loaded by a program and emptied by the SCI in real time. As long as the number of data bytes never exceeds the buffer size, there will be no overflow or underflow of the buffer. Registers M0-M3 must be loaded with the buffer size minus one to make pointer registers R0-R3 work as circular pointers. Register N2 is used as a constant to clear the receive buffer empty flag.

The main program starts by filling the transmit buffer with a data packet. When the transmit buffer is full, it calls the subroutine that transmits the slave's address and then jumps to self (SEND jmp SEND), allowing interrupts to transmit and receive the data.

The receive subroutine first checks each byte to see if it is address or data. If it is an address, it compares the address with its own. If the addresses do not match, the SCI is put back to sleep. If the addresses match, the SCI is left awake, and control is returned to the main program. If the byte is data, it is placed in the receive buffer, and the receive buffer empty flag is cleared. Although this flag is not used in this program, it can be used by another program as a simple test to see if data is available. Using N2 as the constant $0 allows the flag to be cleared with a single-word instruction, which can be part of a fast interrupt.

The transmit subroutine transmits a byte and then checks to see if the transmit buffer is empty. If the buffer is not empty, control is returned to the main program, and interrupts are allowed to continue emptying the buffer. If the buffer is empty, the transmit buffer empty flag is set, the transmit interrupt is disabled, and control is returned to the main program.

The wakeup subroutine transmits the slave's address by writing the address to the STXA register and by enabling the transmit interrupt to allow interrupts to empty the transmit buffer. Control is then returned to the main program.

```
;*********************************************************************************
```

```
;  MULTIDROP MASTER/SLAVE WITH INTERRUPTS AND CIRCULAR BUFFERS  *
;*********************************************************************************************

;************************************************
;               SCI and other EQUATES          *
;************************************************

START           EQU     $0040           ;Start of program
TX_BUFF         EQU     $0010           ;Transmit buffer location
RX_BUFF         EQU     $0020           ;Receive buffer location
B_SIZE          EQU     $000E           ;Transmit and receive buffer size
                                        ;(don't allow the TX buffer and RX
                                        ;buffers to overlap).
TX_MTY          EQU     $0000           ;Transmit buffer empty
RX_MTY          EQU     $0001           ;Receive buffer empty
PCC             EQU     $FFE1           ;Port C control register
SCR             EQU     $FFF0           ;SCI interface control register
SCCR            EQU     $FFF2           ;SCI clock control register
STXA            EQU     $FFF3           ;SCI transmit address register
SRX             EQU     $FFF4           ;SCI receive register
STX             EQU     $FFF4           ;SCI transmit register
BCR             EQU     $FFFE           ;Bus control register
IPR             EQU     $FFFF           ;Interrupt priority register

;************************************************
;               RESET VECTOR              *
;************************************************

                ORG     P:$0000
                JMP     START

;************************************************
;     SCI RECEIVE INTERRUPT VECTOR        *
;************************************************

                ORG     P:$0014         ;Load the SCI RX interrupt vectors
                JSR     RX              ;Jump to the receive routine that
                                        ;puts data packet in a circular
                                        ;buffer if it is for this address.
                NOP                     ;Second word of fast interrupt not
                                        ;needed
                ORG     P:$0016         ;This interrupt occurs when data is
                                        ;received with errors. This example
```

```
                NOP                             ;does not trap errors so this
                NOP                             ;interrupt is not used.

;***********************************************
;    SCI TRANSMIT INTERRUPT VECTOR    *
;***********************************************

                ORG     P:$0018                 ;Load the SCI TX interrupt vectors
                JSR     TX                      ;Transmit next byte in buffer
                NOP

;***********************************************
;          INITIALIZE THE SCI PORT            *
;***********************************************

                ORG     P:START                 ;Start the program at location $40
                ORI     #$03,MR                 ;Mask interrupts temporarily
                MOVEP   #$C000,X:IPR            ;Set interrupt priority to 2

                MOVEP   #$0BE6,X:SCR            ;Disable TX, enable RX interrupts
                                                ;Enable transmitter and receiver,
                                                ;Wired-OR mode, Rec. wakeup
                                                ;mode,11-bit multidrop (1 start,
                                                ;8 data,1 data type, 1 stop)
                MOVEP   #$0000,X:SCCR           ;Use internal TX, RX clocks
                                                ;320K BPS
                MOVEP   #>$03,X:PCC             ;Select pins TXD and RXD for SCI

;***********************************************
;INITIALIZE INTERRUPTS, REGISTERS, ETC.*
;***********************************************

                MOVEP   #$0,X:BCR               ;No wait states

                MOVE    #TX_BUFF,R0             ;Load start pointer of transmit
                                                ;buffer
                MOVE    #TX_BUFF,R1             ;Load end   pointer of transmit
                                                ;buffer
                MOVE    #RX_BUFF,R2             ;Load start pointer of receive
                                                ;buffer
                MOVE    #RX_BUFF,R3             ;Load end pointer of receive
                                                ;buffer
                MOVE    #>$41,R5                ;Init data register... R5 contains
```

```
                                        ;the data that will be sent in this
                                        ;example; it is initialized to an
                                        ;ASCII A.
                MOVE    #B_SIZE,M0      ;Load transmit buffer size
                MOVE    #B_SIZE,M1      ;Load transmit buffer size
                MOVE    #B_SIZE,M2      ;Load receive buffer size
                MOVE    #B_SIZE,M3      ;Load receive buffer size
                MOVE    #>$1,N0         ;Load receive address
                MOVE    #>$1,N1         ;Load first slave address
                MOVE    #0,N2           ;Load a constant (0) into N2
                MOVEP   X:SRX,X:(R0)    ;Clear receive register

;***********************************************
;               MAIN PROGRAM            *
;***********************************************

                ANDI    #$FC,MR         ;Re-enable interrupts

                MOVE    (R1)+           ;Temporarily increment the tail
                                        ;pointer
                                        ;Build a packet
LOOP            MOVE    R1,A            ;Check to see if the TX buffer is
                                        ;full
                MOVE    (R1)-           ;(fix tail pointer now that we've
                                        ;used it)
                MOVE    R0,B            ;by comparing the head and tail
                                        ;pointers
                CMP     A,B             ;of the circular transmit buffer.
                JEQ     SND_BUF         ;if equal, transmit completed packet
                MOVE    R5,X:(R1)+      ;if not, put next character in
                                        ;transmit buffer and
                MOVE    (R5)+           ;increment the pointers.
                MOVE    (R1)+           ;Temporarily increment the tail
                                        ;pointer to test buffer again
                JMP     LOOP
SND_BUF         JSR     WAKE_UP         ;Wake up proper slave and send
                                        ;packet
SEND            JMP     SEND            ;and allow interrupts to drain
                                        ;the transmit buffer.

;*************************************************************************************
;SUBROUTINE TO READ SCI AND STORE IN BUFFER USING A LONG INTERRUPT*
```

```
;*****************************************************************************************
RX          JCLR    #7,X:$FFF1,RX_DATA      ;Check if this is address or data.
            MOVEP   X:SRX,A                 ;Compare the received address
            MOVE    N1,B                    ;with the slave address.
            CMP     A,B
            JEQ     END_RX                  ;If address OK, use interrupts to Rx
                                            ;packet
            BSET    #6,X:$FFF0              ;if not, go back to sleep
            JMP     END_RX                  ;and return to previous program.
RX_DATA     MOVEP   X:SRX,X:(R3)+           ;Put data in buffer,
            MOVE    N2,X:RX_MTY             ;and clear the Rx buffer empty flag
END_RX      RTI                             ;Return to previous program

;*****************************************************************************************
;       SUBROUTINE TO WRITE BUFFER TO SCI USING A LONG INTERRUPT        *
;*****************************************************************************************

TX          MOVEP   X:(R0)+,X:STX           ;Transmit a byte and increment the
                                            ;pointer
            MOVE    R0,A                    ;Check to see if the TX buffer is
                                            ;empty
            MOVE    R1,B
            CMP     A,B
            JNE     END_TX                  ;If not, return to main
            MOVE    #$000001,X0             ;If it is, set the TX buffer empty
                                            ;flag
            MOVE    X0,X:TX_MTY
            BCLR    #12,X:SCR               ;disable transmit interrupts, and
END_TX      RTI                             ;return to main

;*****************************************************************************************
;               SUBROUTINE TO WAKE UP THE ADDRESSED SLAVE               *
;*****************************************************************************************

WAKE_UP     MOVEP   N1,X:STXA               ;Transmit slave address using STXA
                                            ;not STX
            BSET    #12,X:SCR               ;Enable transmit interrupts to send
                                            ;packet
AWAKE       RTI

            END                             ;End of example.
```

Figure 11-35 Multidrop Transmit/Receive Example

11.2.9 SCI Timer

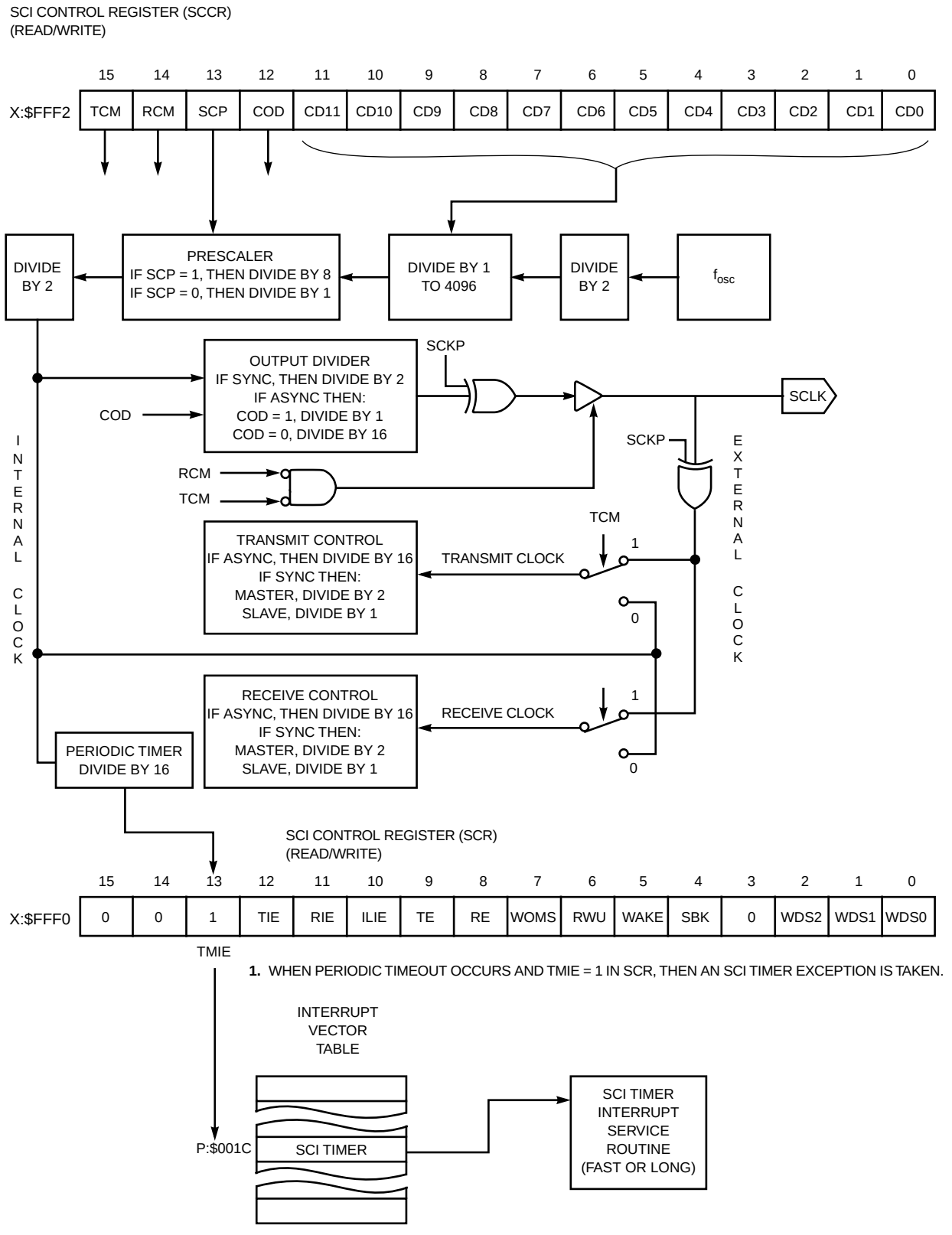
The SCI clock used to determine the data transmission rate can also be used to cause a periodic interrupt. This interrupt can be used as an event timer or for any other timing function. Figure 11-36 illustrates how the SCI timer is programmed. Only bits CD11–CD0 and SCP in the SCCR are used to determine the time base. The crystal oscillator f_{osc} is first divided by 2 and then divided by the number CD11–CD0 in the SCCR. The oscillator is then divided by 1 (if SCP=0) or eight (if SCP=1). Finally, it is divided by 2 and then by 16. If TMIE in the SCR is set (1) when the periodic timeout occurs, the SCI timer interrupt is recognized and pending. The SCI timer interrupt is automatically cleared when the interrupt is serviced. This interrupt will occur every time the periodic timer times out. If only the timer function is being used (i.e., PC0, PC1, and PC2 pins have been programmed as parallel I/O pins), the transmit interrupts should be turned off (TIE=0). Under individual reset, TDRE will remain set, continuously generating interrupts.

Figure 11-36 shows that an external clock can be used for SCI receive and/or transmit, which frees the SCI timer to be programmed for a different interrupt rate. In addition, both the SCI timer interrupt and the SCI can use the internal time base if the SCI receiver and/or transmitter require the same clock period as the SCI timer.

The following program (see Figure 11-37) configures the SCI to interrupt the DSP at fixed intervals. The program starts by setting equates for convenience and clarity and then points the reset vector to the start of the program. The SCI timer interrupt vector location contains “move (R0)+”, incrementing the contents of R0, which serves as an elapsed time counter.

The timer initialization consists of enabling the SCI timer interrupt, setting the SCI baud rate counters for the desired interrupt rate, setting the interrupt mask, enabling the interrupt, and then enabling the SCI state machine.

```
;*********************************************************************************************
;                          TIMER USING SCI TIMER INTERRUPT                                  *
;*********************************************************************************************

;*************************************************
;              SCI and other EQUATES              *
;*************************************************

START       EQU     $0040           ;Start of program
SCR         EQU     $FFF0           ;SCI control register
SCCR        EQU     $FFF2           ;SCI clock control register
IPR         EQU     $FFFF           ;Interrupt priority register
```



1. WHEN PERIODIC TIMEOUT OCCURS AND TMIE = 1 IN SCR, THEN AN SCI TIMER EXCEPTION IS TAKEN.

2. PENDING TIMER INTERRUPT IS AUTOMATICALLY CLEARED WHEN INTERRUPT IS SERVICED.

Figure 11-36 SCI Timer Operation

```
;*************************************************
;               RESET VECTOR                    *
;*************************************************

                ORG     P:$0000
                JMP     START

;*************************************************
;     SCI TIMER INTERRUPT VECTOR                *
;*************************************************

                ORG     P:$001C         ;Load the SCI timer interrupt vectors
                MOVE    (R0)+           ;Increment the timer interrupt counter
                NOP                     ;This timer routine is implemented
                                        ;as a fast interrupt.

;*************************************************
;          INITIALIZE THE SCI PORT              *
;*************************************************

                ORG     P:START         ;Start the program at location $40
                MOVE    #0,R0           ;Initialize the timer interrupt counter
                MOVEP   #$2000,X:SCR    ;Select the timer interrupt
                MOVEP   #$013F,X:SCCR   ;Set the interrupt rate at 1 ms.
                                        ;(arbitrarily chosen).
                                        ;Interrupts/second =
                                        ;fosc/(64×(7(SCP)-+1)×(CD+1))
                                        ;Note that this is the same equation
                                        ;as for SCI async baud rate.
                                        ;For 1 ms, SCP=0,
                                        ;CD=0001 0011 1111.
                MOVEP   #$C000,X:IPR    ;Set the interrupt priority level–
                                        ;application specific.
                ANDI    #$FC,MR         ;Enable interrupts, set MR bits I1 and
                                        ;I0=0

END             JMP     END             ;Normally something more useful
                                        ;would be put here.

                END                     ;End of example.
```

Figure 11-37 SCI Timer Example

11.2.10 Example Circuits

The SCI can be used in a number of configurations to connect multiple processors. The synchronous mode shown in Figure 11-38 shows the DSP acting as a slave. The 8051 provides the clock that clocks data in and out of the SCI, which is possible because the SCI shift register mode timing is compatible with the timing for 8051/8096 processors. Transmit data is changed on the negative edge of the clock, and receive data is latched on the positive edge of the clock. A protocol must be used to prevent both processors from transmitting simultaneously. The DSP is also capable of being the master device.

A multimaster system can be configured (see Figure 11-39) using a single transmit/ receive line, multidrop word format, and wired-OR. The use of wired-OR requires a pul-lup resistor as shown. A protocol must be used to prevent collisions. This scheme is physically the simplest multiple DSP interconnection because it uses only one wire and one resistor.

The master-slave system shown in Figure 11-40 is different in that it is full duplex. The clock pin is not required; thus, it is configured as a parallel I/O pin. Communication is asynchronous. The slave's transmitters must be wire-ORed because more than one transmitter is on one line. The master's transmitter does not need to be wire-ORed.

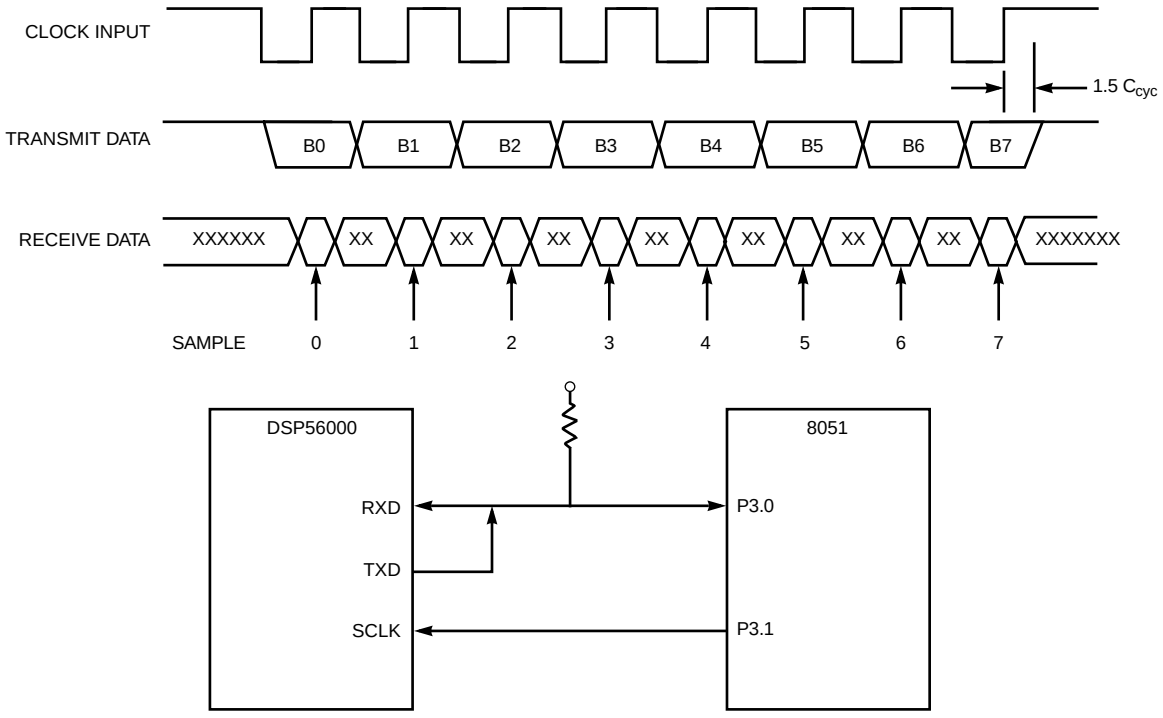


Figure 11-38 Synchronous Mode Example

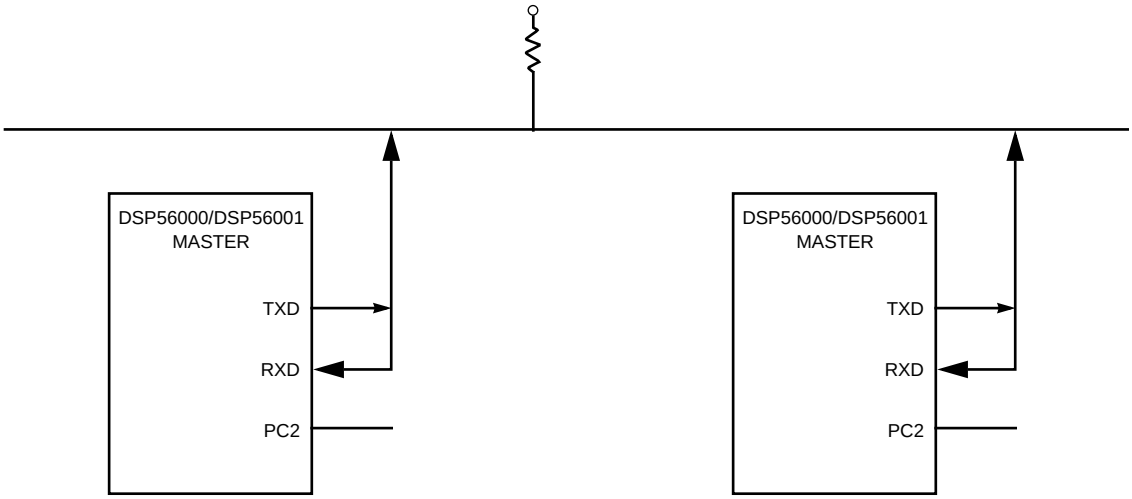


Figure 11-39 Multimaster System Example

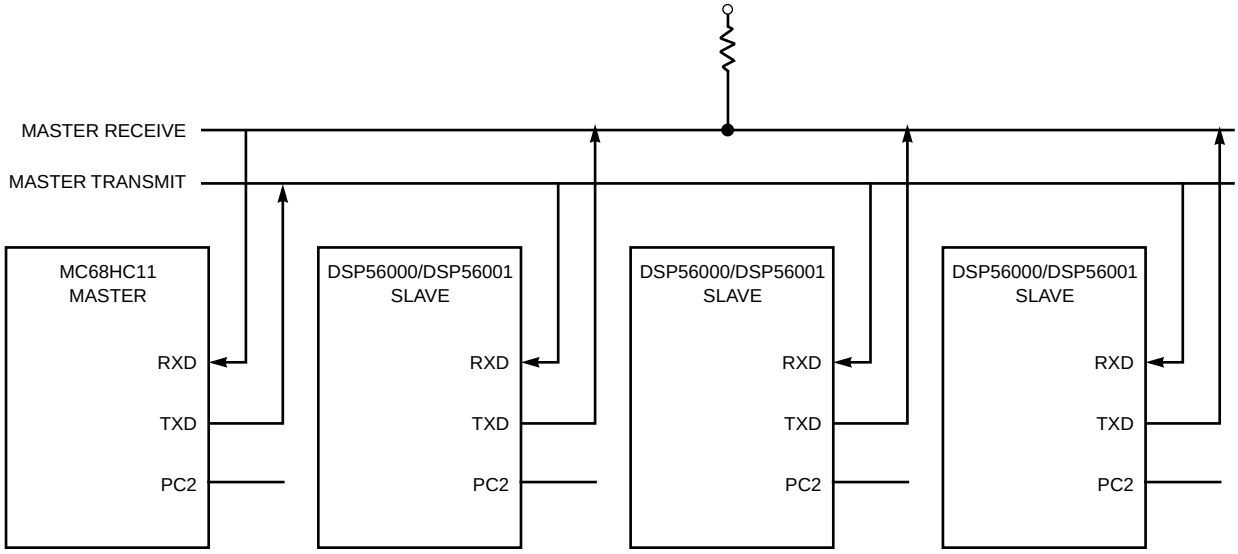


Figure 11-40 Master-Slave System Example

11.3 SYNCHRONOUS SERIAL INTERFACE (SSI)

The synchronous serial interface (SSI) provides a full-duplex serial port for serial communication with a variety of serial devices including one or more industry-standard codecs, other DSPs, microprocessors, and peripherals which implement the Motorola SPI. The SSI consists of independent transmitter and receiver sections and a common SSI clock generator. Three to six pins are required for operation, depending on the operating mode selected.

The following is a short list of SSI features:

- A 6.75 Mbps at 27 MHz ($f_{osc}/4$) serial interface
- Double Buffered
- User Programmable
- Separate Transmit and Receive Sections
- Control and Status Bits
- Interface to a Variety of Serial Devices, Including:
  - Codecs (usually without additional logic)
    - MC145500
    - MC145501
    - MC145502
    - MC145503
    - MC145505
    - MC145402 (13-bit linear codec)
    - MC145554 Family of Codecs
  - Serial Peripherals (A/D, D/A)
    - Most Industry-Standard A/D, D/A
    - DSP56ADC16 (16-bit linear A/D)
  - DSP56000 to DSP56000 Networks
  - Motorola SPI Peripherals and Processors
  - Shift Registers
- Interface to Time Division Multiplexed Networks without Additional Logic
- Six Pins:
  - STD SSI Transmit Data
  - SRD SSI Receive Data
  - SCK SSI Serial Clock
  - SC0 Serial Control 0 (defined by SSI mode)
  - SC1 Serial Control 1 (defined by SSI mode)
  - SC2 Serial Control 2 (defined by SSI mode)
- On-chip Programmable Functions Include:
  - Clock – Continuous, Gated, Internal, External
  - Synchronization Signals:
    - – Bit Length
    - – Word Length
  - TX/RX Timing – Synchronous, Asynchronous
  - Operating Modes – Normal, Network, On-Demand
  - Word Length – 8, 12, 16, 24 Bits
  - Serial Clock and Frame Sync Generator

- Four Interrupt Vectors:
  Receive
  Receive with Exception
  Transmit
  Transmit with Exception

This interface is named synchronous because all serial transfers are synchronized to a clock. Additional synchronization signals are used to delineate the word frames. The normal mode of operation is used to transfer data at a periodic rate, but only one word per period. The network mode is similar in that it is also intended for periodic transfers; however, it will support up to 32 words (time slots) per period. This mode can be used to build time division multiplexed (TDM) networks. In contrast, the on-demand mode is intended for nonperiodic transfers of data. This mode can be used to transfer data serially at high speed when the data becomes available. This mode offers a subset of the SPI protocol.

11.3.1 SSI Data and Control Pins

The SSI has three dedicated I/O pins (see Figure 11-1), which are used for transmit data (STD), receive data (SRD), and serial clock (SCK), where SCK may be used by both the transmitter and the receiver for synchronous data transfers or by the transmitter only for asynchronous data transfers. Three other pins may also be used, depending on the mode selected; they are serial control pins SC0, SC1, and SC2. These serial control pins may be programmed as SSI control pins in the port C control register. Table 11-4 shows the definition of SC0, SC1, SC2, and SCK in the various configurations. The following paragraphs describe the uses of these pins for each of the SSI operating modes.

Figure 11-41 and Figure 11-42 show the internal clock path connections in block diagram form. The receiver and transmitter clocks can be internal or external depending on the SYN, SCD0, and SCKD bits in CRB.

11.3.1.1 Serial Transmit Data Pin (STD)

STD is used for transmitting data from the serial transmit shift register. STD is an output when data is being transmitted. Data changes on the positive edge of the bit clock. STD goes to high impedance on the negative edge of the bit clock of the last data bit of the word (i.e., during the second half of the last data bit period) with external gated clock, regardless of the mode. With an internally generated bit clock, the STD pin becomes high impedance after the last data bit has been transmitted for a full clock period, assuming another data word does not follow immediately. If a data word follows immediately, there will not be a high-impedance interval.

Codecs label the MSB as bit 0; whereas, the DSP labels the LSB as bit 0. Therefore, when using a standard codec, the DSP MSB (or codec bit 0) is shifted out first when SHFD=0, and the DSP LSB (or codec bit 7) is shifted out first when SHFD=1. STD may be programmed as a general-purpose pin called PC8 when the SSI STD function is not being used.

Table 11-4 Definition of SC0, SC1, SC2, and SCK

SSI Pin Name (Control Bit Name)	Asynchronous (SYN=0)		Synchronous (SYN=1)	
	Continuous Clock (GCK=0)	Gated Clock (GCK=1)	Continuous Clock (GCK=0)	Gated Clock (GCK=1)
SC0=0 (in) SC0=1 (out) (SCD0)	RXC External RXC Internal	RXC External RXC Internal	Input F0 Output F0	Input F0 Output F0
SC1=0 (in) SC1=1 (out) (SCD1)	FSR External FSR Internal	Not Used FSR Internal	Input F1 Output F1	Input F1 Output F1
SC2=0 (in) SC2=1 (out) (SCD2)	FST External FST Internal	Not Used FST Internal	FS* External FS* Internal	Not Used FS* Internal
SCK=0 (in) SCK=1 (out (SCKD)	TXC External TXC Internal	TXC External TXC Internal)	*XC External *XC Internal	*XC External *XC Internal

TXC – Transmitter Clock
RXC – Receiver Clock
*XC – Transmitter/Receiver Clock (synchronous operation)
FST – Transmitter Frame Sync
FSR – Receiver Frame Sync
FS* – Transmitter/Receiver Frame Sync (synchronous operation)
PF0 – Flag 0
F1 – Flag 1

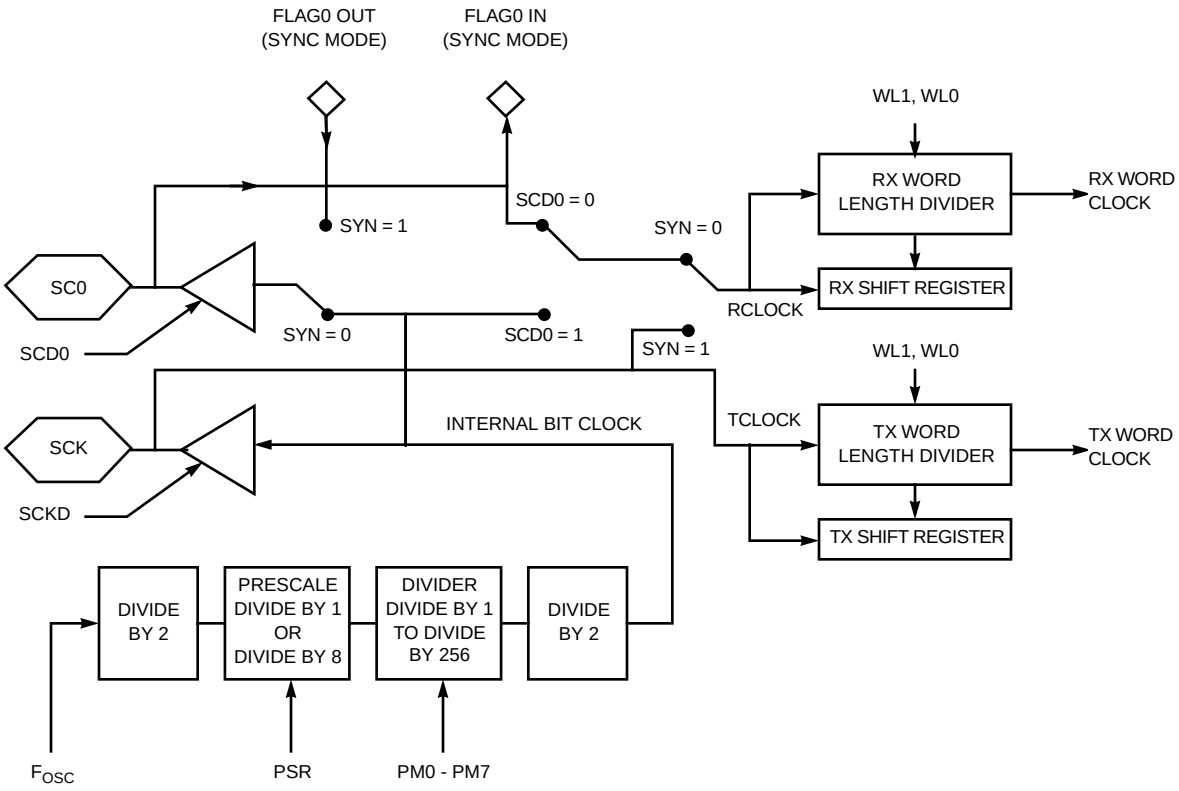


Figure 11-41 SSI Clock Generator Functional Block Diagram

11.3.1.2 Serial Receive Data Pin (SRD). SRD receives serial data and transfers the data to the SSI receive shift register. SRD may be programmed as a general-purpose I/O pin called PC7 when the SSI SRD function is not being used. Data is sampled on the negative edge of the bit clock.

11.3.1.3 Serial Clock (SCK). SCK is a bidirectional pin providing the serial bit rate clock for the SSI interface. The SCK is a clock input or output used by both the transmitter and receiver in synchronous modes or by the transmitter in asynchronous modes (see Table11-5).

NOTE

Although an external serial clock can be independent of and asynchronous to the DSP system clock, it must exceed the minimum clock cycle time of at least 3.6 times the external SSI clock frequency. Using a ratio of 3.6:1 allows for a clock skew between two SSIs of up to ±5%.

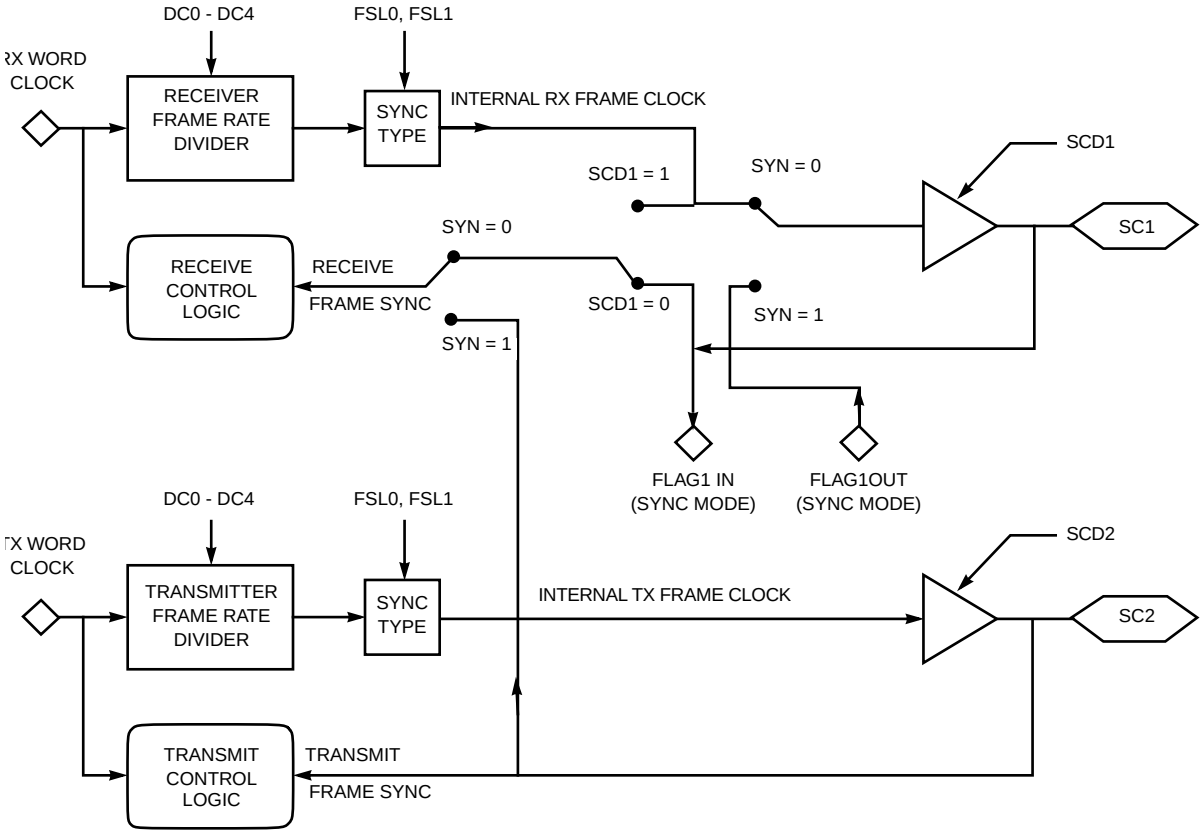


Figure 11-42 SSI Frame Sync Generator Functional Block Diagram

11.3.1.4 Serial Control Pin (SC0)

The function of this pin is determined solely on the selection of either synchronous or asynchronous mode (see Tables 11-4 and 11-5). For asynchronous mode, this pin will be used for the receive clock I/O. For synchronous mode, this pin is used for serial flag I/O. A typical application of flag I/O would be multiple device selection for addressing in codec systems. The direction of this pin is determined by the SCD0 bit in the CRB as described in the following table. When configured as an output, this pin will be either serial output flag 0, based on control bit OF0 in CRB, or a receive shift register clock output. When configured as an input, this pin may be used either as serial input flag 0, which will control status bit IF0 in the SSISR, or as a receive shift

Table 11-5 SSI Clock Sources, Inputs, and Outputs

SYN	SCKD	SCD0	R Clock Source	RX Clock Out	T Clock Source	TX Clock Out
Asynchronous						
0	0	0	EXT, SC0	–	EXT, SCK	–
0	0	1	INT	SC0	EXT, SCK	–
0	1	0	EXT, SC0	–	INT	SCK
0	1	1	INT	SC0	EXT	SCK
Synchronous						
1	0	0	EXT, SCK	–	EXT, SCK	–
1	0	1	EXT, SCK	–	EXT, SCK	–
1	1	0	INT	SCK	INT	SCK
1	1	1	INT	SCK	INT	SCK

EXT – External Pin Name
INT – Internal Bit Clock

register clock input.

SYN	GCK	SCD0	Operation
Synchronous	Continuous	Input	Flag 0 Input
Synchronous	Continuous	Output	Flag 0 Output
Synchronous	Gated	Input	Flag 0 Input
Synchronous	Gated	Output	Flag 0 Output
Asynchronous	Continuous	Input	Rx Clock – External
Asynchronous	Continuous	Output	Rx Clock – Internal
Asynchronous	Gated	Input	Rx Clock – External
Asynchronous	Gated	Output	Rx Clock – Internal

11.3.1.5 Serial Control Pin (SC1)

The function of this pin is determined solely on the selection of either synchronous or asynchronous mode (see Table 11-4). In asynchronous mode (such as a single codec with asynchronous transmit and receive), this pin is the receiver frame sync I/O. For synchronous mode with continuous clock, this pin is serial flag SC1 and operates like the previously described SC0. SC0 and SC1 are independent serial I/O flags but may be used together for multiple serial device selection. SC0 and SC1 can be used unencoded to select up to two codecs or may be decoded externally to select up to four codecs. The direction of this pin is determined by the SCD1 bit in the CRB. When configured as

an output, this pin will be either a serial output flag, based on control bit OF1, or it will make the receive frame sync signal available. When configured as an input, this pin may be used as a serial input flag, which will control status bit IF1 in the SSI status register, or as a receive frame sync from an external source for continuous clock mode. In the gated clock mode, external frame sync signals are not used.

SYN	GCK	SCD1	Operation
Synchronous	Continuous	Input	Flag 1 Input
Synchronous	Continuous	Output	Flag 1 Output
Synchronous	Gated	Input	Flag 1 Input
Synchronous	Gated	Output	Flag 1 Output
Asynchronous	Continuous	Input	RX Frame Sync – External
Asynchronous	Continuous	Output	RX Frame Sync – Internal
Asynchronous	Gated	Input	–
Asynchronous	Gated	Output	RX Frame Sync – Internal

11.3.1.6 Serial Control Pin (SC2)

This pin is used for frame sync I/O (see Table 11-4). SC2 is the frame sync for both the transmitter and receiver in synchronous mode and for the transmitter only in asynchronous mode. The direction of this pin is determined by the SCD2 bit in CRB. When configured as an output, this pin is the internally generated frame sync signal. When configured as an input, this pin receives an external frame sync signal for the transmitter (and the receiver in synchronous operation). In the gated clock mode, external frame sync signals are not used.

SYN	GCK	SCD2	Operation
Synchronous	Continuous	Input	TX and RX Frame Sync
Synchronous	Continuous	Output	TX and RX Frame Sync
Synchronous	Gated	Input	–
Synchronous	Gated	Output	TX and RX Frame Sync
Asynchronous	Continuous	Input	TX Frame Sync – External
Asynchronous	Continuous	Output	TX Frame Sync – Internal
Asynchronous	Gated	Input	–
Asynchronous	Gated	Output	TX Frame Sync – Internal

11.3.2 SSI Interface Programming Model

The SSI can be viewed as two control registers, one status register, a transmit register, a receive register, and special-purpose time slot register. These registers are illustrated in Figure 11-43 and Figure 11-44. The following paragraphs give detailed descriptions and operations of each of the bits in the SSI registers. The SSI registers are not prefaced with an "S" (for serial) as are the SCI registers.

11.3.2.1 SSI Control Register A (CRA)

CRA is one of two 16-bit read/write control registers used to direct the operation of the SSI. The CRA controls the SSI clock generator bit and frame sync rates, word length, and number of words per frame for the serial data. The high-order bits of CRA are read as zeros by the DSP CPU. The CRA control bits are described in the following paragraphs.

11.3.2.1.1 CRA Prescale Modulus Select (PM7–PM0) Bits 0–7

The PM0–PM7 bits specify the divide ratio of the prescale divider in the SSI clock generator. A divide ratio from 1 to 256 (PM=0 to $FF) may be selected. The bit clock output is available at the transmit clock (SCK) and/or the receive clock (SC0) pins of the DSP. The bit clock output is also available internally for use as the bit clock to shift the transmit and receive shift registers. Careful choice of the crystal oscillator frequency and the prescaler modulus will allow the industry-standard codec master clock frequencies of 2.048 MHz, 1.544 MHz, and 1.536 MHz to be generated. Hardware and software reset clear PM0–PM7.

11.3.2.1.2 CRA Frame Rate Divider Control (DC4–DC0) Bits 8–12

The DC4–DC0 bits control the divide ratio for the programmable frame rate dividers used to generate the frame clocks (see Figure 11-42). In network mode, this ratio may be interpreted as the number of words per frame minus one. In normal mode, this ratio determines the word transfer rate. The divide ratio may range from 1 to 32 (DC=00000 to 11111) for normal mode and 2 to 32 (DC=00001 to 11111) for network mode.

A divide ratio of one (DC=00000) in network mode is a special case (see 11.3.7.4 ON-DEMAND MODE). In normal mode, a divide ratio of one (DC=00000) provides continuous periodic data word transfers. A bit-length sync (FSL1=1, FSL0=0) must be used in this case. Hardware and software reset clear DC4–DC0.

11.3.2.1.3 CRA Word Length Control (WL0, WL1) Bits 13 and 14

The WL1 and WL0 bits are used to select the length of the data words being transferred via the SSI. Word lengths of 8, 12, 16, or 24 bits may be selected according to the following assignments:

WL1	WL0	Number of Bits/Word
0	0	8
0	1	12
1	0	16
1	1	24

These bits control the number of active clock transitions in the gated clock modes and control the word length divider (see Figure 11-41 and Figure 11-42), which is part of the frame rate signal generator for continuous clock modes. The WL control bits also control the frame sync pulse length when FSL0 and FSL1 select a WL bit clock (see Figure 11-41). Hardware and software reset clear WL0 and WL1.

11.3.2.1.4 CRA Prescaler Range (PSR) Bit 15

The PSR controls a fixed divide-by-eight prescaler in series with the variable prescaler. This bit is used to extend the range of the prescaler for those cases where a slower bit clock is desired (see Figure 11-41). When PSR is cleared, the fixed prescaler is bypassed. When PSR is set, the fixed divide-by-eight prescaler is operational. This allows a 128-kHz master clock to be generated for MC14550x series codecs.

The maximum internally generated bit clock frequency is fosc/4, the minimum internally generated bit clock frequency is fosc/4/8/256=fosc/8192. Hardware and software reset clear PSR.

11.3.2.2 SSI Control Register B (CRB)

The CRB is one of two 16-bit read/write control registers used to direct the operation of the SSI. CRB controls the SSI multifunction pins, SC2, SC1, and SC0, which can be used as clock inputs or outputs, frame synchronization pins, or serial I/O flag pins. The serial output flag control bits and the direction control bits

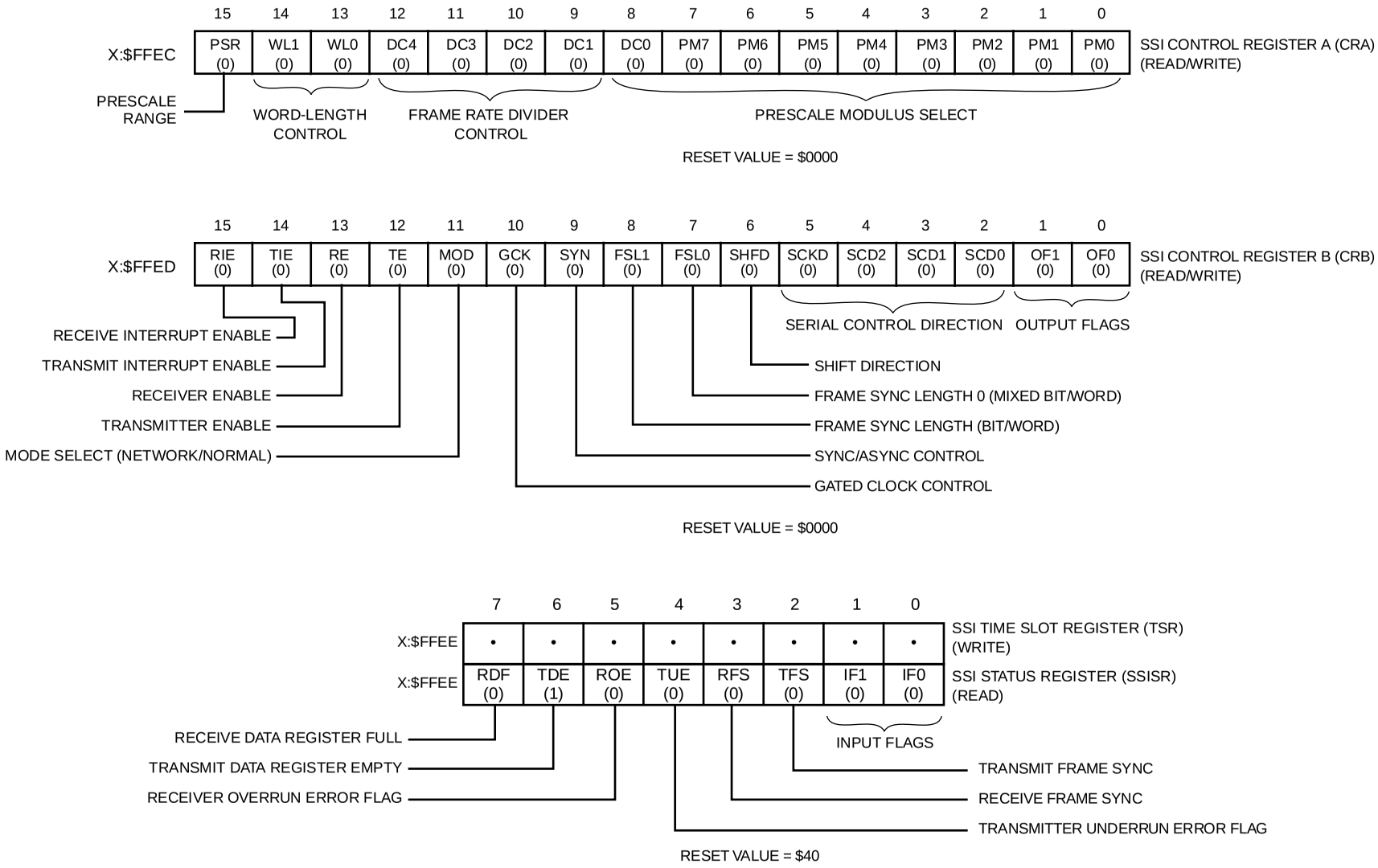
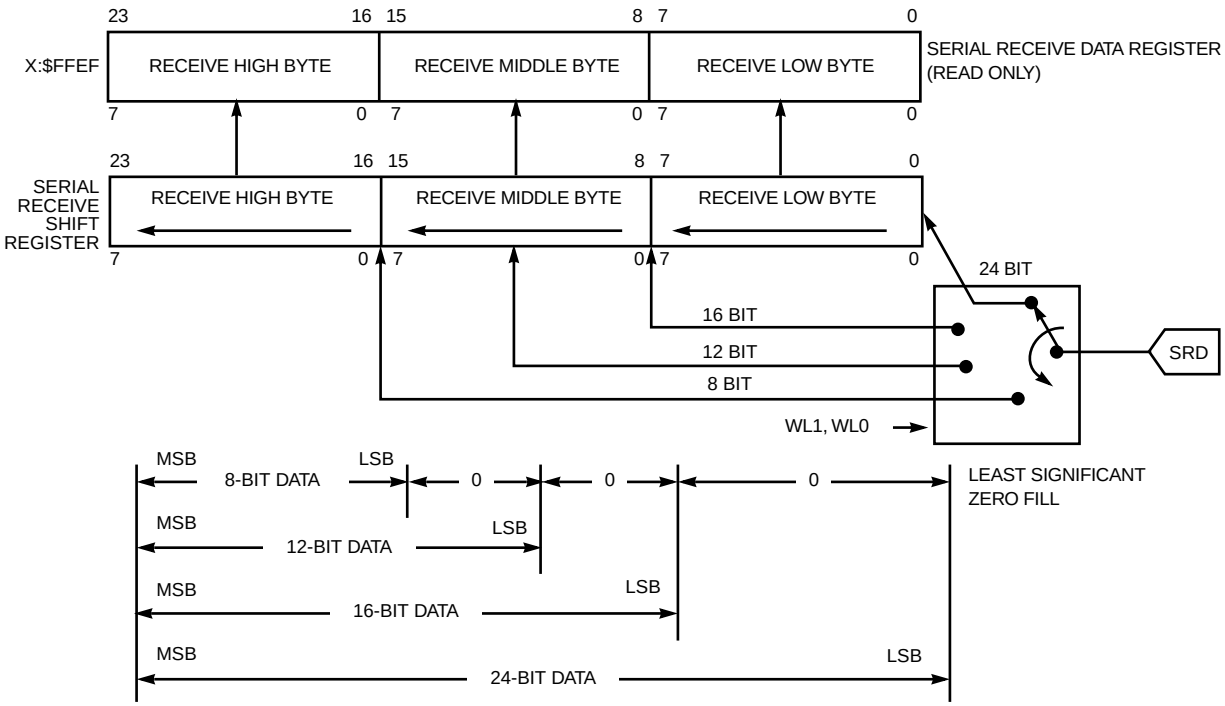
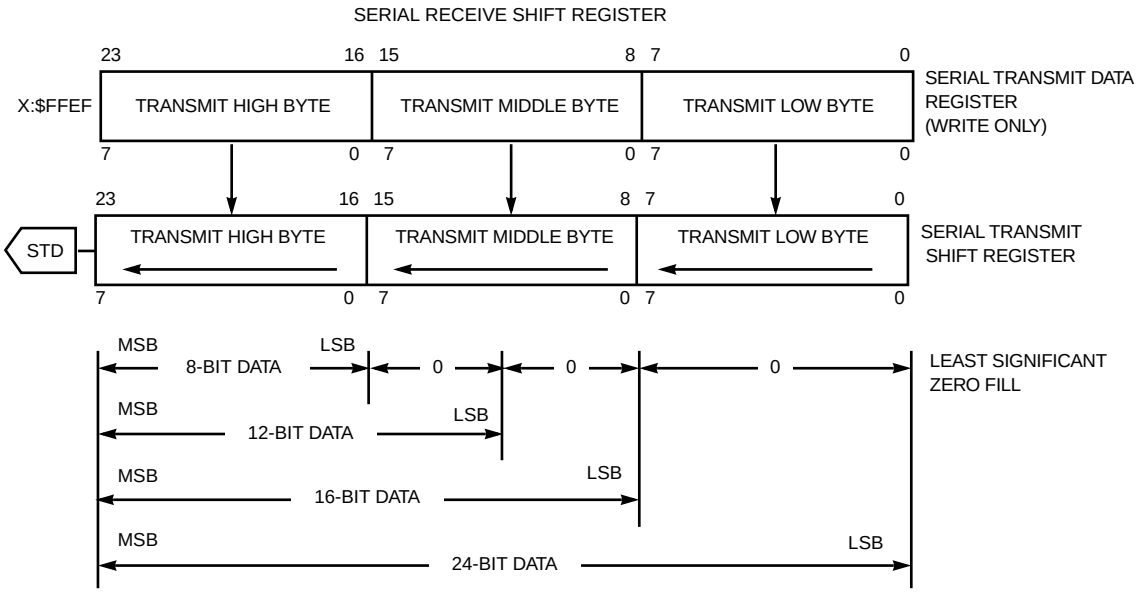


Figure 11-43 SSI Programming Model — Control and Status Registers



NOTES:
1. Data is received MSB first if SHFD = 0.
2. Compatible with fractional format.

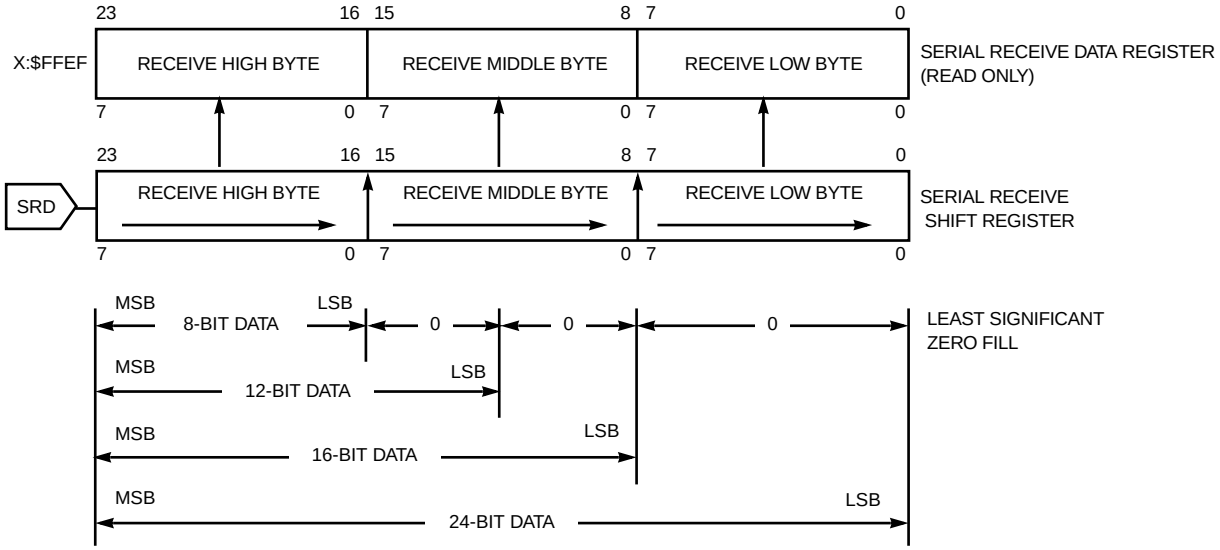
(a) Receive Registers for SHFD = 0



NOTES:
1. Data is sent MSB first if SHFD = 0.
2. Compatible with fractional format.

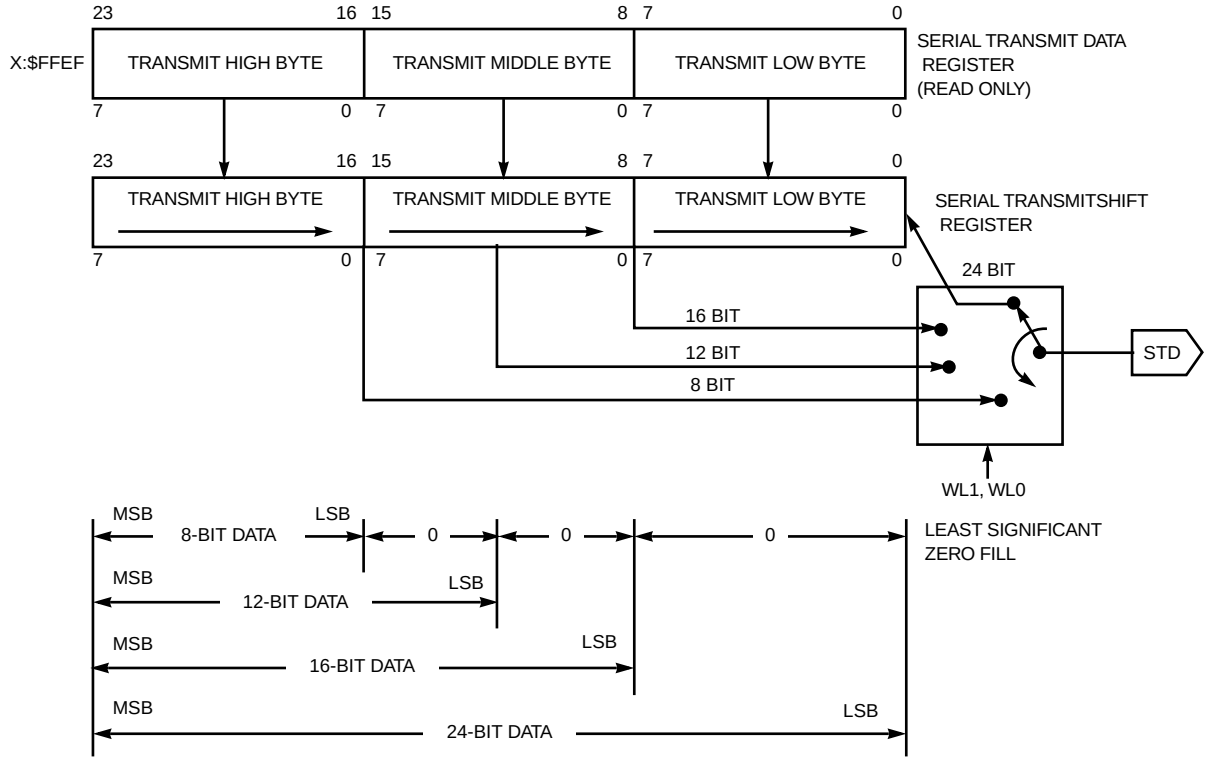
(b) Transmit Registers for SHFD = 0

Figure 11-44 SSI Programming Model (Sheet 1 of 2)



NOTES:
1. Data is received LSB first if SHFD = 1.
2. Compatible with fractional format.

(c) Receive Registers for SHFD = 1



NOTES:
1. Data is received LSB first if SHFD = 1.
2. compatible with fractional format.

(d) Transmit Registers for SHFD = 1

Figure 11-44 SSI Programming Model (Sheet 2 of 2)

for the serial control pins are in the SSI CRB. Interrupt enable bits for each data register interrupt are provided in this control register. When read by the DSP, CRB appears on the two low-order bytes of the 24-bit word, and the high-order byte reads as zeros. Operating modes are also selected in this register. Hardware and software reset clear all the bits in the CRB. The relationships between the SSI pins (SC0, SC1, SC2, and SCK) and some of the CRB bits are summarized in Tables 11-4, 11-6, and 11-7. The SSI CRB bits are described in the following paragraphs.

11.3.2.2.1 CRB Serial Output Flag 0 (OF0) Bit 0. When the SSI is in the synchronous clock mode and the serial control direction zero bit (SCD0) is set, indicating that the SC0 pin is an output, then data present in OF0 will be written to SC0 at the beginning of the frame in normal mode or at the beginning of the next time slot in network mode. Hardware and software reset clear OF0.

11.3.2.2.2 CRB Serial Output Flag 1 (OF1) Bit 1. When the SSI is in the synchronous clock mode and the serial control direction one (SCD1) bit is set, indicating that the SC1 pin is an output, then data present in OF1 will be written to the SC1 pin at the beginning of the frame in normal mode or at the beginning of the next time slot in network mode (see 11.3.7 Operating Modes – Normal, Network, and On-Demand).

The normal sequence for setting output flags when transmitting data is to poll TDE (TX empty), to first write the flags, and then write the transmit data to the TX register. OF0 and OF1 are double buffered so that the flag states appear on the pins when the TX data is transferred to the transmit shift register (i.e., the flags are synchronous with the data). Hardware and software reset clear OF1.

NOTE

The optional serial output pins (SC0, SC1, and SC2) are controlled by the frame timing and are not affected by TE or RE.

11.3.2.2.3 CRB Serial Control 0 Direction (SCD0) Bit 2. SCD0 controls the direction of the SC0 I/O line. When SCD0 is cleared, SC0 is an input; when SCD0 is set, SC0 is an output (see Tables 11-4, 11-5, and Figure 11-45). Hardware and software reset clear SCD0.

11.3.2.2.4 CRB Serial Control 1 Direction (SCD1) Bit 3. SCD1 controls the direction of the SC1 I/O line. When SCD1 is cleared, SC1 is an input; when SCD1 is set, SC1 is an output (see Tables 11-4, 11-5 and Figure 11-45). Hardware and software reset clear SCD1.

11.3.2.2.5 CRB Serial Control 2 Direction (SCD2) Bit 4. SCD2 controls the direction of the SC2 I/O line. When SCD2 is cleared, SC2 is an input; when SCD2 is set, SC2 is an output (see Tables 11-4, 11-5, and Figure 11-45). Hardware and software reset clear SCD2.

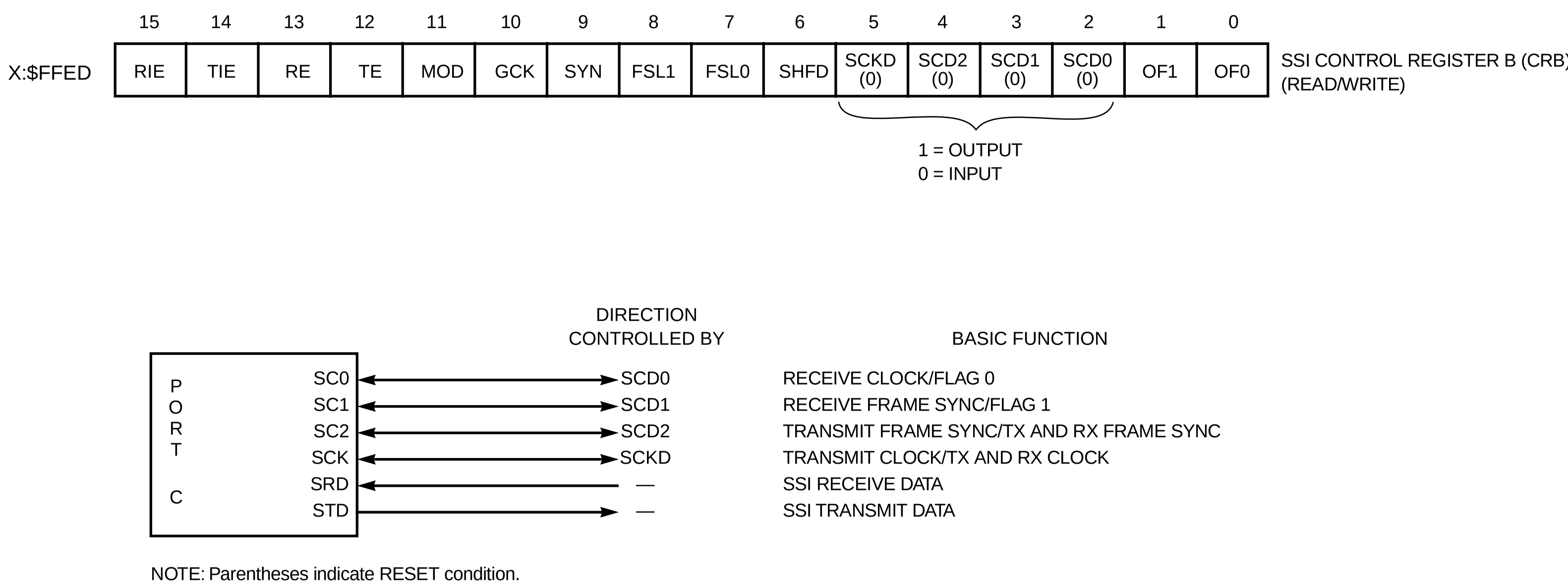


Figure 11-45 Serial Control, Direction Bits

11.3.2.2.6 CRB Clock Source Direction (SCKD) Bit 5. SCKD selects the source of the

clock signal used to clock the transmit shift register in the asynchronous mode and both the transmit shift register and the receive shift register in the synchronous mode. When SCKD is set, the internal clock source becomes the bit clock for the transmit shift register and word length divider and is the output on the SCK pin. When SCKD is cleared, the clock source is external; the internal clock generator is disconnected from the SCK pin, and an external clock source may drive this pin. Hardware and software reset clear SCKD.

11.3.2.2.7 CRB Shift Direction (SHFD) Bit 6. This bit causes the transmit shift register to shift data out MSB first when SHFD equals zero or LSB first when SHFD equals one. Receive data is shifted in MSB first when SHFD equals zero or LSB first when SHFD equals one. Hardware reset and software reset clear SHFD.

11.3.2.2.8 CRB Frame Sync Length (FSL0 and FSL1) Bits 7 and 8. These bits select the type of frame sync to be generated or recognized. If FSL1 equals zero and FSL0 equals zero, a word-length frame sync is selected for both TX and RX that is the length of the data word defined by bits WL1 and WL0. If FSL1 equals one and FSL0 equals zero, a 1-bit clock period frame sync is selected for both TX and RX. When FSL0 equals one, the TX and RX frame syncs are different lengths. Hardware reset and software reset clear FSL0 and FSL1.

FSL1	FSL0	Frame Sync Length
0	0	WL bit clock for both TX/RX
0	1	One-bit clock for TX and WL bit clock for RX
1	0	One-bit clock for both TX/RX
1	1	One-bit clock for RX and WL bit clock for TX

11.3.2.2.9 CRB Sync/Async (SYN) Bit 9. SYN controls whether the receive and transmit functions of the SSI occur synchronously or asynchronously with respect to each other. When SYN is cleared, asynchronous mode is chosen and separate clock and frame sync signals are used for the transmit and receive sections. When SYN is set, synchronous mode is chosen and the transmit and receive sections use common clock and frame sync signals. Hardware reset and software reset clear SYN.

11.3.2.2.10 CRB Gated Clock Control (GCK) Bit 10. GCK is used to select between a continuously running data clock or a clock that runs only when there is data to be sent in the transmit shift register. When GCK is cleared, a continuous clock is selected; when GCK is set, the clock will be gated. Hardware reset and software reset clear GCK.

NOTE

For gated clock mode with externally generated bit clock, internally generated frame sync is **not** defined.

11.3.2.2.11 CRB SSI Mode Select (MOD) Bit 11

MOD selects the operational mode of the SSI. When MOD is cleared, the normal mode is selected; when MOD is set, the network mode is selected. In the normal mode, the frame rate divider determines the word transfer rate – one word is transferred per frame sync during the frame sync time slot. In network mode, a word is (possibly) transferred every time slot. For more details, see 11.3.3 OPERATIONAL MODES AND PIN DEFINITIONS. Hardware and software reset clear MOD.

11.3.2.2.12 CRB SSI Transmit Enable (TE) Bit 12

TE enables the transfer of data from TX to the transmit shift register. When TE is set and a frame sync is detected, the transmit portion of the SSI is enabled for that frame. When TE is cleared, the transmitter will be disabled after completing transmission of data currently in the SSI transmit shift register. The serial output is three-stated, and any data present in TX will not be transmitted (i.e., data can be written to TX with TE cleared; TDE will be cleared, but data will not be transferred to the transmit shift register).

The normal mode transmit enable sequence is to write data to TX or TSR before setting TE. The normal transmit disable sequence is to clear TE and TIE after TDE equals one.

In the network mode, the operation of clearing TE and setting it again will disable the transmitter after completing transmission of the current data word until the beginning of the next frame. During that time period, the STD pin will remain in the high-impedance state. Hardware reset and software reset clear TE.

The on-demand mode transmit enable sequence can be the same as the normal mode, or TE can be left enabled.

Note: TE does not inhibit TDE or transmitter interrupts. TE does not affect the generation of frame sync or output flags.

11.3.2.2.13 CRB SSI Receive Enable (RE) Bit 13

When RE is set, the receive portion of the SSI is enabled. When this bit is cleared, the receiver will be disabled by inhibiting data transfer into RX. If data is being received while this bit is cleared, the remainder of the word will be shifted in and transferred to the SSI receive data register.

RE must be set in the normal mode and on-demand mode to receive data. In network mode, the operation of clearing RE and setting it again will disable the receiver after reception of the current data word until the beginning of the next data frame. Hardware and software reset clear RE.

Note: RE does not inhibit RDF or receiver interrupts. RE does not affect the gener-

ation of a frame sync.

11.3.2.2.14 CRB SSI Transmit Interrupt Enable (TIE) Bit 14

The DSP will be interrupted when TIE and the TDE flag in the SSI status register is set. When TIE is cleared, this interrupt is disabled. However, the TDE bit will always indicate the transmit data register empty condition even when the transmitter is disabled with the TE bit. Writing data to TX or TSR will clear TDE, thus clearing the interrupt. Hardware and software reset clear RE.

There are two transmit data interrupts that have separate interrupt vectors:

1. Transmit data with exceptions – This interrupt is generated on the following condition:
   TIE=1, TDE=1, and TUE=1
2. Transmit data without exceptions – This interrupt is generated on the following condition:
   TIE=1, TDE=1, and TUE=0

See SECTION 8 PROCESSING STATES for more information on exceptions.

11.3.2.2.15 CRB SSI Receive Interrupt Enable (RIE) Bit 15

When RIE is set, the DSP will be interrupted when RDF in the SSI status register is set. When RIE is cleared, this interrupt is disabled. However, the RDF bit still indicates the receive data register full condition. Reading the receive data register will clear RDF, thus clearing the pending interrupt. Hardware and software reset clear RIE.

There are two receive data interrupts that have separate interrupt vectors:

1. Receive data with exceptions – This interrupt is generated on the following condition:
   RIE=1, RDF=1, and ROE=1
2. Receive data without exceptions – This interrupt is generated on the following condition:
   RIE=1, RDF=1, and ROE=0

See SECTION 8 PROCESSING STATES for more information on exceptions.

11.3.2.3 SSI Status Register (SSISR)

The SSISR is an 8-bit read-only status register used by the DSP to interrogate the status and serial input flags of the SSI. When the SSISR is read to the internal data bus, the register contents occupy the low-order byte of the data bus, and the high-order portion is zero filled. The status bits are described in the following paragraphs.

11.3.2.3.1 SSISR Serial Input Flag 0 (IF0) Bit 0

The SSI latches data present on the SC0 pin during reception of the first received bit after frame sync is detected. IF0 is updated with this data when the receive shift register is transferred into the receive data register. The IF0 bit is enabled only when SCD0 is cleared and SYN is set, indicating that SC0 is an input and the synchronous mode is selected (see Table 11-4); otherwise, IF0 reads as a zero when it is not enabled. Hardware, software, SSI individual, and STOP reset clear IF0.

11.3.2.3.2 SSISR Serial Input Flag 1 (IF1) Bit 1

The SSI latches data present on the SC1 pin during reception of the first received bit after frame sync is detected. The IF1 flag is updated with the data when the receiver shift register is transferred into the receive data register. The IF1 bit is enabled only when SCD1 is cleared and SYN is set, indicating that SC1 is an input and the synchronous mode is selected (see Table 11- 4); otherwise, IF1 reads as a zero when it is not enabled. Hardware, software, SSI individual, and STOP reset clear IF1.

11.3.2.3.3 SSISR TRANSMIT FRAME SYNC FLAG (TFS) Bit 2

When set, TFS indicates that a transmit frame sync occurred in the current time slot. TFS is set at the start of the first time slot in the frame and cleared during all other time slots. Data written to the transmit data register during the time slot when TFS is set will be transmitted (in network mode) during the second time slot in the frame. TFS is useful in network mode to identify the start of a frame.

Note: In normal mode, TFS will always read as a one when transmitting data because there is only one time slot per frame – the “frame sync” time slot.

TFS, which is cleared by hardware, software, SSI individual, or STOP reset, is not affected by TE.

11.3.2.3.4 SSISR Receive Frame Sync Flag (RFS) Bit 3

When set, RFS indicates that a receive frame sync occurred during reception of the word in the serial receive data register. This indicates that the data word is from the first time slot in the frame. When RFS is clear and a word is received, it indicates (only in the network mode) that the frame sync did not occur during reception of that word.

Note: In normal mode, RFS will always read as a one when reading data because there is only one time slot per frame – the “frame sync” time slot.

RFS, which is cleared by hardware, software, SSI individual, or STOP reset, is not affected by RE.

11.3.2.3.5 SSISR Transmitter Underrun Error Flag (TUE) Bit 4

TUE is set when the serial transmit shift register is empty (no new data to be transmitted) and a transmit time slot occurs. When a transmit underrun error occurs, the previous data (which is still present in the TX) will be retransmitted.

In the normal mode, there is only one transmit time slot per frame. In the network mode, there can be up to 32 transmit time slots per frame.

TUE does not cause any interrupts; however, TUE does cause a change in the interrupt vector used for transmit interrupts so that a different interrupt handler may be used for a transmit underrun condition. If a transmit interrupt occurs with TUE set, the transmit data with exception status interrupt will be generated; if a transmit interrupt occurs with TUE clear, the transmit data without errors interrupt will be generated.

Hardware, software, SSI individual, and STOP reset clear TUE. TUE is also cleared by reading the SSISR with TUE set, followed by writing TX or TSR.

11.3.2.3.6 SSISR Receiver Overrun Error Flag (ROE) Bit 5

This flag is set when the serial receive shift register is filled and ready to transfer to the receiver data register (RX) and RX is already full (i.e., RDF=1). The receiver shift register is not transferred to RX. ROE does not cause any interrupts; however, ROE does cause a change in the interrupt vector used for receive interrupts so that a different interrupt handler may be used for a receive error condition. If a receive interrupt occurs with ROE set, the receive data with exception status interrupt will be generated; if a receive interrupt occurs with ROE clear, the receive data without errors interrupt will be generated.

Hardware, software, SSI individual, and STOP reset clear ROE. ROE is also cleared by reading the SSISR with ROE set, followed by reading the RX. Clearing RE does not affect ROE.

11.3.2.3.7 SSISR SSI Transmit Data Register Empty (TDE) Bit 6

This flag is set when the contents of the transmit data register are transferred to the transmit shift register; it is also set for a disabled time slot period in network mode (as if data were being transmitted after the TSR was written). Thirdly, it can be set by the hardware, software, SSI individual, or STOP reset. When set, TDE indicates that data should be written to the TX or to the time slot register (TSR). TDE is cleared when the DSP writes to the transmit data register or when the DSP writes to the TSR to disable transmission of the next time slot. If TIE is set, a DSP transmit data interrupt request will be issued when TDE is set. The vector of the interrupt will depend on the state of the transmitter underrun bit.

11.3.2.3.8 SSISR SSI Receive Data Register Full (RDF) Bit 7

RDF is set when the contents of the receive shift register are transferred to the receive data register. RDF is cleared when the DSP reads the receive data register or cleared by hardware, software, SSI individual, or STOP reset. If RIE is set, a DSP receive data interrupt request will be issued when RDF is set. The vector of the interrupt request will depend on the state of the receiver overrun bit.

11.3.2.3.9 SSI Receive Shift Register

This 24-bit shift register receives the incoming data from the serial receive data pin. Data is shifted in by the selected (internal/external) bit clock when the associated frame sync I/O (or gated clock) is asserted. Data is assumed to be received MSB first if SHFD equals zero and LSB first if SHFD equals one. Data is transferred to the SSI receive data register after 8, 12, 16, or 24 bits have been shifted in, depending on the word-length control bits in the CRA (see Figure 11-46).

11.3.2.3.10 SSI Receive Data Register (RX)

RX is a 24-bit read-only register that accepts data from the receive shift register as it becomes full. The data read will occupy the most significant portion of the receive data register (see Figure 11-46). The unused bits (least significant portion) will read as zeros. The DSP is interrupted whenever RX becomes full if the associated interrupt is enabled.

11.3.2.3.11 SSI Transmit Shift Register

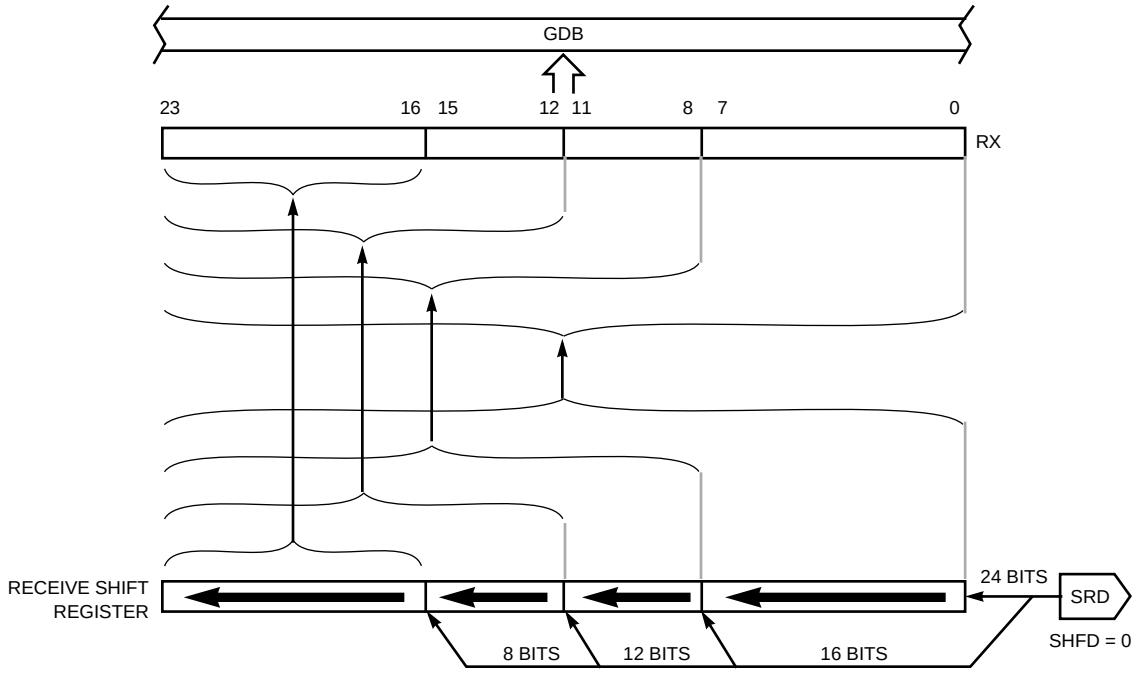
This 24-bit shift register contains the data being transmitted. Data is shifted out to the serial transmit data pin by the selected (internal/external) bit clock when the associated frame sync I/O (or gated clock) is asserted. The number of bits shifted out before the shift register is considered empty and may be written to again can be 8, 12, 16, or 24 bits (determined by the word-length control bits in CRA). The data to be transmitted occupies the most significant portion of the shift register. The unused portion of the register is ignored. Data is shifted out of this register MSB first if SHFD equals zero and LSB first if SHFD equals one (see Figure 11-47).

11.3.2.3.12 SSI Transmit Data Register (TX)

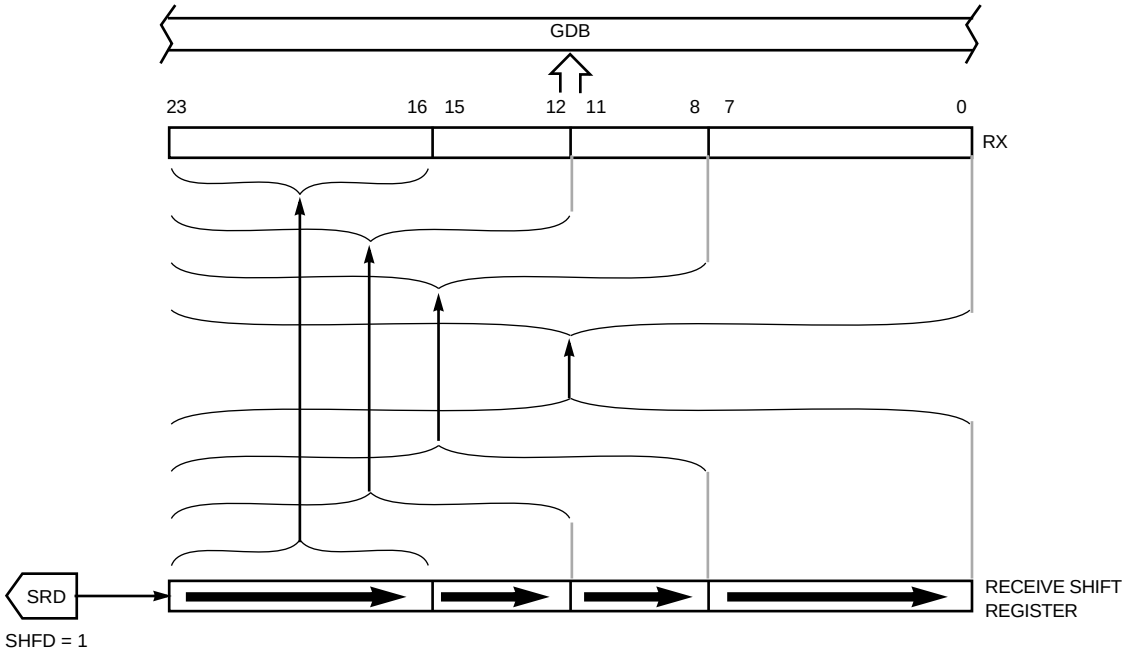
TX is a 24-bit write-only register. Data to be transmitted is written into this register and is automatically transferred to the transmit shift register. The data written (8, 12, 16, or 24 bits) should occupy the most significant portion of TX (see Figure 11-47). The unused bits (least significant portion) of TX are don’t care bits. The DSP is interrupted whenever TX becomes empty if the transmit data register empty interrupt has been enabled.

11.3.2.3.13 Time Slot Register (TSR)

TSR is effectively a null data register that is used when the data is not to be transmitted



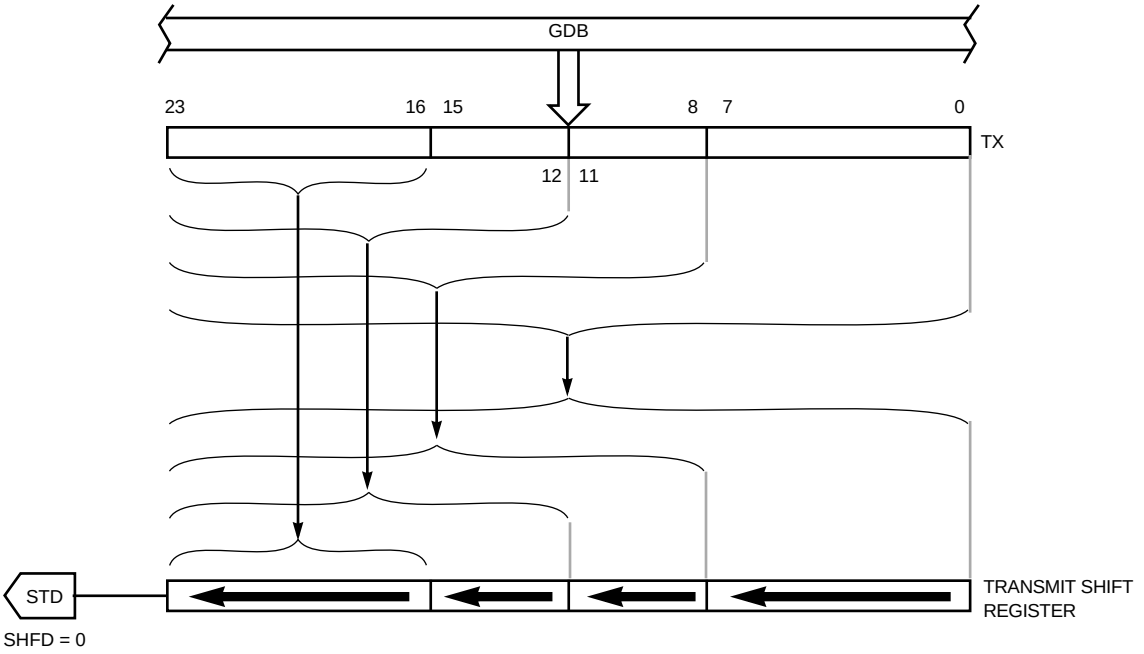
(a) SHFD = 0



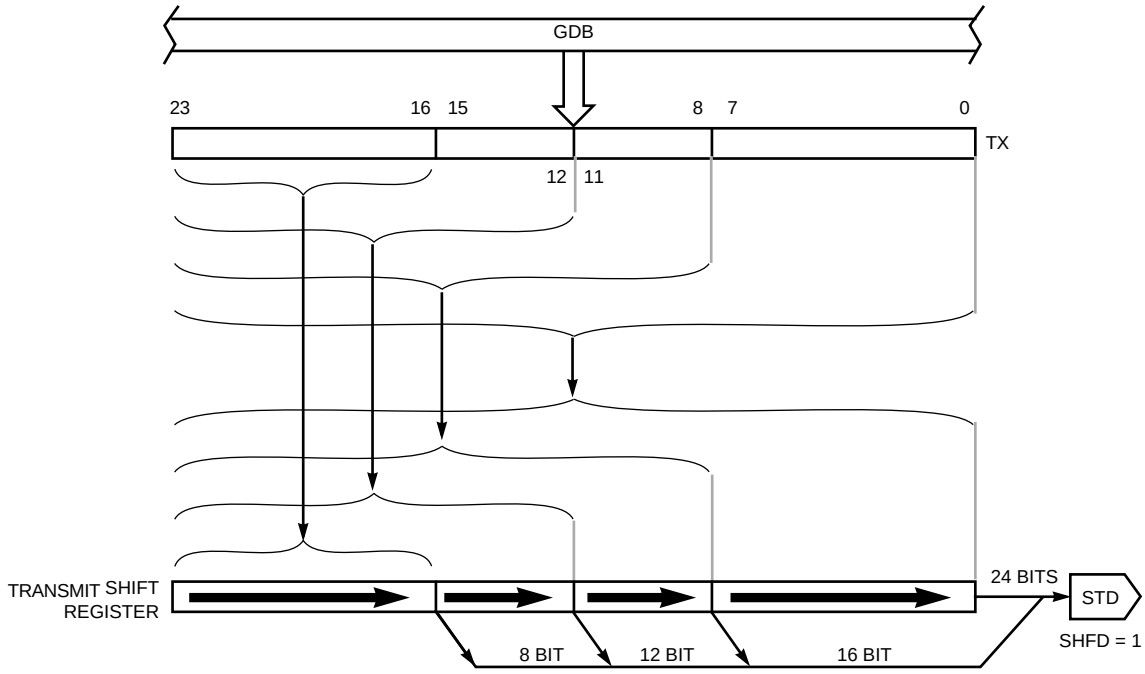
(b) SHFD = 1

Figure 11-46 Receive Data Path

in the available transmit time slot. For the purposes of timing, TSR is a write-only register



(a) SHFD = 0



(b) SHFD = 1

Figure 11-47 Transmit Data Path

that behaves like an alternative transmit data register, except that, rather than transmitting data, the transmit data pin is in the high-impedance state for that time slot.

11.3.3 Operational Modes and Pin Definitions

Tables 11-6 and 11-7 completely describe the SSI operational modes and pin definitions (Table 11-4 is a simplified version of these tables). The operational modes are as follows:

1. Continuous Clock
   Mode 1 – Normal with Internal Frame Sync

Table 11-6 Mode and Pin Definition Table — Continuous Clock

Control Bits								Mode		SC0		SC1		SC2		SCK	
MOD	GCLK	SYN	SCD2	SCD1	SCD0	SCKD	DC4-DC0	TX	RX	In	Out	In	Out	In	Out	In	Out
0	0	0	1	1	X	X	X	1	1	RXC	RXC	—	FSR	—	FST	TXC	TXC
0	0	1	1	X	X	X	X	1	1	F0	F0	F1	F1	—	FS*	*XC	*XC
1	0	0	1	1	X	X	1	2	2	RXC	RXC	—	FSR	—	FST	TXC	TXC
1	0	1	1	X	X	X	1	2	2	F0	F0	F1	F1	—	FS*	*XC	*XC
0	0	0	0	1	X	X	X	3	1	RXC	RXC	—	FSR	FST	—	TXC	TXC
0	0	0	1	0	X	X	X	1	3	RXC	RXC	FSR	—	—	FST	TXC	TXC
0	0	0	0	0	X	X	X	3	3	RXC	RXC	FSR	—	FST	—	TXC	TXC
0	0	1	0	X	X	X	X	3	3	F0	F0	F1	F1	FS*	—	*XC	*XC
1	0	0	0	1	X	X	X	4	2	RXC	RXC	—	FSR	FST	—	TXC	TXC
1	0	0	1	0	X	X	1	2	4	RXC	RXC	FSR	—	—	FST	TXC	TXC
1	0	0	0	0	X	X	X	4	4	RXC	RXC	FSR	—	FST	—	TXC	TXC
1	0	1	0	X	X	X	X	4	4	F0	F0	F1	F1	FS*	—	*XC	*XC
1	0	0	1	1	X	X	0	8	2	RXC	RXC	—	FSR	—	FST	TXC	TXC
1	0	1	1	X	X	X	0	8	9	F0	F0	F1	F1	—	FS*	*XC	*XC
1	0	0	1	0	X	X	0	8	4	RXC	RXC	FSR	—	—	FST	TXC	TXC

DC4-DC0 = 0 means that bits DC4 = 0, DC3 = 0, DC2 = 0, DC1 = 0, and DC0 = 0.
DC4-DC0 = 1 means that bits DC4-DC0≠0.
TXC — Transmitter Clock
RXC — Receiver Clock
*XC — Transmitter/Receiver Clock (Synchronous Operation)
FST — Transmitter Frame Sync
FSR — Receiver Frame Sync
FS* — Transmitter/Receiver Frame Sync (Synchronous Operation)
F0 — Flag 0
F1 — Flag 1

Mode 2 – Network with Internal Frame Sync
Mode 3 – Normal with External Frame Sync
Mode 4 – Network with External Frame Sync

2. Gated Clock
   Mode 5 – External Gated Clock
   Mode 6 – Normal with Internal Gated Clock
   Mode 7 – Network with Internal Gated Clock

3. Special Case (Both Gated and Continuous Clock)
   Mode 8 – On-Demand Mode (Transmitter Only)
   Mode 9 – Receiver Follows Transmitter Clocking

11.3.4 Registers After Reset

Hardware or software reset clears the port control register bits, which configure all I/O as general-purpose input. The SSI will remain in reset while all SSI pins are programmed as general-purpose I/O (CC8–CC3=0) and will become active only when at least one of the SSI I/O pins is programmed as not general-purpose I/O. Table 11-8 shows how each type of reset affects each SSI register bit.

Table 11-7 Mode and Pin Definition Table — Gated Clock

Control Bits								Mode		SC0		SC1		SC2		SCK	
MOD	GCLK	SYN	SCD2	SCD1	SCD0	SCKD	DC4-DC0	TX	RX	In	Out	In	Out	In	Out	In	Out
0	1	0	X	X	1	1	X	6	6	—	RXC	?	FSR	?	FST	—	TXC
0	1	1	X	X	X	1	X	6	6	F0	F0	F0	F1	?	FS*	—	*XC
0	1	0	X	X	1	0	X	5	6	—	RXC	?	FSR	?	?	TXC	—
0	1	0	X	X	0	0	X	5	5	RXC	—	?	?	?	?	TXC	—
0	1	1	X	X	X	0	X	5	5	F0	F0	F1	F1	?	?	*XC	—
1	1	0	X	X	1	1	0	8	7	—	RXC	?	FSR	?	FST	—	TXC
1	1	0	X	X	0	1	0	8	5	RXC	—	?	?	?	FST	—	TXC
1	1	1	X	X	X	1	0	8	9	F0	F0	F1	F1	?	FS*	—	*XC
0	1	0	X	X	0	1	X	6	5	RXC	—	?	?	?	FST	—	TXC

DC4–DC0=0 means that bits DC4=0, DC3=0, DC2=0, DC1=0, and DC0=0.
TXC – Transmitter Clock
RXC – Receiver Clock
*XC – Transmitter/Receiver Clock (Synchronous Operation)
FST – Transmitter Frame Sync
FSR – Receiver Frame Sync
FS* – Transmitter/Receiver Frame Sync (Synchronous Operation)
F0 – Flag 0
F1 – Flag 1
? – Undefined

Table 11-8 SSI Registers After Reset

Register Name	Register Data	Bit Number	Reset			
			HW Reset	SW Reset	Individual Reset	ST Reset
CRA	PSR	15	0	0	–	–
	WL(2–0)	13,14	0	0	–	–
	DC(4–0)	8–12	0	0	–	–
	PM(7–0)	0–7	0	0	–	–
CRB	RIE	15	0	0	–	–
	TIE	14	0	0	–	–
	RE	13	0	0	–	–
	TE	12	0	0	–	–
	MOD	11	0	0	–	–
	GCK	10	0	0	–	–
	SYN	9	0	0	–	–
	FSL1	8	0	0	–	–
	FSL0	7	0	0	–	–
	SHFD	6	0	0	–	–
	SCKD	5	0	0	–	–
	SCD(2–0)	2–4	0	0	–	–
	OF(1–0)	0,1	0	0	–	–
SSISR	RDF	7	0	0	0	0
	TDE	6	1	1	1	1
	ROE	5	0	0	0	0
	TUE	4	0	0	0	0
	RFS	3	0	0	0	0
	TFS	2	0	0	0	0
	IF(1–0)	0,1	0	0	0	0
RDR	RDR (23–0)	23–0	–	–	–	–
TDR	TDR (23–0)	23–0	–	–	–	–
RSR	RDR (23–0)	23–0	–	–	–	–
TSR	RDR (23–0)	23–0	–	–	–	–

NOTES:
1. RSR – SSI receive shift register
2. TSR – SSI transmit shift register
3. HW – Hardware reset is caused by asserting the external pin $\overline{\text{RESET}}$.
4. SW – Software reset is caused by executing the RESET instruction.
5. IR – Individual reset is caused by SSI peripheral pins (i.e., PCC(3–8)) being configured as general-purpose I/O.
6. ST – Stop reset is caused by executing the STOP instruction.

11.3.5 SSI Initialization

The correct way to initialize the SSI is as follows:

1. Hardware, software, SSI individual, or STOP reset
2. Program SSI control registers
3. Configure SSI pins (at least one) as not general-purpose I/O

During program execution, CC8–CC3 may be cleared, causing the SSI to stop serial activity and enter the individual reset state. All status bits of the interface will be set to their reset state; however, the contents of CRA and CRB are not affected. This procedure allows the DSP program to reset each interface separately from the other internal peripherals.

The DSP program must use an SSI reset when changing the MOD, GCK, SYN, SCKD, SCD2, SCD1, or SCD0 bits to ensure proper operation of the interface. Figure 11-48 is a flowchart illustrating the three initialization steps previously listed. Figure 11-49, Figure 11-50, and Figure 11-51 provide additional detail to the flowchart.

Figure 11-51 shows the six control bits in the PCC, which select the six SSI pins as either general-purpose I/O or as SSI pins. The STD pin can only transmit data; the SRD pin can only receive data. The other four pins can be inputs or outputs, depending on how they are programmed. This programming is accomplished by setting bits in CRA and CRB as shown in Figure 11-45. The CRA (see Figure 11-49) sets the SSI bit rate clock with PSR and PM0–PM7, sets the word length with WL1 and WL0, and sets the number of words in a frame with DC0–DC4. There is a special case where DC4–DC0 equals zero (one word per frame). Depending on whether the normal or network mode is selected (MOD=0 or MOD=1, respectively), either the continuous periodic data mode is selected, or the on-demand data driven mode is selected. The continuous periodic mode requires that FSL1 equals one and FSL0 equals zero. Figure 11-50 shows the meaning of each individual bit in the CRB. These bits should be set according to the application requirements.

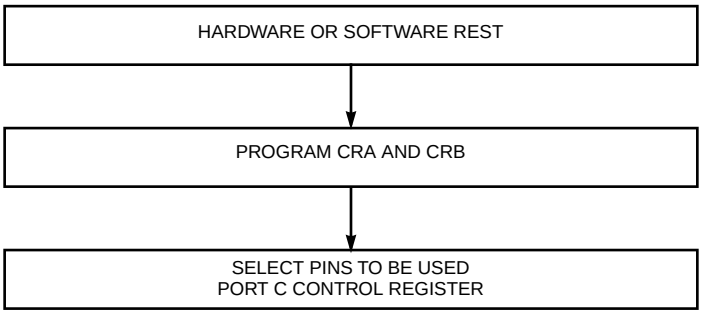
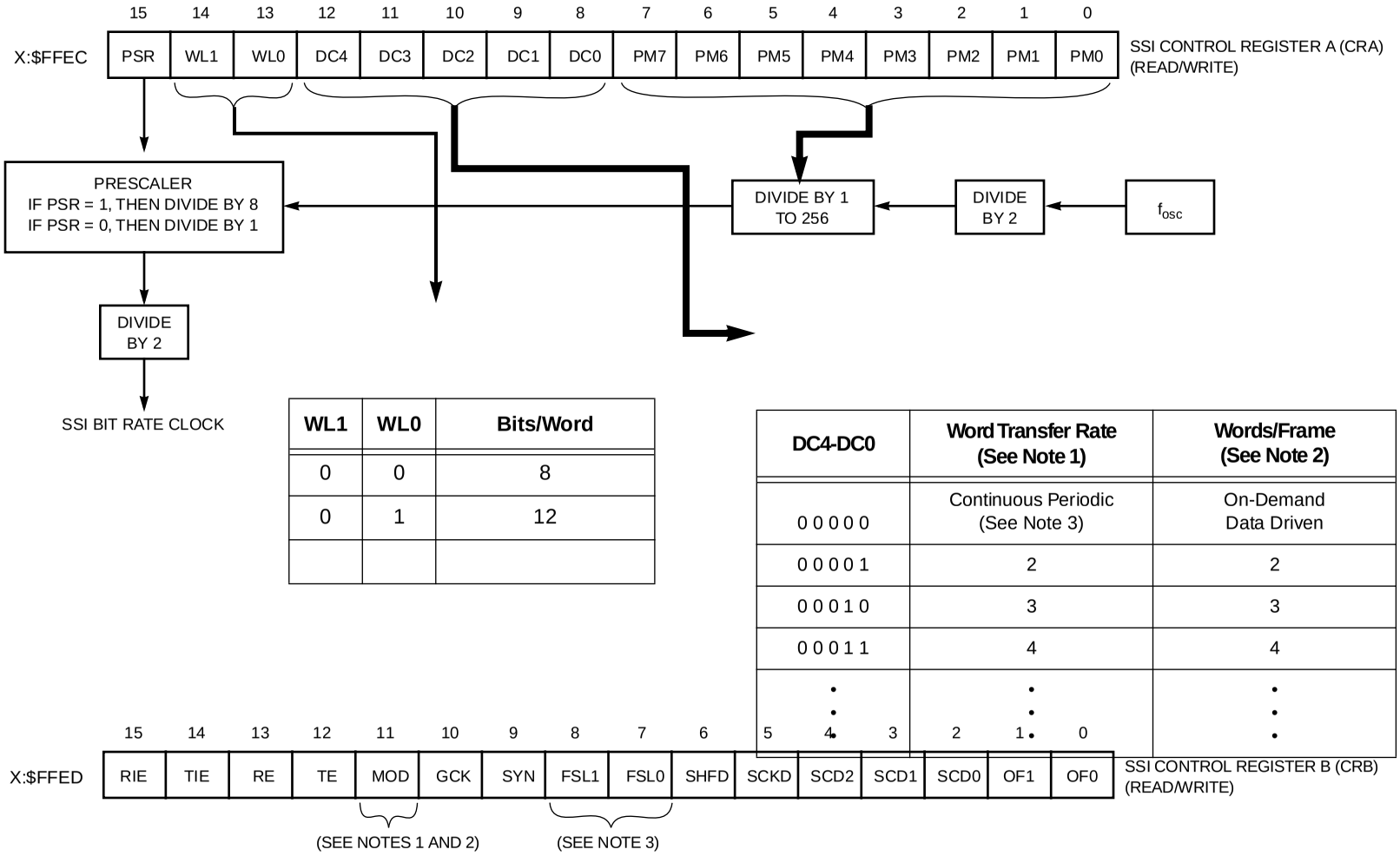


Figure 11-48 SSI Initialization Block Diagram



WL1	WL0	Bits/Word
0	0	8
0	1	12

DC4-DC0	Word Transfer Rate (See Note 1)	Words/Frame (See Note 2)
0 0 0 0 0	Continuous Periodic (See Note 3)	On-Demand Data Driven
0 0 0 0 1	2	2
0 0 0 1 0	3	3
0 0 0 1 1	4	4
• • •	• • •	• • •

NOTES:
1. NORMAL — MOD = 0
2. NETWORK — MOD = 1
3. FSL1 = 1, FSL0 = 0

Figure 11-49 SSI CRA Initialization Procedure

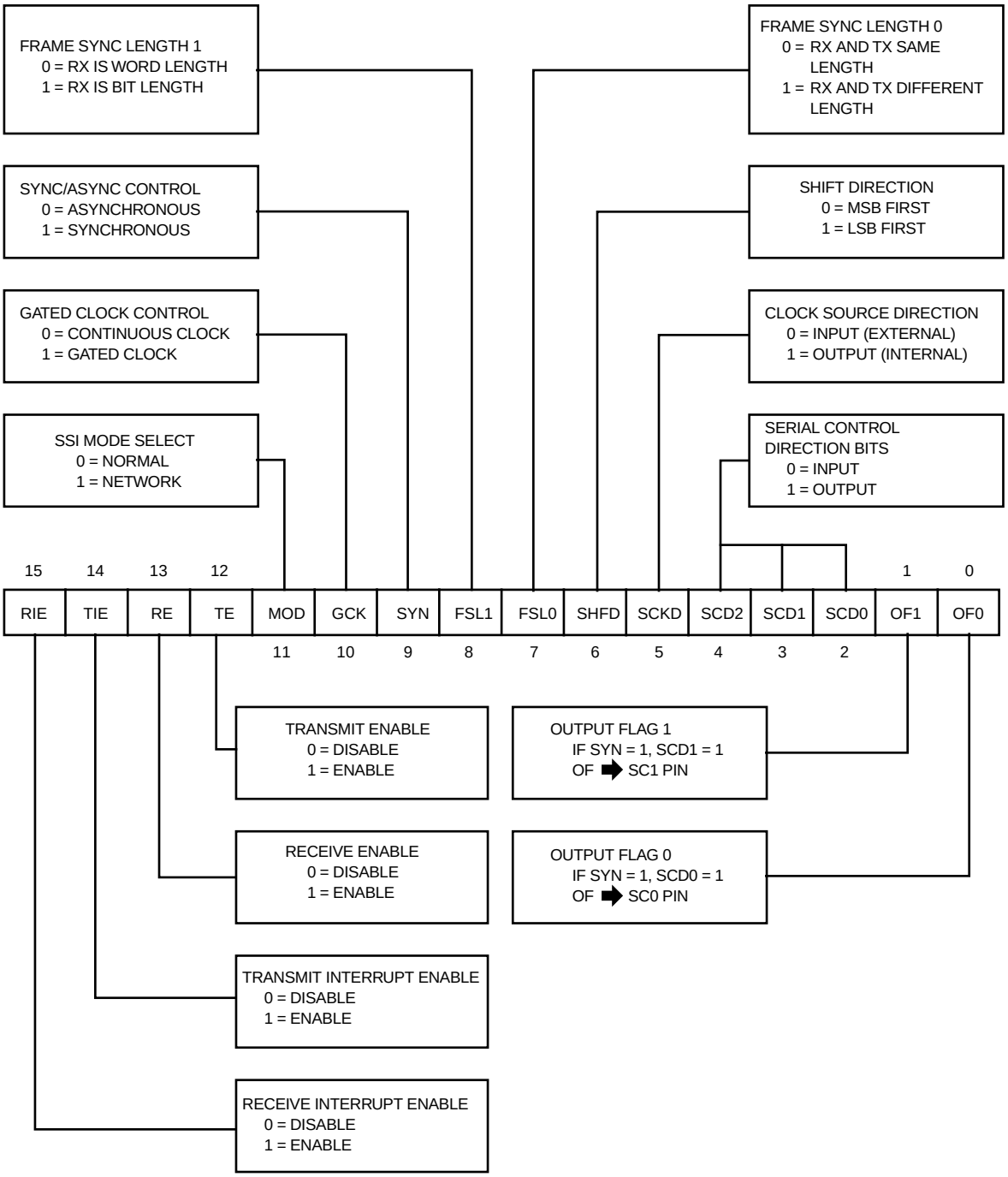
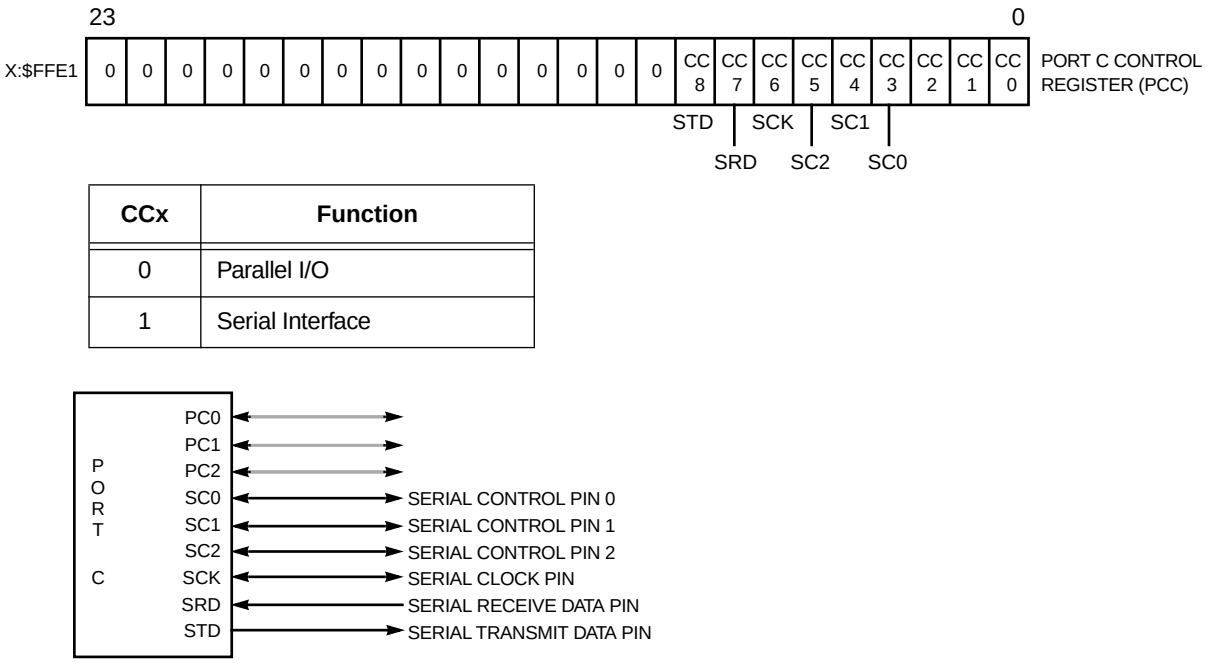


Figure 11-50 SSI CRB Initialization Procedure

Tables 11-9(a) and 11-9(b) provide a convenient listing of PSR and PM0–PM7 settings for the common data communication rates and the highest rate possible for the SSI for the chosen crystal frequencies. The crystal frequency selected for Table 11-9(a) is the one used by the DSP56000ADS board; the one selected for Table 11-9(b) is the closest one to 27 MHz that divides down to exactly 128 kHz. If an exact baud rate is required, the crystal frequency may have to be selected. Table 11-10 gives the PSR and PM0–



CCx	Function
0	Parallel I/O
1	Serial Interface

Figure 11-51 SSI Initialization Procedure

PM7 settings in addition to the required crystal frequency for three common telecommunication frequencies.

11.3.6 SSI Exceptions

The SSI can generate four different exceptions (see Figure 11-52 and Figure 11-53):

1. SSI Receive Data – occurs when the receive interrupt is enabled, the receive data register is full, and no receive error conditions exist. Reading RX clears the pending interrupt. This error-free interrupt can use a fast interrupt service routine for minimum overhead.
2. SSI Receive Data with Exception Status – occurs when the receive interrupt is enabled, the receive data register is full, and a receiver overrun error has occurred. ROE is cleared by first reading the SSISR and then reading RX.
3. SSI Transmit Data – occurs when the transmit interrupt is enabled, the transmit data register is empty, and no transmitter error conditions exist. Writing to TX or the TSR will clear this interrupt. This error-free interrupt may use a fast interrupt service routine for minimum overhead.
4. SSI Transmit Data with Exception Status – occurs when the transmit interrupt is enabled, the transmit data register is empty, and a transmitter underrun error has occurred. TUE is cleared by first reading the SSISR and then writing to TX or the TSR to clear the pending interrupt.

Table 11-9(a) SSI Baud Rates for a 20.48-MHz Crystal

Baud Rate (BPS)	PSR	PM
1000	1	$27F
2000	1	$13F
4000	1	$9F
8000	1	$4F
16K	1	$27
32K	1	$13
64K	0	$4F
128K	0	$27
5.12M	0	$00

$BPS = f_{osc} \div (4 \times (7(PSR) + 1) \times (PM + 1))$
where f_{osc}=20.48 MHz
PSR = 0 or 1
PM = 0 to $FFF

Table 11-9(b) SSI Baud Rates for a 26.624-MHz Crystal

Baud Rate (BPS)	PSR	PM
1000	1	$33F
2000	1	$19F
4000	1	$CF
8000	1	$67
16K	1	$33
32K	1	$19
64K	0	$67
128K	0	$33
6.656M	0	$00

$BPS = f_{osc} \div (4 \times (7(PSR) + 1) \times (PM + 1))$
where f_{osc}=26.624 MHz
PSR = 0 or 1
PM = 0 to $FFF

Table 11-10 Crystal Frequencies Required for Codecs

Baud Rate (BPS)	PSR	PM	Crystal Frequency
1.536M	0	$03	24.576 MHz
1.544M	0	$03	24.707 MHz
2.048M	0	$02	24.576 MHz

$BPS = f_{osc} \div (4 \times (7(PSR) + 1) \times (PM + 1))$
PSR = 0 or 1
PM = 0 to $FFF

11.3.7 Operating Modes – Normal, Network, and On-Demand

The SSI has three basic operating modes and many data/operation formats. These modes can be programmed by several bits in the SSI control registers. Table 11-11 lists the SSI operating modes and some of the typical applications in which they may be used.

The data/operation formats are selected by choosing between gated and continuous clocks, synchronization of transmitter and receiver, selection of word or bit frame sync, and whether the LSB is transferred first or last. The following paragraphs describe how to select a particular data/operation format and describe examples of normal-mode and network-mode applications. The on-demand mode is selected as a special case of the network mode.

The SSI can function as an SPI master or SPI slave, using additional logic for arbitration, which is required because the SSI interface does not perform SPI master/slave arbitration. An SPI master device always uses an internally generated clock; whereas, an SPI slave device always uses an external clock.

11.3.7.1 Data/Operation Formats

The data/operation formats available to the SSI are selected by setting or clearing con-

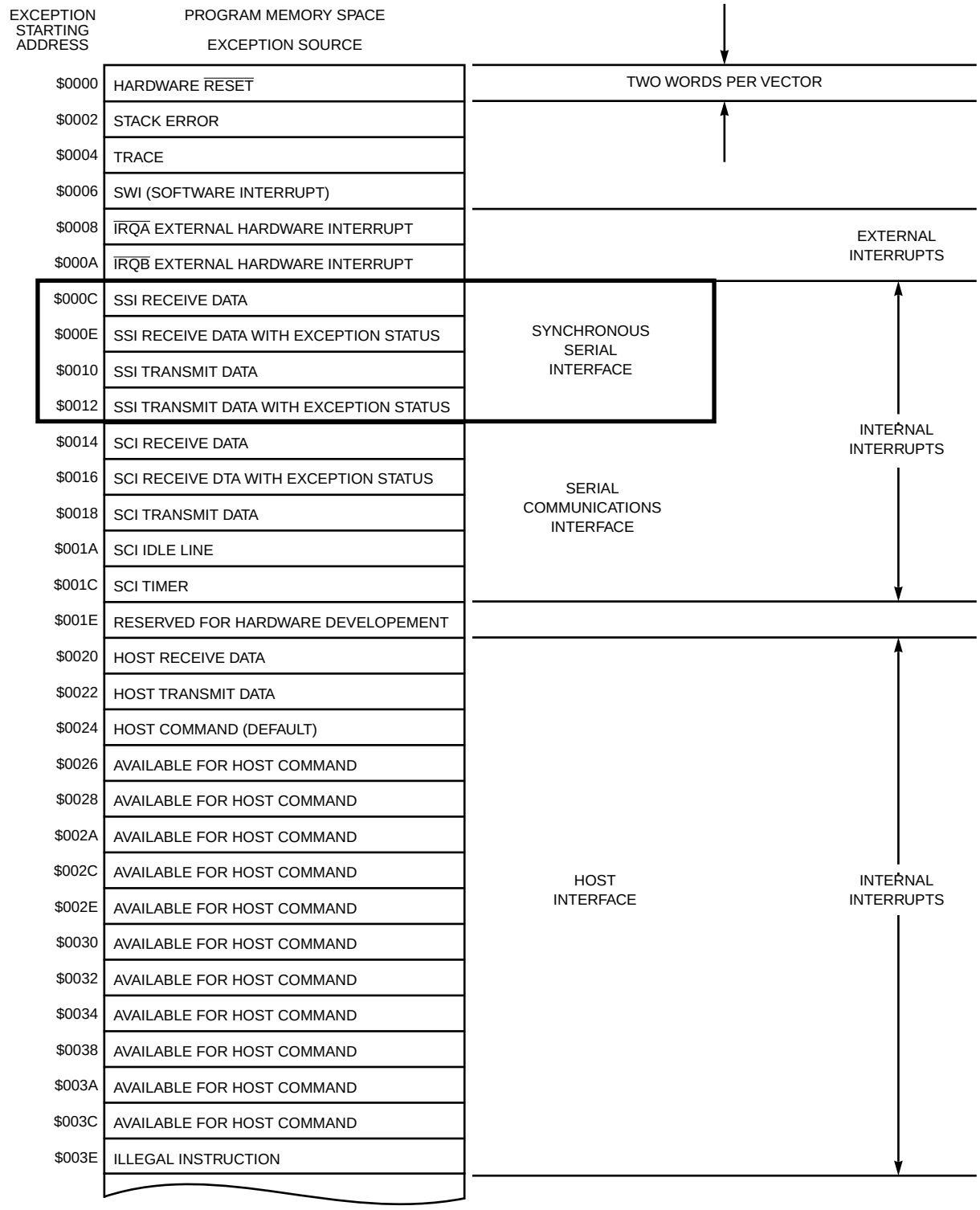


Figure 11-52 SSI Exception Vector Locations

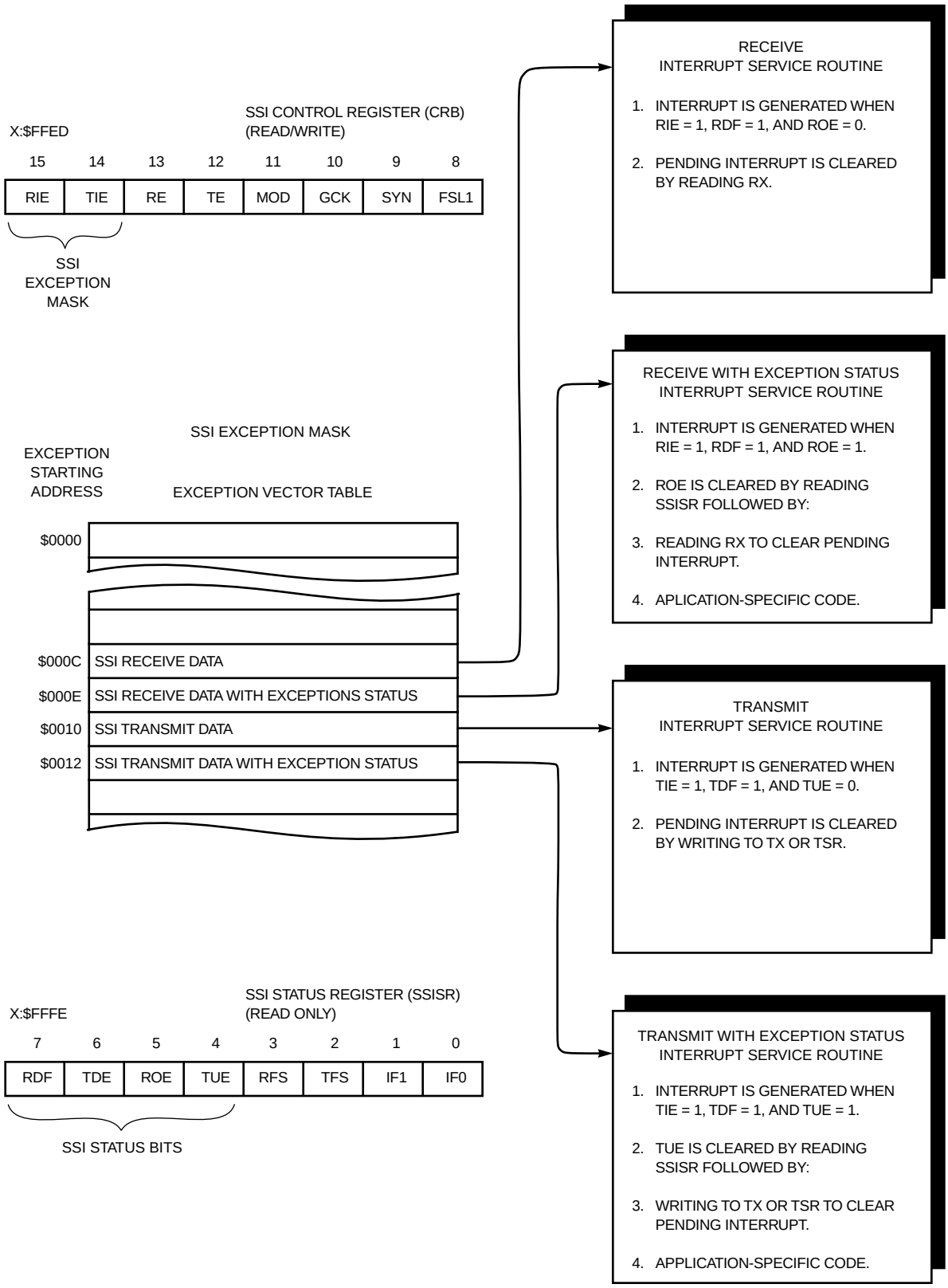


Figure 11-53 SSI Exceptions

Table 11-11 SSI Operating Modes

Operating Format	Serial Clock	TX, RX Sections	Typical Applications
Normal	Continuous	Asynchronous	Single Asynchronous Codec; Stream-Mode Channel Interface
Normal	Continuous	Synchronous	Multiple Synchronous Codecs
Normal	Gated	Asynchronous	DSP-to-DSP; Serial Peripherals (A/D,D/A)
Normal	Gated	Synchronous	SPI-Type Devices; DSP to MCU
Network	Continuous	Asynchronous	TDM Networks
Network	Continuous	Synchronous	TDM Codec Networks, TDM DSP Networks
On-Demand	Gated	Asynchronous	Parallel-to-Serial and Serial-to-Parallel Conversion
On-Demand	Gated	Synchronous	DSP to SPI Peripherals

11.3.7.1.1 Normal/Network Mode Selection

Selecting between the normal mode and network mode is accomplished by clearing or setting the MOD bit in the CRB (see Figure 11-54). For normal mode, the SSI functions with one data word of I/O per frame (see Figure 11-55). For the network mode, 2 to 32 data words of I/O may be used per frame. In either case, the transfers are periodic. The normal mode is typically used to transfer data to/from a single device. Network mode is typically used in time division multiplexed (TDM) networks of codecs or DSPs with multiple words per frame (see Figure 11-56, which shows two words in a frame with either word-length or bit-length frame sync). The frame sync shown in Figure 11-54 is the word-length frame sync. A bit-length frame sync can be chosen by setting FSL1 and FSL0 for the configuration desired.

11.3.7.1.2 Continuous/Gated Clock Selection

The TX and RX clocks may be programmed as either continuous or gated clock signals by the GCK bit in the CRB. A continuous TX and RX clock is required in applications such as communicating with some codecs where the clock is used for more than just data transfer. A gated clock, in which the clock only toggles while data is being transferred, is useful for many applications and is required for SPI compatibility. The frame sync outputs may be used as a start conversion signal by some A/D and D/A devices.

Figure 11-57 illustrates the difference between continuous clock and gated clock systems. A separate frame-sync signal is required in continuous clock systems to delimit the active clock transitions. Although the word-length frame sync is shown in Figure 11-57, a bit-length frame sync can be used (see Figure 11-58). In gated clock systems, frame synchronization is inherent in the clock signal; thus a separate sync signal is not required (see Figure 11-59 and Figure 11-60). The SSI can be programmed to generate frame sync outputs in gated clock mode but does not use frame sync inputs.

Input flags (see Figure 11-59 and Figure 11-60) are latched on the negative edge of the first data bit of a frame. Output flags are valid during the entire frame.

11.3.7.1.3 Synchronous/Asynchronous Operating Modes

The transmit and receive sections of this interface may be synchronous or asynchronous – i.e., the transmitter and receiver may use common clock and synchronization signals (synchronous operating mode, see Figure 11-61) or they may have their own separate clock and sync signals (asynchronous operating mode). The SYN bit in CRB selects synchronous or asynchronous operation. Since the SSI is designed to operate either synchronously or asynchronously, separate receive and transmit interrupts are provided.

Figure 11-62 illustrates the operation of the SYN bit in the CRB. When SYN equals zero, the SSI TX and RX clocks and frame sync sources are independent. If SYN equals one, the SSI TX and RX clocks and frame sync come from the same source (either external or internal).

Data clock and frame sync signals can be generated internally by the DSP or may be obtained from external sources. If internally generated, the SSI clock generator is used to derive bit clock and frame sync signals from the DSP internal system clock. The SSI clock generator consists of a selectable fixed prescaler and a programmable prescaler for bit rate clock generation and also a programmable frame-rate divider and a word-length divider for frame-rate sync-signal generation.

Figures 11-63, 11-64, 11-65, and 11-66 show the definitions of the SSI pins during each

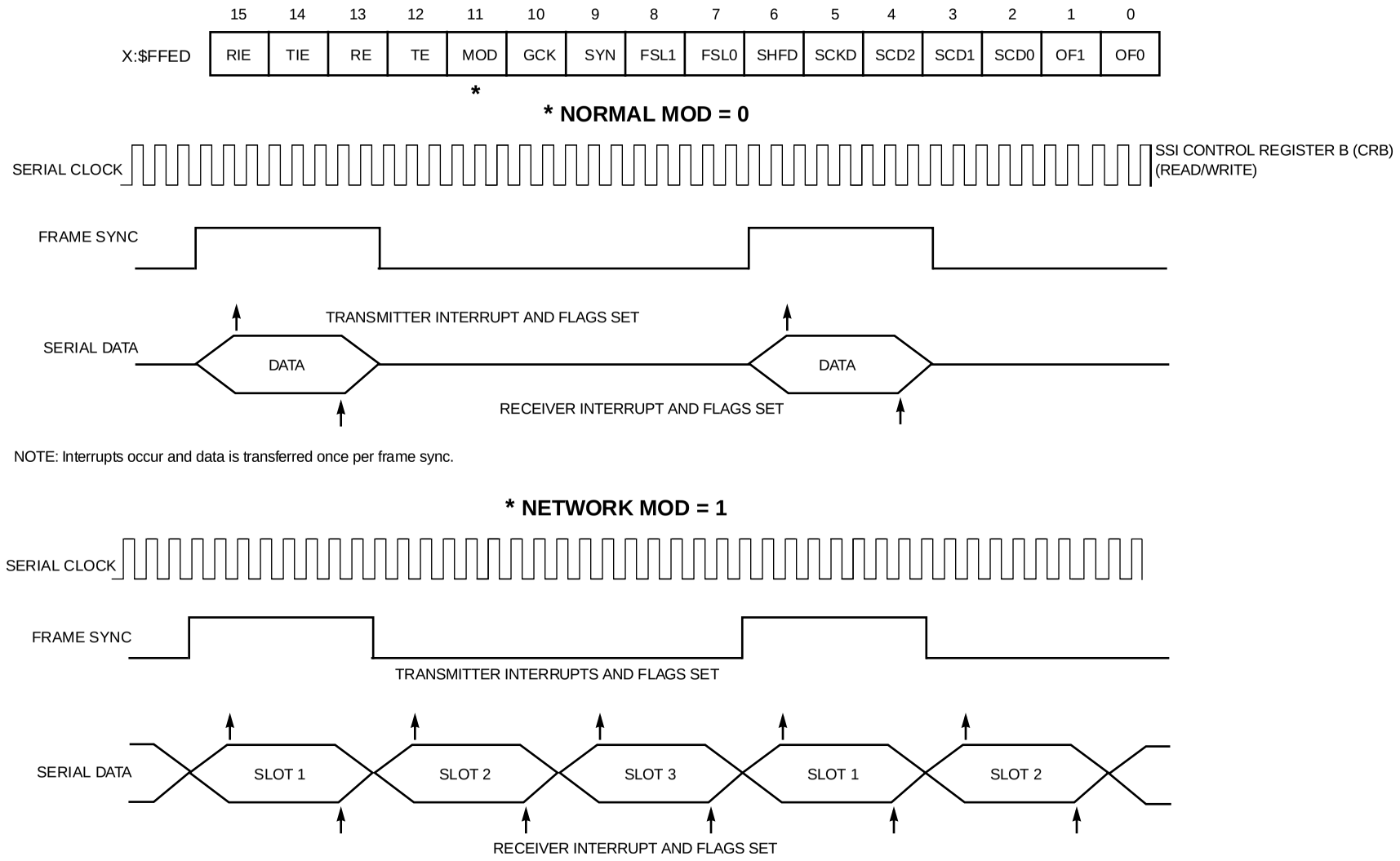


Figure 11-54 CRB MOD Bit Operation

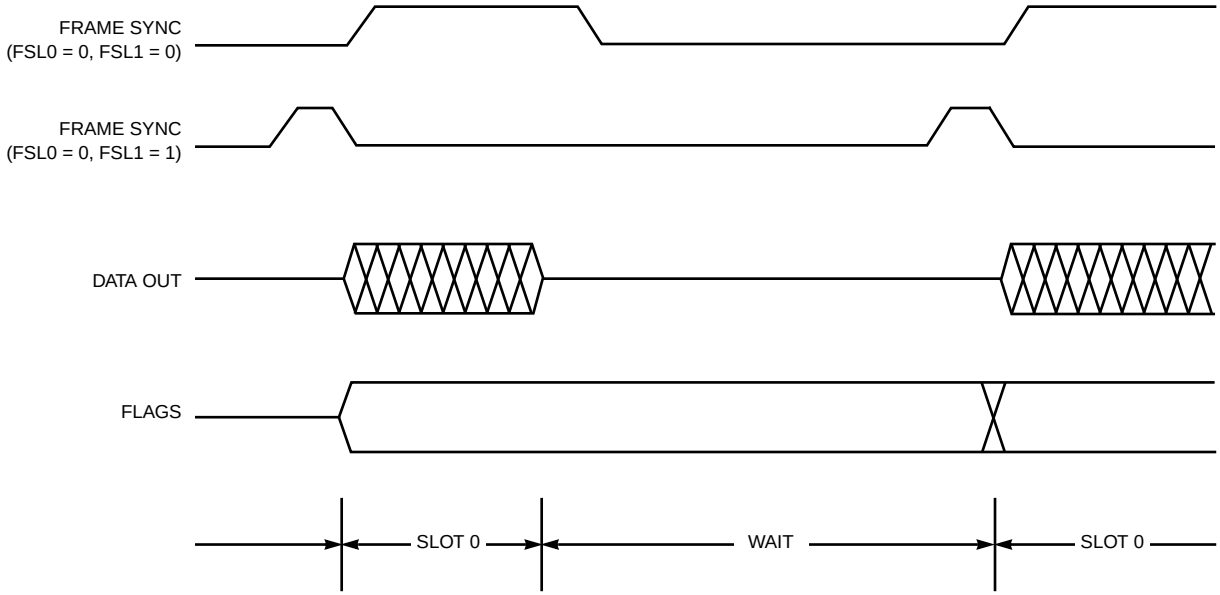


Figure 11-55 Normal Mode, External Frame Sync (8 Bit, 1 Word in Frame)

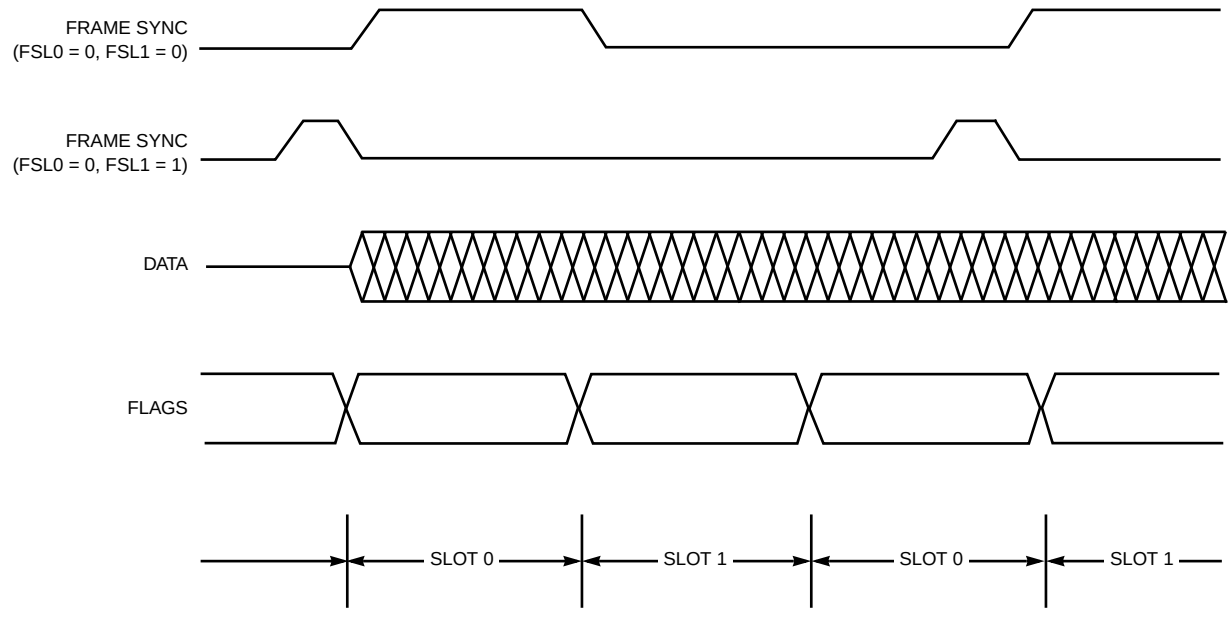


Figure 11-56 Network Mode, External Frame Sync (8 Bit, 2 Words in Frame)

of the four main operating modes of the SSI I/O interface. Figure 11-63 uses a gated

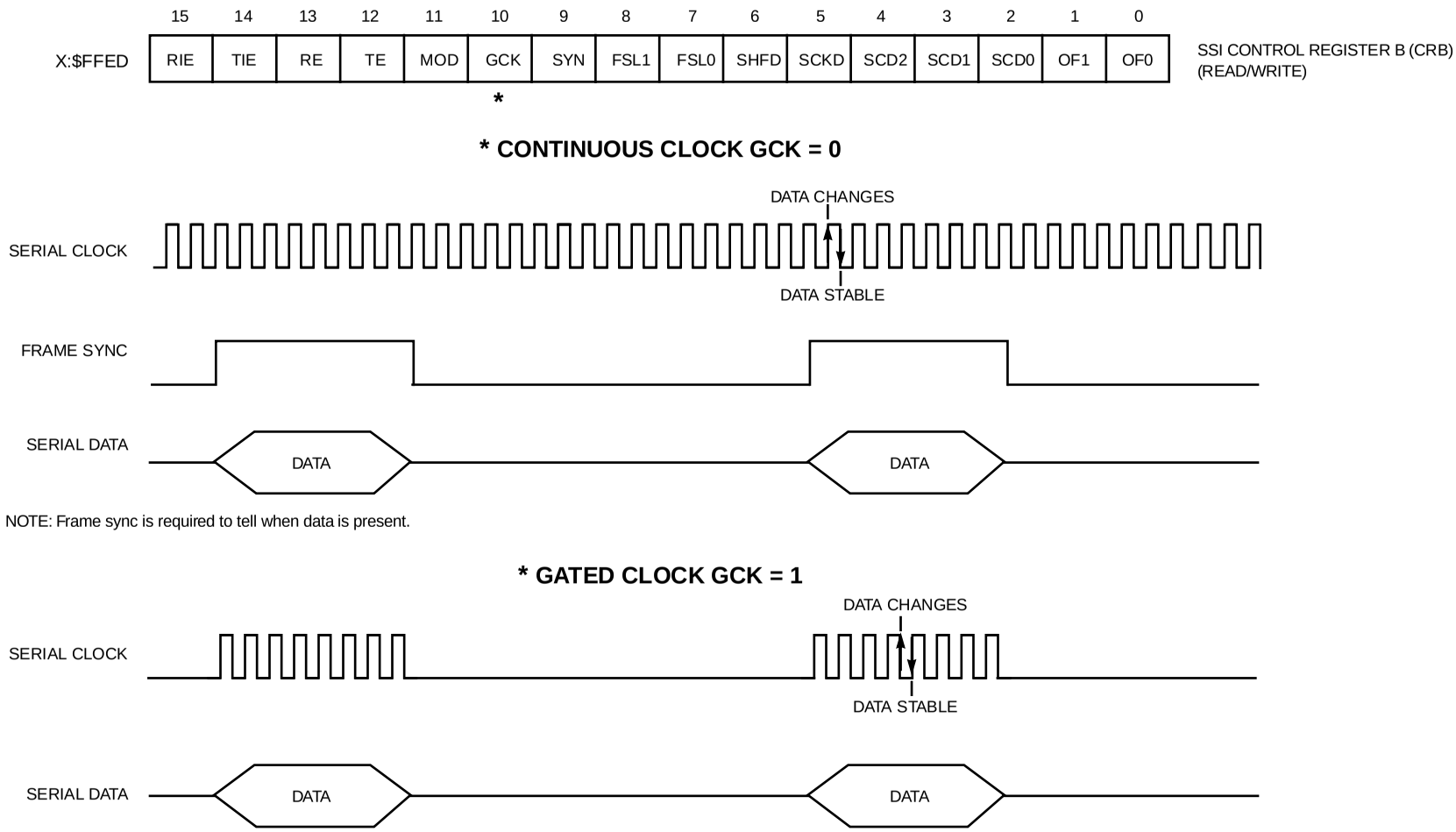


Figure 11-57 CRB GCK Bit Operation

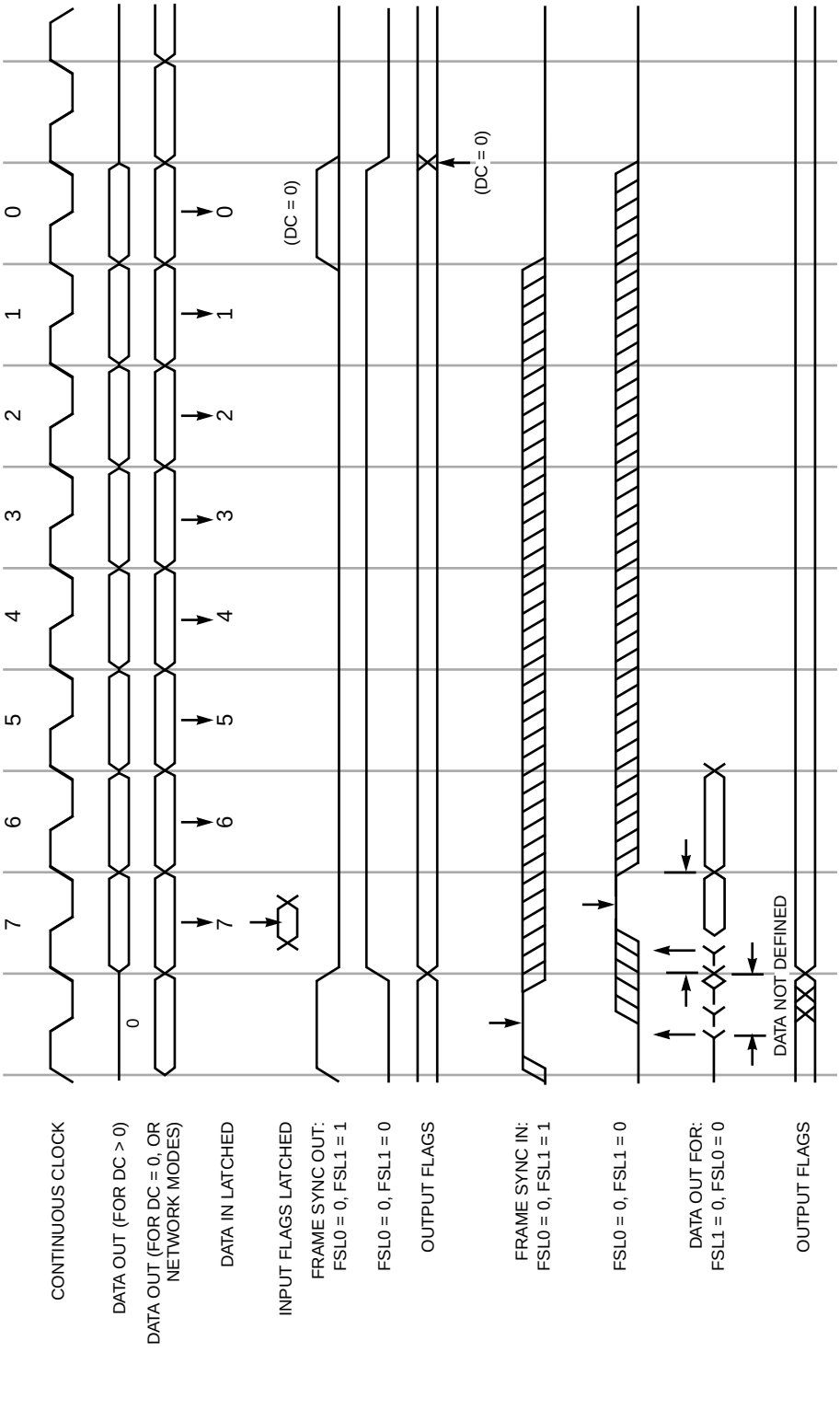


Figure 11-58 Continuous Clock Timing Diagram (8-Bit Example)

NOTES:
1. For FSL1 - 0 the frame sync is latched and enables the STD output buffer, but data may not be valid until rising edge of bit clock.
2. WL bit frame sync (FSL0 = 0, FSL1 = 0) is not defined for DC = 0 in continuous clock mode.
3. Data and flags transition after external frame sync but not before rising edge of clock.

clock (from either an external source or the internal clock), which means that frame sync

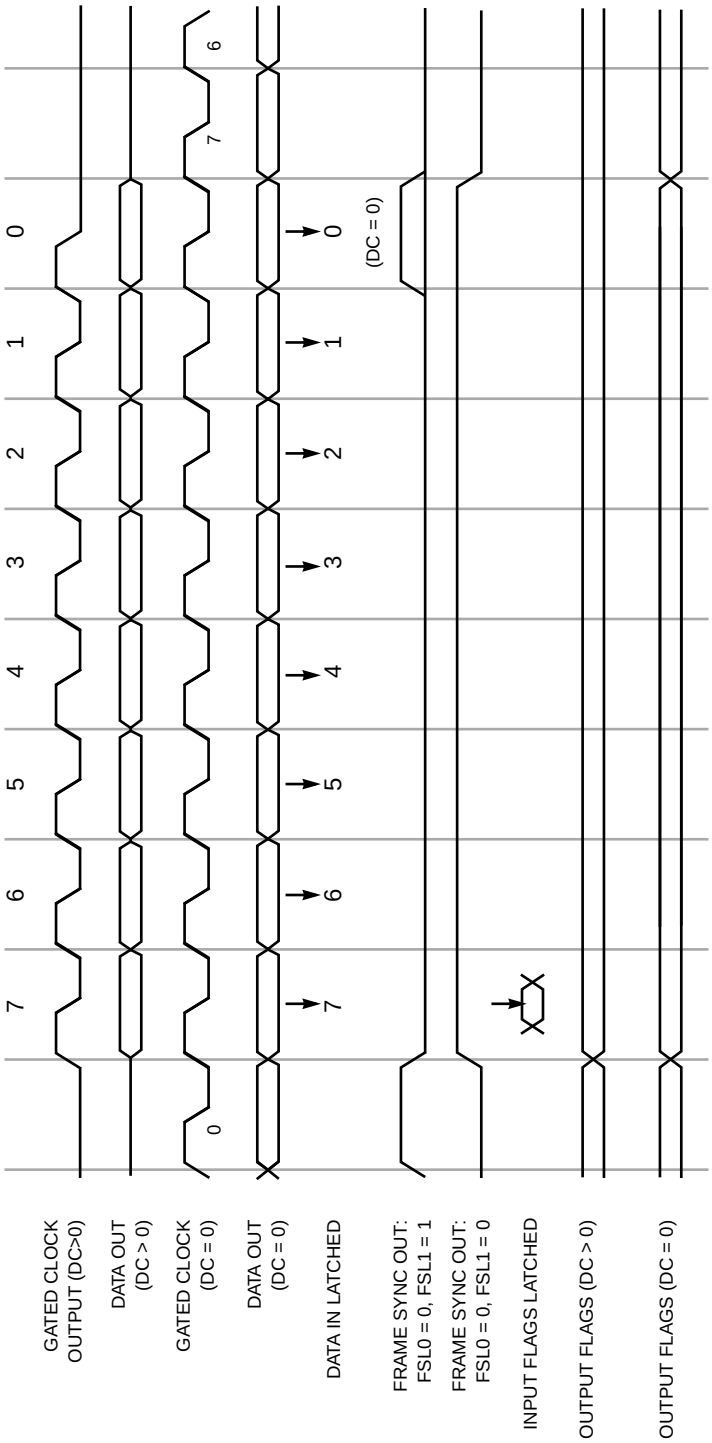
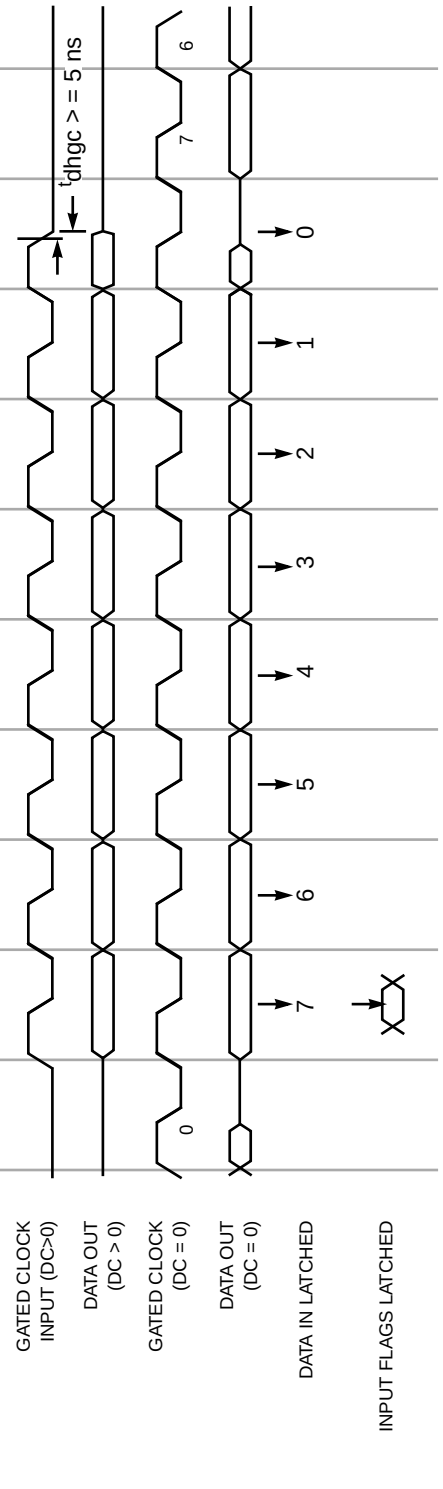


Figure 11-59 Internally Generated Clock Timing (8-Bit Example)

is inherent in the clock. Since both the transmitter and receiver use the same clock



NOTES:
1. Output enabled on rising edge of first clock input.
2. Output disabled on falling edge of last clock pulse.
3. t_{dhgc} is guaranteed by circuit design.
4. Frame syncs (in or out) are not defined for external gated clock mode.

Figure 11-60 Externally Generated Clock Timing (8-Bit Example)

(synchronous configuration), both use the SCK pin. SC0 and SC1 are designated as

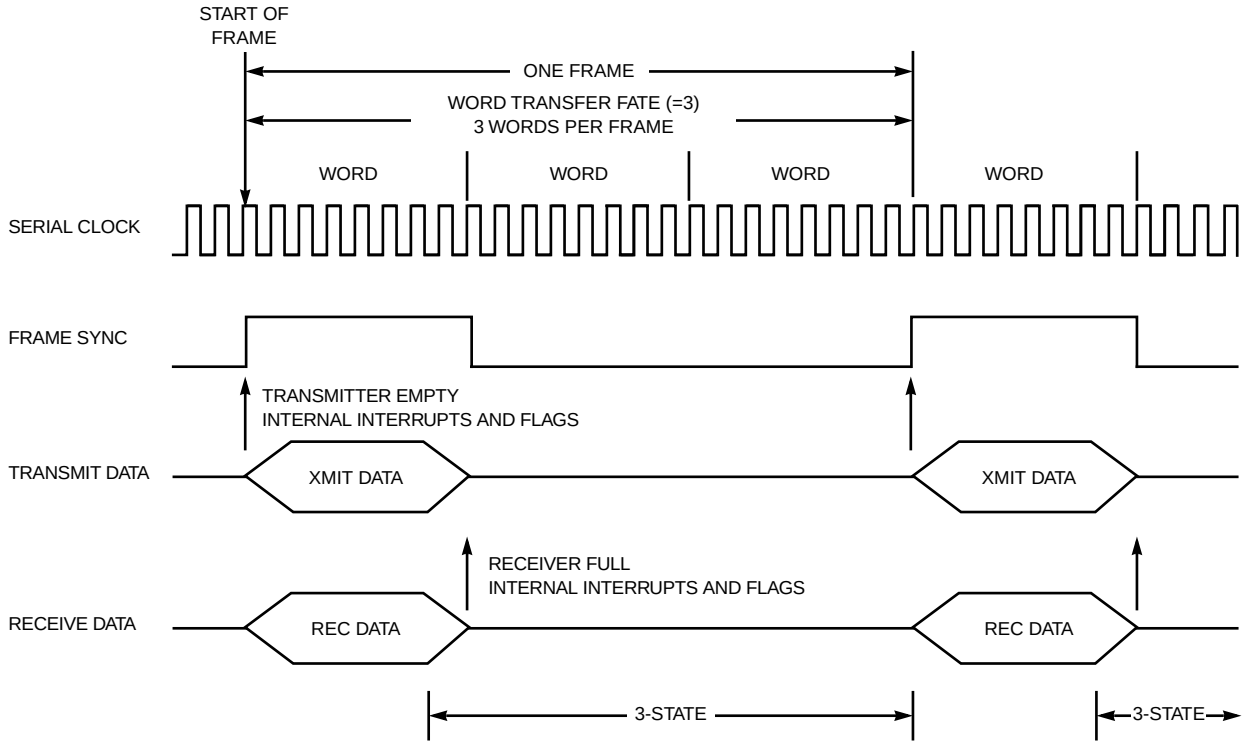


Figure 11-61 Synchronous Communication

flags or can be used as general purpose-parallel I/O. SC2 is not defined if it is an input; SC2 is the transmit and receive frame sync if it is an output.

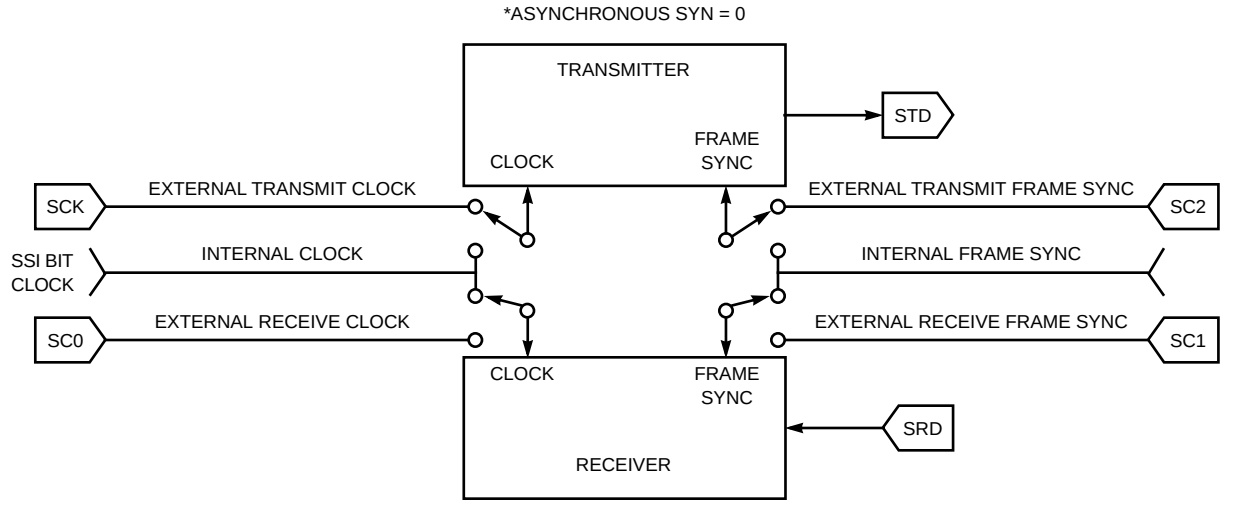
Figure 11-64 shows a gated clock (from either an external source or the internal clock), which means that frame sync is inherent in the clock. Since this configuration is asynchronous, SCK is the transmitter clock pin (input or output) and SC0 is the receiver clock pin (input or output). SC1 and SC2 are designated as receive or transmit frame sync, respectively, if they are selected to be outputs; these bits are undefined if they are selected to be inputs. SC1 and SC2 can also be used as general-purpose parallel I/O.

Figure 11-65 shows a continuous clock (from either an external source or the internal clock), which means that frame sync must be a separate signal. SC2 is used for frame sync, which can come from an internal or external source. Since both the transmitter and receiver use the same clock (synchronous configuration), both use the SCK pin. SC0 and SC1 are designated as flags or can be used as general-purpose parallel I/O.

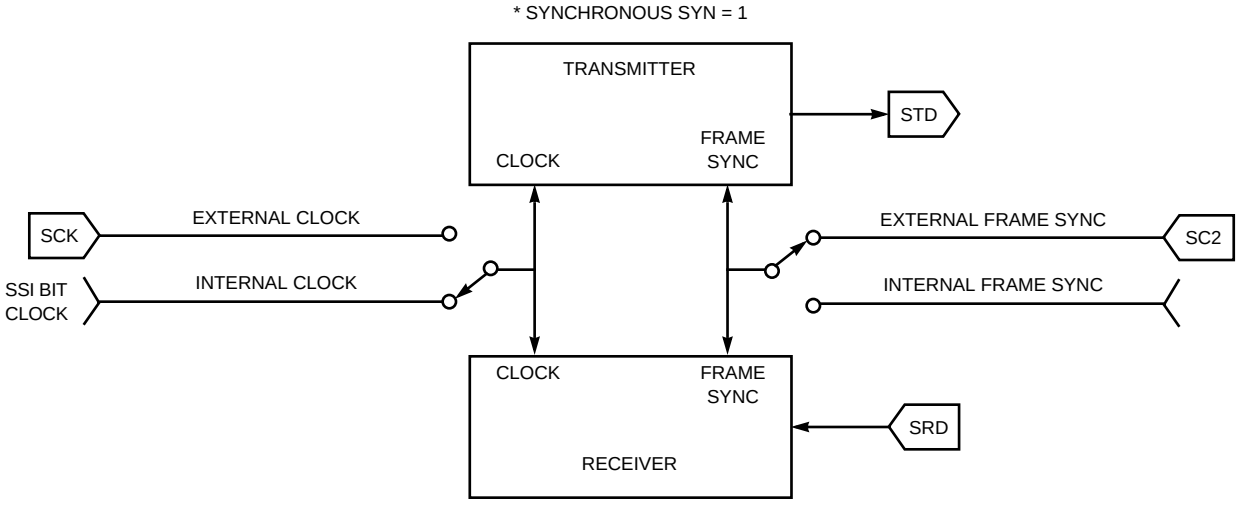
Figure 11-66 shows a continuous clock (from either an external source or the internal clock), which means that frame sync must be a separate signal. SC1 is used for the receive frame sync, and SC2 is used for the transmit frame sync. Either frame sync can come from an internal or external source. Since the transmitter and receiver use different clocks (asynchronous configuration), SCK is used for the transmit clock, and SC0 is

SSI CONTROL REGISTER B (CRB)
(READ/WRITE)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X:$FFED	RIE	TIE	RE	TE	MOD	GCK	SYN	FSL1	FSL0	SHFD	SCKD	SCD2	SCD1	SCD0	OF1	OF0
							*									



NOTE: Transmitter and receiver may have different clocks and frame syncs.



NOTE: Transmitter and receiver may have the same clock frame syncs.

Figure 11-62 CRB SYN Bit Operation

used for the receive clock.

11.3.7.1.4 Frame Sync Selection

The transmitter and receiver can operate totally independent of each other. The transmitter can have either a bit-long or word-long frame-sync signal format, and the receiver can have the same or opposite format. The selection is made by programming FSL0 and FSL1 in the CRB as shown in Figure 11-67.

1. If FSL1 equals zero (see Figure 11-68), the RX frame sync is asserted during the entire data transfer period. This frame sync length is compatible with Motorola codecs, SPI serial peripherals, serial A/D and D/A converters, shift registers, and telecommunication PCM serial I/O.
2. If FSL1 equals one (see Figure 11-69), the RX frame sync pulses active for one bit clock immediately before the data transfer period. This frame sync length is compatible with Intel and National components, codecs, and telecommunication PCM serial I/O.

The ability to mix frame sync lengths is useful in configuring systems in which data is received from one type device (e.g., codec) and transmitted to a different type device.

FSL0 controls whether RX and TX have the same frame sync length (see Figure 11-67). If FSL0 equals zero, RX and TX have the same frame sync length, which is selected by FSL1. If FSL0 equals one, RX and TX have different frame sync lengths, which are selected by FSL1.

The SSI receiver looks for a receive frame sync leading edge only when the previous frame is completed. If the frame sync goes high before the frame is completed (or before the last bit of the frame is received in the case of a bit frame sync), the current frame sync will not be recognized, and the receiver will be internally disabled until the next frame sync. Frames do not have to be adjacent – i.e., a new frame sync does not have to immediately follow the previous frame. Gaps of arbitrary periods can occur between frames. The transmitter will be three-stated during these gaps.

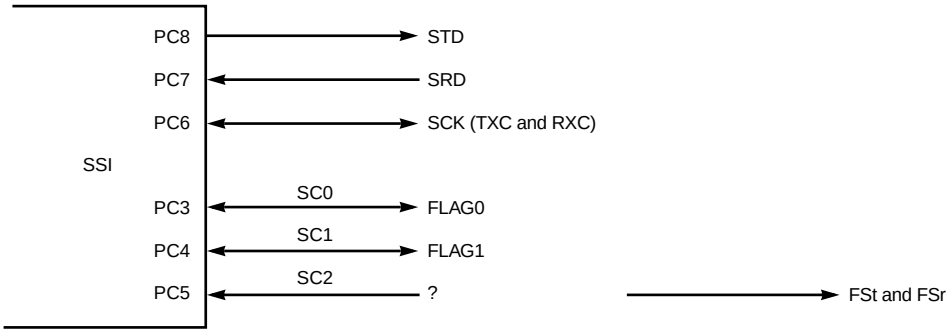


Figure 11-63 Gated Clock — Synchronous Operation

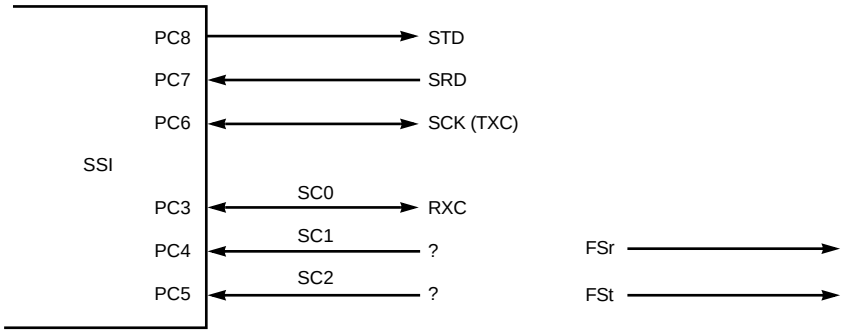


Figure 11-64 Gated Clock — Asynchronous Operation

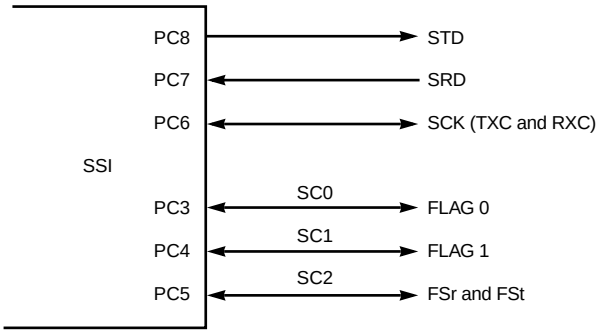


Figure 11-65 Continuous Clock — Synchronous Operation

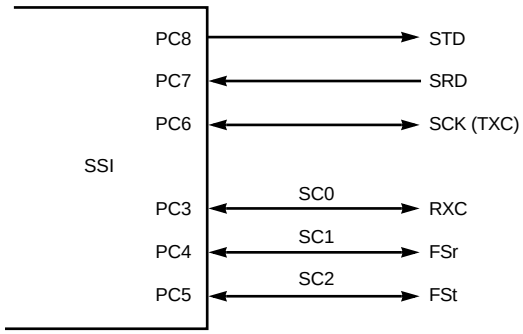
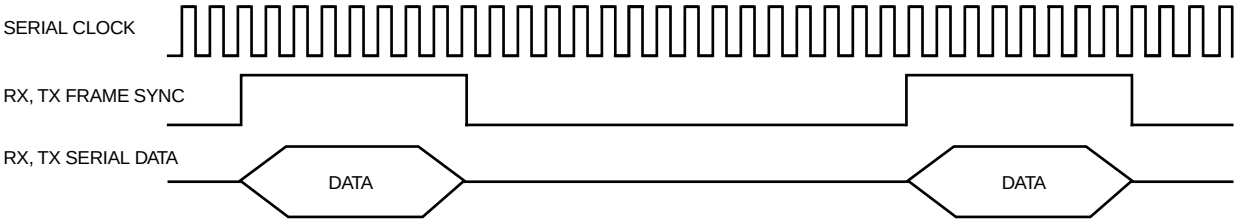


Figure 11-66 Continuous Clock — Asynchronous Operation

SSI CONTROL REGISTER B (CRB)
(READ/WRITE)

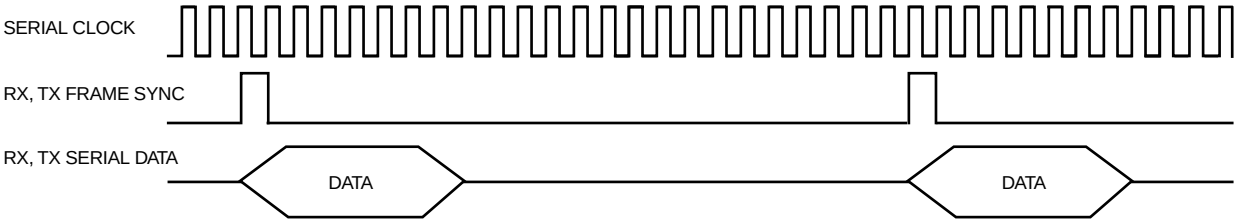
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X:$FFED	RIE	TIE	RE	TE	MOD	GCK	SYN	FSL1	FSL0	SHFD	SCKD	SCD2	SCD1	SCD0	OF1	OF0
								*	*							

*** WORD LENGTH: FSL1 = 0, FSL0 = 0**



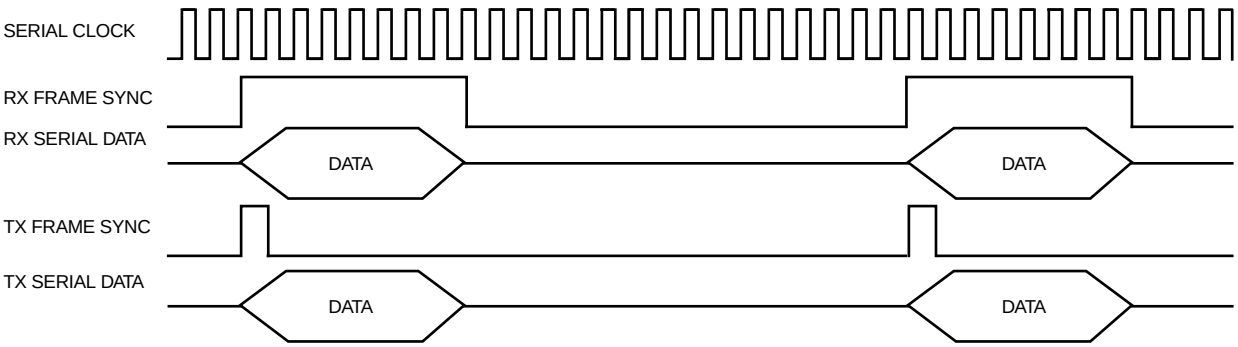
NOTE: Frame sync occurs while data is valid.

*** ONE BIT: FSL1 = 1, FSL0 = 0**



NOTE: Frame sync occurs for one bit time preceding the data.

*** MIXED FRAME LENGTH: FSL1 = 0, FSL0 = 1**



*** MIXED FRAME LENGTH: FSL1 = 1, FSL0 = 1**

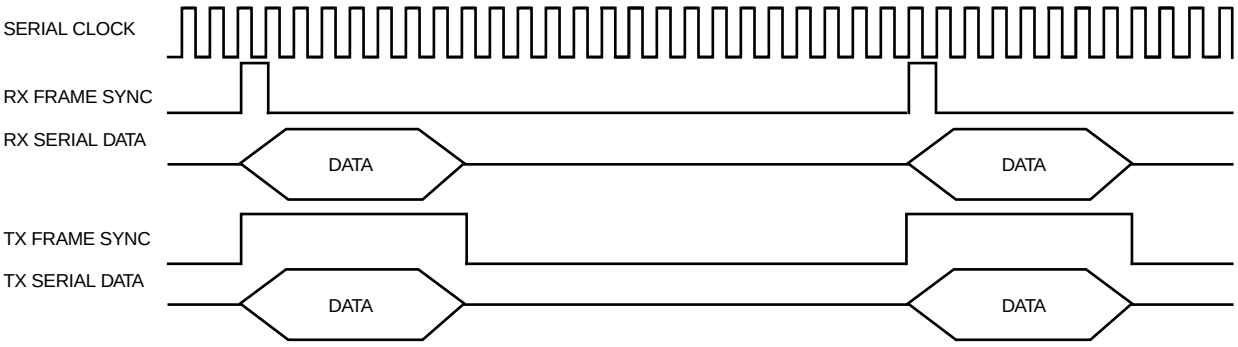


Figure 11-67 CRB FSL0 and FSL1 Bit Operation

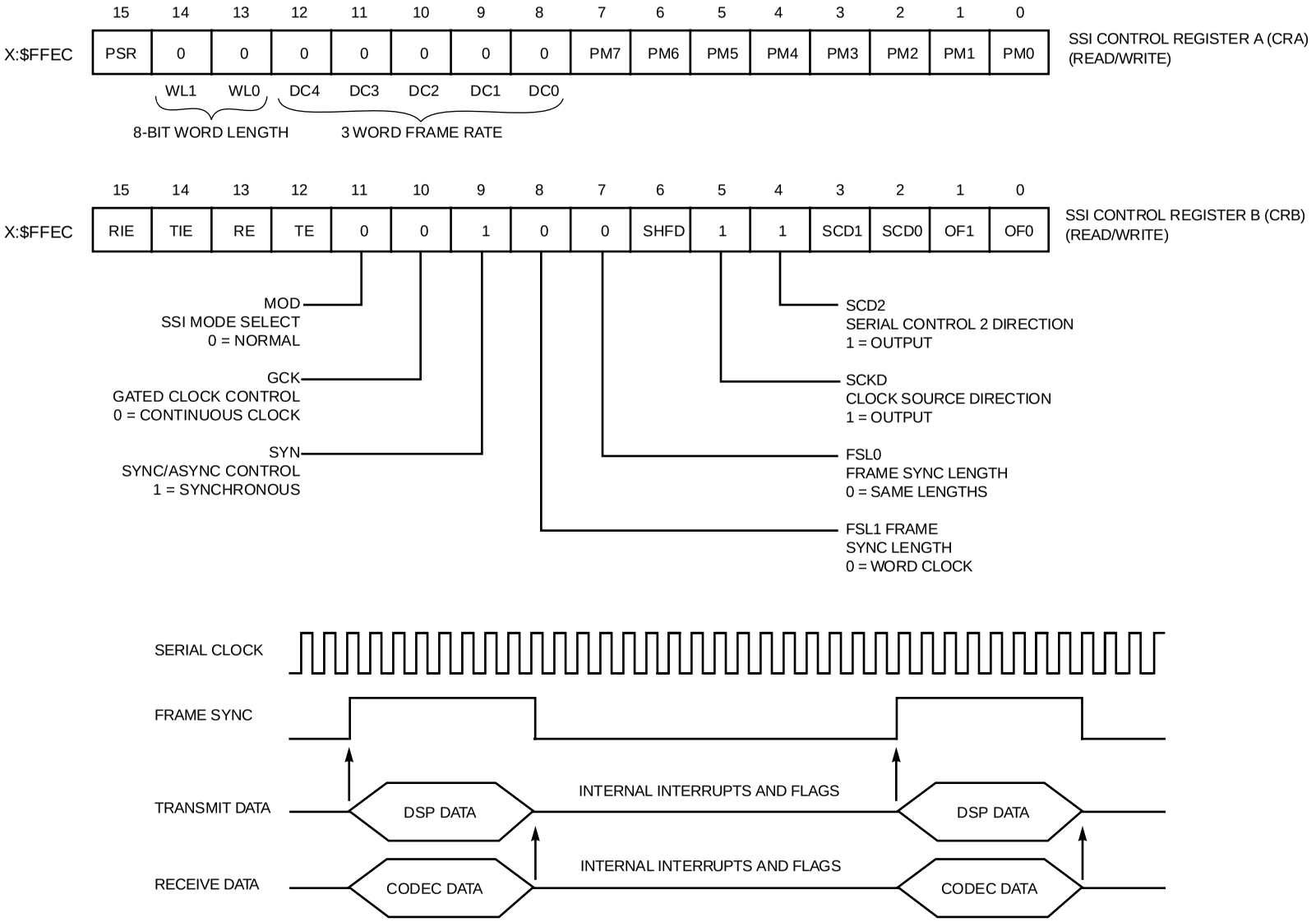


Figure 11-68 Normal Mode Initialization for FLS1=0 and FSL0=0

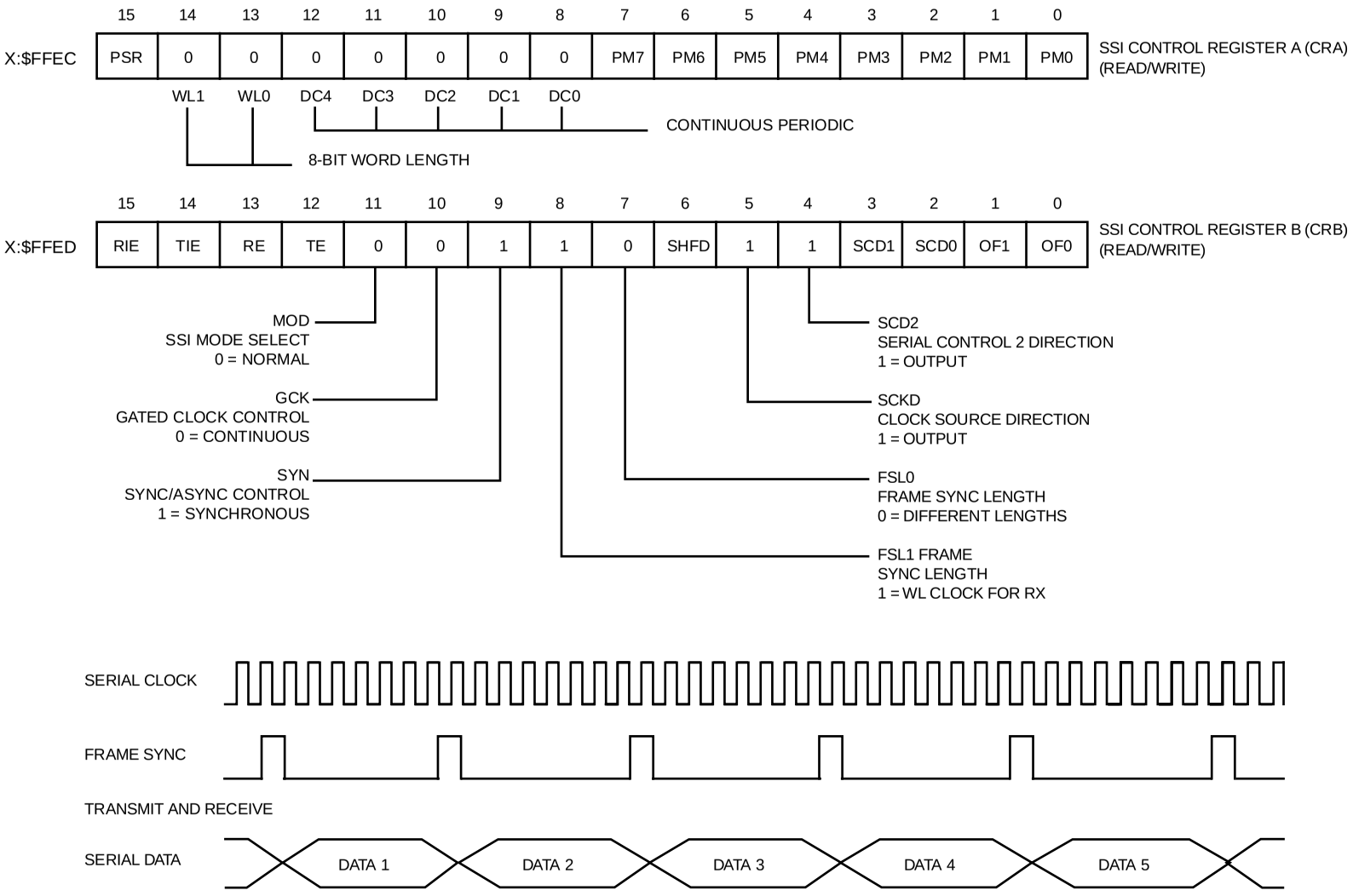
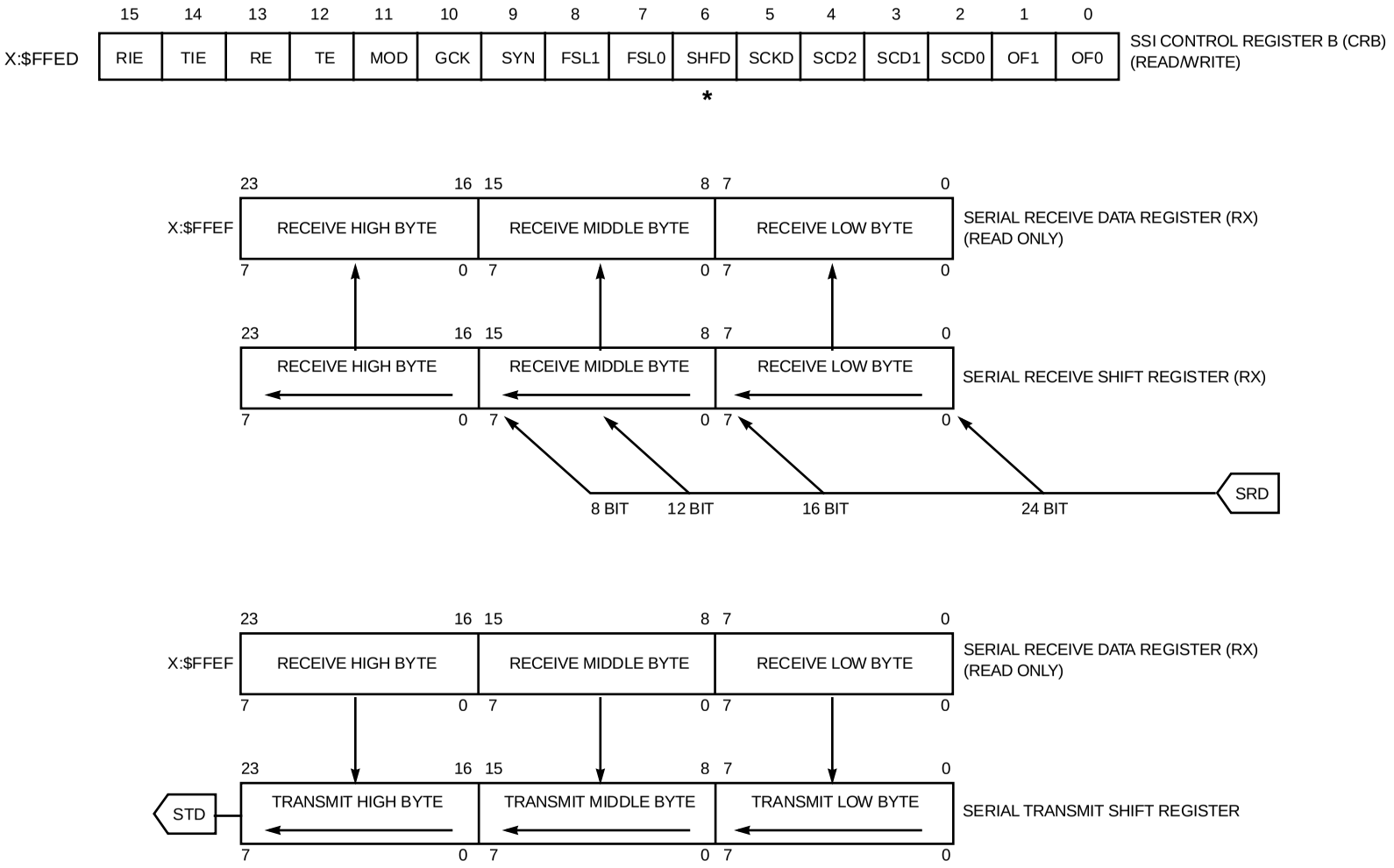


Figure 11-69 Normal Mode Initialization for FSL1=1 and FSL0=0



(a) SHFD = 0

Figure 11-70 CRB SHFD Bit Operation (Sheet 1 of 2)

11.3.7.1.5 Shift Direction Selection

Some data formats, such as those used by codecs, specify MSB first other data formats, such as the AES-EBU digital audio, specify LSB first. To interface with devices from both systems, the shift registers in the SSI are bidirectional. The MSB/LSB selection is made by programming SHFD in the CRB.

Figure 11-70 illustrates the operation of the SHFD bit in the CRB. If SHFD equals zero (see Figure 11-70(a)), data is shifted into the receive shift register MSB first and shifted out of the transmit shift register MSB first. If SHFD equals one (see Figure 11-71(b)), data is shifted into the receive shift register LSB first and shifted out of the transmit shift register LSB first.

11.3.7.2 Normal Mode Examples

The normal SSI operating mode characteristically has one time slot per serial frame, and data is transferred every frame sync. When the SSI is not in the normal mode, it is in the network mode. The MSB is transmitted first (SHFD=0), with overrun and underrun errors detected by the SSI hardware. Transmit flags are set when data is transferred from the transmit data register to the transmit shift register. The receive flags are set when data is transferred from the receive shift register to the receive data register.

Figure 11-72 shows an example of using the SSI to connect an MC15500 codec with a DSP56000/DSP56001. No glue logic is needed. The serial clock, which is generated internally by the DSP, provides the transmit and receive clocks (synchronous operation) for the codec. SC2 provides all the necessary handshaking. Data transfer begins when the frame sync is asserted. Transmit data is clocked out and receive data is clocked in with the serial clock while the frame sync is asserted (word-length frame sync). At the end of the data transfer, DSP internal interrupts programmed to transfer data to/from will occur, and the SSISR will be updated.

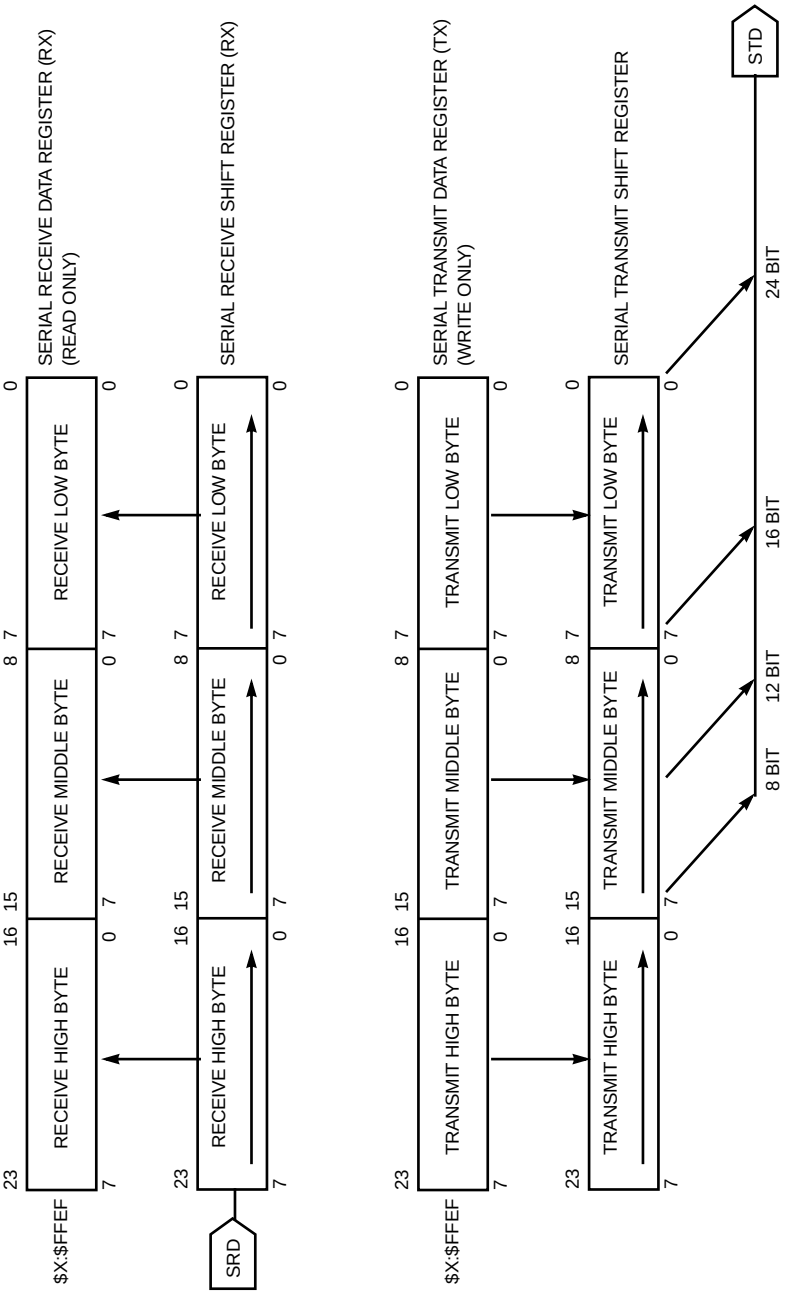
11.3.7.2.1 Normal Mode Transmit

The conditions for data transmission from the SSI are as follows:

1. Transmitter is Enabled (TE=1).
2. Frame sync (or clock in gated clock mode) is active.

When these conditions occur in normal mode, the next data word will be transferred from TX to the transmit shift register, the TDE flag will be set (transmitter empty), and the transmit interrupt will occur if TIE equals one (transmit interrupt enabled.) The new data word will be transmitted immediately.

The transmit data output (STD) is three-stated, except during the data transmission period. The optional frame sync output, flag outputs, and clock outputs are not three-



(b) SHFD=1

Figure 11-71 CRB SHFD Bit Operation (Sheet 2 of 2)

stated even if both receiver and transmitter are disabled.

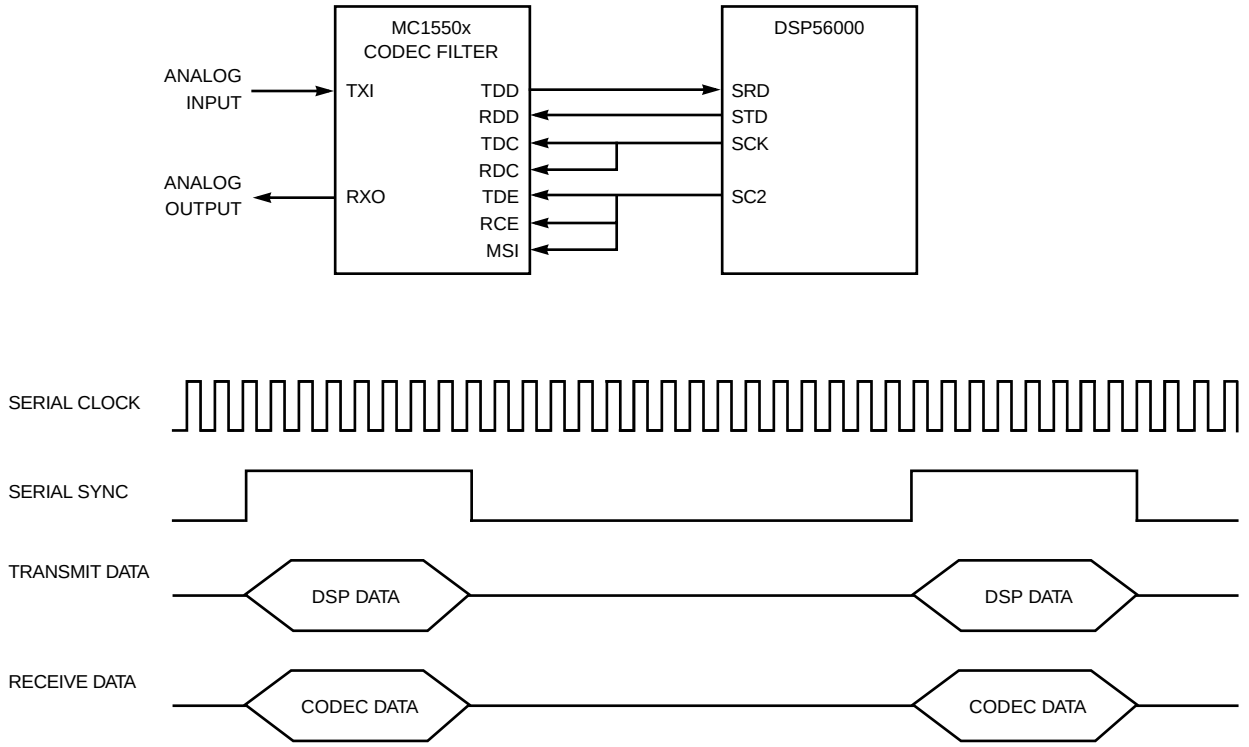


Figure 11-72 Normal Mode Example

The optional output flags are always updated at the beginning of the frame, regardless of TE. The state of the flag does not change for the entire frame.

Figure 11-72 is an example of transmitting data using the SSI in the normal mode with a continuous clock, a bit-length frame sync, and 16-bit data words. The purpose of the program is to interleave and transmit right and left channels in a compact disk player. Four SSI pins are used: SC0 is used as an output flag to indicate right-channel data (OF0=1) or left-channel data (OF0=0); SC2 is TX and RX frame sync out; STD is transmit data out; and SCK clocks the transmit data out. Equates are set for convenience and readability. Test data is then put in the low X: memory locations. The transmit interrupt vector contains a JSR instruction (which forms a long interrupt). The data pointer and channel flag are initialized before initializing CRA and CRB. It is assumed that the DSP CPU and SSI have been previously reset. At this point, the SSI is ready to transmit except that the interrupt is masked because the MR was cleared on reset and port C is still configured a general-purpose I/O. Unmasking the interrupt and enabling the SSI pins allows transmission to begin. A “jump to self” instruction causes the DSP to hang and wait for interrupts to transmit the data. When an interrupt occurs, a JSR instruction at the interrupt vector location causes the XMT routine to be executed. Data is then moved to the TX register, and the data pointer is incremented. The flag is tested by the JSET instruction and, if it is set, a jump to left occurs, and the code for the left channel is executed. If the flag is not

set, the code for the right channel is executed. In either case, the channel flag in X0 and then the output flag are set to reflect the channel being transmitted. Control is then returned to the main program, which will wait for the next interrupt.

```
;************************************************
;            SSI and other I/O EQUATES         *
;************************************************

IPR             EQU     $FFFF
CRA             EQU     $FFEC
CRB             EQU     $FFED
PCC             EQU     $FFE1
TX              EQU     $FFEF
FLG             EQU     $0010

                ORG     X:0
                DC      $AAAA00         ;Data to transmit.
                DC      $333300
                DC      $CCCC00
                DC      $F0F000

;************************************************
;              INTERRUPT VECTOR            *
;************************************************

                ORG     P:$0010
                JSR     XMT

;************************************************
;              MAIN PROGRAM                *
;************************************************

                ORG     P:$40
                MOVE    #0,R0           ;Pointer to data buffer.
                MOVE    #3,M0           ;Set modulus to 4.
                MOVE    #0,X0           ;Initialize channel flag for SSI flag.
                MOVE    X0,X:FLG        ;Start with right channel first.

;************************************************
;              Initialize SSI Port         *
;************************************************

                MOVEP   #$3000,X:IPR    ;Set interrupt priority register for SSI.
                MOVEP   #$401F,X:CRA    ;Set continuous clock=5.12/32 MHz
```

```
                                                ;word length=16.
                MOVEP   #$5334,X:CRB            ;Enable TIE and TE; make clock and
                                                ;frame sync outputs; frame
                                                ;sync=bit mode; synchronous mode;
                                                ;make SC0 an output.

;************************************************
;               Init SSI Interrupt              *
;************************************************

                ANDI    #$FC,MR                 ;Unmask interrupts.
                MOVEP   #$01F8,X:PCC            ;Turn on SSI port.

                JMP     *                       ;Wait for interrupt.

;************************************************
;          MAIN INTERRUPT ROUTINE               *
;************************************************

XMT             MOVEP   X:(R0);pl,X:TX          ;Move data to TX register.
                JSET    #0,X:FLG,LEFT           ;Check channel flag.

RIGHT           BCLR    #0,X:CRB                ;Clear SC0 indicating right channel
                                                ;data
                MOVE    #>$01,X0                ;Set channel flag to 1 for next data.
                MOVE    X0,X:FLG
                RTI

LEFT            BSET    #0,X:CRB                ;Set SC0 indicating left channel data.
                MOVE    #>$00,X0                ;Clear channel flag for next data.
                MOVE    X0,X:FLG
                RTI

                END
```

Figure 11-72 Normal Mode Transmit Example

11.3.7.2.2 Normal Mode Receive

If the receiver is enabled, a data word will be clocked in each time the frame sync signal is generated (internal) or detected (external). After receiving the data word, it will be transferred from the SSI receive shift register to the receive data register (RX), RDF will be set (receiver full), and the receive interrupt will occur if it is enabled (RIE=1).

The DSP program has to read the data from RX before a new data word is transferred

from the receive shift register; otherwise, the receiver overrun error will be set (ROE=1).

Figure 11-73 illustrates the program that receives the data transmitted by the program shown in Figure 11-72. Using the flag to identify the channel, the receive program receives the right- and left-channel data and separates the data into a right data buffer and a left data buffer. The program shown in Figure 11-73 begins by setting equates and then using a JSR instruction at the receive interrupt vector location to form a long interrupt. The main program starts by initializing pointers to the right and left data buffers. The IPR, CRA, and CRB are then initialized. The clock divider bits in the CRA do not have to be set since an external receive clock is specified (SCKD=0). Pin SC0 is specified as an input flag (SYN=1, SCD0=0); pin SC2 is specified as TX and RX frame sync (SYN=1, SCD2=0). The SSI port is then enabled and interrupts are unmasked, which allows the SSI port to begin data reception. A jump-to-self instruction is then used to hang the processor and allow interrupts to receive the data. Normally, the processor would execute useful instructions while waiting for the receive interrupts. When an interrupt occurs, the JSR instruction at the interrupt vector location transfers control to the RCV subroutine. The input flag is tested, and data is put in the left or right data buffer depending on the results of the test. The RTI instruction then returns control to the main program, which will wait for the next interrupt.

```
;************************************************
;            SSI and other I/O EQUATES         *
;************************************************

IPR             EQU     $FFFF
SSISR           EQU     $FFEE
CRA             EQU     $FFEC
CRB             EQU     $FFED
PCC             EQU     $FFE1
RX              EQU     $FFEF
FLG             EQU     $0010

;************************************************
;              INTERRUPT VECTOR                *
;************************************************

                ORG     P:$000C
                JSR     RCV

;************************************************
;              MAIN PROGRAM                    *
;************************************************
```

```
                ORG     P:$40
                MOVE    #0,R0           ;Pointer to memory buffer for
                MOVE    #$08,R1         ;received data. Note data will be
                MOVE    #1,M0           ;split between two buffers which are
                MOVE    #1,M1           ;modulus 2.

;***********************************************
;               Initialize SSI Port             *
;***********************************************

                MOVEP   #$3000,X:IPR    ;Set interrupt priority register for SSI.
                MOVEP   #$4000,X:CRA    ;Set word length = 16 bits.
                MOVEP   #$A300,X:CRB    ;Enable RIE and RE; synchronous
                                        ;mode with bit frame sync;
                                        ;clock and frame sync are
                                        ;external; SC0 is an output.

;***********************************************
;               Init SSI Interrupt              *
;***********************************************

                ANDI    #$FC,MR         ;Unmask interrupts.
                MOVEP   #$01F8,X:PCC    ;Turn on SSI port.
                JMP     *               ;Wait for interrupt.

;***********************************************
;          MAIN INTERRUPT ROUTINE               *
;***********************************************

RCV             JSET    #0,X:SSISR, RIGHT ;Test SCO flag.
LEFT            MOVEP   X:RX,X:(RO)+    ;If SCO clear, receive data
                RTI                     ;into left buffer (R0).
RIGHT           MOVEP   X:RX,X:(R1)+    ;If SCO set, receive data
                RTI                     ;into right buffer (R1).
                END
```

Figure 11-73 Normal Mode Receive Example

11.3.7.3 Network Mode Examples

The network mode, the typical mode in which the DSP would interface to a TDM codec network or a network of DSPs, is compatible with Bell and CCITT PCM data/operation formats. The DSP may be a master device (see Figure 11-74) that controls its own private network or a slave device that is connected to an existing TDM network, occupying

one or more time slots. The key characteristic of the network mode is that each time slot (data word time) is identified by an interrupt or by polling status bits, which allows the option of ignoring the time slot or transmitting data during the time slot. The receiver operates in the same manner except that data is always being shifted into the receive shift register and transferred to the RX. The DSP reads the receive data register and uses or discards the contents. Overrun and underrun errors are detected.

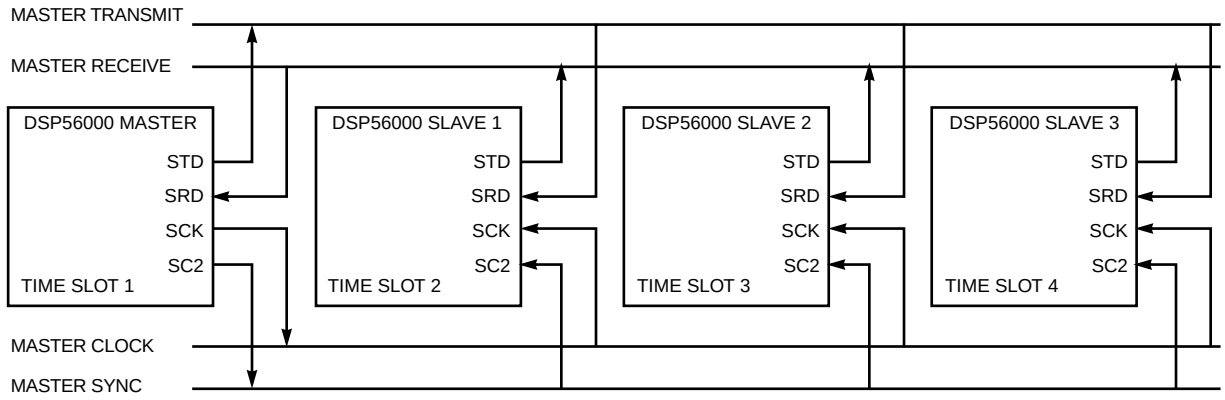


Figure 11-74 Network Mode Example

The frame sync signal indicates the beginning of a new data frame. Each data frame is divided into time slots; transmission or reception can occur in each time slot (rather than in just the frame sync time slot as in normal mode). The frame rate dividers (controlled by DC4, DC3, DC2, DC1, and DC0) control the number of time slots per frame from 2 to 32. Time-slot assignment is totally under software control. Devices can transmit on multiple time slots, receive multiple time slots, and the time-slot assignment can be changed dynamically.

A simplified flowchart showing operation of the network mode is shown in Figure 11-75. Two counters are used to track the current transmit and receive time slots. Slot counter one (SLOTCT1) is used to track the transmit time slot; slot counter two (SLOTCT2) is used for receive. When the transmitter is empty, it generates an interrupt; a test is then made to see if it is the beginning of a frame. If it is the beginning of a frame, SLOTCT1 is cleared to start counting the time slots. If it is not the beginning of a frame, SLOTCT1 is incremented. The next test checks to see if the SSI should transmit during this time slot. If it is time to transmit, data is written to the TX; otherwise, dummy data is written to the TSR, which prevents a transmit underrun error from occurring and three-states the STD pin. The DSP can then return to what it was doing before the interrupt and wait for the next interrupt to occur. SLOTCT1 should reflect the data in the shift registers to coincide with TFS. Software must recognize that the data being written to TX will be transmitted in time slot SLOTCT1 plus one.

The receiver operates in a similar manner. When the receiver is full, an interrupt is gen-

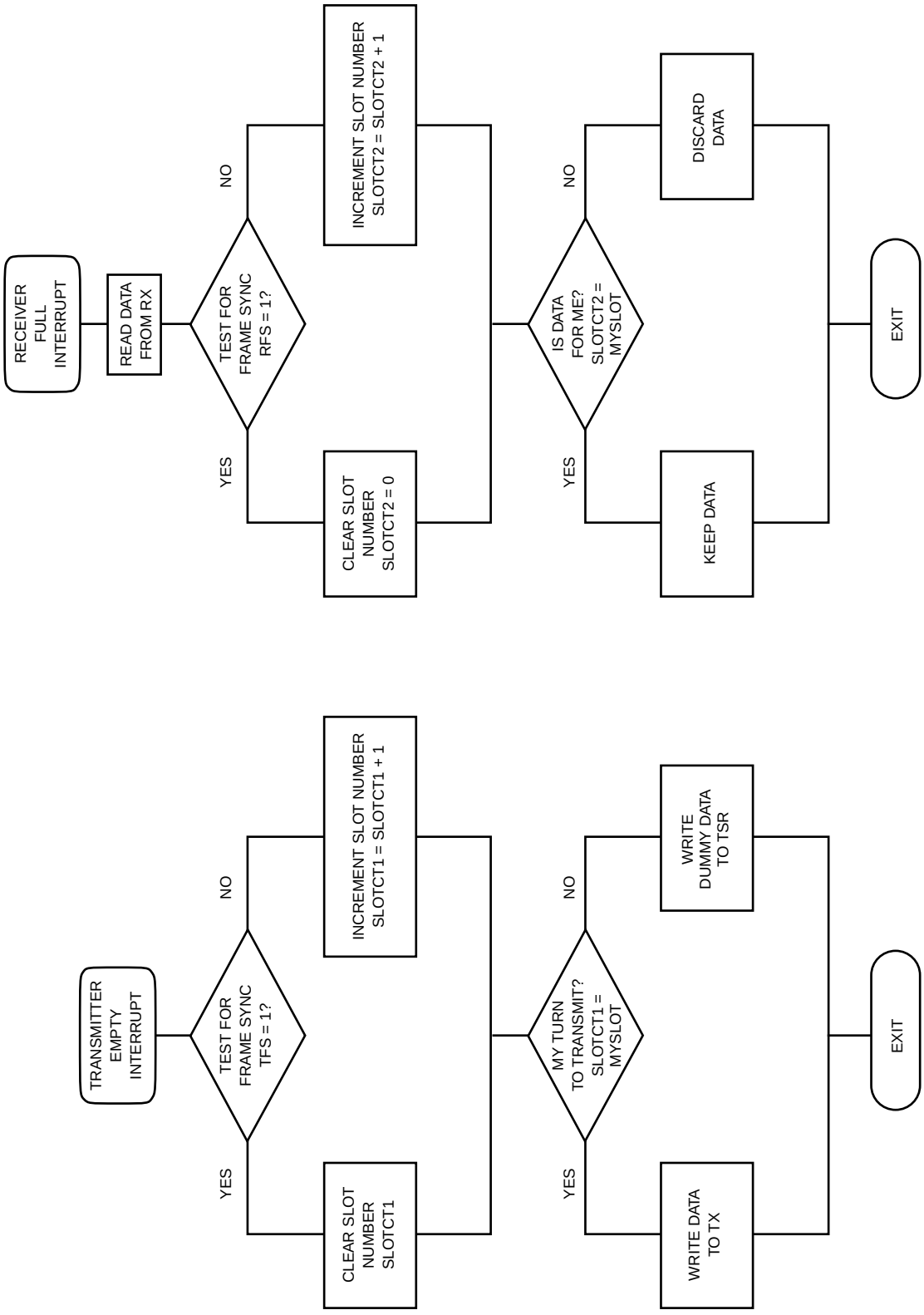


Figure 11-75 TDM Network Software Flowchart

erated, and a test is made to see if this is the beginning of a frame. If it is the beginning of

a frame, SLOTCT2 is cleared to start counting the time slots. If it is not the beginning of a frame, SLOTCT2 is incremented. The next test checks to see if the data received is intended for this DSP. If the current time slot is the one assigned to the DSP receiver, the data is kept; otherwise, the data is discarded, and the DSP can then return to what it was doing before the interrupt. SLOTCT2 should reflect the data in the receive shift register to coincide with the RFS flag. Software must recognize that the data being read from RX is for time slot SLOTCT2 minus two.

Initializing the network mode is accomplished by setting the bits in CRA and CRB as follows (see Figure 11-76):

12. The word length must be selected by setting WL1 and WL0. In this example, an 8-bit word length was chosen (WL1=0 and WL0=0).
13. The number of time slots is selected by setting DC4–DC0. Four time slots were chosen for this example (DC4–DC0=$03).
14. The serial clock rate must be selected by setting PSR and PM7–PM0 (see Tables 11-9 and 11-10).
15. RE and TE must be set to activate the transmitter and receiver. If interrupts are to be used, RIE and TIE should be set. RIE and TIE are usually set after everything else is configured and the DSP is ready to receive interrupts.
16. The network mode must be selected (MOD=1).
17. A continuous clock is selected in this example by setting GCK=0.
18. Although it is not required for the network mode, synchronous clock control was selected (SYN=1).
19. The frame sync length was chosen in this example as word length (FSL1=0) for both transmit and receive frame sync (FSL0=0). Any other combinations could have been selected, depending on the application.
20. Control bits SHFD, SCKD, SCD2, SCD1, SCD0, and the flag bits (OF1 and OF0) should be set as needed for the application.

11.3.7.3.1 Network Mode Transmit

When TE is set, the transmitter will be enabled only after detection of a new data frame sync. This procedure allows the SSI to synchronize to the network timing.

Normal startup sequence for transmission in the first time slot is to write the data to be transmitted to TX, which clears the TDE flag. Then set TE and TIE to enable the transmitter on the next frame sync and to enable transmit interrupts.

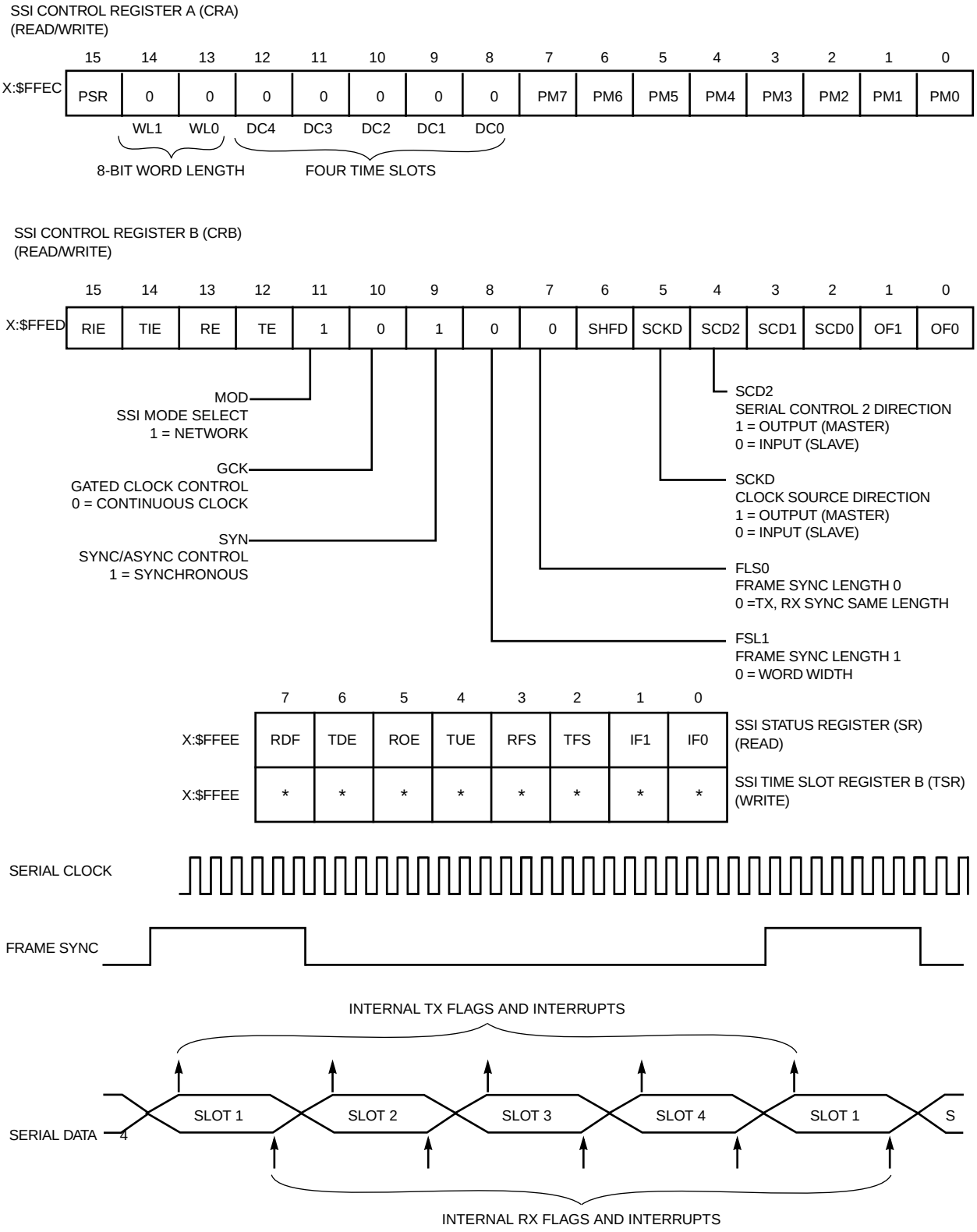


Figure 11-76 Network Mode Initialization

Alternatively, the DSP programmer may decide not to transmit in the first time slot by

writing any data to the time slot register (TSR). This will clear the TDE flag just as if data were going to be transmitted, but the STD pin will remain in the high-impedance state for the first time slot. The programmer then sets TE and TIE.

When the frame sync is detected (or generated), the first data word will be transferred from TX to the transmit shift register and will be shifted out (transmitted). TX being empty will cause TDE to be set, which will cause a transmitter interrupt. Software can poll TDE or use interrupts to reload the TX register with new data for the next time slot. Software can also write to TSR to prevent transmitting in the next time slot. Failing to reload TX (or writing to the TSR) before the transmit shift register is finished shifting (empty) will cause a transmitter underrun. The TUE error bit will be set, causing the previous data to be retransmitted.

The operation of clearing TE and setting it again will disable the transmitter after completion of transmission of the current data word until the beginning of the next frame sync period. During that time, the STD pin will be three-stated. When it is time to disable the transmitter, TE should be cleared after TDE is set to ensure that all pending data is transmitted.

The optional output flags are updated every time slot regardless of TE.

To summarize, the network mode transmitter generates interrupts every time slot and requires the DSP program to respond to each time slot. These responses can be

1. Write data register with data to enable transmission in the next time slot.
2. Write the time slot register to disable transmission in the next time slot.
3. Do nothing – transmit underrun will occur the at beginning of the next time slot, and the previous data will be transmitted.

Figure 11-77 is essentially the same program shown in Figure 11-72 except that this program uses the network mode to transmit only right-channel data. A time slot is assigned for the left-channel data, which could be inserted by another DSP using the network mode. In the “Initialize SSI Port” section of the program, two words per frame are selected using CRA, and the network mode is selected by setting MOD to one in the CRB. The main interrupt routine, which waits to move the data to TX, only transmits data if the current time slot is for the right channel. If the current time slot is for the left channel, the TSR is written, which three-states the output to allow another DSP to transmit the left channel during the time slot.

```
;*************************************************
;               SSI and other I/O EQUATES         *
;*************************************************
```

```
IPR             EQU     $FFFF
CRA             EQU     $FFEC
CRB             EQU     $FFED
PCC             EQU     $FFE1
TX              EQU     $FFEF
TSR             EQU     $FFEE
FLG             EQU     $0010

                ORG     X:0
                DC      $AAAA00         ;Data to transmit.
                DC      $333300
                DC      $CCCC00
                DC      $F0F000

;************************************************
;               INTERRUPT VECTOR                *
;************************************************

                ORG     P:$0010
                JSR     XMT

;************************************************
;               MAIN PROGRAM                    *
;************************************************

                ORG     P:$40

                MOVE    #0,R0           ;Pointer to data buffer.
                MOVE    #3,M0           ;Set modulus to 4.
                MOVE    #0,X0           ;Initialize user flag for SSI flag.
                MOVE    X0,X:FLG        ;Start with the right channel.

;************************************************
;               Initialize SSI Port             *
;************************************************

                MOVEP   #$3000,X:IPR    ;Set interrupt priority register for SSI.
                MOVEP   #$411F,X:CRA    ;Set continuous clock=5.12/32 MHz
                                        ;word length=16.
                MOVEP   #$5B34,X:CRB    ;Enable TIE and TE; make clock and
                                        ;frame sync outputs; frame
                                        ;sync=bit mode; synchronous mode;
                                        ;make SC0 an output.
```

```
;************************************************
;                Init SSI Interrupt             *
;************************************************

                ANDI    #$FC,MR         ;Unmask interrupts.
                MOVEP   #$01F8,X:PCC    ;Turn on SSI port.
                JMP     *               ;Wait for interrupt.

;************************************************
;          MAIN INTERRUPT ROUTINE       *
;************************************************

XMT
                JSET    #0,X:FLG,LEFT   ;Check user flag.

RIGHT           BCLR    #0,X:CRB        ;Clear SC0 indicating right channel
                                        ;data
                MOVEP   X:(R0)+,X:TX    Move data to TX register.
                MOVE    #>$01,X0        ;Set user flag to 1
                MOVE    X0,X:FLG        ;for next data.
                RTI

LEFT            BSET    #0,X:CRB        ;Set SC0 indicating left channel data.
                MOVEP   X0,X:TSR        ;Write to TSR register.
                MOVE    #>$00,X0        ;Clear user flag
                MOVE    X0,X:FLG        ;for next data.
                RTI

                END
```

Figure 11-77 Network Mode Transmit Example Program

```
;************************************************
;          SSI and other I/O EQUATES      *
;************************************************

IPR             EQU     $FFFF
SSISR           EQU     $FFEE
CRA             EQU     $FFEC
CRB             EQU     $FFED
PCC             EQU     $FFE1
RX              EQU     $FFEF
```

```
;***********************************************
;              INTERRUPT VECTOR                 *
;***********************************************

                ORG     P:$000C
                JSR     RCV

;***********************************************
;              MAIN PROGRAM                     *
;***********************************************

                ORG     P:$40
                MOVE    #0,R0                   ;Pointer to memory buffer for
                MOVE    #$08,R1                 ;received data. Note data will be
                MOVE    #3,M0                   ;split between two buffers which are
                MOVE    #3,M1                   ;modulus 4.

;***********************************************
;              Initialize SSI Port              *
;***********************************************

                MOVEP   #$3000,X:IPR            ;Set interrupt priority register for SSI.
                MOVEP   #$4100,X:CRA            ;Set word length = 16 bits.
                MOVEP   #$AB00,X:CRB            ;Enable RIE and RE; synchronous
                                                ;mode with bit frame sync;
                                                ;clock and frame sync are
                                                ;external; SC0 is an input.

;***********************************************
;              Init SSI Interrupt               *
;***********************************************

                ANDI    #$FC,MR                 ;Unmask interrupts.
                MOVEP   #$01F8,X:PCC            ;Turn on SSI port.
                JMP     *                       ;Wait for interrupt.

;***********************************************
;          MAIN INTERRUPT ROUTINE               *
;***********************************************

RCV             JSET    #0,X:SSISR, RIGHT       ;Test SCO flag.

LEFT            MOVEP   X:RX,X:(RO)+            ;If SCO clear, receive data
                RTI                             ;into left buffer (R0).
```

```
RIGHT       MOVEP   X:RX,X:(R1)+    ;If SCO set, receive data
            RTI                     ;into right buffer (R1).

            END
```

Figure 11-78 Network Mode Receive Example Program

11.3.7.3.2 Network Mode Receive

The receive enable will occur only after detection of a new data frame with RE set. The first data word is shifted into the receive shift register and is transferred to the RX, which sets RDF if a frame sync was received (i.e., this is the start of a new frame). Setting RDF will cause a receive interrupt to occur if the receiver interrupt is enabled (RIE=1).

The second data word (second time slot in the frame) begins shifting in immediately after the transfer of the first data word to the RX. The DSP program has to read the data from RX (which clears RDF) before the second data word is completely received (ready to transfer to RX), or a receive overrun error will occur (ROE=1), and the data in the receiver shift register will not be transferred and will be lost.

If RE is cleared and set again by the DSP program, the receiver will be disabled after receiving the current time slot in progress until the next frame sync (first time slot). This mechanism allows the DSP programmer to ignore data in the last portion of a data frame.

Note: The optional frame sync output and clock output signals are not affected, even if the transmitter and/or receiver are disabled. TE and RE do not disable bit clock and frame sync generation.

To summarize, the network mode receiver receives every time slot data word unless the receiver is disabled. An interrupt can occur after the reception of each data word, or the programmer can poll RDF. The DSP program response can be

1. Read RX and use the data.
2. Read RX and ignore the data.
3. Do nothing – the receiver overrun exception will occur at the end of the current time slot.
4. Toggle RE to disable the receiver until the next frame, and read RX to clear RDF.

Figure 11-78 is essentially the same program shown in Figure 11-73 except that this program uses the network mode to receive only right-channel data. In the “Initialize SSI Port” section of the program, two words per frame are selected using the DC bits in the CRA, and the network mode is selected by setting MOD to one in the CRB. If the program in Figure 11-77 is used to transmit to the program in Figure 11-78, the correct data will appear in the data buffer for the right channel, but the buffer for the left channel will probably contain $000000 or $FFFFFF, depending on whether the transmitter output was high or low when TSR was written and whether the output was three-stated.

11.3.7.4 On-Demand Mode Examples

A divide ratio of one (DC=00000) in the network mode is defined as the on-demand mode of the SSI because it is the only data-driven mode of the SSI – i.e., data is transferred whenever data is present (see Figure 11-79 and Figure 11-80). STD and SCK from DSP1 are connected to DSP2 – SRD and SC0, respectively. SC0 is used as an input clock pin in this application. Receive data and receive data clock are separate from the transmit signals. On-demand data transfers are nonperiodic, and no time slots are defined. When there is a clock in the gated clock mode, data is transferred. Although they are not necessarily needed, frame sync and flags are generated when data is transferred. Transmitter underruns (TUE) are impossible in this mode and are therefore disabled. In the on-demand transmit mode, two additional SSI clock cycles are automatically inserted between each data word transmitted. This procedure guarantees that frame sync will be low between every transmitted data word or that the clock will not be continuous between two consecutive words in the gated clock mode. The on-demand mode is similar to the SCI shift register mode with SSFTD equals one and SCKP equals one. The receiver should be configured to receive the bit clock and, if continuous clock is used, to receive an external frame sync. Therefore, for all full-duplex communication in on-demand mode, the asynchronous mode should be used. The on-demand mode is SPI compatible.

Initializing the on-demand mode for the example illustrated in Figure 11-80 is accomplished by setting the bits in CRA and CRB as follows:

1. The word length must be selected by setting WL1 and WL0. In this example, a 24-bit word length was chosen (WL1=1 and WL0=1).
2. The on-demand mode is selected by clearing DC4–DC0.
3. The serial clock rate must be selected by setting PSR and PM7–PM0 (see Tables 11-9 and 11-10).

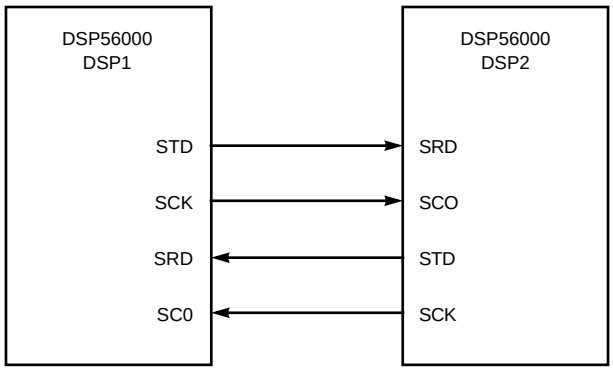
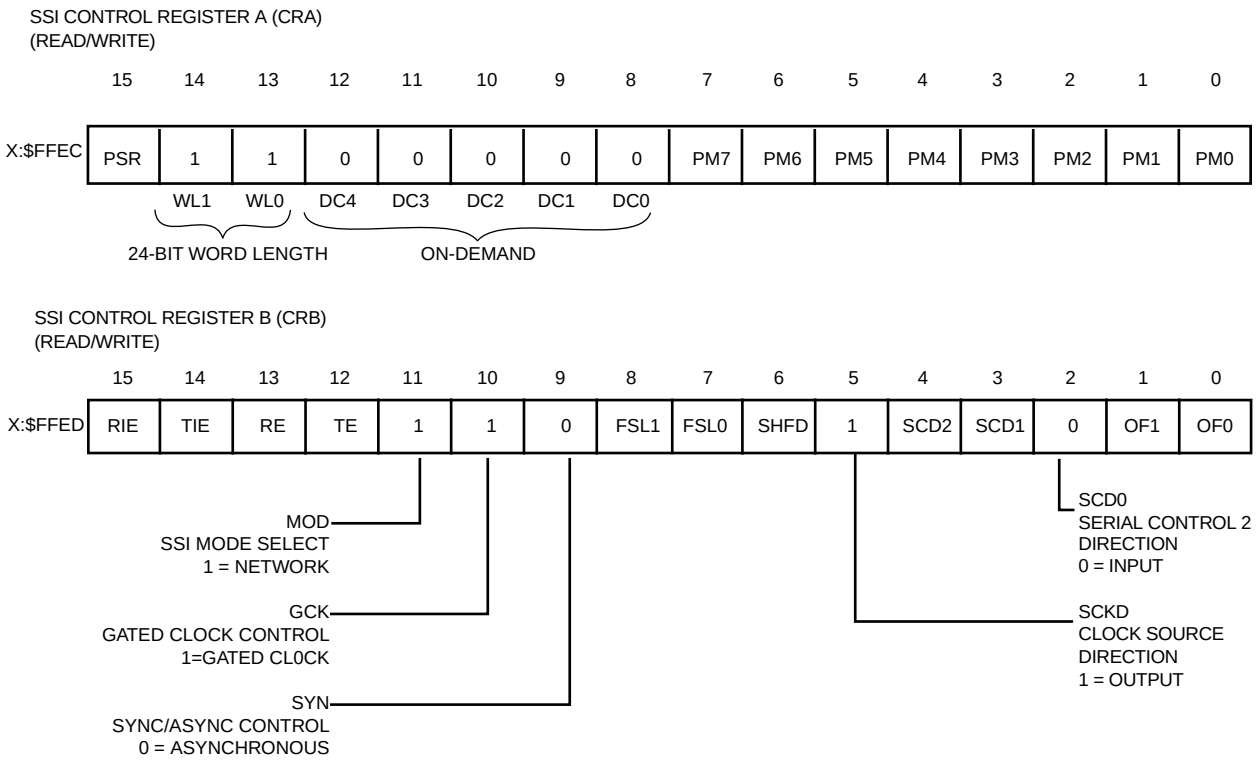


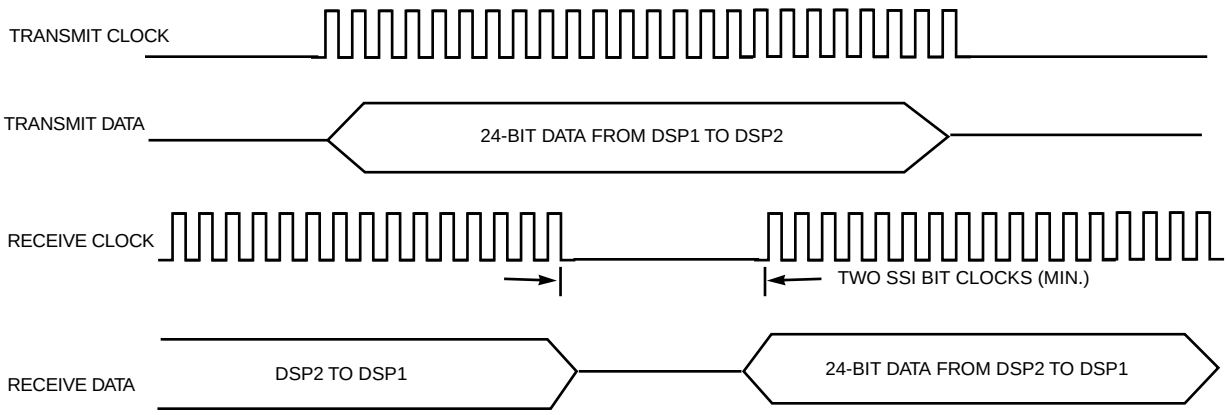
Figure 11-79 On Demand Example

4. RE and TE must be set to activate the transmitter and receiver. If interrupts are to be used, RIE and TIE should be set. RIE and TIE are usually set after everything else is configured and the DSP is ready to receive interrupts.
5. The network mode must be selected (MOD=1).
6. A gated clock (GCK=1) is selected in this example. A continuous clock example is shown in Figure 11-77.
7. Asynchronous clock control was selected (SYN=0) in this example.
8. Since gated clock is used, the frame sync is not necessary. FSL1 and FSL0 can be ignored.
9. SCKD must be an output (SCKD=1).
10. SCD0 must be an input (SCD0=0).
11. Control bit SHFD should be set as needed for the application. Pins SC1 and SC2 are undefined in this mode (see Table 11-7) and should be programmed as general-purpose I/O pins.

11.3.7.4.1 On-Demand Mode – Continuous Clock

This special case will not generate a periodic frame sync. A frame sync pulse will be generated only when data is available to transmit (see Figure 11-81(a)). The frame sync signal indicates the first time slot in the frame. The on-demand mode requires that the transmit frame sync be internal (output) and the receive frame sync be external (input). Therefore, for simplex operation, the synchronous mode could be used; however, for full-duplex operation, the asynchronous mode must be used. Data transmission that is data driven is enabled by writing data into TX. Although the SSI is double buffered, only one word can be written to TX, even if the transmit shift register is empty. The receive and transmit interrupts function as usual using TDE and RDF; however, transmit and receive underruns are impossible for on-demand transmission and are disabled. This mode is useful for interfacing to codecs requiring a continuous clock.





NOTE: Two SSI bit clock times are automatically inserted between each data word. This guarantees frame sync will be low between every data word transmitted and the clock will not be continuous for two consecutive data words.

Figure 11-80 Network Mode Initialization

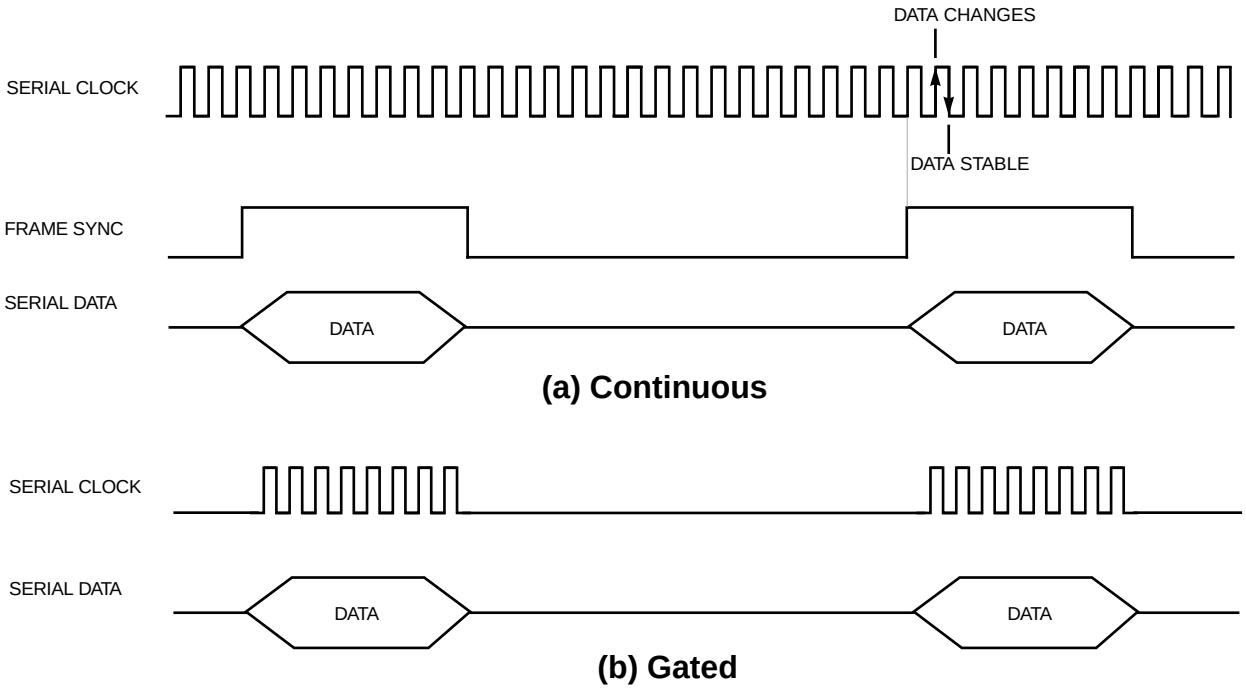


Figure 11-81 Clock Modes

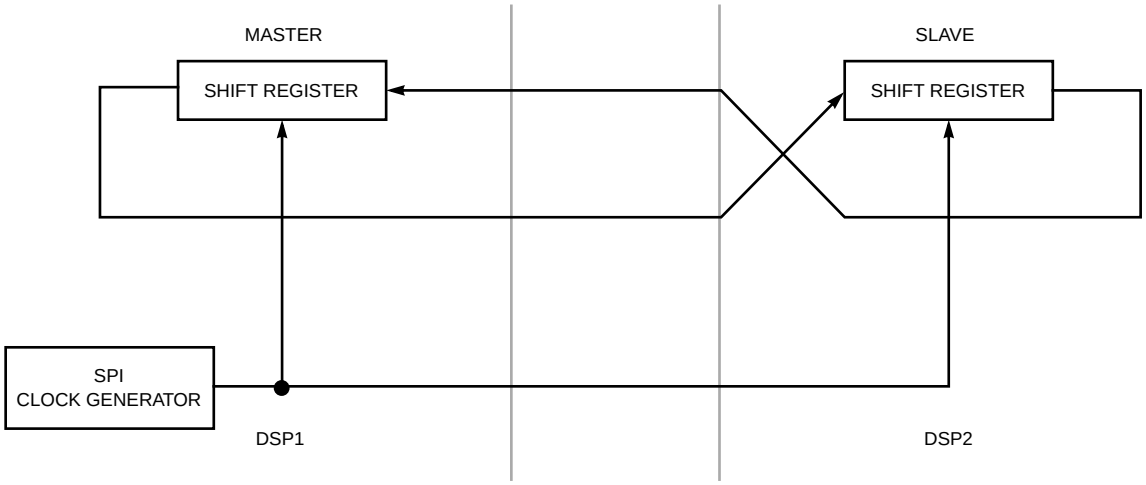


Figure 11-82 SPI Configuration

11.3.7.4.2 On-Demand Mode – Gated Clock

Gated clock mode (see Figure 11-81(b)) is defined for on-demand mode, but the gated clock mode is considered a frame sync source; therefore, in gated clock mode, the transmit clock must be internal (output) and the receive clock must be external (input). For on-demand mode, with internal (output) synchronous gated clock, output clock is enabled for the transmitter and receiver when TX data is transferred to the transmit data shift register. This SPI master operating mode is shown in Figure 11-82. Word sync is inherent in the clock signal, and the operation format must provide frame synchronization.

Figure 11-83 is the block diagram for the program presented in Figure 11-84. This program contains a transmit test program that was written as a scoping loop (providing a repetitive sync) using the on-demand, gated, synchronous mode with no interrupts (polling) to transmit data to the program shown in Figure 11-85. The program also demonstrates using parallel I/O pins as general-purpose control lines. PC3 is used as an external strobe or enable for hardware such as an A/D converter. The transmit program sets equates for convenience and readability. Test data is then written to X: memory, and the data pointer is initialized. Setting M0 to two makes the buffer circular (modulo 3), which saves the step of resetting the pointer each loop. PC3 is configured as a general-purpose output for use as a scope sync, and CRA and CRB are then initialized. Setting the PCC bits begins SSI operation; however, no data will be transmitted until data is written to TX. PC3 is set high at the beginning of data transmission; data is then moved to TX to begin transmission. A JCLR instruction is then used to form a wait loop until TDE equals one and the SSI is ready for another data word to be transmitted. Two more data words are transmitted in this fashion (this is an arbitrary number chosen for this test loop). An additional wait is included to make sure that the frame sync has gone low before PC3 is cleared, indicating on the scope that transmission is complete. A wait of 100 NOPs is implemented by using the REP instruction before starting the loop again.

```
;*********************************************
;            SSI and other I/O EQUATES        *
;*********************************************

CRA             EQU     $FFEC
CRB             EQU     $FFED
PCC             EQU     $FFE1
PCD             EQU     $FFE5
SSISR           EQU     $FFEE
TX              EQU     $FFEF
PCDDR           EQU     $FFE3
```

```
                ORG     X:0
                DC      $AA0000         ;Data to transmit.
                DC      $330000
                DC      $F00000

;***********************************************
;               MAIN PROGRAM                    *
;***********************************************

                ORG     P:$40

                MOVE    #0,R0           ;Pointer to data buffer
                MOVE    #2,M0           ;Length off buffer is 3

                MOVEP   #$08,X:PCDDR    ;SC0 (PC3) as general
                                        ;purpose output.

                MOVEP   #$001F,X:CRA    ;Set Word Length=8, CLK=5.12/32
                                        ;MHz.
                MOVEP   #$1E30,X:CRB    ;Enable transmitter, Mode=On-
                                        ;Demand,
                                        ;Gated clock on, synchronous mode,
                                        ;Word frame sync selected, frame
                                        ;sync and clock are internal and
                                        ;output to port pins.
                MOVEP   #$1F0,X:PCC     ;Set PCC for SSI and
```

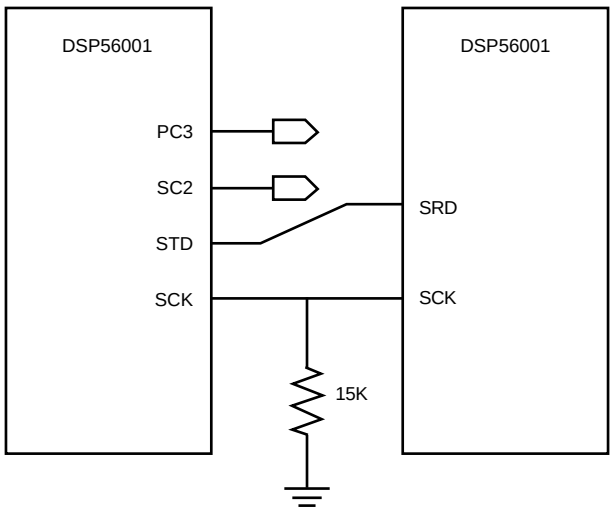


Figure 11-83 On-Demand Mode Example — Hardware Configuration

```
LOOP0       BSET    #3,X:PCD            ;Set PC3 high (this is example enable
                                        ;or strobe for an external device
                                        ;such as an ADC).
            MOVEP   X:(R0);pl,X:TX      ;Move data to TX register
TDE1        JCLR    #6,X:SSISR,TDE1     ;Wait for TDE (transmit data register
                                        ;empty) to go high.
            MOVEP   X:(R0);pl,X:TX      ;Move next data to TX.
TDE2        JCLR    #6,X:SSISR,TDE2     ;Wait for TDE to go high.
            MOVEP   X:(R0);pl,X:TX      ;Move data to TX.
TDE3        JCLR    #6,X:SSISR,TDE3     ;Wait for TDE=1.

FSC         JSET    #5,X:PCD,FSC        ;Wait for frame sync to go low. NOTE:
                                        ;State of frame sync is directly
                                        ;determined by reading PC5.

            BCLR    #3,X:PCD            ;Set PC3 lo (example external
                                        ;enable).

;anything goes here (i.e., any processing)

            REP     #100
            NOP

            JMP     LOOP0               ;Continue sequence forever.

            END
```

Figure 11-84 On-Demand Mode Transmit Example Program

Figure 11-85 is the receive program for the scoping loop program presented in Figure 11-84. The receive program also uses the on-demand, gated, synchronous mode with no interrupts (polling). Initialization for the receiver is slightly different than for the transmitter. In CRB, RE is set rather than TE, and SCKD and SCD2 are inputs rather than outputs. After initialization, a JCLR instruction is used to wait for a data word to be received (RDF=1). When a word is received, it is put into the circular buffer and loops to wait for another data word. The data in the circular buffer will be overwritten after three words are received (does not matter in this application).

```
;************************************************
;            SSI and other I/O EQUATES         *
;************************************************

CRA         EQU     $FFEC
CRB         EQU     $FFED
```

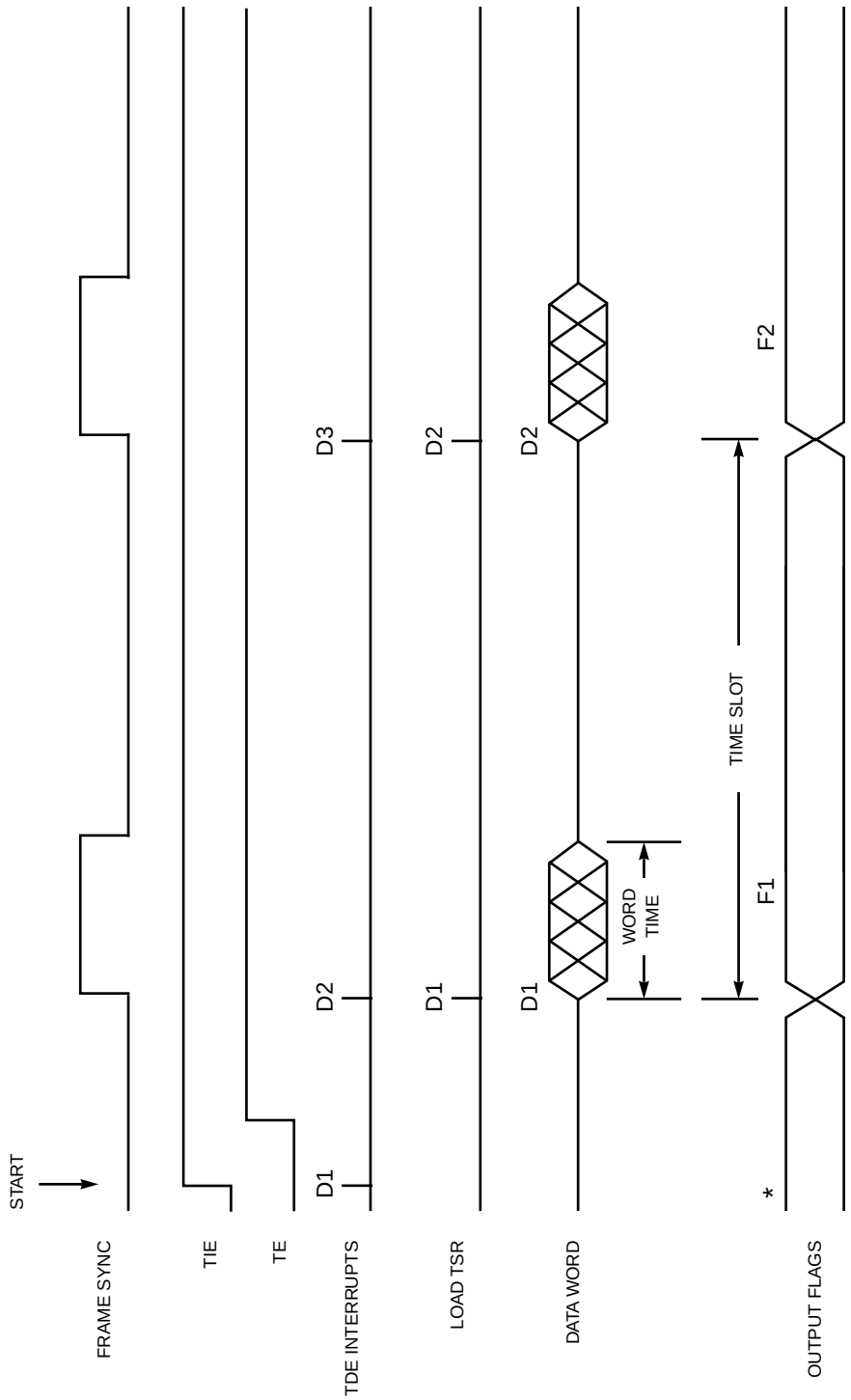
```
PCC             EQU     $FFE1
PCD             EQU     $FFE5
SSISR           EQU     $FFEE
RX              EQU     $FFEF
PCDDR           EQU     $FFE3

;************************************************
;               MAIN PROGRAM                    *
;************************************************

                ORG     P:$40

                MOVE    #0,R0           ;Pointer to data buffer
                MOVE    #2,M0           ;Length of buffer is 3

                MOVEP   #$001F,X:CRA    ;Set Word Length=8, CLK=5.12/32
                                        ;MHz.
                MOVEP   #$1E30,X:CRB    ;Enable receiver, Mode=On-
                                        ;Demand, gated clock on,
                                        synchronous mode,
                                        ;Word frame sync selected, frame
                                        ;sync and clock are external.
                MOVEP   #$1F0,X:PCC     ;Set PCC for SSI

LOOP0

RDF1            JCLR    #7,X:SSISR,RDF1 ;Wait for RDF (receive data register
                                        ;Full) go to high.
                MOVEP   X:RX,X:(R0)+    ;Read data from RX into memory.

                JMP     LOOP            ;Continue sequence forever.

                END
```

Figure 11-85 On-Demand Mode Receive Example Program

11.3.8 Flags

Two SSI pins (SC1 and SC0) are available in the synchronous mode for use as serial I/O flags. The control bits (OF1 and OF0) and status bits (IF1 and IF0) are double buffered to/from SC1 and SC0. Double buffering the flags keeps them in sync with TX and RX. The direction of SC1 and SC0 is controlled by SCD1 and SCD0 in CRB.

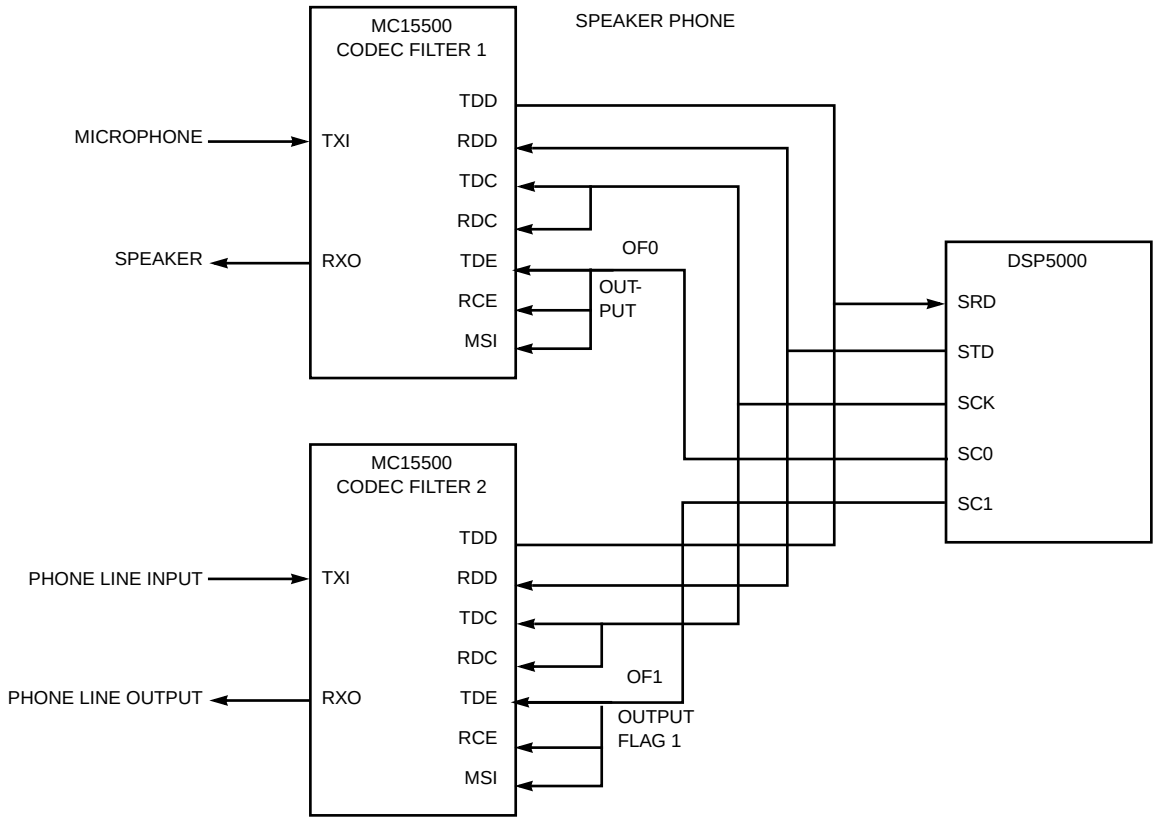
Figure 11-86 shows the flag timing for a network mode example. Initially, neither TIE nor



NOTES:
1. Fn = flags associated with Dn data.
2. Output flags are double buffered with transmit data.
3. Output flags change when data is transferred from TX to the transmit data shift register.
4. Initial flag outputs (*) = last flag output value.
5. Data and flags transition after external frame sync but not before rising edge of clock.

Figure 11-86 Output Flag Timing

TE is set, and the flag outputs are the last flag output value. When TIE is set, a TDE



NOTE: SC0 and SC1 are output flag 0 and 1 used to software select either filter 1 or 2.

Figure 11-87 Output Flag Example

interrupt occurs (the transmitter does not have to be enabled for this interrupt to occur). Data (D1) is written to TX, which clears TDE, and the transmitter is enabled by software. When the frame sync occurs, data (D1) is transferred to the transmit shift register, setting TDE. Data (D1) is shifted out during the first word time, and the output flags are updated. These flags will remain stable until the next frame sync. The TDE interrupt is then serviced by writing data (D2) to TX, clearing TDE. After the TSR completes transmission, the transmit pin is three-stated until the next frame sync

Figure 11-87 shows a speaker phone example that uses a DSP56000 and two codecs. No additional logic is required to connect the codecs to the DSP. The two serial output flags in this example (OF1 and OF0) are used as chip selects to enable the appropriate codec for I/O. This procedure allows the transmit lines to be ORed together. The appropriate output flag pin changes at the same time as the first bit of the transmit word and remains stable until the next transmit word (see Figure 11-88). Applications include serial-device chip selects, implementing multidrop protocols, generating Bell PCM signaling frame syncs, and outputting status information.

Initializing the flags (see Figure 11-88) is accomplished by setting SYN, SCD1, and

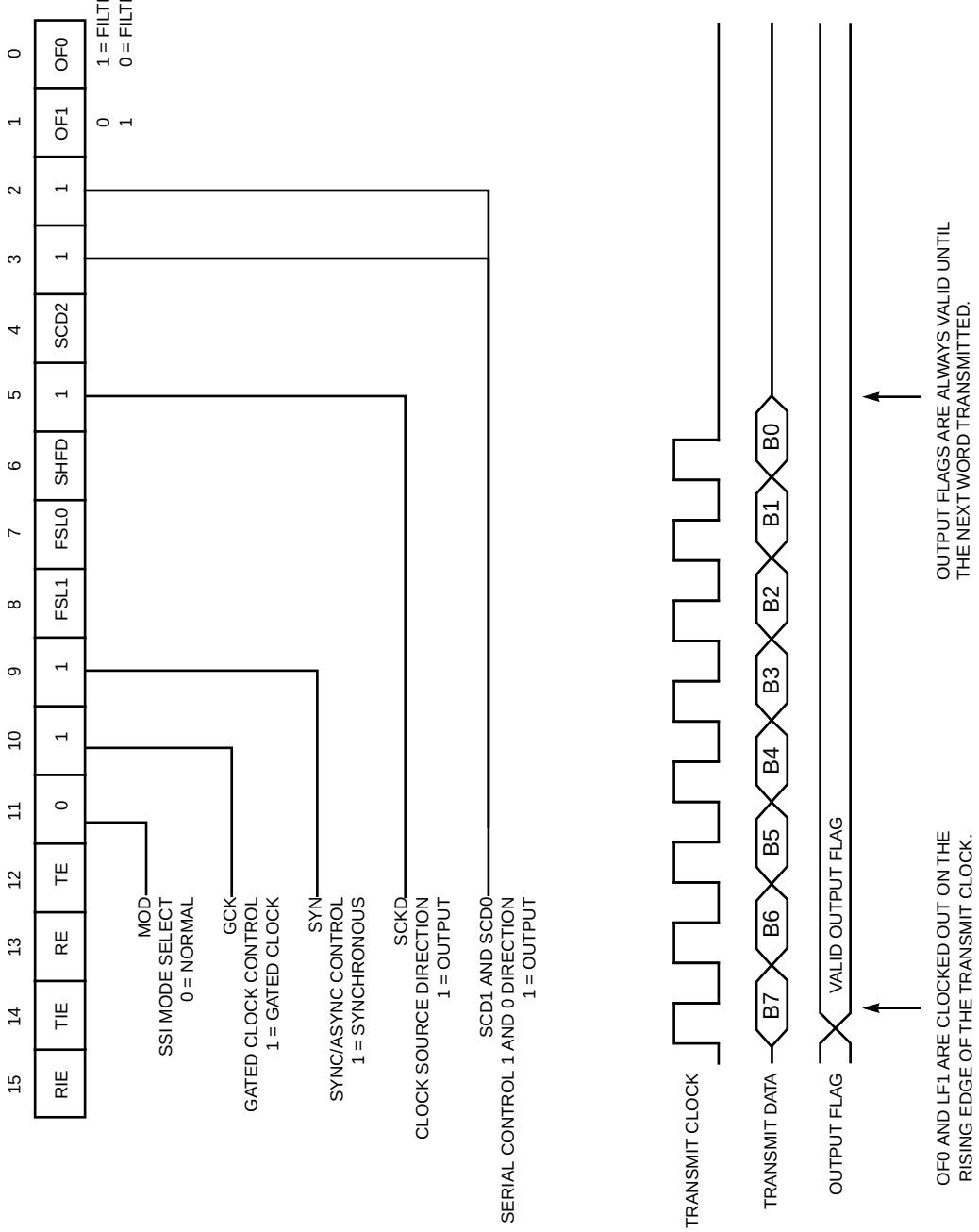


Figure 11-88 Output Flag Initialization

SCD0. No other control bits affect the flags. The synchronous control bit must be set

(SYN=1) to select the SC1 and SC0 pins as flags. SCD1 and SCD0 select whether SC1 and SC0 are inputs or outputs (input=0, output=1). The other bits selected in Figure 11-88 are chosen for the speaker phone example in Figure 11-87. In this example, the codecs require that the SSI be set for normal mode (MOD=0) with a gated clock (GCK=1) out (SCKD=1).

Serial input flags, IF1 and IF0, are latched at the same time as the first bit is sampled in the receive data word (see Figure 11-89). Since the input was latched, the signal on the input flag pin can change without affecting the input flag until the first bit of the next receive data word. To initialize SC1 or SC0 as input flags, the synchronous control bit in CRB must be set to one (SYN=1) and SCD1 set to zero for pin SC1, and SCD0 must be set to zero for pin SC0. The input flags are bits 1 and 0 in the SSISR (at X:$FFEE).

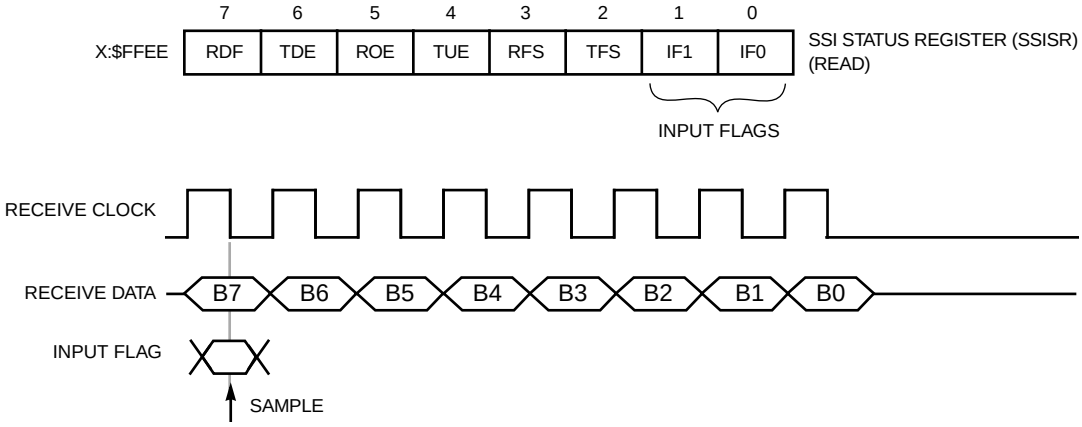


Figure 11-89 Input Flags

11.3.9 Example Circuits

The DSP-to-DSP serial network shown in Figure 11-90 uses no additional logic chips for the network connection. All serial data is synchronized to the data source (all serial clocks and serial syncs are common). This basic configuration is useful for decimation and data reduction when more processing power is needed than one DSP can provide. Cascading DSPs in this manner is useful in several network topologies including star and ring networks.

TDM networks are useful to reduce the wiring needed for connecting multiple processors. A TDM parallel topology, such as the one shown in Figure 11-91, is useful for interpolating filters. Serial data can be received simultaneously by all DSPs, processing can occur in parallel, and the results are then multiplexed to a single serial data out line. This configuration can be cascaded and/or looped back on itself as needed to fit a particular application (see Figure 11-92). The serial and parallel configurations can be combined to form the array processor shown in Figure 11-93. A nearest neighbor array, which is

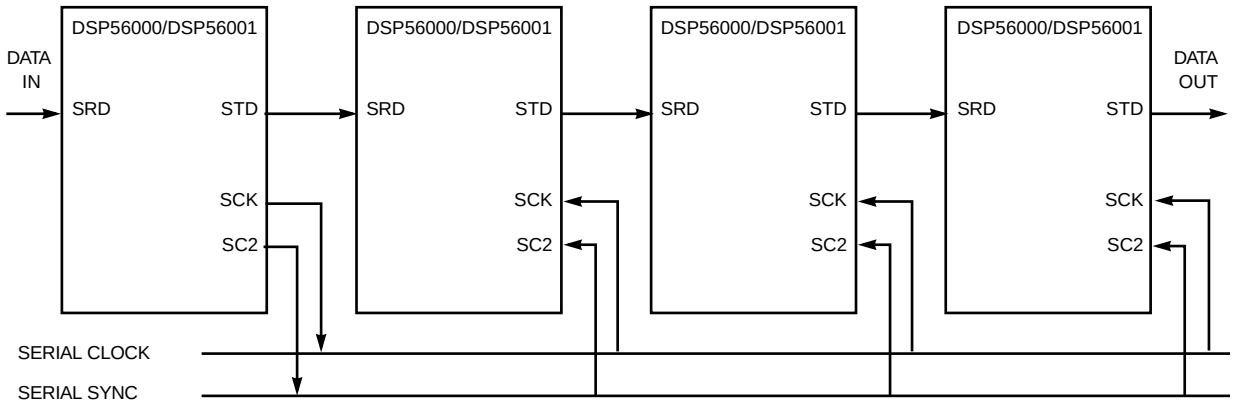


Figure 11-90 SSI Cascaded Multi-DSP System

applicable to matrix relaxation processing, is shown in Figure 11-94. To simplify the drawing, only the center DSP is connected in this illustration. In use, all DSPs would have four three-state buffers connected to their STD pin. The flags (SC0 and SC1) on the control master operate the three-state buffers, which control the direction that data is transferred in the matrix (north, south, east, or west).

The bus architecture shown in Figure 11-97 allows data to be transferred between any two DSPs. However, the bus must be arbitrated by hardware or a software protocol to prevent collisions. The master/slave configuration shown in Figure 11-96 also allows data to be transferred between any two DSPs but simplifies network control.

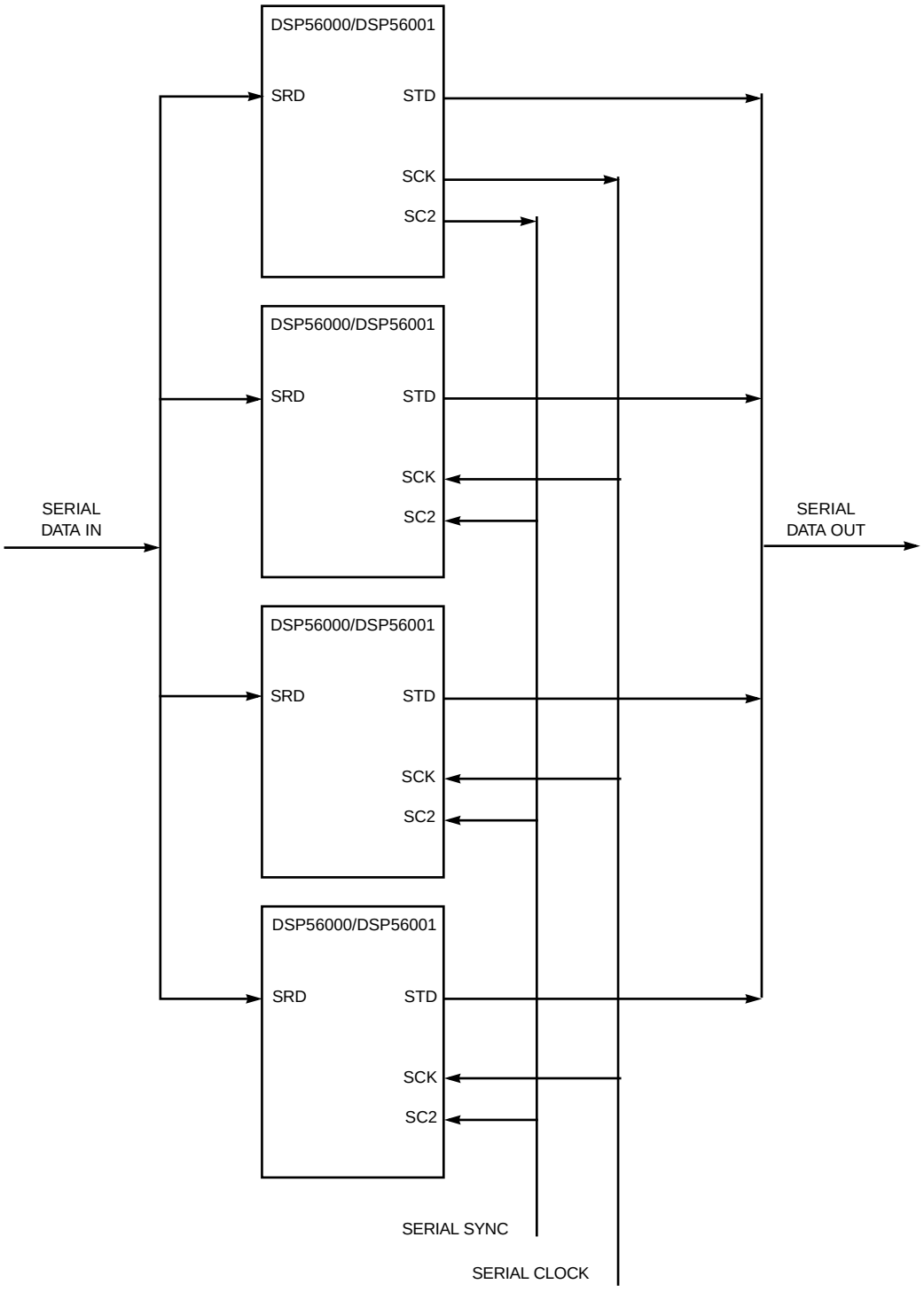


Figure 11-91 SSI TDM Parallel DSP Network

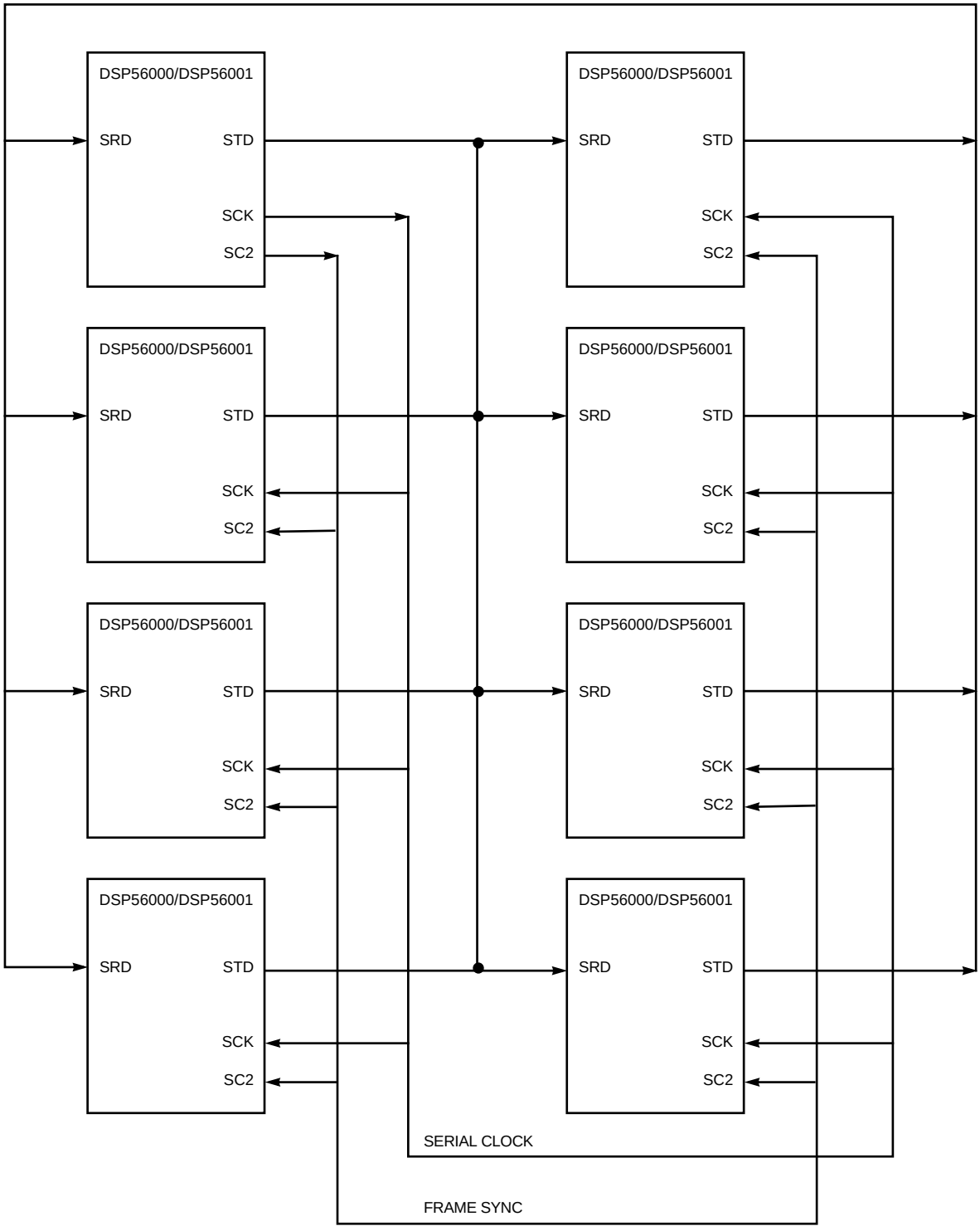


Figure 11-92 SSI TDM Connected Parallel Processing Array

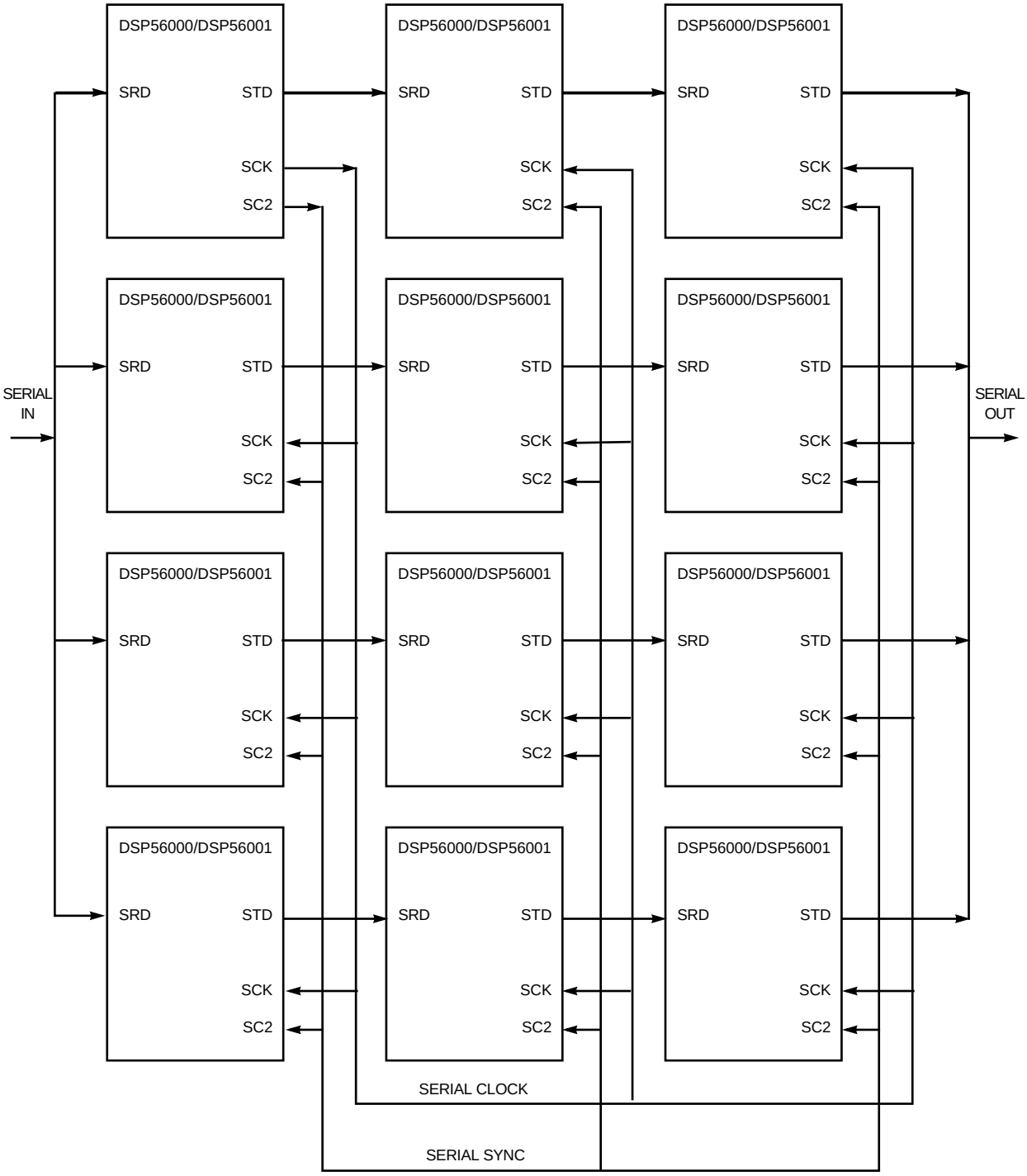


Figure 11-93 SSI TDM Serial/Parallel Processing Array

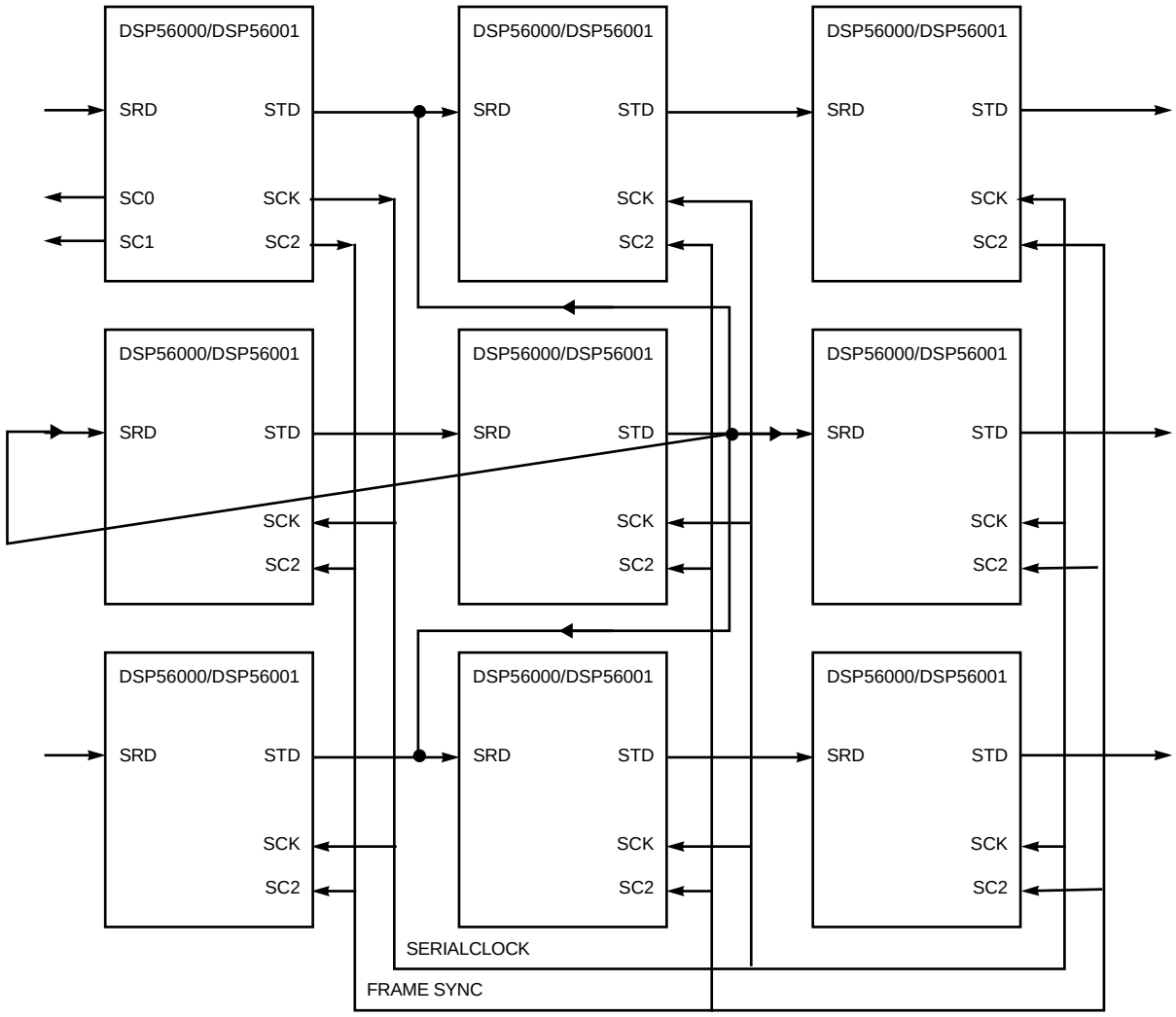
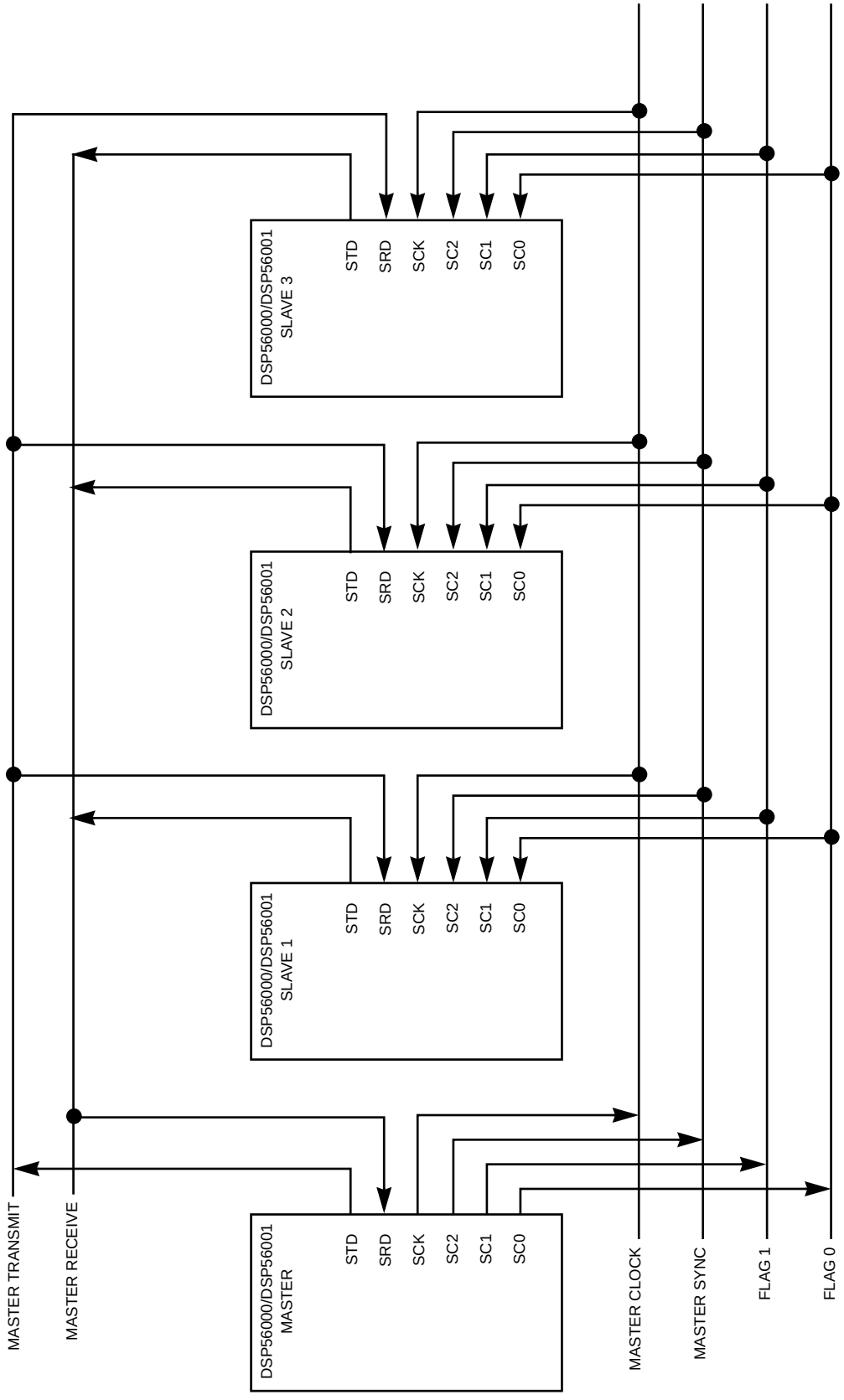


Figure 11-94 SSI Parallel Processing — Nearest Neighbor Array



NOTE: Flags can specify data types: control, address, and data.

Figure 11-96 SSI TDM Master-Slave DSP Network

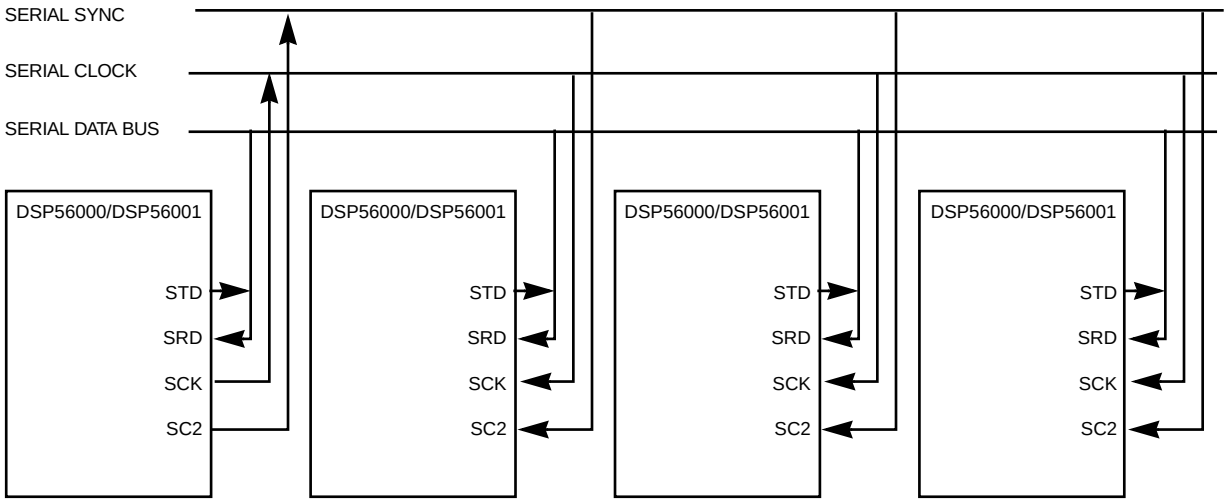


Figure 11-97 SSI TDM Bus DSP Network

APPENDIX A
INSTRUCTION SET DETAILS

This appendix contains detailed information about each instruction in the DSP56000/DSP56001 instruction set. An instruction guide is presented first to help understand the individual instruction descriptions. This guide is followed by sections on notation and addressing modes. Since parallel moves are allowed with many of the instructions, they are discussed before the instructions. The instructions are then discussed in alphabetical order.

A.1 INSTRUCTION GUIDE

The following information is included in each instruction description with the goal of making each description self-contained:

1. **Name and Mnemonic:** The mnemonic is highlighted in **bold** type for easy reference.

2. **Assembler Syntax and Operation:** For each instruction syntax, the corresponding operation is symbolically described. If there are several operations indicated on a single line in the operation field, those operations do not necessarily occur in the order shown but are generally assumed to occur in parallel. If a parallel data move is allowed, it will be indicated in parenthesis in both the assembler syntax and operation fields. If a letter in the mnemonic is optional, it will be shown in parenthesis in the assembler syntax field.

3. **Description:** A complete text description of the instruction is given together with any special cases and/or condition code anomalies of which the user should be aware when using that instruction.

4. **Example:** An example of the use of the instruction is given. The example is shown in DSP56000/DSP56001 assembler source code format. Most arithmetic and logical instruction examples include one or two parallel data moves to illustrate the many types of parallel moves that are possible. The example includes a complete explanation, which discusses the contents of the registers referenced by the instruction (but not those referenced by the parallel moves) both before and after the execution of the instruction. Most examples are designed to be easily understood without the use of a calculator.

5. **Condition Codes:** The status register is depicted with the condition code bits which can be affected by the instruction highlighted in **bold** type. Not all bits in the status register are used. Those which are reserved are indicated with a double asterisk and are read as zeros.

6. **Instruction Format:** The instruction fields, the instruction opcode, and the instruction extension word are specified for each instruction syntax. When the extension

word is optional, it is so indicated. The values which can be assumed by each of the variables in the various instruction fields are shown under the instruction field's heading. Note that the symbols used in decoding the various opcode fields of an instruction are **completely arbitrary**. Furthermore, the opcode symbols used in one instruction are **completely independent** of the opcode symbols used in a different instruction.

7. **Timing:** The number of oscillator clock cycles required for each instruction syntax is given. This information provides the user a basis for comparison of the execution times of the various instructions in oscillator clock cycles. Refer to Table A-1 and A.7 INSTRUCTION TIMING for a complete explanation of instruction timing, including the meaning of the symbols ''aio'', ''ap'', ''ax'', ''ay'', ''axy'', ''ea'', ''jx'', ''mv'', ''mvb'', ''mvc'', ''mvm'', ''mvp'', ''rx'', ''wio'', ''wp'', ''wx'', and ''wy''.

8. **Memory:** The number of program memory words required for each instruction syntax is given. This information provides the user a basis for comparison of the number of program memory locations required for each of the various instructions in 24-bit program memory words. Refer to Table A-1 and A.7 INSTRUCTION TIMING for a complete explanation of instruction memory requirements, including the meaning of the symbols ''ea'' and ''mv''.

A.2 NOTATION

Each instruction description contains symbols used to abbreviate certain operands and operations. Table A-1 lists the symbols used and their respective meanings. Depending on the context, registers refer to either the register itself or the contents of the register.

A.3 ADDRESSING MODES

The addressing modes are grouped into three categories: register direct, address register indirect, and special. These addressing modes are summarized in Table A-2. All address calculations are performed in the address ALU to minimize execution time and loop overhead. Addressing modes, which specify whether the operands are in registers, in memory, or in the instruction itself (such as immediate data), 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 X1, X0, Y1, Y0, X, Y, A2, A1, A0, B2, B1, B0, A, and B. The control registers include SR, OMR, SP, SSH, SSL, LA, LC, CCR, and MR.

Address register indirect modes use an address register Rn (R0–R7) to point to locations in X, Y, and P memory. The contents of the Rn address register (Rn) is the effective address (ea) of the specified operand, except in the ''indexed by offset'' mode where the effective address (ea) is (Rn+Nn). Address register indirect modes use an address modifier register Mn to specify the type of arithmetic to be used to update the address register Rn. If an addressing mode specifies an address offset register Nn, the given address offset register is used to update the corresponding address register Rn. The Rn address register may only use the corresponding address offset register Nn and the corresponding address modifier register Mn. For example, the address register R0 may only use the

Table A-1 Instruction Description Notation

Data ALU Registers Operands

Xn	Input Register X1 or X0 (24 Bits)
Yn	Input Register Y1 or Y0 (24 Bits)
An	Accumulator Registers A2, A1, A0 (A2 — 8 Bits, A1 and A0 — 24 Bits)
Bn	Accumulator Registers B2, B1, B0 (B2 — 8 Bits, B1 and B0 — 24 Bits)
X	Input Register X = X1: X0 (48 Bits)
Y	Input Register Y = Y1: Y0 (48 Bits)
A	Accumulator A = A2: A1: A0 (56 Bits)*
B	Accumulator B = B2: B1: B0 (56 BIts)*
AB	Accumulators A and B = A1: B1 (48 Bits)*
BA	Accumulators B and A = B1: A1 (48 Bits)*
A10	Accumulator A = A1: A0 (48 Bits)
B10	Accumulator B= B1:B0 (48 bits)
*** NOTE:** In data move operations, shifting and limiting are performed when this register is specified as a source operand. When specified as a destination operand, sign extension and possibly zeroing are performed.	

Address ALU Registers Operands

Rn	Address Registers R0 - R7 (16 Bits)
Nn	Address Offset Registers N0 - N7 (16 Bits)
Mn	Address Modifier Registers M0 - M7 (16 Bits)

N0 address offset register and the M0 address modifier register during actual address computation and address register update operations. This unique implementation is extremely powerful and allows the user to easily address a wide variety of DSP-oriented data structures. All address register indirect modes use at least one set of address registers (Rn, Nn, and Mn), and the XY memory reference uses two sets of address registers, one for the X memory space and one for the Y memory space.

The special addressing modes include immediate and absolute addressing modes as well as implied references to the program counter (PC), the system stack (SSH or SSL), and program (P) memory.

Addressing modes may also be categorized by the ways in which they may be used.

Table A-1 Instruction Description Notation (Continued)

Program Controller Registers Operands

PC	Program Counter Register (16 Bits)
MR	Mode Register (8 Bits)
CCR	Condition Code Register (8 Bits)
SR	Status Register = MR:CCR (16 Bits)
OMR	Operating Mode Register (8 Bits)
LA	Hardware Loop Address Register (16 Bits)
LC	Hardware Loop Counter Register (16 Bits)
SP	System Stack Pointer Register (6 Bits)
SSH	Upper Portion of the Current Top of the Stack (16 Bits)
SSL	Lower Portion of the Current Top of the Stack (16 Bits)
SS	System Stack RAM = SSH: SSL (15 Locations by 32 Bits)

Address Operands

ea	Effective Address
eax	Effective Address for X Bus
eay	Effective Address for Y Bus
xxxx	Absolute Address (16 Bits)
xxx	Short Jump Address (12 Bits)
aa	Absolute Short Address (6 Bits, Zero Extended)
pp	I/O Short Address (6 Bits, Ones Extended)
<. . .>	Specifiies the Contents of the Specified Address
X:	X Memory Reference
Y:	Y Memory Reference
L:	Long Memory Reference = X:Y
P:	Program Memory Reference

Table A-3 shows the various categories to which each addressing mode belongs. The following classifications will be used in the instruction descriptions.

Table A-3. DSP56000/DSP56001 Addressing Mode Encoding

These addressing mode categories may be combined so that additional, more restrictive classifications may be defined. For example, the instruction descriptions may use a

Table A-1 Instruction Description Notation (Continued)

Miscellaneous Operands

S, Sn	Source Operand Register
D, Dn	Destination Operand Register
D [n]	Bit n of D Destination Operand Register
#n	Immediate Short Data (5 Bits)
#xx	Immediate Short Data (8 Bits)
#xxx	Immediate Short Data (12 Bits)
#xxxxxx	Immediate Data (24 Bits)

Unary Operators

-	Negation Operator
—	Logical NOT Operator
PUSH	Push Specified Value onto the System Stack (SS) Operator
PULL	Pull Specified Value from the System Stack (SS) Operator
READ	Read the Top of the System Stack (SS) Operator
PURGE	Delete the Top Value on the System Stack (SS) Operator
| |	Absolute Value Operator

Binary Operators

+	Addition Operator
-	Subtraction Operator
*	Multiplication Operator
÷, /	Division Operator
+	Logical Inclusive OR Operator
•	Logical AND Operator
⊕	Logical Exclusive OR Operator
➡	“Is Transferred To” Operator
:	Concatenation Operator

memory alterable classification, which refers to addressing modes that are **both** mem-

Table A-1 Instruction Description Notation (Continued)

Addressing Mode Operators

<<	I/O Short Addressing Mode Force Operator
<	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

Mode Register (MR) Symbols

LF	Loop Flag Bit Indicating When a DO Loop is in Progress
T	Trace Mode Bit Indicating if the Tracing Function has been Enabled
S1, S0	Scaling Mode Bits Indicating the Current Scaling Mode
I1, I0	Interrupt Mask Bits Indicating the Current Interrupt Priority Level

Condition Code Register (CCR) Symbols

Standard Definitions (Table A - 3 Describes Exceptions)

L	Limit Bit Indicating Arithmetic Overflow and/or Data Shifting/Limiting
E	Extension Bit Indicating if the Integer Portion of A or B is in Use
U	Unnormalized Bit Indicating if the A or B Result is Unnormalized
N	Negative Bit Indicating if Bit 55 of the A or B Result is Set
Z	Zero Bit Indicating if the A or B Result Equals Zero
V	Overflow Bit Indicating if Arithmetic Overflow has Occurred in A or B
C	Carry Bit Indicating if a Carry or Borrow Occurred in A or B Result

ory addressing modes **and** alterable addressing modes. Thus, memory alterable addressing modes use address register indirect and absolute addressing modes.

The address register indirect addressing modes require that the offset register number be the same as the address register number. However, future family members may allow the offset register number to be different from the address register number. The assembler syntax ‘‘Nn’’ supports the future feature. The assembler syntax ‘‘N’’ may be used

Table A-1 Instruction Description Notation (Continued)

Instruction Timing Symbols

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
ay	Time Required to Access a Y Memory Operand
axy	Time Required to Access XY Memory Operands
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
rx	Time Required to Execute Part of an TRTI or RTS Instruction
wio	Number of Wait States Used in Accessing External I/O
wp	Number of Wait States Used in Accessing External P Memory
wx	Number of Wait States Used in Accessing External X Memory
wy	Number of Wait States Used in Accessing External Y Memory

Other Symbols

( )	Optional Letter, Operand, or Operation
(. . . . .)	Any Arithmetic or Logical Instruction Which Allows Parallel Moves
EXT	Extension Register Portion of an Accumulator (A2 or B2)
LS	Least Significant
LSP	Least Significant Portion of an Accumulator (A0 or B0)
MS	Most Significant
MSP	Most Significant Portion of a n Accumulator (A1 or B1)
r	Rounding constant
S/L	Shifting and/or Limiting on a Data ALU Register
Sign Ext	Sign Externsion of a Data ALU Register
Zero	Zeroing of a Data ALU Register

Table A-2 DSP 56000/56001 Addressing Modes

Addressing Mode	Uses Mn Modifier	Operand Reference								
		S	C	D	A	P	X	Y	L	XY
Register Direct										
Data or Control Register	No	X	X	X						
Address Register Rn	No				X					
Address Modifier Register Mn	No				X					
Address Offset Register Nn	No				X					
Address Register Indirect										
No Update	Yes					X	X	X	X	X
Postincrement by 1	Yes					X	X	X	X	X
Postdecrement by 1	Yes					X	X	X	X	X
Postincrement by Offset Nn	Yes					X	X	X	X	X
Postdecrement by Offset Nn	Yes					X	X	X	X	
Indexed by Offset Nn	Yes					X	X	X	X	
Predecrement by 1	Yes					X	X	X	X	
Special										
Immediate Data	No					X				
Absolute Address	No					X	X	X	X	
Immediate Short Data	No					X				
Short Jump Address	No					X				
Absolute Short Address	No					X	X	X	X	
I/O Short Address	No						X	X		
Implicit	No	X	X			X				

NOTE:S = System Stack Reference
C = Program Controller Register Reference
D = Data ALU Register Reference
A = Address ALU Register Reference
P = Program Memory Reference
X = X Memory Reference
Y = Y Memory Reference
L = L Memory Reference
XY = XY Memory Reference

instead of ‘‘Nn’’ in the address register indirect memory addressing modes. If ‘‘N’’ is specified, the offset register number is the same as the address register number.

Table A-3 DSP56000/56001 Addressing Mode Encoding

Addressing Mode	Mode MMM	Reg RRR	Addressing Categories				Assembler Syntax
			U	P	M	A	
Register Direct							
Data or Control Register	—	—				X	(SeeTable A-1)
Address Register	—	—				X	Rn
Address Offset Register	—	—				X	Nn
Address Modifier Register	—	—				X	Mn
Address Register Indirect							
No Update	100	Rn		X	X	X	(Rn)
Postincrement by 1	011	Rn	X	X	X	X	(Rn) +
Postdecrement by 1	010	Rn	X	X	X	X	(Rn) -
Postincrement by Offset Nn	001	Rn	X	X	X	X	(Rn) + Nn
Postdecrement by Offset Nn	000	Rn	X		X	X	(RN) - Nn
Indexed by Offset Nn	101	Rn			X	X	(Rn + Nn)
Predecrement by 1	111	Rn			X	X	- (Rn)
Special							
Immediate Data	110	100			X		#xxxxxx
Absolute Address	110	000			X	X	xxxx
Immediate Short Data	—	—					#xx
Short Jump Address	—	—				X	xxx
Absolute Short Address	—	—				X	aa
I/O Short Address	—	—				X	pp
Implicit	—	—				X	

Update Mode (U) The update addressing mode is used to modify address registers without any associated data move.

Parallel Mode (P) The parallel addressing mode is used in instructions where two effective addresses are required.

Memory Mode (M) The memory addressing mode is used to refer to operands in memory using an effective addressing field.

Alterable Mode (A) The alterable addressing mode is used to refer to alterable or writable registers or memory.

A.3.1 Addressing Mode Modifiers

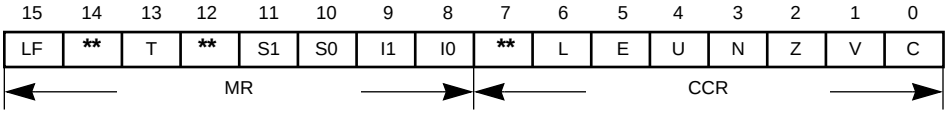
The addressing mode selected in the instruction word is further specified by the contents of the address modifier register Mn. The addressing mode update modifiers (M0–M7) are shown in Table A-4. There are no restrictions on the use of modifier types with any address register indirect addressing mode.

Table A-4 Addressing Mode Modifier Summary

16-Bit Modifier Reg. (M0 - M7) MMMMMMMMMMMMMMMM*	Address Calculation Arithmetic
0000000000000000	Reverse Carry (Bit Reversed)
0000000000000001	Modulo 2
0000000000000010	Modulo 3
•	• •
•	• •
0111111111111110	Modulo 32767
0111111111111111	Modulo 32768
1000000000000000	Undefined
•	• •
1111111111111110	Undefined
1111111111111111	Linear (Modulo 65536)

***MMMMMMMMMMMMMMMM = 16-Bit Modifier Reg. Contents

A.4 CONDITION CODE COMPUTATION



The condition code register (CCR) portion of the status register (SR) consists of seven defined bits:

L — Limit Bit
E — Extension Bit
U — Unnormalized Bit
N — Negative Bit
Z — Zero Bit
V — Overflow Bit
C — Carry Bit

The E, U, N, Z, V, and C bits are **true** condition code bits that reflect the condition of the **result of a data ALU operation**. These condition code bits are **not latched** and are **not affected by address ALU calculations or by data transfers** over the X, Y, or global data buses. The L bit is a **latching overflow bit** which indicates that an overflow has occurred in the data ALU or that data limiting has occurred when moving the contents of the Aand/or[lz B accumulators.

The standard definition of the condition code bits is as follows. Exceptions to these standard definitions are given in Table A-5.

L (Limit Bit)

Set if the overflow bit V is set or if the data shifter/limiters perform a limiting operation. Not affected otherwise. This bit is **latched** and must be reset by the user.

E (Extension Bit)

Cleared if all the bits of the **signed integer portion** of the A or B result are the **same** =m i.e., the bit patterns are either 00 . . . 00 or 11 . . . 11. Set otherwise. The signed integer portion is defined by the scaling mode as shown in the following table:

S1	S0	Scaling Mode	Signed Integer Portion
0	0	No Scaling	Bits 55, 54, ... 48, 47
0	1	Scale Down	Bits 55, 54, ... 49, 48
1	0	Scale Up	Bits 55, 54, ... 47, 46

Note that the **signed integer portion** of an accumulator **IS NOT** necessarily the same as the **extension register portion** of that accumulator. The signed integer portion of an accumulator consists of the MS 8, 9, or 10 bits of that accumulator, depending on the scaling mode being used. The extension register portion of an accumulator (A2 or B2) is always the MS 8 bits of that accumulator. **The E bit refers to the signed integer portion of an accumulator and NOT the extension register portion of that accumulator.** For example, if the current scaling mode is set for no scaling (i.e., S1=S0=0), the signed integer portion of the A or B accumulator consists of **bits 47 through 55**. If the A accumulator contained the signed 56-bit value $00:800000:000000 as a **result of a data ALU operation**, the E bit **would** be set (E=1) since the **9** MS bits of that accumulator were not all the same (i.e., neither 00 . . 00 nor 11 . . 11). This means that data limiting **will** occur if that 56-bit value is specified as a **source** operand in a move-type operation. This limiting operation will result in either a positive or negative, 24-bit or 48-bit saturation constant being stored in the specified destination. The **only** situation in which the signed integer portion of an accumulator and the extension register portion of an accumulator are the same is in the ''Scale Down'' scaling mode (i.e., S1=0 and S0=1).

U (Unnormalized Bit)

Set if the two MS bits of the MSP portion of the A or B result are the same. Cleared otherwise. The MSP portion is defined by the scaling mode. The U bit is computed as follows:

S1	S0	Scaling Mode	U Bit Computation
0	0	No Scaling	$U=(\overline{\text{Bit 47}} \oplus \overline{\text{Bit 46}})$
0	1	Scale Down	$U=(\overline{\text{Bit 48}} \oplus \overline{\text{Bit 47}})$
1	0	Scale Up	$U=(\overline{\text{Bit 46}} \oplus \overline{\text{Bit 45}})$

N (Negative Bit)

Set if the MS bit 55 of the A or B result is set. Cleared otherwise.

Z (Zero Bit)

Set if the A or B result equals zero. Cleared otherwise.

V (Overflow Bit)

Set if an arithmetic overflow occurs in the 56-bit A or B result. This indicates that the result cannot be represented in the 56-bit accumulator; thus, the accumulator has overflowed. Cleared otherwise.

C (Carry Bit)

Set if a carry is generated out of the MS bit of the A or B result of an addition or if a borrow is generated out of the MS bit of the A or B result of a subtraction. The carry or borrow is generated out of bit 55 of the A or B result. Cleared otherwise.

Table A-5 details how each instruction affects the condition codes. The convention for the notation that is used is shown at the bottom of Table A-5.

A.5 PARALLEL MOVE DESCRIPTIONS

Many of the instructions in the DSP56000/DSP56001 instruction set allow optional parallel data bus movement. A.6 INSTRUCTION DESCRIPTIONS indicates the parallel move option in the instruction syntax with the statement ''(parallel move)''. The MOVE instruction is equivalent to a NOP with parallel moves. Therefore, a detailed description of each parallel move is given with the MOVE instruction details in A.6 INSTRUCTION DESCRIPTIONS.

A.6 INSTRUCTION DESCRIPTIONS

The following section describes each instruction in the DSP56000/DSP56001 instruction set in complete detail. The format of each instruction description is given in A.1 INSTRUCTION GUIDE. Instructions which allow parallel moves include the notation ''(parallel move)'' in both the **Assembler Syntax** and the **Operation** fields. The example given with each instruction discusses the contents of all the registers and memory locations referenced by the opcode-operand portion of that instruction but not those referenced by the parallel move portion of that instruction. Refer to A.5 PARALLEL MOVE DESCRIPTIONS for a complete discussion of parallel moves, including examples which discuss the contents of all the registers and memory locations referenced by the parallel move portion of an instruction.

Whenever an instruction uses an accumulator as both a destination operand for a data ALU operation and as a source for a parallel move operation, the parallel move operation will use the value in the accumulator prior to execution of any data ALU operation.

Whenever a bit in the condition code register is defined according to the **standard** definition given in 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 will be given in the **Condition Code** section of that instruction in **bold** text to alert the user to any special conditions concerning its use.

Table A-5 Condition Code Computations

Mnemonic	L	E	U	N	Z	V	C	Notes	Mnemonic	L	E	U	N	Z	V	C	Notes
ABS	*	*	*	*	*	*	—		MAC	*	*	*	*	*	*	—	
ADC	*	*	*	*	*	*	*		MACR	*	*	*	*	*	*	—	
ADD	*	*	*	*	*	*	*		MOVE	*	—	—	—	—	—	—	
ADDL	*	*	*	*	*	?	*	1	MOVEC	?	?	?	?	?	?	?	13
ADDR	*	*	*	*	*	*	*		MOVEM	?	?	?	?	?	?	?	13
AND	*	—	—	?	?	0	—	8, 9	MOVEP	?	?	?	?	?	?	?	13
ANDI	?	?	?	?	?	?	?	2	MPY	*	*	*	*	*	*	—	
ASL	*	*	*	*	*	?	?	1, 3	MPYR	*	*	*	*	*	*	—	
ASR	*	*	*	*	*	0	?	4	NEG	*	*	*	*	*	*	—	
BCHG	?	—	—	—	—	—	?	5, 14	NOP	—	—	—	—	—	—	—	
BCLR	?	—	—	—	—	—	?	5, 14	NORM	*	*	*	*	*	?	—	1
BSET	?	—	—	—	—	—	?	5, 14	NOT	*	—	—	?	?	0	—	8, 9
BTST	?	—	—	—	—	—	?	5, 14	OR	*	—	—	?	?	0	—	8, 9
CLR	*	*	*	*	*	0	—		ORI	?	?	?	?	?	?	?	6
CMP	*	*	*	*	*	*	*		REP	*	—	—	—	—	—	—	
CMPM	*	*	*	*	*	*	*		RESET	—	—	—	—	—	—	—	
DIV	*	—	—	—	—	?	?	1, 7	RND	*	*	*	*	*	*	—	
DO	*	—	—	—	—	—	—		ROL	*	—	—	?	?	0	?	8, 9, 10
ENDDO	—	—	—	—	—	—	—		ROR	*	—	—	?	?	0	?	8, 9, 11
EOR	*	—	—	?	?	0	—	8, 9	RTI	?	?	?	?	?	?	?	12
Jcc	—	—	—	—	—	—	—		RTS	*	—	—	—	—	—	—	
JCLR	—	—	—	—	—	—	—		SBC	*	*	*	*	*	*	*	
JMP	—	—	—	—	—	—	—		STOP	—	—	—	—	—	—	—	
JScc	—	—	—	—	—	—	—		SUB	*	*	*	*	*	*	*	
JSCLR	—	—	—	—	—	—	—		SUBL	*	*	*	*	*	?	*	1
JSET	—	—	—	—	—	—	—		SUBR	*	*	*	*	*	*	*	
JSR	—	—	—	—	—	—	—		SWI	—	—	—	—	—	—	—	
JSSET	—	—	—	—	—	—	—		Tcc	—	—	—	—	—	—	—	
LSL	*	—	—	?	?	0	?	8, 9, 10	TFR	*	—	—	—	—	—	—	
LSR	*	—	—	?	?	0	?	8, 9, 11	TST	*	*	*	*	*	0	—	
LUA	—	—	—	—	—	—	—		WAIT	—	—	—	—	—	—	—	

where: * Set according to the **standard** definition of the operation
— Not affected by the operation
? Set according to a **special** definition and can be a 0 or 1
0 The V bit is cleared

NOTES:

1 V Set if an arithmetic overflow occurs in the 56-bit result. Also set if the MS bit of the destination operand is changed as a result of the left shift. Cleared otherwise.
2 ? Cleared if the corresponding bit in the immediate data is cleared when the operand is the CCR. Not affected otherwise.
3 C Set if bit 55 of the source operand is set. Cleared otherwise.
4 C Set if bit 0 of the source operand is set. Cleared otherwise.
5 C Set if bit #n of the source operand is set. Cleared otherwise.
6 ? Set if the corresponding bit in the immediate data is set when the operand is the CCR. Not affected otherwise.
7 C Set if bit 55 of the result is cleared. Cleared otherwise.
8 N Set if bit 47 of the result is set. Cleared otherwise.
9 Z Set if bits 47 - 24 of the result are zero. Cleared otherwise.
10 C Set if bit 47 of the source operand is set. Cleared otherwise.
11 C Set if bit 24 of the source operand is set. Cleared otherwise.
12 ? Set according to the value pulled from the stack.
13 ? If the status register (SR) is specified as a destination operand, set according to the corresponding bit of the source operand. If SR is not specified as a destination operand, the L bit is set if data limiting occurred. All ? bits are not affected otherwise.
14 ? Set if limiting occurs, not affected otherwise.

The definition and thus the computation of both the E (extension) and U (unnormalized) bits of the condition code register (CCR) varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

NOTE

The signed integer portion of an accumulator is NOT necessarily the same as either the A2 or B2 extension register portion of that accumulator. The signed integer portion of an accumulator is defined according to the scaling mode being used and can consist of the MS 8, 9, or 10 bits of an accumulator. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Operation:
| D | → D (parallel move)

Assembler Syntax:
ABS D (parallel move)

Description: Take the absolute value of the destination operand D and store the result in the destination accumulator.

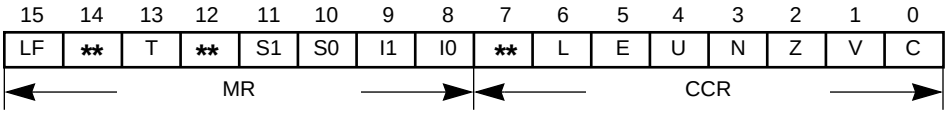
Example:

```
        :
ABS     A       #$123456,X0     A,Y0        ;take abs. value, set up X0, save value
        :
```

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $FF:FFFFFF:FFFFF2. Since this is a negative number, the execution of the ABS instruction takes the twos complement of that value and returns $00:000000:00000E.

Note: For the case in which the D operand equals $80:000000:000000 (-256.0), the ABS instruction will cause an overflow to occur since the result cannot be correctly expressed using the standard 56-bit, fixed-point, twos-complement data representation. Data limiting does not occur (i.e., A is not set to the limiting value of $7F:FFFFFF:FFFFFF).

Condition Codes:



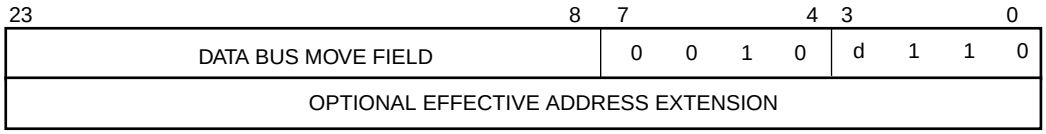
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

ABS D

Opcode:



Instruction Fields:

D d

A 0

B 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
S+C+D → D (parallel move)

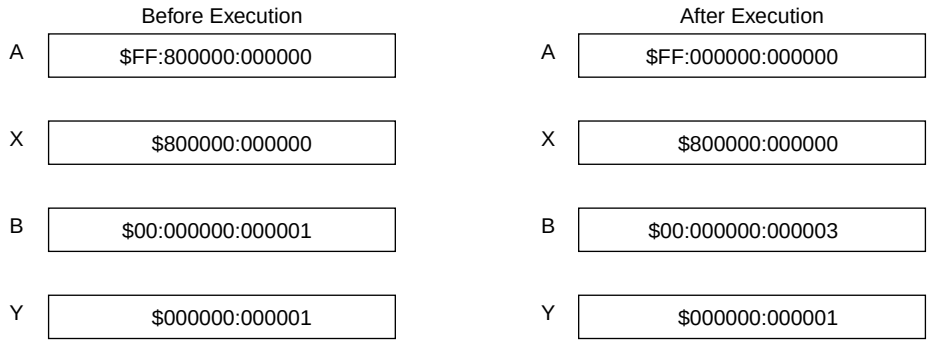
Assembler Syntax:
ADC S,D (parallel move)

Description: Add the source operand S and the carry bit C of the condition code register to the destination operand D and store the result in the destination accumulator. Long words (48 bits) may be added to the (56-bit) destination accumulator.

Note: The carry bit is set correctly for multiple precision arithmetic using long-word operands if the extension register of the destination accumulator (A2 or B2) is the sign extension of bit 47 of the destination accumulator (A or B).

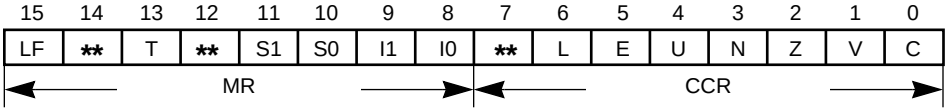
Example:

```
        :
MOVE L:<$0,X        ;get a 48-bit LS long-word operand in X
MOVE L:<$1,A        ;get other LS long word in A (sign ext.)
MOVE L:<$2,Y        ;get a 48-bit MS long-word operand in Y
ADD X,A L:<$3,B     ;add LS words; get other MS word in B
ADC Y,B A10,L:<$4   ;add MS words with carry, save LS sum
MOVE B10,L:<$5      ;save MS sum
        :
```



Explanation of Example: This example illustrates long-word double-precision (96-bit) addition using the ADC instruction. Prior to execution of the ADD and ADC instructions, the double-precision 96-bit value $000000:000001:800000:000000 is loaded into the Y and X registers (Y:X), respectively. The other double-precision 96-bit value $000000:000001:800000:000000 is loaded into the B and A accumulators (B:A), respectively. Since the 48-bit value loaded into the A accumulator is automatically sign extended to 56 bits and the other 48-bit long-word operand is internally sign extended to 56 bits during instruction execution, the carry bit will be set correctly after the execution of the ADD X,A instruction. The ADC Y,B instruction then produces the correct MS 56-bit result. The actual 96-bit result is stored in memory using the A10 and B10 operands (instead of A and B) because shifting and limiting is not desired.

Condition Codes:



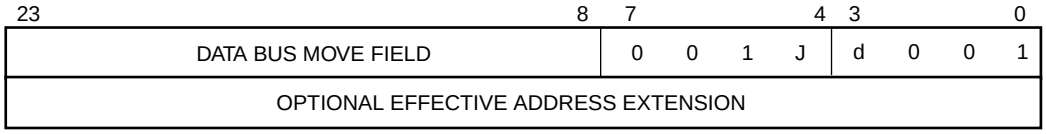
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

ADC S,D

Opcode:



Instruction Fields:

S,D	J d
X,A	0 0
X,B	0 1
Y,A	1 0
Y,B	1 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
S+D→D (parallel move

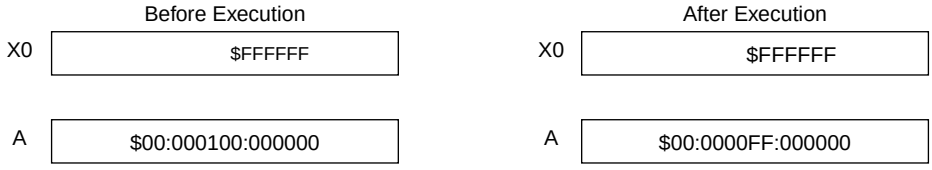
Assembler Syntax:
ADD S,D (parallel move)

Description: Add the source operand S to the destination operand D and store the result in the destination accumulator. Words (24 bits), long words (48 bits), and accumulators (56 bits) may be added to the destination accumulator.

Note: The carry bit is set correctly using word or long-word source operands if the extension register of the destination accumulator (A2 or B2) is the sign extension of bit 47 of the destination accumulator (A or B). Thus, the carry bit is always set correctly using accumulator source operands, but can be set incorrectly if A1, B1, A10, or B10 are used as source operands and A2 and B2 are not replicas of bit 47.

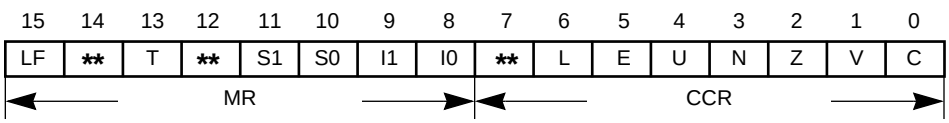
Example:

```
        :
ADD   X0,A A,X1     A,Y:(R1)+I      ;24-bit add, set up X1, save prev. result
        :
```



Explanation of Example: Prior to execution, the 24-bit X0 register contains the value $FFFFFF and the 56-bit A accumulator contains the value $00:000100:000000. The ADD instruction automatically appends the 24-bit value in the X0 register with 24 LS zeros, sign extends the resulting 48-bit long word to 56 bits, and adds the result to the 56-bit A accumulator. Thus, 24-bit operands are added to the MSP portion of A or B (A1 or B1) because all arithmetic instructions assume a fractional, twos complement data representation. Note that 24-bit operands can be added to the LSP portion of A or B (A0 or B0) by loading the 24-bit operand into X0 or Y0, forming a 48-bit word by loading X1 or Y1 with the sign extension of X0 or Y0 and executing an ADD X,A or ADD Y,A instruction.

Condition Codes:



L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

ADD S,D

Opcode:

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

Instruction Fields:

S,D	J J J d	S,D	J J J d	S,D	J J J d
B,A	0 0 1 0	X0,A	1 0 0 0	Y1,A	1 1 1 0
A,B	0 0 1 1	X0,B	1 0 0 1	Y1,B	1 1 1 1
X,A	0 1 0 0	Y0,A	1 0 1 0		
X,B	0 1 0 1	Y0,B	1 0 1 1		
Y,A	0 1 1 0	X1,A	1 1 0 0		
Y,B	0 1 1 1	X1,B	1 1 0 1		

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

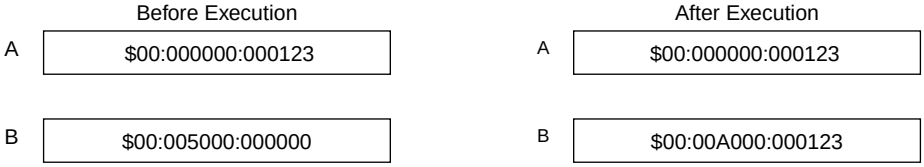
Operation:
S+2∗D➞D (parallel move)

Assembler Syntax:
ADDL S,D (parallel move)

Description: Add the source operand S to two times the destination operand D and store the result in the destination accumulator. The destination operand D is arithmetically shifted one bit to the left, and a zero is shifted into the LS bit of D prior to the addition operation. The carry bit is set correctly if the source operand does not overflow as a result of the left shift operation. The overflow bit may be set as a result of either the shifting or addition operation (or both). This instruction is useful for efficient divide and decimation in time (DIT) FFT algorithms.

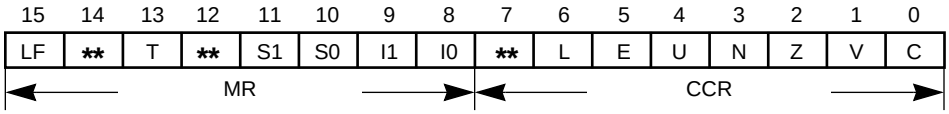
Example:

```
        :
ADDL A,B #$0,R0          ;A+2∗B➞B, set up addr. reg. R0
        :
```



Explanation of Example: Prior to execution, the 56-bit accumulator contains the value $00:000000:000123, and the 56-bit B accumulator contains the value $00:005000:000000. The ADDL A,B instruction adds two times the value in the B accumulator to the value in the A accumulator and stores the 56-bit result in the B accumulator.

Condition Codes:



L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — **Set if overflow has occurred in A or B result or if the MS bit of the destination operand is changed as a result of the instruction's left shift**
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

ADDL S,D

Opcode:

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

Instruction Fields:

S,D	**d**
B,A	0
A,B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
S+D / 2→D (parallel move)

Assembler Syntax:
ADDR S,D (parallel move)

Description: Add the source operand S to one-half the destination operand D and store the result in the destination accumulator. The destination operand D is arithmetically shifted one bit to the right while the MS bit of D is held constant prior to the addition operation. In contrast to the ADDL instruction, the carry bit is always set correctly, and the overflow bit can only be set by the addition operation and not by an overflow due to the initial shifting operation. This instruction is useful for efficient divide and decimation in time (DIT) FFT algorithms.

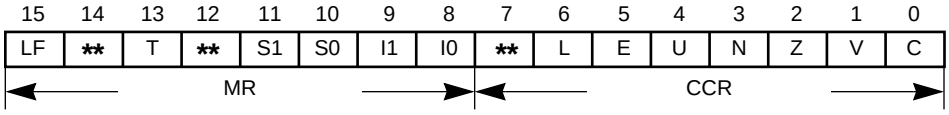
Example:

```
        :
ADDR B,A X0,X:(R1)+N1 Y0,Y:(R4)–        ;B+A / 2→A, save X0 and Y0
        :
```

	Before Execution	After Execution
A	$80:000000:2468AC	$C0:013570:123456
B	$00:013570:000000	$00:013570:000000

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $80:000000:2468AC, and the 56-bit B accumulator contains the value $00:013570:000000. The ADDR B,A instruction adds one-half the value in the A accumulator to the value in the B accumulator and stores the 56-bit result in the A accumulator.

Condition Codes:



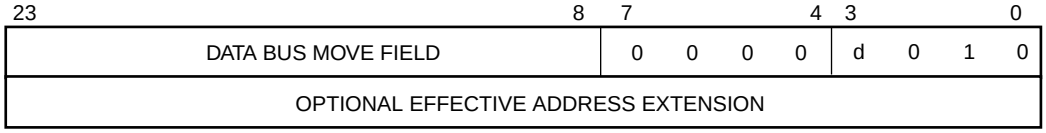
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

ADDR S,D

Opcode:



Instruction Fields:

S,D	d
B,A	0
A,B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

S • D[47:24]→D[47:24] (parallel move)
where •denotes the logical AND operator

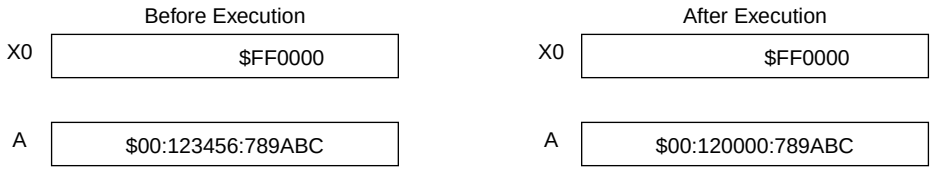
Assembler Syntax:

AND S,D (parallel move)

Description: Logically AND the source operand S with bits 47–24 of the destination operand D and store the result in bits 47–24 of the destination accumulator. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

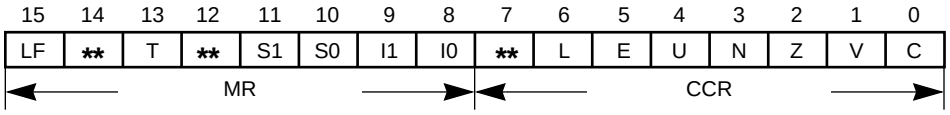
Example:

```
        :
AND X0,A (R5)–N5        ;AND X0 with A1, update R5 using N5
        :
```



Explanation of Example: Prior to execution, the 24-bit X0 register contains the value $FF0000, and the 56-bit A accumulator contains the value $00:123456:789ABC. The AND X0,A instruction logically ANDs the 24-bit value in the X0 register with bits 47–24 of the A accumulator (A1) and stores the result in the A accumulator with bits 55–48 and 23–0 unchanged.

Condition Codes:

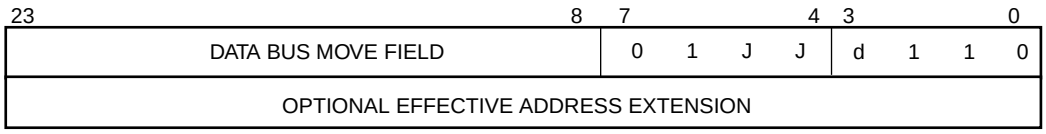


L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z— **Set if bits 47–24 of A or B result are zero**
V — Always cleared

Instruction Format:

AND S,D

Opcode:



Instruction Fields:

S	**J J**	**D d**
X,0	0 0	A 0 (only A1 is changed)
X,1	1 0	B 1 (only B1 is changed)
Y,0	0 1	
Y,1	1 1	

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

#xx • D→D

where • denotes the logical AND operator

Assembler Syntax:

AND(I) #xx,D

Description: Logically AND the 8-bit immediate operand (#xx) with the contents of the destination control register D and store the result in the destination control register. The condition codes are affected only when the condition code register (CCR) is specified as the destination operand.

Restrictions: The ANDI #xx,MR instruction cannot be used **immediately before** an ENDDO or RTI instruction and cannot be one of the **last three** instructions in a DO loop (at LA-2, LA-1, or LA).

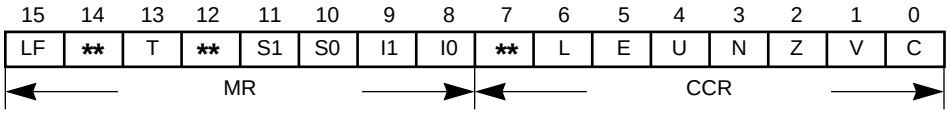
The ANDI #xx,CCR instruction cannot be used **immediately before** an RTI instruction.

Example:

```
        :
AND #$FE,CCR        ;clear carry bit C in cond. code register
        :
```

Explanation of Example: Prior to execution, the 8-bit condition code register (CCR) contains the value $31. The AND #$FE,CCR instruction logically ANDs the immediate 8-bit value $FE with the contents of the condition code register and stores the result in the condition code register.

Condition Codes:



For CCR Operand:

L — **Cleared if bit 6 of the immediate operand is cleared**

E — **Cleared if bit 5 of the immediate operand is cleared**

U — **Cleared if bit 4 of the immediate operand is cleared**

N — **Cleared if bit 3 of the immediate operand is cleared**

Z— **Cleared if bit 2 of the immediate operand is cleared**

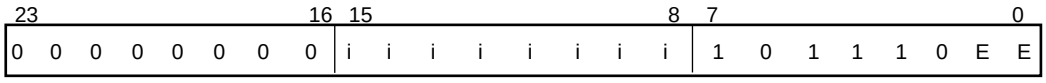
V — **Cleared if bit 1 of the immediate operand is cleared**
C — **Cleared if bit 0 of the immediate operand is cleared**

For MR and OMR Operands: The condition codes are not affected using these operands.

Instruction Format:

AND(I) #xx,D

Opcode:



Instruction Fields:

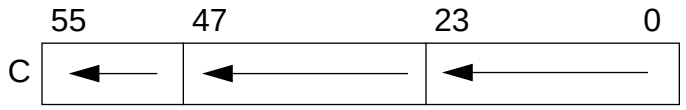
#xx=8-bit Immediate Short Data — i i i i i i i i

D	**E E**
MR	0 0
CCR	0 1
OMR	1 0

Timing: 2 oscillator clock cycles

Memory: 1 program word

Operation:

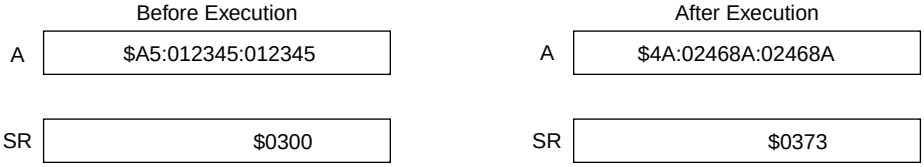


Assembler Syntax: ASL D (parallel move)

Description: Arithmetically shift the destination operand D one bit to the left and store the result in the destination accumulator. The MS bit of D prior to instruction execution is shifted into the carry bit C and a zero is shifted into the LS bit of the destination accumulator D. If a zero shift count is specified, the carry bit is cleared. The difference between ASL and LSL is that ASL operates on the entire 56 bits of the accumulator and therefore sets the V bit if the number overflowed.

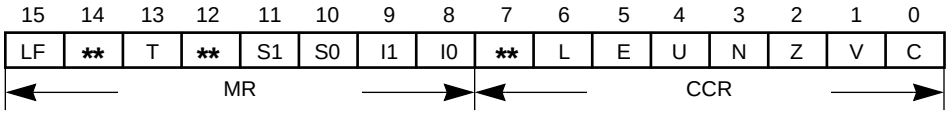
Example:

```
        :
ASL A   (R3)–           ;multiply A by 2, update R3
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $A5:012345:012345. The execution of the ASL A instruction shifts the 56-bit value in the A accumulator one bit to the left and stores the result back in the A accumulator.

Condition Codes:



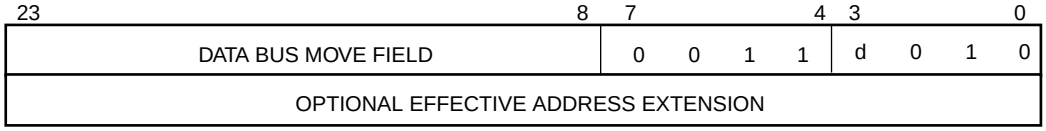
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — **Set if bit 55 of A or B result is changed due to left shift**
C — **Set if bit 55 of A or B was set prior to instruction execution**

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

ASL D

Opcode:



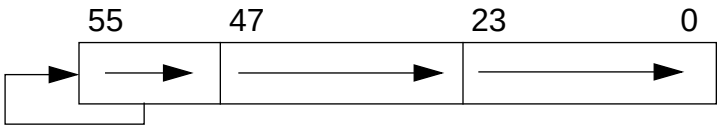
Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

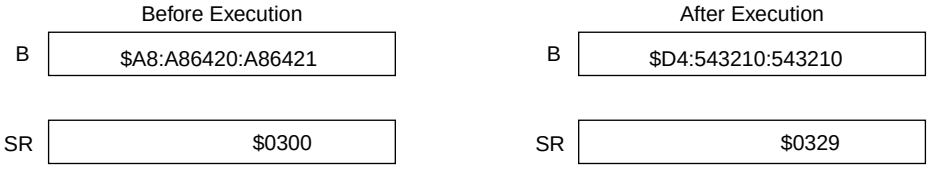


Assembler Syntax: ASR D (parallel move)

Description: Arithmetically shift the destination operand D one bit to the right and store the result in the destination accumulator. The LS bit of D prior to instruction execution is shifted into the carry bit C, and the MS bit of D is held constant.

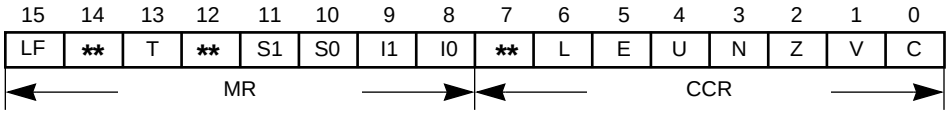
Example:

```
        :
ASR B   X:–(R3),R3      ;divide B by 2, update R3, load R3
        :
```



Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $A8:A86420:A86421. The execution of the ASR B instruction shifts the 56-bit value in the B accumulator one bit to the right and stores the result back in the B accumulator.

Condition Codes:



L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Always cleared
C — **Set if bit 0 of A or B was set prior to instruction execution**

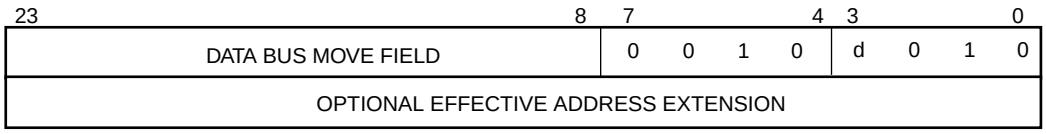
Note: The definition of the E and U bits varies according to the scaling mode being used.

Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

ASR D

Opcode:



Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

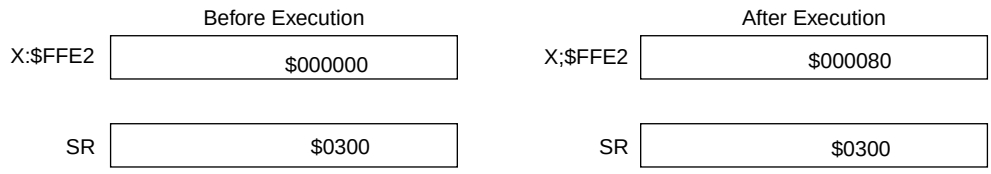
Memory: 1+mv program words

Operation:	**Assembler Syntax:**
D[n] → C; $\overline{D[n]}$ → D[n]	BCHG #n,X:ea
D[n] → C; $\overline{D[n]}$ → D[n]	BCHG #n,X:aa
D[n] → C; $\overline{D[n]}$ → D[n]	BCHG #n,X:pp
D[n] → C; $\overline{D[n]}$ → D[n]	BCHG #n,Y:ea
D[n] → C; $\overline{D[n]}$ → D[n]	BCHG #n,Y:aa
D[n] → C; $\overline{D[n]}$ → D[n]	BCHG #n,Y:pp
D[n] → C; $\overline{D[n]}$ → D[n]	BCHG #n,D

Description: Test the n^{th} bit of the destination operand D, complement it, and store the result in the destination location. The state of the n^{th} bit is stored in the carry bit C of the condition code register. After the test, the n^{th} bit of the destination location is complemented. The bit to be tested is selected by an immediate bit number from 0–23. This instruction performs a read-modify-write operation on the destination location using two destination accesses before releasing the bus. This instruction provides a test-and-change capability which is useful for synchronizing multiple processors using a shared memory. This instruction can use all memory alterable addressing modes.

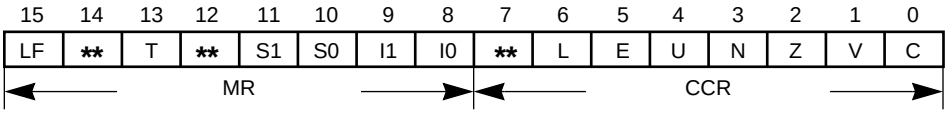
Example:

```
        :
BCHG    #$7,X:<<$FFE2    ;test and change bit 7 in I/O Port B DDR
        :
```



Explanation of Example: Prior to execution, the 24-bit X location X:$FFE2 (I/O port B data direction register) contains the value $000000. The execution of the BCHG #$7,X:<<$FFE2 instruction tests the state of the 7th bit in X:$FFE2, sets the carry bit C accordingly, and then complements the 7th bit in X:$FFE2.

Condition Codes:



CCR Condition Codes:

For destination operand SR:

C — Changed if bit 0 is specified. Not affected otherwise.
V — Changed if bit 1 is specified. Not affected otherwise.
Z — Changed if bit 2 is specified. Not affected otherwise.
N — Changed if bit 3 is specified. Not affected otherwise.
U — Changed if bit 4 is specified. Not affected otherwise.
E — Changed if bit 5 is specified. Not affected otherwise.
L — Changed if bit 6 is specified. Not affected otherwise.

For other destination operands:

C — Set if bit tested is set. Cleared otherwise.
V — Not affected
Z — Not affected
N — Not affected
U — Not affected
E —Not affected
L —Not affected

MR Status Bits:

For destination operand SR:

I0 — Changed if bit 8 is specified. Not affected otherwise.
I1 — Changed if bit 9 is specified. Not affected otherwise.
S0 — Changed if bit 10 is specified. Not affected otherwise.
S1 — Changed if bit 11 is specified. Not affected otherwise.
T — Changed if bit 13 is specified. Not affected otherwise.
LF — Changed if bit 15 is specified. Not affected otherwise.

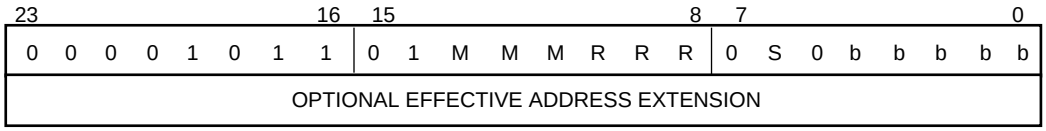
For other destination operands:

I0 — Not affected
I1 — Not affected
S0 — Not affected
S1 — Not affected
T — Not affected
LF — Not affected

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r
Absolute address	1	1	0	0	0	0

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where “rrr” refers to an address register R0-R7

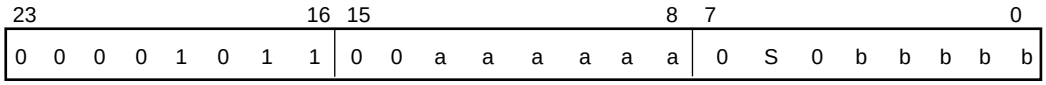
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

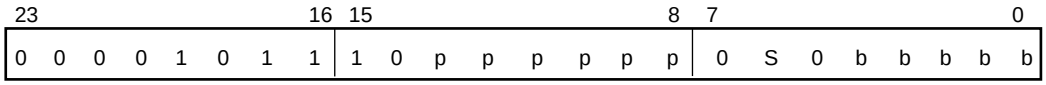
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit I/O Short Address=pppppp

I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

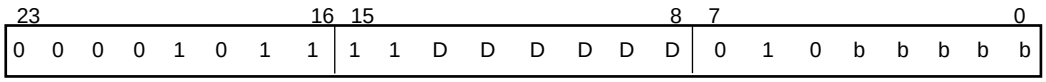
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

BCHG #n,D

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
D=destination register=DDDDDD
xxxx=16-bit Absolute Address in extension word

Destination Register	**D D D D D D**	**Bit Number bbbbb**
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 0 1 G G G	

See A.9 INSTRUCTION ENCODING and Table A-18 for specific register encodings.

Timing: 4+mvb oscillator clock cycles

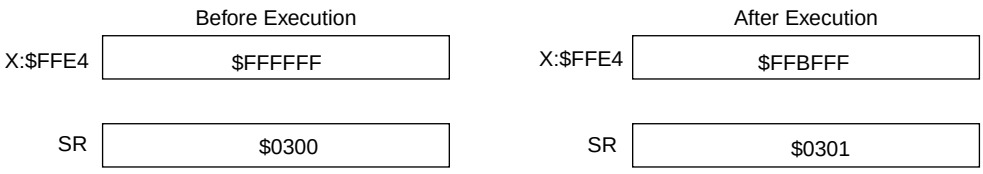
Memory: 1+ea program words

Operation:	**Assembler Syntax:**
D[n] → C; 0 → D[n]	BCLR #n,X:ea
D[n] → C; 0 → D[n]	BCLR #n,X:aa
D[n] → C; 0 → D[n]	BCLR #n,X:pp
D[n] → C; 0 → D[n]	BCLR #n,Y:ea
D[n] → C; 0 → D[n]	BCLR #n,Y:aa
D[n] → C; 0 → D[n]	BCLR #n,Y:pp
D[n] → C; 0 → D[n]	BCLR #n,D

Description: Test the n^{th} bit of the destination operand D, clear it and store the result in the destination location. The state of the n^{th} bit is stored in the carry bit C of the condition code register. After the test, the n^{th} bit of the destination location is cleared. The bit to be tested is selected by an immediate bit number from 0–23. This instruction performs a read-modify-write operation on the destination location using two destination accesses before releasing the bus. This instruction provides a test-and-clear capability which is useful for synchronizing multiple processors using a shared memory. This instruction can use all memory alterable addressing modes.

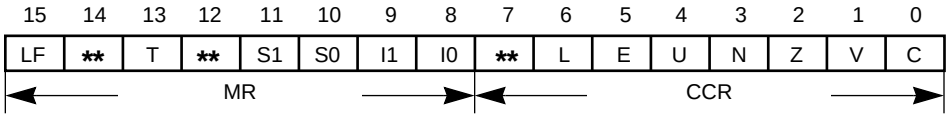
Example:

```
        :
BCLR    #$E,X:<<$FFE4    ;test and clear bit 14 in I/O Port B Data Reg.
        :
```



Explanation of Example: Prior to execution, the 24-bit X location X:$FFE4 (I/O port B data register) contains the value $FFFFFF. The execution of the BCLR #$E,X:<<$FFE4 instruction tests the state of the 14th bit in X:$FFE4, sets the carry bit C accordingly, and then clears the 14th bit in X:$FFE4.

Condition Codes:



CCR Condition Codes:

For destination operand SR:

C — Changed if bit 0 is specified. Not affected otherwise.
V — Changed if bit 1 is specified. Not affected otherwise.
Z — Changed if bit 2 is specified. Not affected otherwise.
N — Changed if bit 3 is specified. Not affected otherwise.
U — Changed if bit 4 is specified. Not affected otherwise.
E — Changed if bit 5 is specified. Not affected otherwise.
L — Changed if bit 6 is specified. Not affected otherwise.

For other destination operands:

C — Set if bit tested is set. Cleared otherwise.
V — Not affected
Z — Not affected
N — Not affected
U — Not affected
E —Not affected
L —Not affected

MR Status Bits:

For destination operand SR:

I0 — Changed if bit 8 is specified. Not affected otherwise.
I1 — Changed if bit 9 is specified. Not affected otherwise.
S0 — Changed if bit 10 is specified. Not affected otherwise.
S1 — Changed if bit 11 is specified. Not affected otherwise.
T — Changed if bit 13 is specified. Not affected otherwise.
LF — Changed if bit 15 is specified. Not affected otherwise.

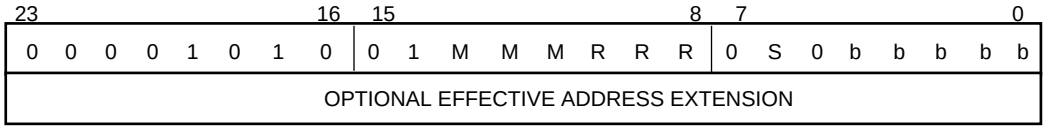
For other destination operands:

I0 — Not affected
I1 — Not affected
S0 — Not affected
S1 — Not affected
T — Not affected
LF — Not affected

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r
Absolute address	1	1	0	0	0	0

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where “rrr” refers to an address register R0-R7

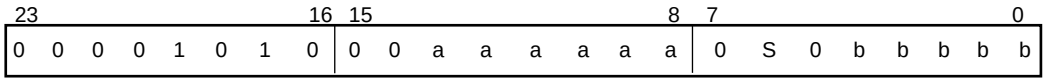
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory Space S		Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			•
111111			10111

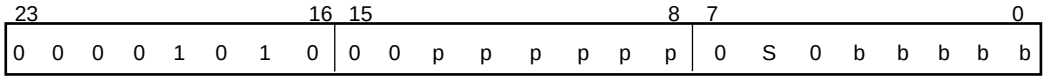
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit I/O Short Address=pppppp

I/O Short Address pppppp	Memory Space S		Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			•
111111			10111

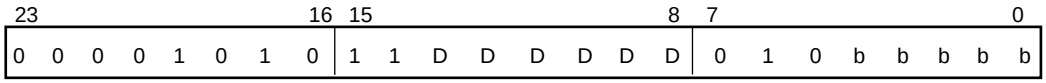
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

BCLR #n,D

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
D=destination register=DDDDDD
xxxx=16-bit Absolute Address in extension word

Destination Register	**D D D D D D**	**Bit Number bbbbb**
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 0 1 G G G	

See A.9 INSTRUCTION ENCODING and Table A-18 for specific register encodings.

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Operation:	**Assembler Syntax:**
D[n] → C; 1 → D[n]	BSET #n,X:ea
D[n] → C; 1 → D[n]	BSET #n,X:aa
D[n] → C; 1 → D[n]	BSET #n,X:pp
D[n] → C; 1 → D[n]	BSET #n,Y:ea
D[n] → C; 1 → D[n]	BSET #n,Y:aa
D[n] → C; 1 → D[n]	BSET #n,Y:pp
D[n] → C; 1 → D[n]	BSET #n,D

Description: Test the n^{th} bit of the destination operand D, set it, and store the result in the destination location. The state of the n^{th} bit is stored in the carry bit C of the condition code register. After the test, the n^{th} bit of the destination location is set. The bit to be tested is selected by an immediate bit number from 0–23. This instruction performs a read-modify-write operation on the destination location using two destination accesses before releasing the bus. This instruction provides a test-and-set capability which is useful for synchronizing multiple processors using a shared memory. This instruction can use all memory alterable addressing modes.

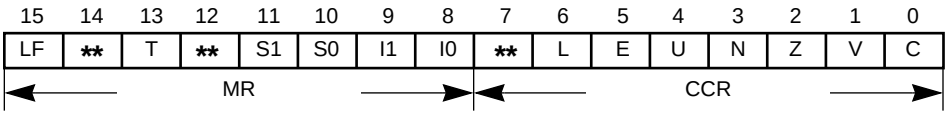
Example:

```
        :
BSET #$0,X:<<$FFE5;test and clear bit 14 in I/O Port B Data Reg.
        :
```

	Before Execution		After Execution
X:$FFE5	$000000	X:$FFE5	$000001
SR	$0300	SR	$0300

Explanation of Example: Prior to execution, the 24-bit X location X:$FFE5 (I/O port C data register) contains the value $000000. The execution of the BSET #$0,X:<<$FFE5 instruction tests the state of the 0^{th} bit in X:$FFE5, sets the carry bit C accordingly, and then sets the 0th bit in X:$FFE5.

Condition Codes:



CCR Condition Codes:

For destination operand SR:

C — Changed if bit 0 is specified. Not affected otherwise.
V — Changed if bit 1 is specified. Not affected otherwise.
Z — Changed if bit 2 is specified. Not affected otherwise.
N — Changed if bit 3 is specified. Not affected otherwise.
U — Changed if bit 4 is specified. Not affected otherwise.
E — Changed if bit 5 is specified. Not affected otherwise.
L — Changed if bit 6 is specified. Not affected otherwise.

For other destination operands:

C — **Set if bit tested is set. Cleared otherwise.**
V — Not affected
Z — Not affected
N — Not affected
U — Not affected
E —Not affected
L —Not affected

MR Status Bits:

For destination operand SR:

I0 — Changed if bit 8 is specified. Not affected otherwise.
I1 — Changed if bit 9 is specified. Not affected otherwise.
S0 — Changed if bit 10 is specified. Not affected otherwise.
S1 — Changed if bit 11 is specified. Not affected otherwise.
T — Changed if bit 13 is specified. Not affected otherwise.
LF — Changed if bit 15 is specified. Not affected otherwise.

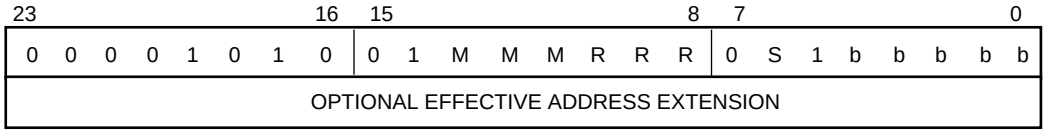
For other destination operands:

I0 — Not affected
I1 — Not affected
S0 — Not affected
S1 — Not affected
T — Not affected
LF — Not affected

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r
Absolute address	1	1	0	0	0	0

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where “rrr” refers to an address register R0-R7

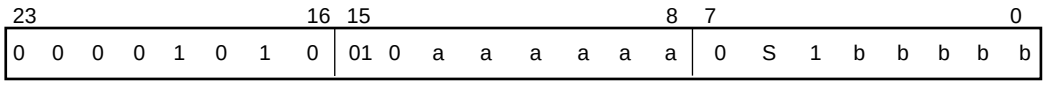
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory SpaceS		Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

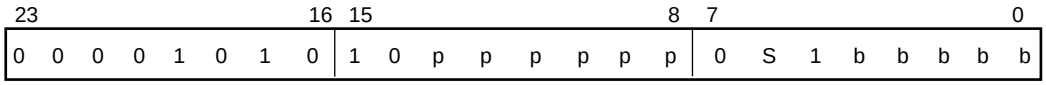
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit I/O Short Address=pppppp

I/O Short Address pppppp	Memory SpaceS		Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

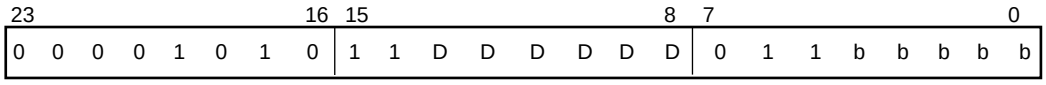
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

BSET #n,D

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
D=destination register=DDDDDD
xxxx=16-bit Absolute Address in extension word

Destination Register	**D D D D D D**	**Bit Number bbbbb**
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 0 1 G G G	

See A.9 INSTRUCTION ENCODING and Table A-18 for specific register encodings.

Timing: 4+mvb oscillator clock cycles

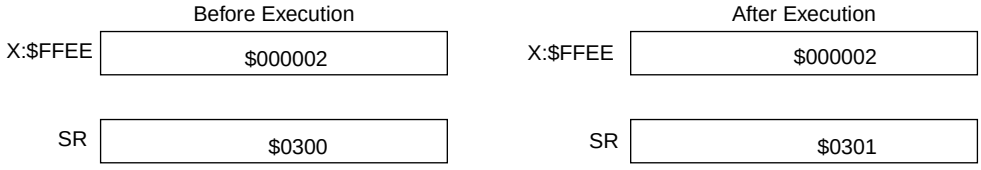
Memory: 1+ea program words

Operation:	**Assembler Syntax:**	
D[n] → C;	BTST	#n,X:ea
D[n] → C;	BTST	#n,X:aa
D[n] → C;	BTST	#n,X:pp
D[n] → C;	BTST	#n,Y:ea
D[n] → C;	BTST	#n,Y:aa
D[n] → C;	BTST	#n,Y:pp
D[n] → C;	BTST	#n,D

Description: Test the n^{th} bit of the destination operand D. The state of the n^{th} bit is stored in the carry bit C of the condition code register. The bit to be tested is selected by an immediate bit number from 0–23. This instruction is useful for performing serial to parallel conversion when used with the appropriate rotate instructions. This instruction can use all memory alterable addressing modes.

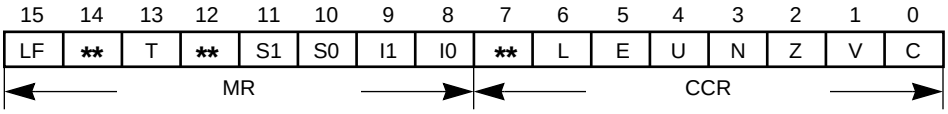
Example:

```
        :
BTST    #$0,X:<<$FFEE   ;read SSI serial input flag IF1 into C bit
ROL     A               ;rotate carry bit C into LSB of A1
        :
```



Explanation of Example: Prior to execution, the 24-bit X location X:$FFEE (I/O SSI status register) contains the value $000002. The execution of the BTST #$1,X:<<$FFEE instruction tests the state of the 1st bit (serial input flag IF1) in X:$FFEE and sets the carry bit C accordingly. This instruction sequence illustrates serial to parallel conversion using the carry bit C and the 24-bit A1 register.

Condition Codes:



CCR Condition Codes:

C — **Set if bit tested is set. Cleared otherwise.**
V — Not affected
Z — Not affected
N — Not affected
U — Not affected
E —Not affected
L —Not affected

MR Status bits are not affected.

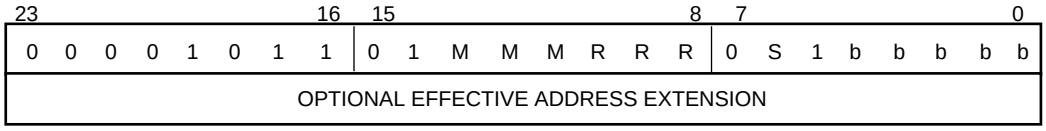
SP — Stack Pointer:

For destination operand SSH: SP — Decrement by 1.
For other destination operands:

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r
Absolute address	1	1	0	0	0	0

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where “rrr” refers to an address register R0-R7

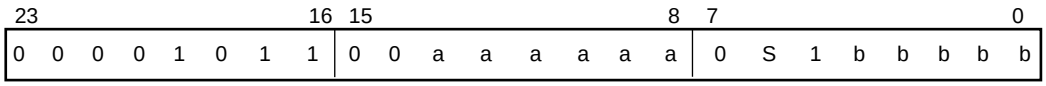
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa
000000
•
•
111111

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
10111

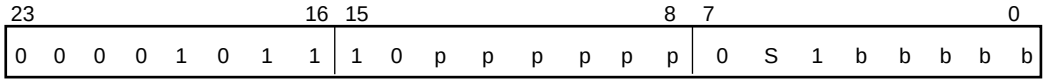
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

BTST #n,X:pp
BTST #n,Y:pp

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit I/O Short Address=pppppp

I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

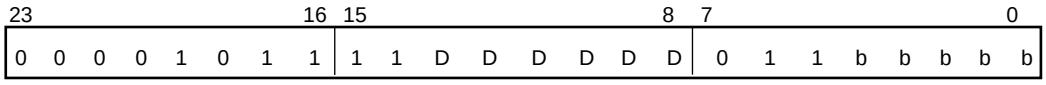
Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

BTST #n,D

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
D=destination register=DDDDDD,
xxxx=16-bit Absolute Address in extension word

Destination Register	**D D D D D D**	**Bit Number bbbbb**
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 0 1 G G G	

See A.9 INSTRUCTION ENCODING and Table A-18 for specific register encodings.

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

Operation:
0 → D (parallel move)

Assembler Syntax:
CLR D (parallel move)

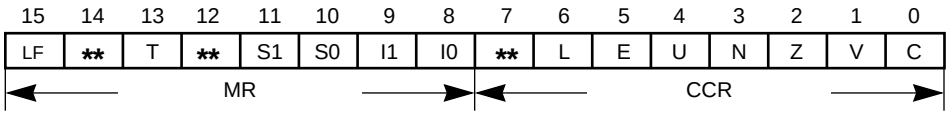
Description: Clear the destination accumulator. This is a 56-bit clear instruction.

Example:

```
        :
CLR  A     #$7F,N              ;clear A, set up N0 addr. reg.
        :
```

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $12:345678:9ABCDE. The execution of the CLR A instruction clears the 56-bit A accumulator to zero.

Condition Codes:

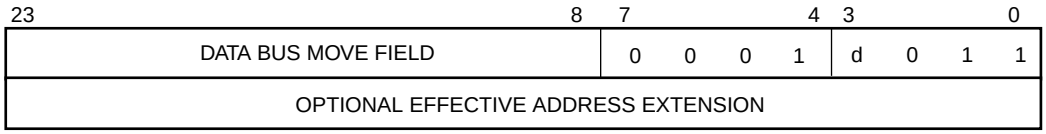


L — Set if data limiting has occurred during parallelmove
E — Always cleared
U — Always set
N — Always cleared
Z— Always set
V — Always cleared

Instruction Format:

CLR D

Opcode:



Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
S2 – S1(parallel move)

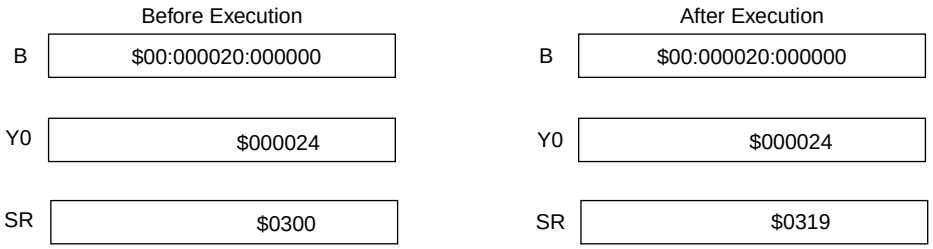
Assembler Syntax:
CMP S1, S2 (parallel move)

Description: Subtract the source one operand, S1, from the source two accumulator, S2, and update the condition code register. The result of the subtraction operation is not stored.

Note: This instruction subtracts 56-bit operands. When a word is specified as S1, it is sign extended and zero filled to form a valid 56-bit operand. For the carry to be set correctly as a result of the subtraction, S2 must be properly sign extended. S2 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 where it is extended to compare 24-bit operands such as X0 with A1.

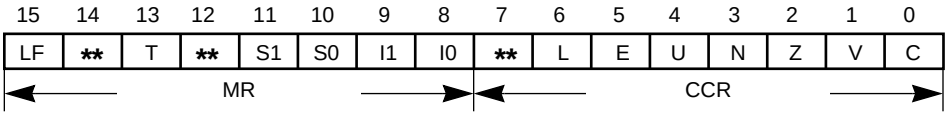
Example:

```
        :
CMP   Y0,B     X0,X:(R6)+N6     Y1,Y:(R0)–     ;comp. Y0 and B, save X0, Y1
        :
```



Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $00:000020:000000 and the 24-bit Y0 register contains the value $000024. The execution of the CMP Y0,B instruction automatically appends the 24-bit value in the Y0 register with 24 LS zeros, sign extends the resulting 48-bit long word to 56 bits, subtracts the result from the 56-bit B accumulator and updates the condition code register.

Condition Codes:



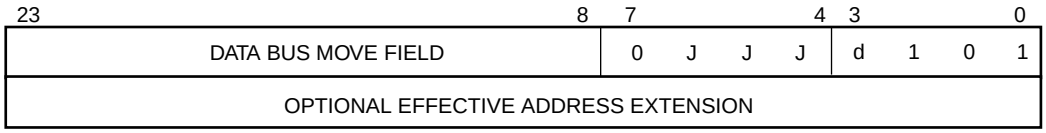
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

CMP S1, S2

Opcode:



Instruction Fields:

S1,S2	J J J d	S1,S2	J J J d
B,A	0 0 0 0	Y0,B	1 0 1 1
A,B	0 0 0 1	X1,A	1 1 0 0
X0,A	1 0 0 0	X1,B	1 1 0 1
X0,B	1 0 0 1	Y1,A	1 1 1 0
Y0,A	1 0 1 0	Y1,B	1 1 1 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
|S2| – |S1|(parallel move)

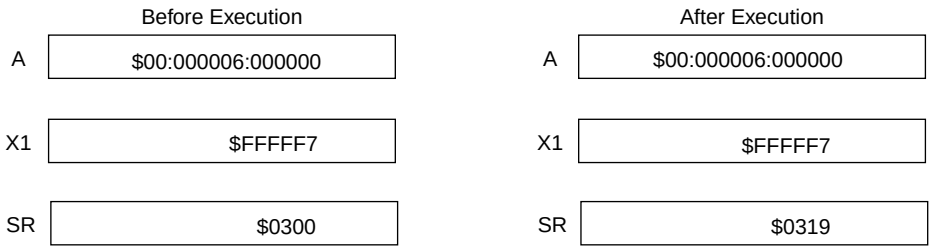
Assembler Syntax:
CMPM S1, S2 (parallel move)

Description: Subtract the absolute value (magnitude) of the source one operand, S1, from the absolute value of the source two accumulator, S2, and update the condition code register. The result of the subtraction operation is not stored.

Note: This instruction subtracts 56-bit operands. When a word is specified as S1, it is sign extended and zero filled to form a valid 56-bit operand. For the carry to be set correctly as a result of the subtraction, S2 must be properly sign extended. S2 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 where it is extended to compare 24-bit operands such as X0 with A1.

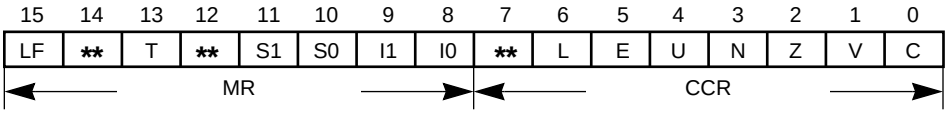
Example:

```
        :
CMPM    X1,A    BA,L:–(R4)          ;comp. Y0 and B, save X0, Y1
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:000006:000000, and the 24-bit X1 register contains the value $FFFFF7. The execution of the CMPM X1,A instruction automatically appends the 24-bit value in the X1 register with 24 LS zeros, sign extends the resulting 48-bit long word to 56 bits, takes the absolute value of the resulting 56-bit number, subtracts the result from the absolute value of the contents of the 56-bit A accumulator, and updates the condition code register.

Condition Codes:



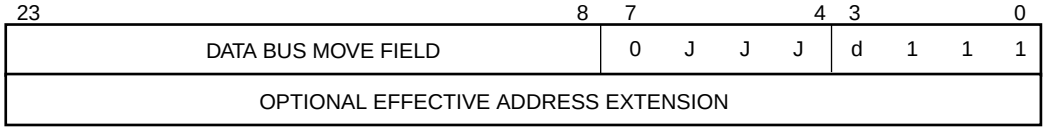
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

CMPM S1, S2

Opcode:



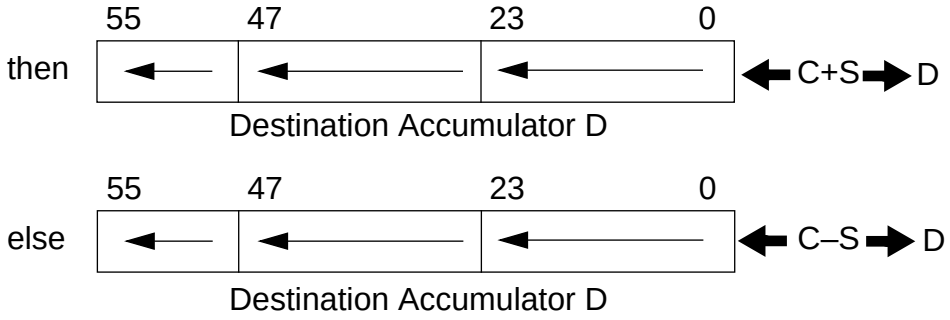
Instruction Fields:

S1,S2	J J J d	S1,S2	J J J d	S1,S2	J J J d
B,A	0 0 0 0	X0,B	1 0 0 1	X1,A	1 1 0 0
A,B	0 0 0 1	Y0,A	1 0 1 0	X1,B	1 1 0 1
X0,A	1 0 0 0	Y0,B	1 0 1 1	Y1,A	1 1 1 0
				Y1,B	1 1 1 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation: If D[55]⊕S[23]=1,



where ⊕ denotes the logical exclusive OR operator

Assembler Syntax: DIV S,D

Description:
Divide the destination operand D by the source operand S and store the result in the destination accumulator D. **The 48-bit dividend must be a positive fraction which has been sign extended to 56-bits and is stored in the full 56-bit destination accumulator D. The 24-bit divisor is a signed fraction and is stored in the source operand S.** Each DIV iteration calculates one quotient bit using a nonrestoring fractional division algorithm (see description on the next page). 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 D and is a signed fraction. The formed quotient occupies the low-order portion of the destination accumulator D (A0 or B0) and is a positive fraction. One bit of the formed quotient is shifted into the LS bit 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. **Valid results are obtained only when** |D| < |S| **and the operands are interpreted as fractions.** Note that this condition ensures that the magnitude of the quotient is less than one (i.e., is fractional) and precludes division by zero.

The DIV instruction calculates one quotient bit based on the divisor and the previous partial remainder. To produce an N-bit quotient, the DIV instruction is executed N times where N is the number of bits of precision desired in the quotient, 1;leN;le24. Thus, for a full-precision (24 bit) quotient, 24 DIV iterations are required. In general, executing the DIV instruction N times produces an N-bit quotient and a 48-bit remainder which has (48–N) bits of precision and whose N MS bits are zeros. The partial remainder is not a true remainder and must be corrected due to the nonrestoring nature of the division algo-

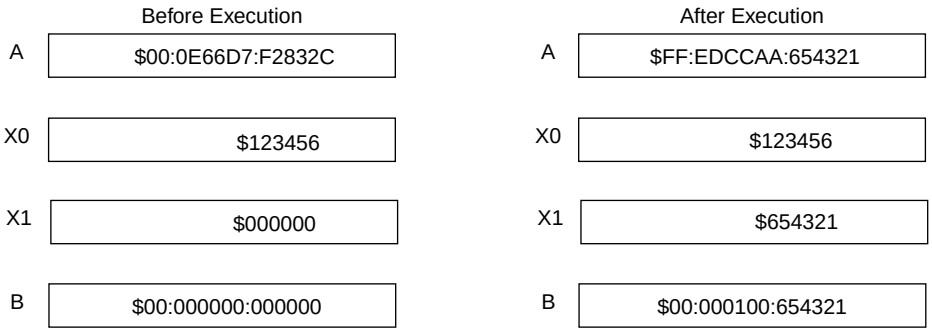
rithm before it may be used. Therefore, once the divide is complete, it is necessary to reverse the last DIV operation and restore the remainder to obtain the true remainder.

The DIV instruction uses a nonrestoring fractional division algorithm which consists of the following operations (see the previous **Operation** diagram):

1. **Compare the source and destination operand sign bits:** An exclusive OR operation is performed on bit 55 of the destination operand D and bit 23 of the source operand S;
2. **Shift the partial remainder and the quotient:** The 55-bit destination accumulator D is shifted one bit to the left. The carry bit C is moved into the LS bit (bit 0) of the accumulator;
3. **Calculate the next quotient bit and the new partial remainder:** The 24-bit source operand S (signed divisor) is either added to or subtracted from the MSP portion of the destination accumulator (A1 or B1), and the result is stored back into the MSP portion of that destination accumulator. If the result of the exclusive OR operation previously described was a ‘‘1’’ (i.e., the sign bits were different), the source operand S is added to the accumulator. If the result of the exclusive OR operation was a ‘‘0’’ (i.e., the sign bits were the same), the source operand S is subtracted from the accumulator. Due to the automatic sign extension of the 24-bit signed divisor, the addition or subtraction operation correctly sets the carry bit C of the condition code register with the next quotient bit.

Example: (4-Quadrant division, 24-bit signed quotient, 48-bit signed remainder)

```
            ABS A A,B               ;make dividend positive, copy A1 to B1
            EOR X0,B B,X:$0         ;save rem. sign in X:$0, quo. sign in N
            AND #$FE,CCR            ;clear carry bit C (quotient sign bit)
            REP #$18                ;form a 24-bit quotient
            DIV X0,A                ;form quotient in A0, remainder in A1
            TFR A,B                 ;save quotient and remainder in B1,B0
            JPL SAVEQUO             ;go to SAVEQUO if quotient is positive
            NEG B                   ;complement quotient if N bit set
SAVEQUO TFR X0,B B0,X1              ;save quo. in X1, get signed divisor
            ABS B                   ;get absolute value of signed divisor
            ADD A,B                 ;restore remainder in B1
            JCLR #23,X:$0,DONE      ;go to DONE if remainder is positive
            MOVE #$0,B0             ;clear LS 24 bits of B
            NEG B                   ;complement remainder if negative
DONE        . . . . . .
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the 56-bit, sign-extended fractional dividend D (D=$00.0E66D7:F2832C=0.112513535894635 approx.) and the 24-bit X0 register contains the 24-bit, signed fractional divisor S (S=$123456=0.142222166061401). Since |D|<|S|, the execution of the previous divide routine stores the correct 24-bit signed quotient in the 24-bit X1 register (A/X0=0.79111111164093=$654321=X1). The partial remainder is restored by reversing the last DIV operation and adding back the absolute value of the signed divisor in X0 to the partial remainder in A1. This produces the correct LS 24 bits of the 48-bit signed remained in the 24-bit B1 register. Note that the remainder is really a 48-bit value which has 24 bits of precision. Thus, the correct 48-bit remainder is $000000:000100 which equals 0.0000000000018190 approximately.

Note that the divide routine used in the previous example assumes that the sign-extended 56-bit signed fractional dividend is stored in the A accumulator and that the 24-bit signed fractional divisor is stored in the X0 register. This routine produces a **full 24-bit signed quotient and a 48-bit signed remainder.**

This routine may be greatly simplified for the case in which only positive, fractional operands are used to produce a 24-bit positive quotient and a 48-bit positive remainder, as shown in the following example:

```
1-Quadrant division, 24-bit unsigned quotient, 48-bit unsigned remainder
AND #$FE,CCR      ;clear carry bit C (quotient sign bit)
REP #$18          ;form a 24-bit quotient and remainder
DIV X0,A          ;form quotient in A0, remainder in A1
ADD X0,A          ;restore remainder in A1
```

Note that this routine assumes that the 56-bit positive, fractional, sign-extended dividend is stored in the A accumulator and that the 24-bit positive, fractional divisor is stored in

the X0 register. After execution, the 24-bit positive fractional quotient is stored in the A0 register; the LS 24 bits of the 48-bit positive fractional remainder are stored in the A1 register.

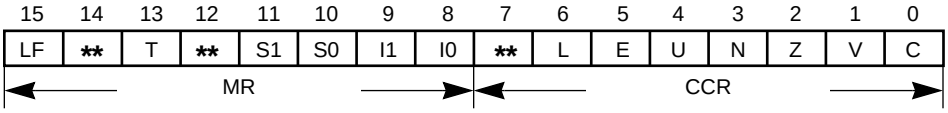
There are many variations possible when choosing a suitable division routine for a given application. The selection of a suitable division routine normally involves specification of the following items:

1. the number of bits of precision in the dividend;
2. the number of bits of precision N in the quotient;
3. whether the value of N is fixed or is variable;
4. whether the operands are unsigned or signed;
5. whether or not the remainder is to be calculated.

A complete discussion of the various division routines is beyond the scope of this manual. For a more complete discussion of these routines, refer to the application note entitled Fractional and Interger Arithmetic Using the DSP56001.

For extended precision division (i.e., for N-bit quotients where N>24), the DIV instruction is no longer applicable, and a user-defined N-bit division routine is required. For further information on division algorithms, refer to pages 524–530 of Theory and Application of Digital Signal Processing by Rabiner and Gold (Prentice-Hall, 1975), pages 190–199 of Computer Architecture and Organization by John Hayes (McGraw-Hill, 1978), pages 213–223 of Computer Arithmetic: Principles, Architecture, and Design by Kai Hwang (John Wiley and Sons, 1979), or other references as required.

Condition Codes:

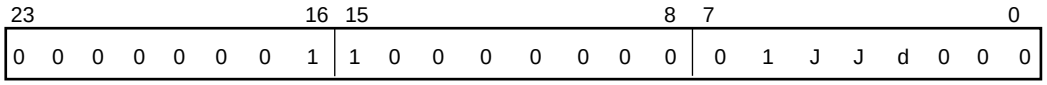


L — Set if overflow bit V is set
V — Set if the MS bit of the destination operand is changed as a result of the instruction's left shift operation
C — Set if bit 55 of the result is cleared.

Instruction Format:

DIV S,D

Opcode:



Instruction Fields:

S,D	J J d	S,D	J J d
X0,A	0 0 0	X1,A	1 0 0
X0,B	0 0 1	X1,B	1 0 1
Y0,A	0 1 0	Y1,A	1 1 0
Y0,B	0 1 1	Y1,B	1 1 1

Timing: 2 oscillator clock cycles

Memory: 1 program word

Operation:	Assembler Syntax:	
SP+1 → SP;LA → SSH;LC → SSL;X:ea → LC SP+1 → SP;PC → SSH;SR → SSL;expr –1 → LA 1 → LF	DO	X:ea,expr
SP+1 → SP;LA → SSH;LC → SSL;X:aa → LC SP+1 → SP;PC → SSH;SR → SSL;expr –1 → LA 1 → LF	DO	X:aa,expr
SP+1 → SP;LA → SSH;LC → SSL;Y:ea → LC SP+1 → SP;PC → SSH;SR → SSL;expr –1 → LA 1 → LF	DO	Y:ea,expr
SP+1 → SP;LA → SSH;LC → SSL;Y:aa → LC SP+1 → SP;PC → SSH;SR → SSL;expr –1 → LA 1 → LF	DO	Y:aa,expr
SP+1 → SP;LA → SSH;LC → SSL;#xxx → LC SP+1 → SP;PC → SSH;SR → SSL;expr –1 → LA 1 → LF	DO	#xxx,expr
SP+1 → SP;LA → SSH;LC → SSL;S → LC SP+1 → SP;PC → SSH;SR → SSL;expr –1 → LA 1 → LF	DO	S,expr

End of Loop:
SSL(LF) → SR;SP–1 → SP
SSH → LA;SSL → LC;SP – 1 → SP

Description: Begin a hardware DO loop that is to be repeated the number of times specified in the instruction's source operand and whose range of execution is terminated by the destination operand (previously shown as ''expr''). No overhead other than the execution of this DO instruction is required to set up this loop. DO loops can be nested and the loop count can be passed as a parameter.

During the first instruction cycle, the current contents of the loop address (LA) and the loop counter (LC) registers are pushed onto the system stack. The DO instruction's source operand is then loaded into the loop counter (LC) register. The LC register contains the remaining number of times the DO loop will be executed and can be accessed from inside the DO loop subject to certain restrictions. If LC equals zero, the DO loop is executed 65,536 times. All address register indirect addressing modes may be used to generate the effective address of the source operand. If immediate short data is speci-

fied, the 12 LS bits of LC are loaded with the 12-bit immediate value, and the four MS bits of LC are cleared.

During the second instruction cycle, the current contents of the program counter (PC) register and the status register (SR) are pushed onto the system stack. The stacking of the LA, LC, PC, and SR registers is the mechanism which permits the nesting of DO loops. The DO instruction's destination operand (shown as ''expr'') is then loaded into the loop address (LA) register. This 16-bit operand is located in the instruction's 24-bit absolute address extension word as shown in the opcode section. The value in the program counter (PC) register pushed onto the system stack is the address of the first instruction following the DO instruction (i.e., the first actual instruction in the DO loop). This value is read (i.e., copied but not pulled) from the top of the system stack to return to the top of the loop for another pass through the loop.

During the third instruction cycle, the loop flag (LF) is set. This results in the PC being repeatedly compared with LA to determine if the last instruction in the loop has been fetched. If LA equals PC, the last instruction in the loop has been fetched and the loop counter (LC) is tested. If LC is not equal to one, it is decremented by one and SSH is loaded into the PC to fetch the first instruction in the loop again. If LC equals one, the ''end-of-loop'' processing begins.

When executing a DO loop, the instructions are actually fetched each time through the loop. Therefore, a DO loop can be interrupted. DO loops can also be nested. When DO loops are nested, the end-of-loop addresses must also be nested and are not allowed to be equal. The assembler generates an error message when DO loops are improperly nested. Nested DO loops are illustrated in the example.

Note: The assembler calculates the end-of-loop address to be loaded into LA (the absolute address extension word) by evaluating the end-of-loop expression ''expr'' and subtracting one. This is done to accommodate the case where the last word in the DO loop is a two-word instruction. Thus, the end-of-loop expression ''expr'' in the source code must represent the address of the instruction AFTER the last instruction in the loop as shown in the example.

During the ''end-of-loop'' processing, the loop flag (LF) from the lower portion (SSL) of SP is written into the status register (SR), the contents of the loop address (LA) register are restored from the upper portion (SSH) of (SP–1), the contents of the loop counter (LC) are restored from the lower portion (SSL) of (SP–1) and the stack pointer (SP) is decremented by two. Instruction fetches now continue at the address of the instruction following the last instruction in the DO loop. Note that LF is the only bit in the status register (SR) that is restored after a hardware DO loop has been exited.

Note: The loop flag (LF) is cleared by a hardware reset.

Restrictions: The ‘‘end-of-loop’’ comparison previously described actually occurs at instruction fetch time. That is, LA is being compared with PC when the instruction at LA–2 is being executed. Therefore, instructions which access the program controller registers and/or change program flow cannot be used in locations LA–2, LA–1, or LA.

Proper DO loop operation is not guaranteed if an instruction **starting** at address **LA–2**, **LA–1**, or **LA** specifies one of the **program controller registers** SR, SP, SSL, LA, LC, or (implicitly) PC as a **destination** register. Similarly, the SSH program controller register may not be specified as a **source or destination** register in an instruction starting at address LA–2, LA–1, or LA. Additionally, the SSH register cannot be specified as a **source** register in the DO instruction itself and LA cannot be used as a **target** for **jumps to subroutine** (i.e., JSR, JScc, JSSET, or JSCLR to LA). A DO instruction cannot be repeated using the REP instruction.

The following instructions cannot **begin** at the indicated position(s) near the end of a DO loop:

At LA–2, LA–1, and LA

DO
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH
MOVEC to LA, LC, SR, SP, SSH, or SSL
MOVEM to LA, LC, SR, SP, SSH, or SSL
MOVEP to LA, LC, SR, SP, SSH, or SSL
ANDI MR
ORI MR
Two-word instructions which read LC, SP, or SSL

At LA–1

Single-word instructions (except REP) which read LC, SP, or SSL, JCLR, JSET, two-word JMP, two-word Jcc

At LA

any two-word instruction*

Jcc	REP
JCLR	RESET
JSET	RTI
JMP	RTS
JScc	STOP
JSR	WAIT

*This restriction applies to the situation in which the DSP56000/DSP56001 simulator’s single-line assembler is used to change the **last** instruction in a DO loop from a one-word instruction to a two-word instruction.

Other Restrictions:

DO SSH,xxxx
JSR to (LA) whenever the loop flag (LF) is set
JScc to (LA) whenever the loop flag (LF) is set
JSCLR to (LA) whenever the loop flag (LF) is set
JSSET to (LA) whenever the loop flag (LF) is set

A DO instruction cannot be repeated using the REP instruction.

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

Similarly, since the DO instruction accesses the program controller registers, the DO instruction must not be immediately preceded by any of the following instructions:

Immediately before DO

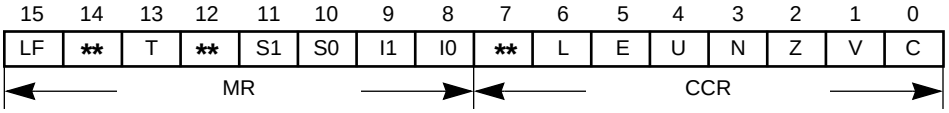
MOVEC to LA, LC, SSH, SSL, or SP
MOVEM to LA, LC, SSH, SSL, or SP
MOVEP to LA, LC, SSH, SSL, or SP
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH

Example:

```
            :
        DO #cnt1, END1          ;begin outer DO loop
            :
        DO #cnt2, END2          ;begin inner DO loop
            :
            :
        MOVE A,X:(R0);p         ;last instruction in inner loop
            :                   ;(in outer loop)
END2                            ;last instruction in outer loop
        ADD  A,B X:(R1)+,X0     first instruction after outer loop
END1        :
            :
```

Explanation of Example: This example illustrates a nested DO loop. The outer DO loop will be executed ‘‘cnt1’’ times while the inner DO loop will be executed (‘‘cnt1’’ * ‘‘cnt2’’) times. Note that the labels END1 and END2 are located at the first instruction past the end of the DO loop, as mentioned above, and are nested properly.

Condition Codes:



LF — Set when a DO loop is in progress

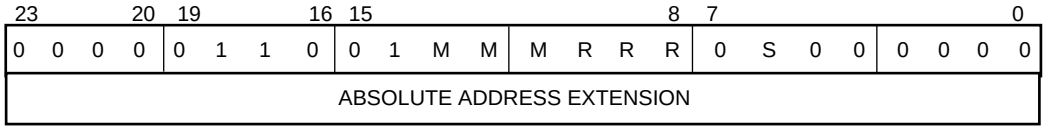
L— Set if data limiting occurred [see Note 2]

Instruction Format:

DO X:ea, expr

DO Y:ea, expr

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR,

expr=16-bit Absolute Address in 24-bit extension word

Effective Addressing Mode	**M M M R R R**	**Memory Space**	**S**
(Rn)-Nn	0 0 0 r r r	X Memory	0
(Rn)+Nn	0 0 1 r r r	Y Memory	1
(Rn)-	0 1 0 r r r		
(Rn)+	0 1 1 r r r		
(Rn)	1 0 0 r r r		
(Rn+Nn)	1 0 1 r r r		
-(Rn)	1 1 1 r r r		

where “rrr” refers to an address register R0-R7

Timing: 6+mv oscillator clock cycles

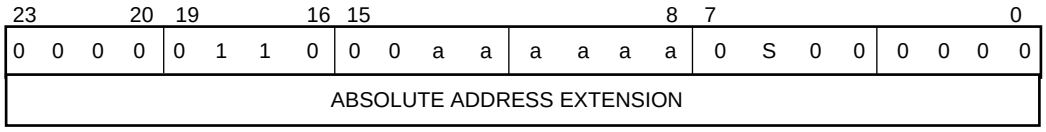
Memory: 2 program words

Instruction Format:

DO X:aa, expr

DO Y:aa, expr

Opcode:



Instruction Fields:

ea=6-bit Effective Short Address=aaaaaa,

expr=16-bit Absolute Address in 24-bit extension word

Absolute Short Address aaaaaa

000000

•

•

111111

Memory SpaceS

X Memory 0

Y Memory 1

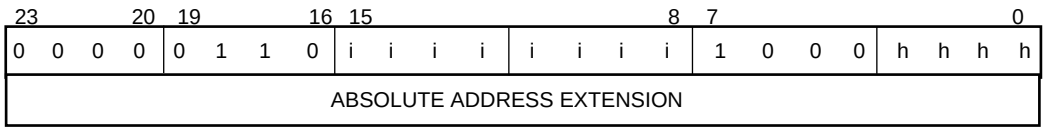
Timing: 6+mv oscillator clock cycles

Memory: 2 program words

Instruction Format:

DO #xxx, expr

Opcode:



Instruction Fields:

#xxx=12-bit Immediate Short Data = hhhhiiiiiiii,

expr=16-bit Absolute Address in 24-bit extension word

Immediate Short Data hhhh i i i i i i i i

000000000000

•

•

111111111111

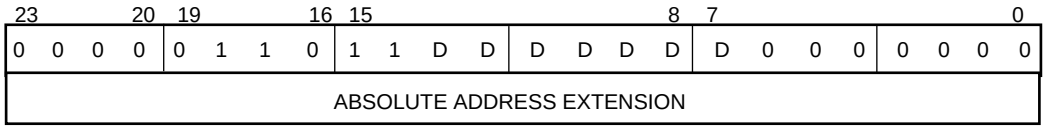
Timing: 6+mv oscillator clock cycles

Memory: 2 program words

Instruction Format:

DO S, expr

Opcode:



Instruction Fields:

S=6-bit Source operand = DDDDDD,
expr=16-bit Absolute Address in 24-bit extension word

Source	D	D	D	D	D	D	S/L	Source	D	D	D	D	D	D
X0	0	0	0	1	0	0	no	SR	1	1	1	0	0	1
X1	0	0	0	1	0	1	no	OMR	1	1	1	0	1	0
Y0	0	0	0	1	1	0	no	SP	1	1	1	0	1	1 [see Note 1]
Y1	0	0	0	1	1	1	no	SSL	1	1	1	1	0	1 [see Note 1]
A0	0	0	1	0	0	0	no	LA	1	1	1	1	1	0
B0	0	0	1	0	0	1	no	LC	1	1	1	1	1	1
A2	o	o	1	o	1	o	no	R0-R7	0	1	0	r	r	r
B2	0	0	1	1	0	0	no	N0-N7	0	1	1	n	n	n
A1	0	0	1	1	0	1	no	M0-M7	1	0	0	m	m	m
A	0	0	1	1	1	0	yes [see Note 2]							
B	0	0	1	1	1	1	yes [see Note 2]							

where rrr=Rn register
where nnn=Nn register
where mmm=Mn register

Note 1:

For DO SP, expr — The actual value that will be loaded into the loop counter (LC) is the value of the stack pointer (SP) before the execution of the DO instruction, incremented by 1.

Thus, if SP=3, the execution of the DO SP,expr instruction will load the loop counter (LC) with the value LC=4.

For	DO SSL, expr	The loop counter (LC) will be loaded with its previous value which was saved on the stack by the DO instruction itself.

Note 2:
If A or B is specified as a source operand, the accumulator value is optionally shifted according to the scaling mode bits in the status register. If the data out of the shifter indicates that the accumulator extension is in use, the 24-bit data is limited to a maximum positive or negative saturation constant. The shifted and limited value is loaded into LC, although A or B remain unchanged.

Timing: 6+mv oscillator clock cycles

Memory: 2 program words

Operation:
SSL(LF) → SR;SP – 1→ SP
SSH → LA; SSL → LC;SP –1 → SP

Assembler Syntax:
ENDDO

Description: Terminate the current hardware DO loop before the current loop counter (LC) equals one. If the value of the current DO loop counter (LC) is needed, it must be read before the execution of the ENDDO instruction. Initially, the loop flag (LF) is restored from the system stack and the remaining portion of the status register (SR) and the program counter (PC) are purged from the system stack. The loop address (LA) and the loop counter (LC) registers are then restored from the system stack.

Restrictions: Due to pipelining and the fact that the ENDDO instruction accesses the program controller registers, the ENDDO instruction must not be immediately preceded by any of the following instructions:

Immediately before ENDDO MOVEC to LA, LC, SR, SSH, SSL, or SP
MOVEM to LA, LC, SR, SSH, SSL, or SP
MOVEP to LA, LC, SR, SSH, SSL, or SP
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH
ORI MR
ANDI MR

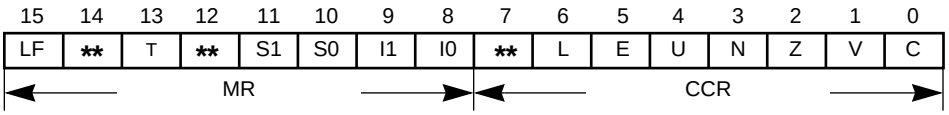
Also, the ENDDO instruction cannot be the last (LA) instruction in a DO loop.

Example:

```
            :
        DO Y0,NEXT          ;exec. loop ending at NEXT (Y0) times
            :
        MOVEC LC,A          ;get current value of loop counter (LC)
        CMP Y1,A            ;compare loop counter with value in Y1
        JNE ONWARD          ;go to ONWARD if LC not equal to Y1
        ENDDO               ;LC equal to Y1, restore all DO registers
        JMP NEXT            ;go to NEXT
ONWARD      :               ;LC not equal to Y1, continue DO loop
            :               ;(last instruction in DO loop)
NEXT MOVE #$123456,X1       ;(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 loop counter (LC) is compared with the value in the Y1 register to determine if execution of the DO loop should continue. Note that 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 instruction (i.e., JMP NEXT as previously shown) must be included after the ENDDO instruction to transfer program control to the first instruction past the end of the DO loop.

Condition Codes:

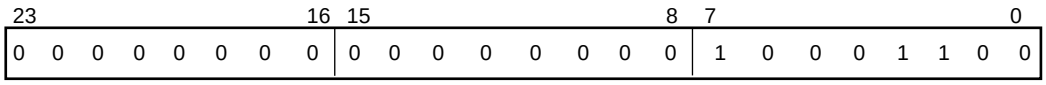


The condition codes are not affected by this instruction.

Instruction Format:

ENDDO

Opcode:



Instruction Fields:

None

Timing: 2 oscillator clock cycles

Memory: 1 program word

Operation:

S ⊕ D[47:24] →D[47:24] (parallel move)

where ⊕ denotes the logical Exclusive OR operator

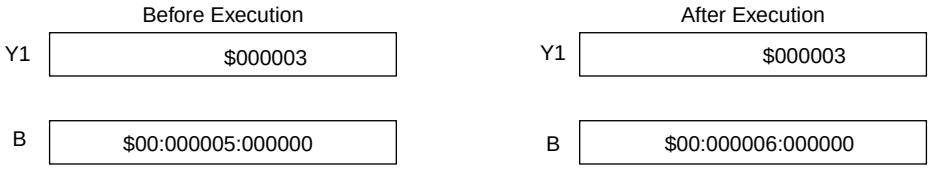
Assembler Syntax:

EOR S,D (parallel move)

Description: Logically exclusive OR the source operand S with bits 47–24 of the destination operand D and store the result in bits 47–24 of the destination accumulator. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

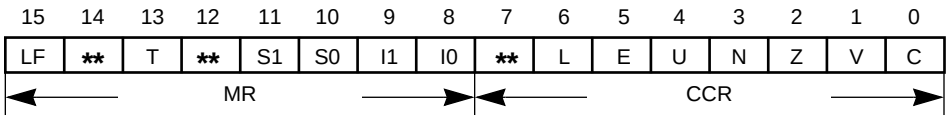
Example:

```
        :
EOR Y1,B    (R2)+       ;Exclusive OR Y1 with B1, update R2
        :
```



Explanation of Example: Prior to execution, the 24-bit Y1 register contains the value $000003, and the 56-bit B accumulator contains the value $00:000005:000000. The EOR Y1,B instruction logically exclusive ORs the 24-bit value in the Y1 register with bits 47–24 of the B accumulator (B1) and stores the result in the B accumulator with bits 55–48 and 23–0 unchanged.

Condition Codes:

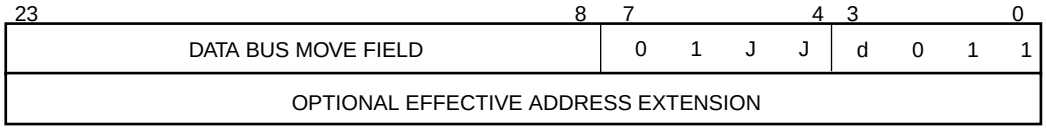


I — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z— **Set if bits 47 - 24 of A or B result are zero**
V — Always cleared

Instruction Format:

EOR S,D

Opcode:



Instruction Fields:Instruction Fields:

S	J J	D	d
X0	0 0	A	0
X1	1 0	B	1
Y0	0 1		
Y1	1 1		

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

Begin Illegal Instruction
exception processing

Assembler Syntax:

ILLEGAL

Description: The ILLEGAL instruction is executed as if it were a NOP instruction. Normal instruction execution is suspended and illegal instruction exception processing is initiated. The interrupt vector address is located at address P:$3E. The interrupt priority level (I1, I0) is set to 3 in the status register if a long interrupt service routine is used. The purpose of the ILLEGAL instruction is to force the DSP into an illegal instruction exception for test purposes. If a fast interrupt is used with the ILLEGAL instruction, an infinite loop will be formed (an illegal instruction interrupt normally returns to the illegal instruction) which can only be broken by a hardware reset. Therefore, only long interrupts should be used. Exiting an illegal instruction is a fatal error. The long exception routine should indicate this condition and cause the system to be restarted.

If the ILLEGAL instruction is in a DO loop at LA and the instruction at 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, etc. are located at LA. This is why JSR, REP, etc. at LA are restricted. Clearly restrictions cannot be imposed on illegal instructions.

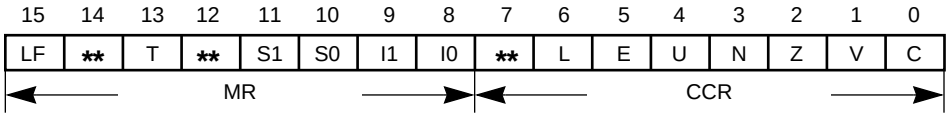
Since REP is uninterruptable, repeating an ILLEGAL instruction results in the interrupt not being initiated until after completion of the REP. After servicing the interrupt, program control will return to the address of the second word following the ILLEGAL instruction. Of course, the ILLEGAL interrupt service routine should abort further processing, and the processor should be reinitialized.

Example:

```
        :
ILLEGAL                 ;begin ILLEGAL exception processing
        :
```

Explanation of Example: The ILLEGAL instruction suspends normal instruction execution and initiates ILLEGAL exception processing.

Condition Codes:

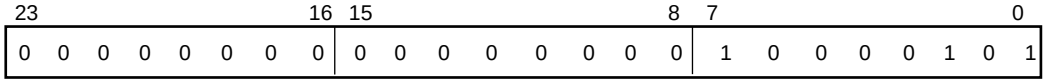


The condition codes are not affected by this instruction.

Instruction Format:

ILLEGAL

Opcode:



Instruction Fields:

None

Timing: 8 oscillator clock cycles

Memory: 1 program word

Operation:

If cc, then 0xxx →PC
else PC+1 →PC

If cc, then ea →PC
else PC+1 →PC

Assembler Syntax:

Jcc xxx

Jcc xxx

Description: Jump to the location in program memory given by the instruction's effective address if the specified condition is true. If the specified condition is false, the program counter (PC) is incremented and the effective address is ignored. However, the address register specified in the effective address field is always updated independently of the specified condition. All memory alterable addressing modes may be used for the effective address. A Fast Short Jump addressing mode may also be used. The 12-bit data is zero extended to form the effective address. See A.8 INSTRUCTION SEQUENCE RESTRICTIONS for restrictions. The term ''cc'' may specify the following conditions:

	''cc'' Mnemonic	**Condition**
CC (HS)	— carry clear (higher or same)	C=0
CS (LO)	— carry set (lower)	C=1
EC	— extension clear	E=0
EQ	— equal	Z=1
ES	— extension set	E=1
GE	— greater than or equal	N ⊕ V=0
GT	— greater than	Z+(N ⊕ V)=0
LC	— limit clear	L=0
LE	— less than or equal	Z+(N ⊕ V)=1
LS	— limit set	L=1
LT	— less than	N ⊕ V=1
MI	— minus	N=1
NE	— not equal	Z=0
NR	— normalized	Z+($\overline{U}$•$\overline{E}$)=1
PL	— plus	N=0
NN	— not normalized	Z+($\overline{U}$•$\overline{E}$)=0

where
$\overline{U}$ denotes the logical complement of U,
+ denotes the logical OR operator,
• denotes the logical AND operator, and
⊕ denotes the logical Exclusive OR operator

Restrictions: A Jcc instruction used **within a DO loop** cannot begin at the address LA within that DO loop.

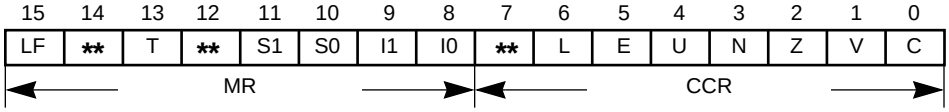
A Jcc instruction cannot be repeated using the REP instruction.

Example:

```
        :
JNN – (R4)                  ;jump to P:(R4) –1 if not normalized
        :
```

Explanation of Example: In this example, program execution is transferred to the address P:(R4)–1 if the result is not normalized. Note that the contents of address register R4 are predecremented by 1, and the resulting address is then loaded into the program counter (PC) if the specified condition is true. If the specified condition is not true, no jump is taken, and the program counter is incremented by one.

Condition Codes:

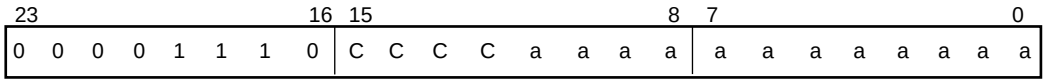


The condition codes are not affected by this instruction.

Instruction Format:

Jcc xxx

Opcode:



Instruction Fields:
cc=4-bit condition code=CCCC,
xxx=12-bit Short Jump Address=aaaaaaaaaaaa

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

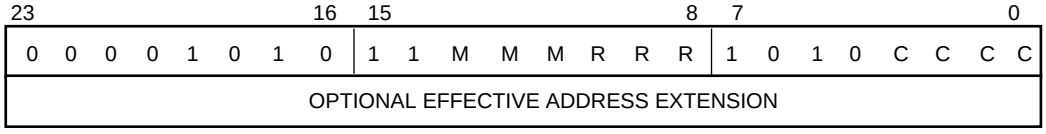
Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

Jcc ea

Opcode:



Instruction Fields:
cc=4-bit condition code=CCCC,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r
Absolute Address	1	1	0	0	0	0

where “rrr” refers to an address register R0-R7

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

Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Operation:	Assembler Syntax:
If S[n]=0, then xxxx→PC else PC+1→PC	JCLR #n,X:ea,xxxx
If S[n]=0, then xxxx →PC else PC+1 →PC	JCLR #n,X:aa,xxxx
If S[n]=0, then xxxx →PC else PC+1 →PC	JCLR #n,X:pp,xxxx
If S[n]=0, then xxxx →PC else PC+1 →PC	JCLR #n,Y:ea,xxxx
If S[n]=0, then xxxx →PC else PC+1 →PC	JCLR #n,Y:aa,xxxx
If S[n]=0, then xxxx →PC else PC+1 →PC	JCLR #n,Y:pp,xxxx
If S[n]=0, then xxxx →PC else PC+1 →PC	JCLR #n,S,xxxx

Description: Jump to the 16-bit absolute address in program memory specified in the instruction's 24-bit extension word if the n^{th} bit of the source operand S is clear. The bit to be tested is selected by an immediate bit number from 0–23. If the specified memory bit is not clear, the program counter (PC) is incremented and the absolute address in the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the n^{th} bit. All address register indirect addressing modes may be used to reference the source operand S. Absolute Short and I/O Short addressing modes may also be used.

Restrictions: A JCLR instruction cannot be repeated using the REP instruction.

A JCLR located at LA, LA–1, or LA–2 of the DO loop cannot specify the program controller registers SR, SP, SSH, SSL, LA, or LC as its target.

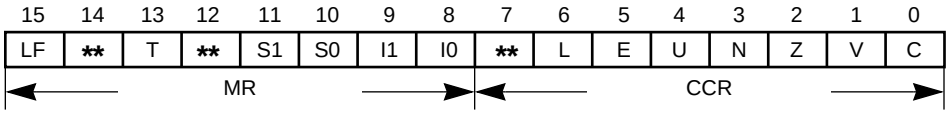
JCLR SSH or JCLR SSL cannot **follow** an instruction that changes the SP.

Example:

```
        :
JCLR #$5,X:<<$FFF1,$1234        ;go to P:$1234 if bit 5 in SCI SSR is clear
        :
```

Explanation of Example: In this example, program execution is transferred to the address P:$1234 if bit 5 (PE) of the 8-bit read-only X memory location X:$FFF1 (I/O SCI interface status register) is a zero. If the specified bit is not clear, no jump is taken, and the program counter (PC) is incremented by one.

Condition Codes:

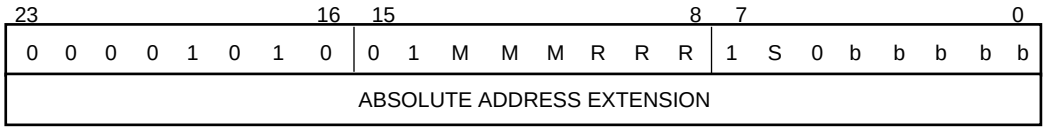


The condition codes are not affected by this instruction.

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR
xxxx=16-bit Absolute Address in extension word

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

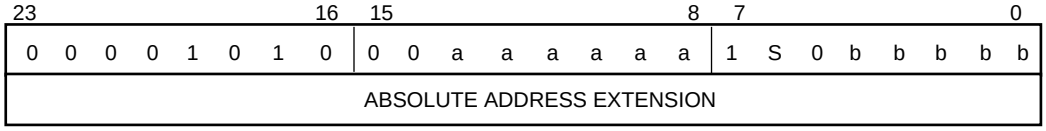
where “rrr” refers to an address register R0-R7

Timing: 6+jx oscillator clock cycles
Memory: 2 program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa
xxxx=16-bit Absolute Address in extension word

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

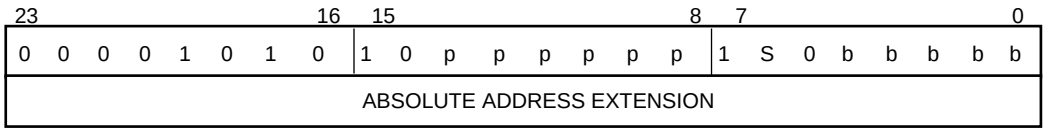
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
pp=6-bit I/O Short Address=pppppp
xxxx=16-bit Absolute Address in extension word

I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

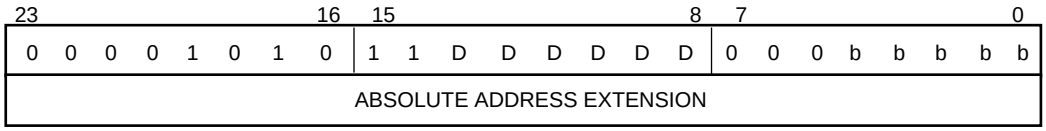
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

JCLR #n,S,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
S=source register=DDDDDD
xxxx=16-bit Absolute Address in extension word

Destination Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See A.9 INSTRUCTION ENCODING and Table A-18 for specific register encodings.

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Operation:

0xxx → PC

ea → PC

Assembler Syntax:

JMP xxx

JMP ea

Description: Jump to the location in program memory given by the instruction's effective address. All memory alterable addressing modes may be used for the effective address. A Fast Short Jump addressing mode may also be used. The 12-bit data is zero extended to form the effective address.

Restrictions: A JMP instruction used **within a DO loop** cannot begin at the address LA within that DO loop.

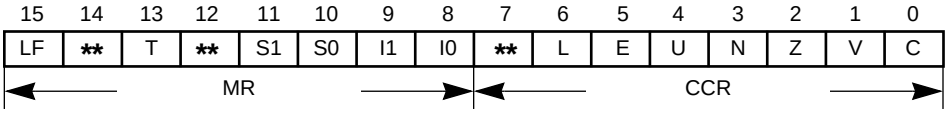
A JMP instruction cannot be repeated using the REP instruction.

Example:

```
        :
JMP (R1+N1)          ;jump to program address P:(R1+N1)
        :
```

Explanation of Example: In this example, program execution is transferred to the program address P:(R1+N1).

Condition Codes:

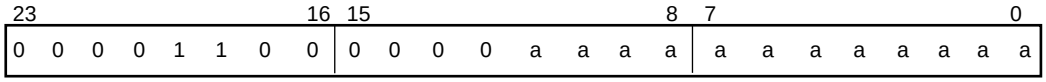


The condition codes are not affected by this instruction.

Instruction Format:

JMP xxx

Opcode:



Instruction Fields:

xxx=12-bit Short Jump Address=aaaaaaaaaaaa

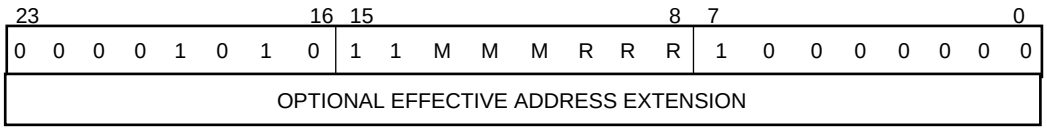
Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

JMP ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r
Absolute address	1	1	0	0	0	0

where “rrr” refers to an address register R0-R7

Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Operation:	**Assembler Syntax:**
If cc, then SP+1→SP; PC→SSH; SR→SSL; 0xxx→PC else PC+1→PC	JScc xxx
If cc, then SP+1→SP; PC→SSH; SR→SSL; ea→PC else PC+1→PC	JScc ea

Description: Jump to the subroutine whose location in program memory is given by the instruction's effective address if the specified condition is true. If the specified condition is true, the address of the instruction immediately following the JScc instruction (PC) and the system status register (SR) are pushed onto the system stack. Program execution then continues at the specified effective address in program memory. If the specified condition is false, the program counter (PC) is incremented, and any extension word is ignored. However, the address register specified in the effective address field is always updated independently of the specified condition. All memory alterable addressing modes may be used for the effective address. A fast short jump addressing mode may also be used. The 12-bit data is zero extended to form the effective address. The term ''cc'' may specify the following conditions:

	''cc'' Mnemonic	**Condition**
CC (HS)	— carry clear (higher or same)	C=0
CS (LO)	— carry set (lower)	C=1
EC	— extension clear	E=0
EQ	— equal	Z=1
ES	— extension set	E=1
GE	— greater than or equal	N ⊕ V=0
GT	— greater than	Z+(N ⊕ V)=0
LC	— limit clear	L=0
LE	— less than or equal	Z+(N ⊕ V)=1
LS	— limit set	L=1
LT	— less than	N ⊕ V=1
MI	— minus	N=1
NE	— not equal	Z=0
NR	— normalized	Z+($\overline{U}$•$\overline{E}$)=1
PL	— plus	N=0
NN	— not normalized	Z+($\overline{U}$•$\overline{E}$)=0

where
$\overline{U}$ denotes the logical complement of U,
+ denotes the logical OR operator,
• denotes the logical AND operator, and
⊕ denotes the logical Exclusive OR operator

Restrictions: A JScc instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JScc instruction used **within in a DO loop** cannot begin at the address LA within that DO loop.

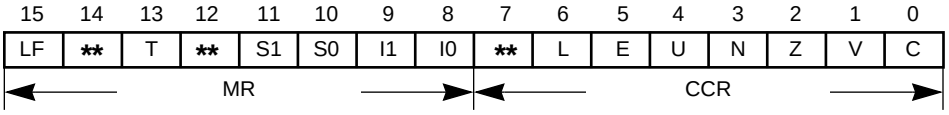
A JScc instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSLS (R3+N3)          ;jump to subroutine at P:(R3+N3) if limit set (L=1)
        :
```

Explanation of Example: In this example, program execution is transferred to the subroutine at address P:(R3+N3) in program memory if the limit bit is set (L=1). Both the return address (PC) and the status register (SR) are pushed onto the system stack prior to transferring program control to the subroutine if the specified condition is true. If the specified condition is not true, no jump is taken and the program counter is incremented by 1.

Condition Codes:

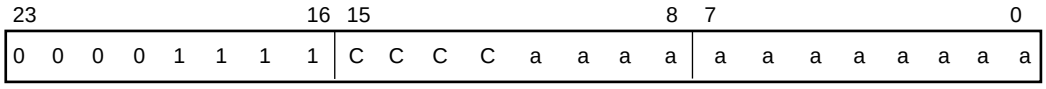


The condition codes are not affected by this instruction.

Instruction Format:

JScc xxx

Opcode:



Instruction Fields:
cc=4-bit condition code=CCCC,
xxx=12-bit Short Jump Address=aaaaaaaaaaaa

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

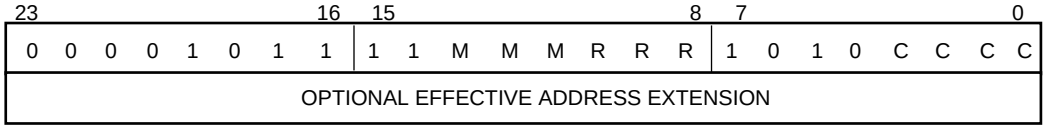
Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

JScc ea

Opcode:



Instruction Fields:

cc=4-bit condition code=CCCC,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R	Mnemonic	C	C	C	C	Mnemonic	C	C	C	C
(Rn)–Nn	0	0	0	r	r	r	CC (HS)	0	0	0	0	CS (LO)	1	0	0	0
(Rn)+Nn0	0	0	1	r	r	r	GE	0	0	0	1	LT	1	0	0	1
(Rn)–	0	1	0	r	r	r	NE	0	0	1	0	EQ	1	0	1	0
(Rn)+	0	1	1	r	r	r	PL	0	0	1	1	MI	1	0	1	1
(Rn)	1	0	0	r	r	r	NN	0	1	0	0	NR	1	1	0	0
(Rn+Nn)	1	0	1	r	r	r	EC	0	1	0	1	ES	1	1	0	1
–(Rn)	1	1	1	r	r	r	LC	0	1	1	0	LS	1	1	1	0
Absolute address	1	1	0	0	0	0	GT	0	1	1	1	LE	1	1	1	1

where ‘‘rrr’’ refers to an address register R0–R7

Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Operation:	Assembler Syntax
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx →PC else PC+1 →PC	JSCLR #n,X:ea,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,X:aa,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,X:pp,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,Y:ea,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,Y:aa,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,Y:pp,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,S,xxxx

Description: Jump to the subroutine at the 16-bit absolute address in program memory specified in the instruction's 24-bit extension word if the n^{th} bit of the source operand S is clear. The bit to be tested is selected by an immediate bit number from 0–23. If the n^{th} bit of the source operand S is clear, the address of the instruction immediately following the JSCLR instruction (PC) and the system status register (SR) are pushed onto the system stack. Program execution then continues at the specified absolute address in the instruction's 24-bit extension word. If the sepcified memory bit is not clear, the program counter (PC) is incremented and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the n^{th} bit. All address register indirect addressing modes may be used to reference the

source operand S. Absolute short and I/O short addressing modes may also be used.

Restrictions: A JSCLR instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JSCLR located at LA, LA–1, or LA–2 of a DO loop, cannot specify the program controller registers SR, SP, SSH, SSL, LA, or LC as its target.

JSCLR SSH or JSCLR SSL cannot **follow** an instruction that changes the SP.

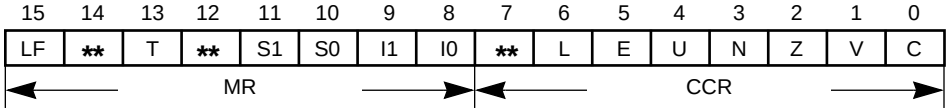
A JSCLR instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSCLR #$1,Y:<<$FFE3,$1357      ;go sub. at P:$1357 if bit 1 in Y:$FFE3 is clear
        :
```

Explanation of Example: In this example, program execution is transferred to the subroutine at absolute address P:$1357 in program memory if bit 1 of the external I/O location Y:<<$FFE3 is a zero. If the specified bit is not clear, no jump is taken and the program counter (PC) is incremented by 1.

Condition Codes:

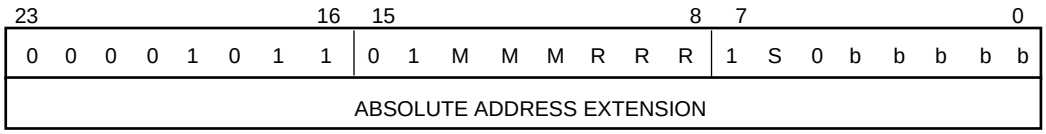


The condition codes are not affected by this instruction.

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR,
xxxx=16-bit Absolute Address in extension word

Effective Addressing Mode	M M M R R R	Memory Space	S	Bit Number bbbbb
(Rn)-Nn	0 0 0 r r r	X Memory	0	00000
(Rn)+Nn	0 0 1 r r r	Y Memory	1	•
(Rn)-	0 1 0 r r r			•
(Rn)+	0 1 1 r r r			•
(Rn)	1 0 0 r r r			10111
(Rn+Nn)	1 0 1 r r r			
-(Rn)	1 1 1 r r r			

where “rrr” refers to an address register R0-R7

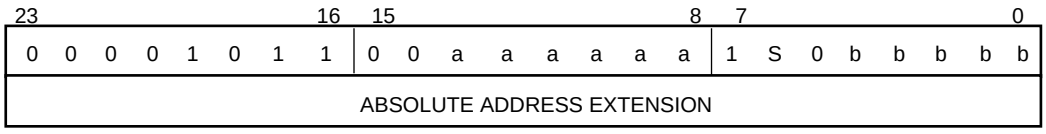
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa,
xxxx=16-bit Absolute Address in extension word

Absolute Short Address aaaaaa	Memory Space S		Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

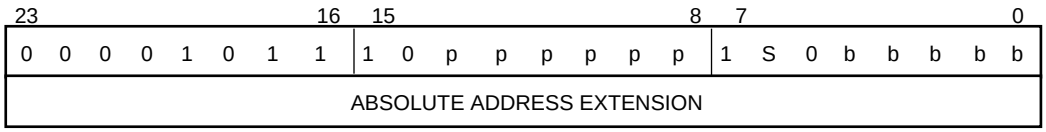
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
pp=6-bit I/O Short Address=pppppp,
xxxx=16-bit Absolute Address in extension word

I/O Short Address aaaaaa	Memory Space S		Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

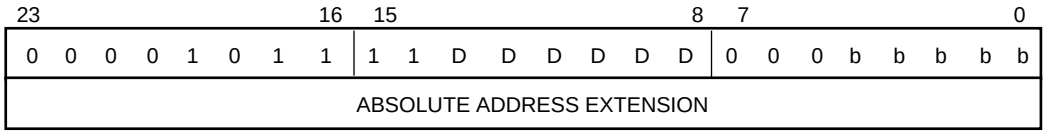
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

JSCLR #n,S,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
S=source register=DDDDDD,
xxxx=16-bit Absolute Address in extension word

Destination Register	**D D D D D D**	**Bit Number bbbbb**
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See A.9 INSTRUCTION ENCODING and Table A-18 for specific register encodings.

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Operation:	Assembler Syntax:	
If S[n]=0, then xxxx→PC else PC+1→PC	JSET	#n,X:ea,xxxx
If S[n]=1, then xxxx→PC else PC+1→PC	JSET	#n,X:ea,xxxx
If S[n]=1, then xxxx→PC else PC+1→PC	JSET	#n,X:aa,xxxx
If S[n]=1, then xxxx →PC else PC+1→PC	JSET	#n,X:pp,xxxx
If S[n]=1, then xxxx→PC else PC+1→PC	JSET	#n,Y:ea,xxxx
If S[n]=1, then xxxx →PC else PC+1→PC	JSET	#n,Y:aa,xxxx
If S[n]=1, then xxxx→PC else PC+1→PC	JSET	#n,Y:pp,xxxx
If S[n]=1, then xxxx→PC else PC+1→PC	JSET	#n,S,xxxx

Description: Jump to the 16-bit absolute address in program memory specified in the instruction's 24-bit extension word if the n^{th} bit of the source operand S is set. The bit to be tested is selected by an immediate bit number from 0–23. If the specified memory bit is not set, the program counter (PC) is incremented, and the absolute address in the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the n^{th} bit. All address register indirect addressing modes may be used to reference the source operand S. Absolute short and I/O short addressing modes may also be used.

Restrictions: A JSET instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JSET located at LA, LA–1, or LA–2 of a DO loop cannot specify the program controller registers SR, SP, SSH, SSL, LA, or LC as its target.

JSET SSH or JSET SSL cannot follow an instruction that changes the SP.

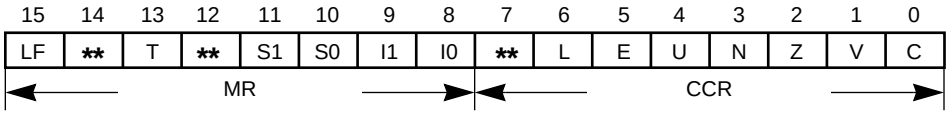
A JSET instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSET #12,X:<<$FFF2,$4321      ;$4321→(PC) if bit 12 (SCI COD) is set
        :
```

Explanation of Example: In this example, program execution is transferred to the address P:$4321 if bit 12 (SCI COD) of the 16-bit read/write I/O register X:$FFF2 is a one. If the specified bit is not set, no jump is taken and the program counter (PC) is incremented by 1.

Condition Codes:

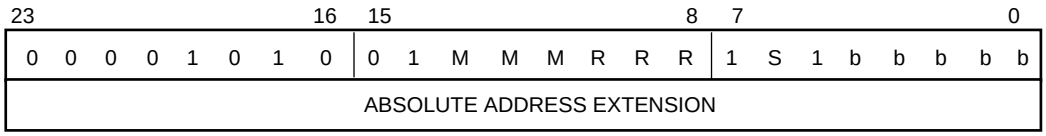


The condition codes are not affected by this instruction.

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR
xxxx=16-bit Absolute Address in extension word

Effective Addressing Mode	**M M M R R R**	**Memory Space S**	**Bit Number bbbbb**
(Rn)-Nn	0 0 0 r r r	X Memory 0	00000
(Rn)+Nn	0 0 1 r r r	Y Memory 1	•
(Rn)-	0 1 0 r r r	•	
(Rn)+	0 1 1 r r r	•	
(Rn)	1 0 0 r r r	10111	
(Rn+Nn)	1 0 1 r r r		
-(Rn)	1 1 1 r r r		

where “rrr” refers to an address register R0-R7

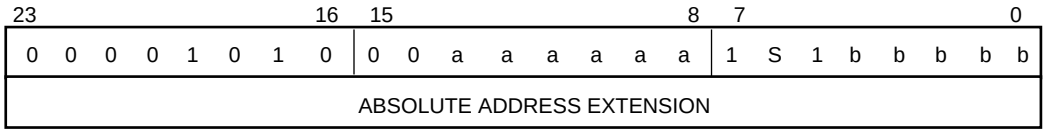
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa,
xxxx=16-bit Absolute Address in extension word

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

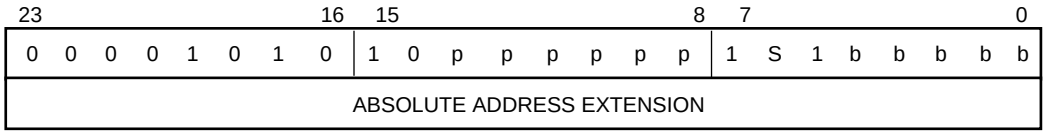
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

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

Opcode:



Instruction Fields:
#n=bit number=bbbbb,
pp=6-bit I/O Short Address=pppppp,
xxxx=16-bit Absolute Address in extension word

I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

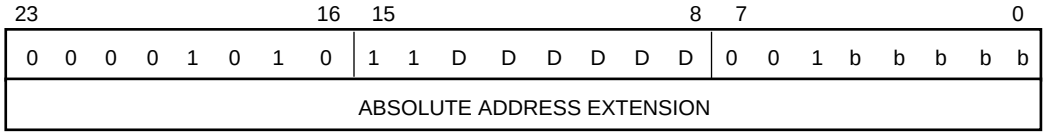
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

JSET #n,S,xxxx

Opcode:



Instruction Fields:
#n=bit number=bbbbb,
S=source register=DDDDDD,
xxxx=16-bit Absolute Address in extension word

Destination Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See A.9 Instruction Encoding and Table A-18 for specific register encodings.

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Operation:	**Assembler Syntax:**
SP+1→SP; PC→SSH; SR→SSL; 0xxx→PC	JSR xxx
SP+→SP; PC→SSH; SR→SSL; ea→PC	JSR ea

Description: Jump to the subroutine whose location in program memory is given by the instruction's effective address. The address of the instruction immediately following the JSR instruction (PC) and the system status register (SR) is pushed onto the system stack. Program execution then continues at the specified effective address in program memory. All memory alterable addressing modes may be used for the effective address. A fast short jump addressing mode may also be used. The 12-bit data is zero extended to form the effective address.

Restrictions: A JSR instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JSR instruction used **within a DO loop** cannot begin at the address LA within that DO loop.

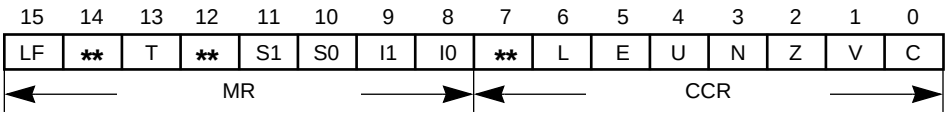
A JSR instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSR (R5)+               ;jump to subroutine at (R5), update R5
        :
```

Explanation of Example: In this example, program execution is transferred to the subroutine at address P:(R5) in program memory, and the contents of the R5 address register are then updated.

Condition Codes:

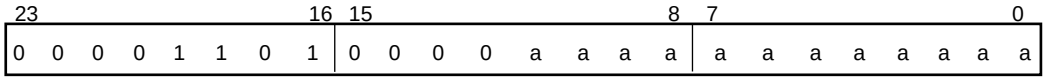


The condition codes are not affected by this instruction.

Instruction Format:

JSR xxx

Opcode:



Instruction Fields:

xxx=12-bit Short Jump Address=aaaaaaaaaaaa

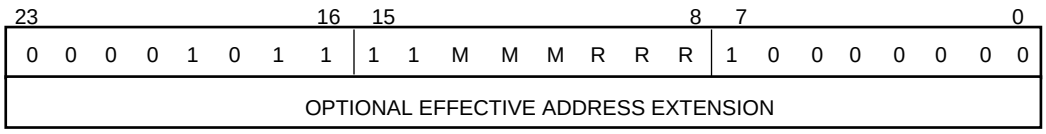
Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

JSR ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	**M**	**M**	**M**	**R**	**R**	**R**
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r
Absolute address	1	1	0	0	0	0

where “rrr” refers to an address register R0-R7

Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Operation:	Assembler Syntax	
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET	#n,X:ea,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET	#n,X:aa,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET	#n,X:pp,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET	#n,Y:ea,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR →SSL; xxxx→PC else PC+1→PC	JSSET	#n,Y:aa,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx →PC else PC+1➡PC	JSSET	#n,Y:pp,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET	#n,S,xxxx

Description: Jump to the subroutine at the 16-bit absolute address in program memory specified in the instruction's 24-bit extension word if the n^{th} bit of the source operand S is set. The bit to be tested is selected by an immediate bit number from 0–23. If the n^{th} bit of the source operand S is set, the address of the instruction immediately following the JSSET instruction (PC) and the system status register (SR) are pushed onto the system stack. Program execution then continues at the specified absolute address in the instruction's 24-bit extension word. If the specified memory bit is not set, the program counter (PC) is incremented, and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the n^{th} bit. All address register indirect addressing modes may be used to reference the source operand S. Absolute short and I/O short addressing modes may also be used.

Restrictions: A JSSET instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JSSET located at LA, LA–1, or LA–2 of a DO loop, cannot specify the program controller registers SR, SP, SSH, SSL, LA, or LC as its target.

JSSET SSH or JSSET SSL cannot **follow** an instruction that changes the SP.

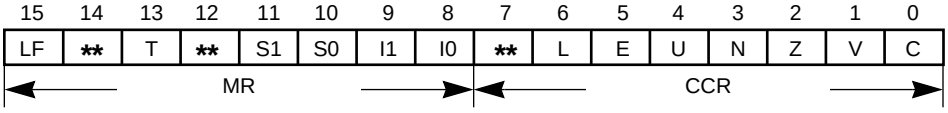
A JSSET instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSSET #$17,Y:<$3F,$100      ;go to sub. at P:$0100 if bit 23 in Y:$3F is set
        :
```

Explanation of Example: In this example, program execution is transferred to the subroutine at absolute address P:$0100 in program memory if bit 23 of Y memory location Y:$003F is a one. If the specified bit is not set, no jump is taken and the program counter (PC) is incremented by 1.

Condition Codes:

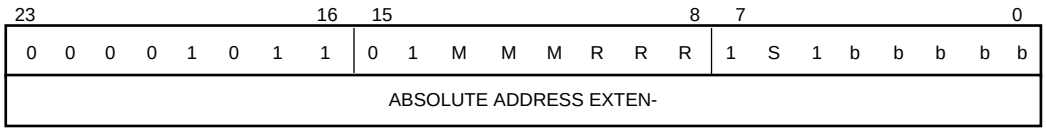


The condition codes are not affected by this instruction.

Instruction Format:

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

Opcode:



Instruction Fields:
#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR,
xxxx=16-bit Absolute Address in extension word

Effective Addressing Mode	M M M R R R	Memory Space	S	Bit Number bbbbb
(Rn)-Nn	0 0 0 r r r	X Memory	0	00000
(Rn)+Nn	0 0 1 r r r	Y Memory	1	•
(Rn)-	0 1 0 r r r			•
(Rn)+	0 1 1 r r r			•
(Rn)	1 0 0 r r r			10111
(Rn+Nn)	1 0 1 r r r			
-(Rn)	1 1 1 r r r			

where “rrr” refers to an address register R0-R7

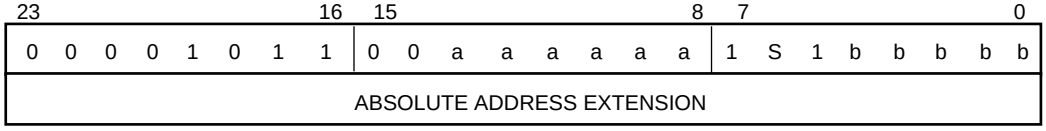
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa,
xxxx=16-bit Absolute Address in extension word

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

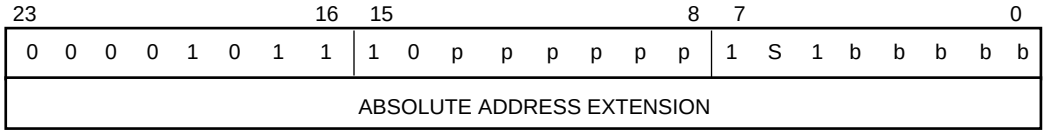
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:

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

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
pp=6-bit I/O Short Address=pppppp,
xxxx=16-bit Absolute Address in extension word

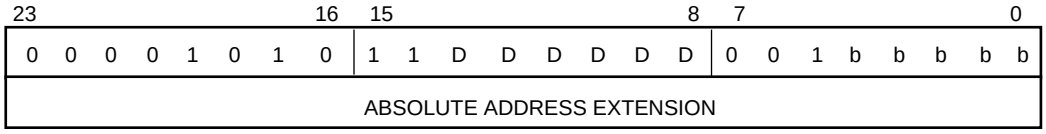
I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			•
111111			10111

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Instruction Format:
JSSET #n,S,xxxx

Opcode:



Instruction Fields:
#n=bit number=bbbbb,
S=source register=DDDDDD,
xxxx=16-bit Absolute Address in extension word

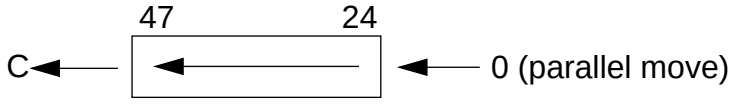
Destination Register	**D D D D D D**	**Bit Number bbbbb**
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See A.9 Instruction Encoding and Table A-18 for specific register encodings.

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Operation:

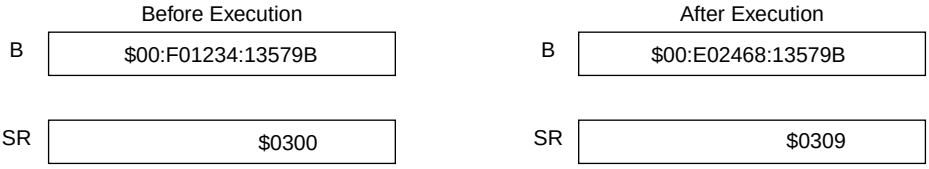


Assembler Syntax: LSL D (parallel move)

Description: Logically shift bits 47–24 of the destination operand D one bit to the left and store the result in the destination accumulator. Prior to instruction execution, bit 47 of D is shifted into the carry bit C, and a zero is shifted into bit 24 of the destination accumulator D. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected. If a zero shift count is specified, the carry bit is cleared. The difference between LSL and ASL is that LSL operates on only A1 or B1 and always clears the V bit.

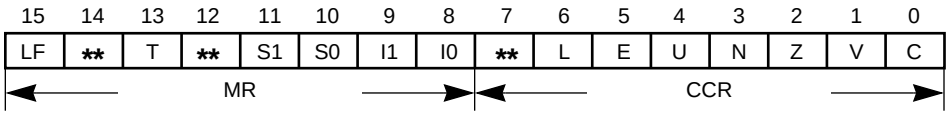
Example:

```
        :
LSL B #$7F,R0           ;shift B1 one bit to the left, set up R0
        :
```



Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $00:F01234:13579B. The execution of the LSL B instruction shifts the 24-bit value in the B1 register one bit to the left and stores the result back in the B1 register.

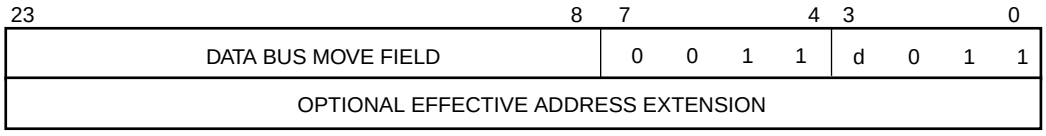
Condition Codes:



L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z— **Set if bits 47–24 of A or B result are zero**
V — Always cleared
C — **Set if bit 47 of A or B was set prior to instruction execution**

Instruction Format:
LSL D

Opcode:



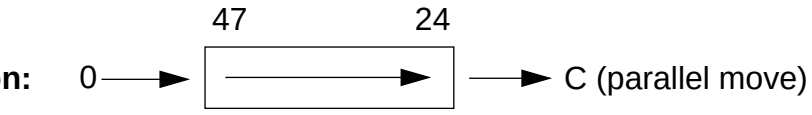
Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

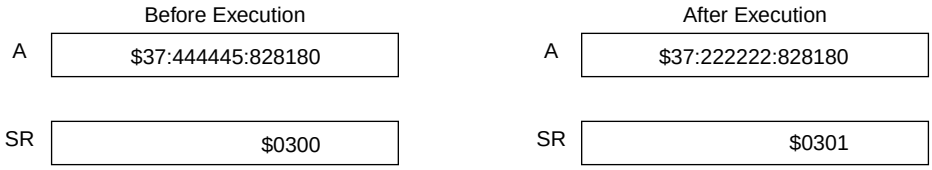


Assembler Syntax: LSR D (parallel move)

Description: Logically shift bits 47–24 of the destination operand D one bit to the right and store the result in the destination accumulator. Prior to instruction execution, bit 24 of D is shifted into the carry bit C, and a zero is shifted into bit 47 of the destination accumulator D. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

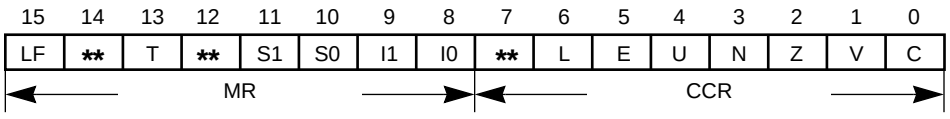
Example:

```
        :
LSR AA1,N4              ;shift A1 one bit to the right, set up N4
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $37:444445:828180. The execution of the LSR A instruction shifts the 24-bit value in the A1 register one bit to the right and stores the result back in the A1 register.

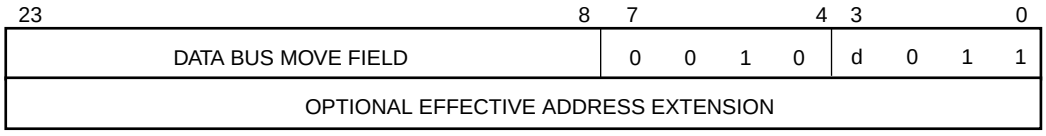
Condition Codes:



L — Set if data limiting has occurred during parallel move
N — **Always cleared**
Z— **Set if bits 47–24 of A or B result are zero**
V — Always cleared
C — **Set if bit 24 of A or B was set prior to instruction execution**

Instruction Format:
LSR D

Opcode:



Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
ea→d

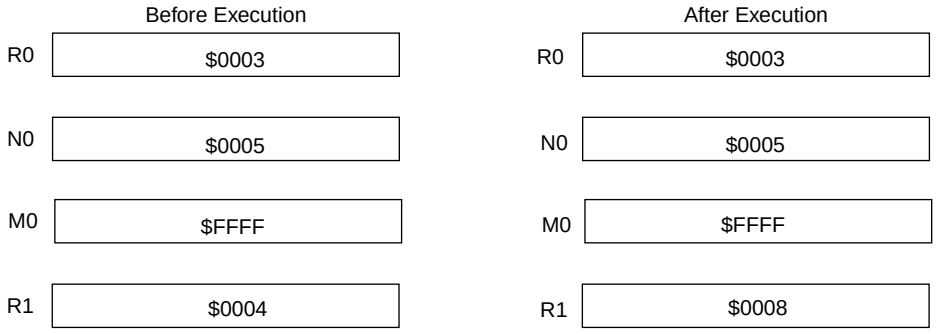
Assembler Syntax:
LUA ea,D

Description: Load the updated address into the destination address register D. The source address register and the update mode used to compute the updated address are specified by the effective address (ea). **Note that the source address register specified in the effective address is not updated.** All update addressing modes may be used.

Note: This instruction is considered to be a move-type instruction. Due to pipelining, the new contents of the destination address register (R0–R7 or N0–N7) will not be available for use during the following instruction (i.e., there is a single instruction cycle pipeline delay).

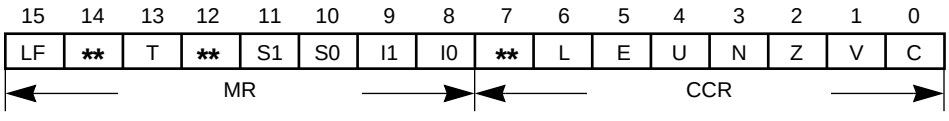
Example:

```
        :
LUA (R0)+N0,R1;update R1 using (R0)+N0
        :
```



Explanation of Example: Prior to execution, the 16-bit address register R0 contains the value $0003, the 16-bit address register N0 contains the value $0005, and the 16-bit address register R1 contains the value $0004. The execution of the LUA (R0)+N0,R1 instruction adds the contents of the R0 register to the contents of the N0 register and stores the resulting updated address in the R1 address register. Note that normally N0 would be added to R0 and deposited in R0; however, for an LUA instruction, the contents of both the R0 and N0 address registers are not affected.

Condition Codes:

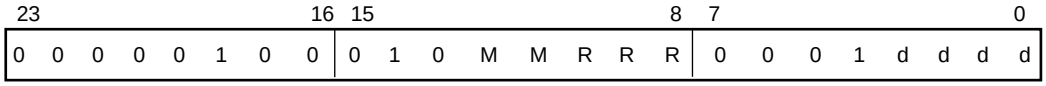


The condition codes are not affected by this instruction.

Instruction Format:

LUA ea,D

Opcode:



Instruction Fields:

ea=5-bit Effective Address=MMRRR,
D=4-bit destination address register=dddd

Effective Addressing Mode	M M M R R R	Dest. Addr. Reg. D	d d d d
(Rn)-Nn	0 0 0 r r r	R0–R7	0 n n n
(Rn)+Nn	0 0 1 r r r	N0–N7	1 n n n
(Rn)-	0 1 0 r r r		
(Rn)+	0 1 1 r r r		

where ‘‘rrr’’ refers to a **source** address register R0–R7
where ‘‘nnn’’ refers to a **destination** address register R0–R7 or N0–N7

Timing: 4 oscillator clock cycles

Memory: 1 program word

Operation:

D±S1∗S2→D (parallel move)

D±S1∗S2→D (parallel move)

Assembler Syntax:

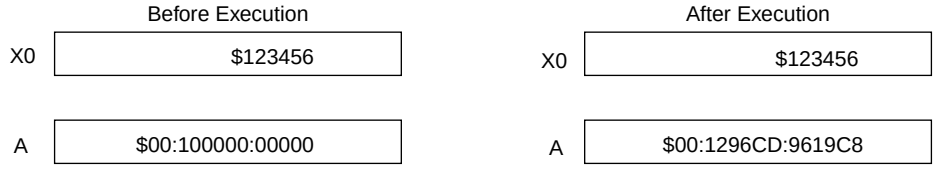
MAC (±)S1,S2,D (parallel move)

MAC (±)S2,S1,D (parallel move)

Description: Multiply the two signed 24-bit source operands S1 and S2 and add/subtract the product to/from the specified 56-bit destination accumulator D. The ‘‘–’’ sign option is used to negate the specified product prior to accumulation. The default sign option is ‘‘+’’.

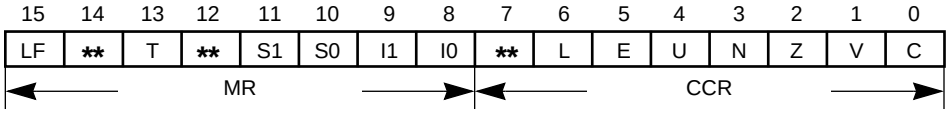
Example:

```
        :
MAC X0,X0,A X:(R2)+N2,Y1        ;square X0 and store in A, update Y1 and R2
        :
```



Explanation of Example: Prior to execution, the 24-bit X0 register contains the value of $123456 (0.142222166), and the 56-bit A accumulator contains the value $00:100000:000000 (0.125). The execution of the MAC X0,X0,A instruction squares the 24-bit signed value in the X0 register and adds the resulting 48-bit product to the 56-bit A accumulator (X0∗X0+lA=0.145227144519197 approximately= $00:1296CD:9619C8=A).

Condition Codes:



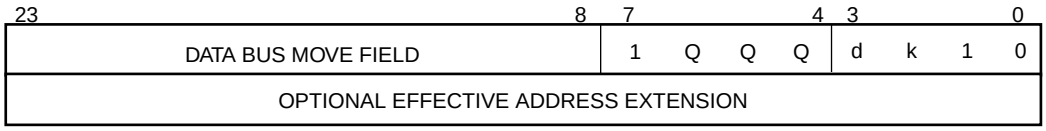
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

MAC (±)S1,S2,D
MAC (±)S2,S1,D

Opcode:



Instruction Fields:

S1∗S2	**Q Q Q**	**Sign**	**k**	**D d**
X0 X0	0 0 0	+	0	A 0
Y0 Y0	0 0 1	–	1	B 1
X1 X0	0 1 0			
Y1 Y0	0 1 1			
X0 Y1	1 0 0			
Y0 X0	1 0 1			
X1 Y0	1 1 0			
Y1 X1	1 1 1			

Note: Only the indicated S1∗S2 combinations are valid. X1∗X1 and Y1∗Y1 are **not** valid.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

D±S1∗S2+r→D (parallel move)

D±S1∗S2+r→ D (parallel move)

Assembler Syntax:

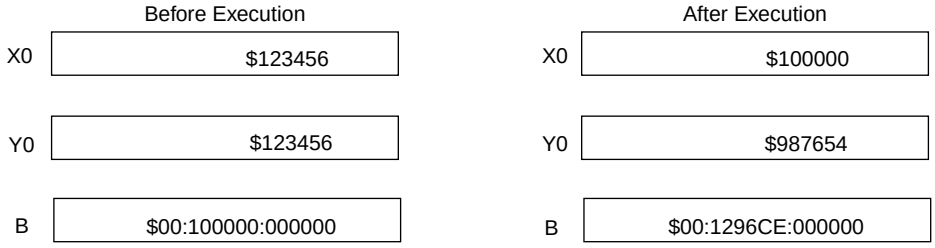
MACR (±)S1,S2,D (parallel move)

MACR (±)S2,S1,D (parallel move)

Description: Multiply the two signed 24-bit source operands S1 and S2, add/subtract the product to/from the specified 56-bit destination accumulator D, and then round the result using convergent rounding. The rounded result is stored in the destination accumulator D. The ‘‘–’’ sign option is used to negate the specified product prior to accumulation. The default sign option is “+’’. The contribution of the LS bits of the result is rounded into the upper portion of the destination accumulator (A1 or B1) by adding a constant to the LS bits of the lower portion of the accumulator (A0 or B0). The value of the constant added is determined by the scaling mode bits S0 and S1 in the status register. Once rounding has been completed, the LS bits of the destination accumulator D (A0 or B0) are loaded with zeros to maintain an unbiased accumulator value which may be reused by the next instruction. The upper portion of the accumulator (A1 or B1) contains the rounded result which may be read out to the data buses. Refer to the RND instruction for more complete information on the convergent rounding process.

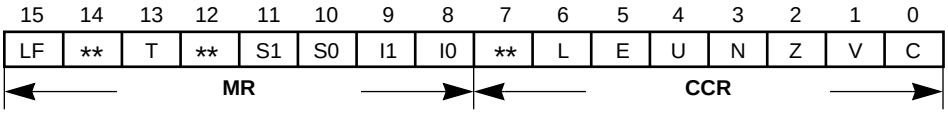
Example:

```
        :
MACR X0,Y0,B    B,X0   Y:(R4)+N4,Y0     ;X0∗Y0+B→B, rnd B, update X0,Y0,R4
        :
```



Explanation of Example: Prior to execution, the 24-bit X0 register contains the value $123456 (0.142222166), the 24-bit Y0 register contains the value $123456 (0.142222166), and the 56-bit B accumulator contains the value $00:100000:000000 (0.125). The execution of the MACR X0,Y0,B instruction multiples the 24-bit signed value in the X0 register by the 24-bit signed value in the Y0 register, adds the resulting product to the 56-bit B accumulator, rounds the result into the B1 portion of the accumulator, and then zeros the B0 portion of the accumulator (X0∗Y0+B=0.145227144519197 approximately =$00:1296CD:9619C8, which is rounded to the value $00:1296CE:000000=0.145227193832397=B).

Condition Codes:



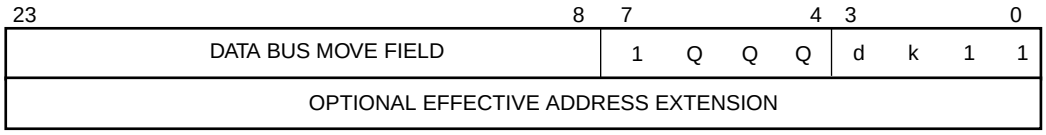
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

MACR (±)S1,S2,D
MACR (±)S2,S1,D

Opcode:



Instruction Fields:

S1∗S2	**Q Q Q**	**Sign**	**k**	**D d**
X0 X0	0 0 0	+	0	A 0
Y0 Y0	0 0 1	–	1	B 1
X1 X0	0 1 0			
Y1 Y0	0 1 1			
X0 Y1	1 0 0			
Y0 X0	1 0 1			
X1 Y0	1 1 0			
Y1 X1	1 1 1			

Note: Only the indicated S1∗S2 combinations are valid. X1∗X1 and Y1∗Y1 are not valid.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
S→D

Assembler Syntax:
MOVE S,D

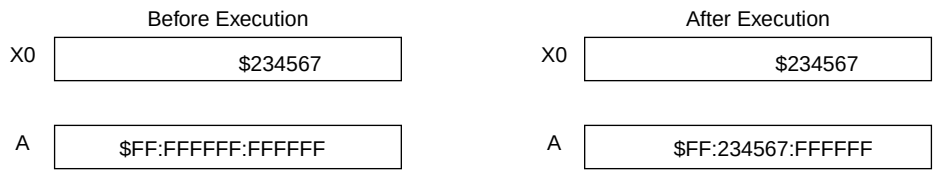
Description: Move the contents of the specified data source S to the specified destination D. This instruction is equivalent to a data ALU NOP with a parallel data move.

When a 56-bit accumulator (A or B) is specified as a **source** operand S, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). 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 24- or 48-bit destination, the value stored in the destination D is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-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 condition code register is latched.

When a 56-bit accumulator (A or B) is specified as a **destination** operand D, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 23) and appending the source operand with 24 LS zeros. Similarly, any 48-bit source data to be loaded into a 56-bit accumulator is automatically sign extended to 56 bits. Note that for 24-bit source operands both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1). Similarly, for 48-bit source operands, the automatic sign-extension feature may be disabled by using the long memory move addressing mode and specifying A10 or B10 as the destination operand.

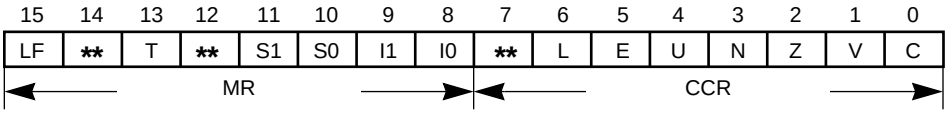
Example:

```
        :
MOVE X0,A1              ;move X0 to A1 without sign ext. or zeroing
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $FF:FFFFFF:FFFFFF, and the 24-bit X0 register contains the value $234567. The execution of the MOVE X0,A1 instruction moves the 24-bit value in the X0 register into the 24-bit A1 register without automatic sign extension and without automatic zeroing.

Condition Codes:

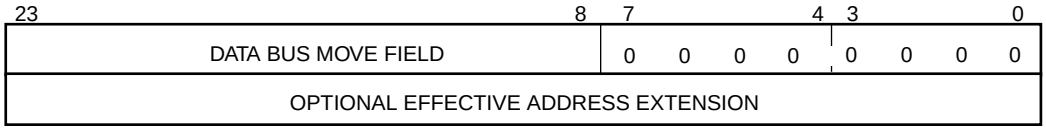


L — Set if data limiting has occurred during parallel move

Instruction Format:

MOVE S,D

Opcode:



Instruction Fields:

See **Parallel Move Descriptions** for data bus move field encoding.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Parallel Move Descriptions: Thirty of the sixty-two instructions provide the capability to specifiy an optional parallel data bus movement over the X and/or Y data bus. This allows a data ALU operation to be executed in parallel with up to two data bus moves during the instruction cycle. Ten types of parallel moves are permitted, including register to register moves, register to memory moves, and memory to register moves. However, not all addressing modes are allowed for each type of memory reference. Addressing mode restrictions which apply to specific types of moves are noted in the individual move operation descriptions. The following section contains detailed descriptions about each type of parallel move operation.

When a 56-bit accumulator (A or B) is specified as a **source** operand S, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). 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 24- or 48-bit destination, the value stored in the destination D is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-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 condition code register is latched.

When a 56-bit accumulator (A or B) is specified as a **destination** operand D, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 23) and appending the source operand with 24 LS zeros. Similarly, any 48-bit source data to be loaded into a 56-bit accumulator is automatically sign extended to 56 bits. Note that for 24-bit source operands both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1). Similarly, for 48-bit source operands, the automatic sign-extension feature may be disabled by using the long memory move addressing mode and specifying A10 or B10 as the destination operand.

Note that the symbols used in decoding the various opcode fields of an instruction or parallel move are **completely arbitrary**. Furthermore, the opcode symbols used in one instruction or parallel move are **completely independent** of the opcode symbols used in a different instruction or parallel move.

Operation:
(. . . . .)

Assembler Syntax:
(. . . . .)

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

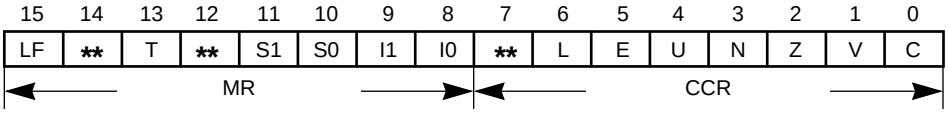
Description: Many (30 of the total 62) instructions in the DSP56000/DSP56001 instruction set allow parallel moves. The parallel moves have been divided into 10 opcode categories. This category is a parallel move NOP and does not involve data bus move activity.

Example:

```
        :
ADD X0,A        ;add X0 to A (no parallel move)
        :
```

Explanation of Example: This is an example of an instruction which allows parallel moves but does not have one.

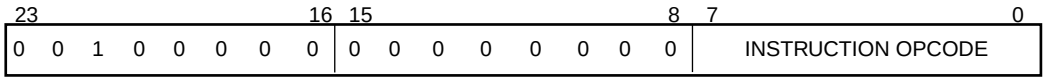
Condition Codes:



The condition codes are not affected by this type of parallel move.

Instruction Format:
(. . . . .)

Opcode:



Instruction Format:
(defined by instruction)

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:
(. . . . .), #xx→D

Assembler Syntax:
(. . . . .) #xx,D

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move the 8-bit immediate data value (#xx) into the destination operand D.

If the destination register D is A0, A1, A2, B0, B1, B2, R0–R7, or N0–N7, the 8-bit immediate short operand is interpreted as an **unsigned integer** and is stored in the specified destination register. That is, the 8-bit data is stored in the eight LS bits of the destination operand, and the remaining bits of the destination operand D are zeroed.

If the destination register D is X0, X1, Y0, Y1, A, or B, the 8-bit immediate short operand is interpreted as a **signed fraction** and is stored in the specified destination register. That is, the 8-bit data is stored in the eight MS bits of the destination operand, and the remaining bits of the destination operand D are zeroed.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D. That is, **duplicate destinations are NOT allowed within the same instruction.**

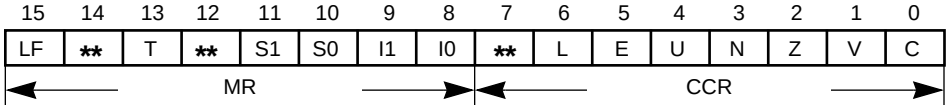
Note: This parallel data move is considered to be a move-type instruction. Due to pipelining, if an address register (R or N) is changed using a move-type instruction, the new contents of the destination address register will not be available for use during the following instruction (i.e., there is a single instruction cycle pipeline delay).

Example:

```
        :
ABS B #$18,R1          ;take absolute value of B, #$18→R1
        :
```

Explanation of Example: Prior to execution, the 16-bit address register R1 contains the value $0000. The execution of the parallel move portion of the instruction, #$18,R1, moves the 8-bit immediate short operand into the eight LS bits of the R1 register and zeros the remaining eight MS bits of that register. The 8-bit value is interpreted as an unsigned integer since its destination is the R1 address register.

Condition Codes:

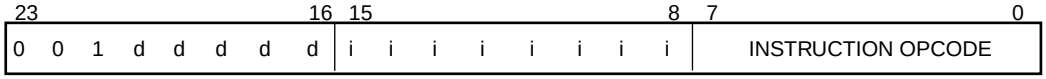


The condition codes are not affected by this type of parallel move.

Instruction Format:

(. . . . .) #xx,D

Opcode:



Instruction Fields:

#xx=8-bit Immediate Short Data=iiiiiiii

D	d	d	d	d	d	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no
X1	0	0	1	0	1	no	no
Y0	0	0	1	1	0	no	no
Y1	0	0	1	1	1	no	no
A0	0	1	0	0	0	no	no
B0	0	1	0	0	1	no	no
A2	0	1	0	1	0	no	no
B2	0	1	0	1	1	no	no
A1	0	1	1	0	0	no	no
B1	0	1	1	0	1	no	no
A	0	1	1	1	0	A2	A0
B	0	1	1	1	1	B2	B0
R0-R7	1	0	r	r	r		
N0-N7	1	1	n	n	n		

where “rrr”=Rn number
where “nnn”=Nn number

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:
(. . . . .); S→D

Assembler Syntax:
(. . . . .) S,D

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move the source register S to the destination register D.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D. That is, **duplicate destinations are NOT allowed within the same instruction**.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S 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**.

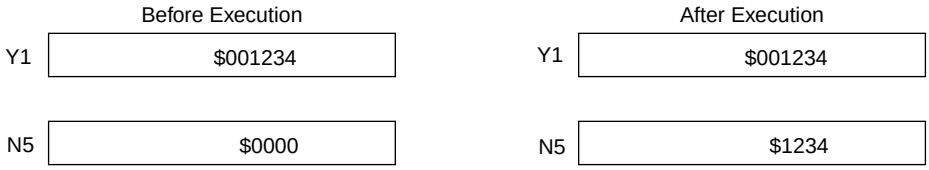
When a 24-bit source operand is moved into a 16-bit destination register, the 16 LS bits of the 24-bit source operand are stored in the 16-bit destination register. When a 16-bit source operand is moved into a 24-bit destination register, the 16 LS bits of the destination register are loaded with the contents of the 16-bit source operand, and the eight MS bits of the 24-bit destination register are zeroed.

Note: The MOVE A,B operation will result in a 24-bit positive or negative saturation constant being stored in the B1 portion of the B accumulator if the signed integer portion of the A accumulator is in use.

Note: This parallel data move is considered to be a move-type instruction. Due to pipelining, if an address register (R or N) is changed using a move-type instruction, the new contents of the destination address register will not be available for use during the following instruction (i.e., there is a single instruction cycle pipeline delay).

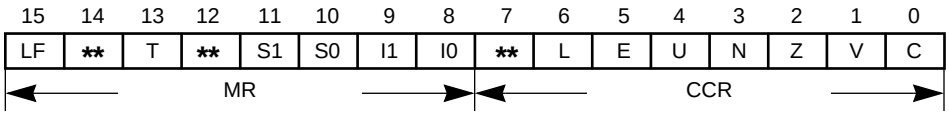
Example:

```
        :
MACR–X0,Y0,A   Y1,N5          ;–X0*Y0+A→A, move Y1→N5
        :
```



Explanation of Example: Prior to execution, the 24-bit Y1 register contains the value $001234 and the 16-bit address offset register N5 contains the value $0000. The execution of the parallel move portion of the instruction, Y1,N5, moves the 16 LS bits of the 24-bit value in the Y1 register into the 16-bit N5 register.

Condition Codes:

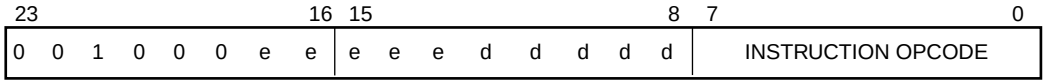


L — Set if data limiting has occurred during parallel move

Instruction Format:

(. . . . .) S,D

Opcode:



Instruction Fields:

S or D	e d	e d	e d	e d	e d	S S/L	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where “rrr”=Rn number
where “nnn”=Nn number

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:
(. . . . .); ea→Rn

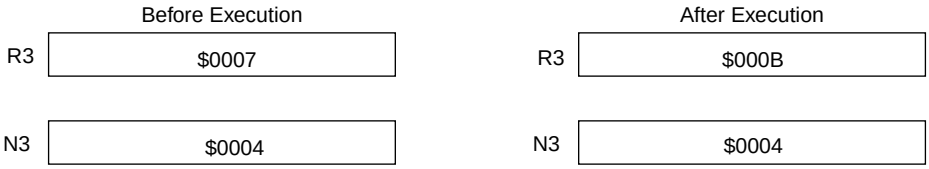
Assembler Syntax:
(. . . . .) ea

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Update the specified address register according to the specified effective addressing mode. All update addressing modes may be used.

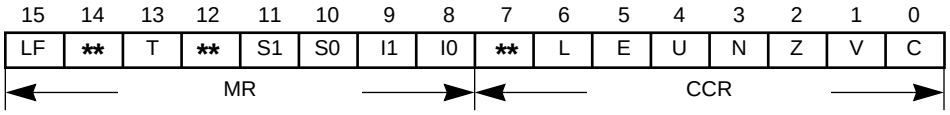
Example:

```
                :
                RND B (R3)+N3;round value in B into B1, R3+N3→R3
        :
```



Explanation of Example: Prior to execution, the 16-bit address register R3 contains the value $0007, and the 16-bit address offset register N3 contains the value $0004. The execution of the parallel move portion of the instruction, (R3)+N3, updates the R3 address register according to the specified effective addressing mode by adding the value in the R3 register to the value in the N3 register and storing the 16-bit result back in the R3 address register.

Condition Codes:

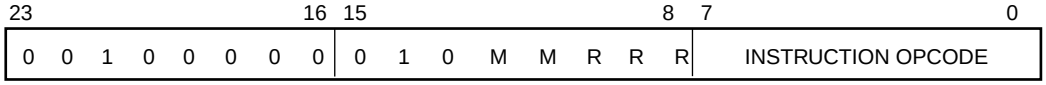


The condition codes are not affected by this type of parallel move.

Instruction Format:

(. . . . .) ea

Opcode:



Instruction Fields:

ea=5-bit Effective Address=MMRRR

Effective Addressing Mode	**M**	**M**	**R**	**R**	**R**
(Rn)-Nn	0	0	r	r	r
(Rn)+Nn	0	1	r	r	r
(Rn)-	1	0	r	r	r
(Rn)+	1	1	r	r	r

where “rrr” refers to an address register R0-R7

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:	Assembler Syntax:
(. . . . .); X:ea→D	(. . . . .) X:ea,D
(. . . . .); X:aa→D	(. . . . .) X:aa,D
(. . . . .); S→X:ea	(. . . . .) S,X:ea
(. . . . .); S→X:aa	(. . . . .) S,X:aa
(. . . . .); #xxxxxx→D	(. . . . .) #xxxxxx,D

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move the specified word operand from/to X memory. All memory addressing modes, including absolute addressing and 24-bit immediate data, may be used. Absolute short addressing may also be used.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D. That is, **duplicate destinations are NOT allowed within the same instruction**.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S 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**.

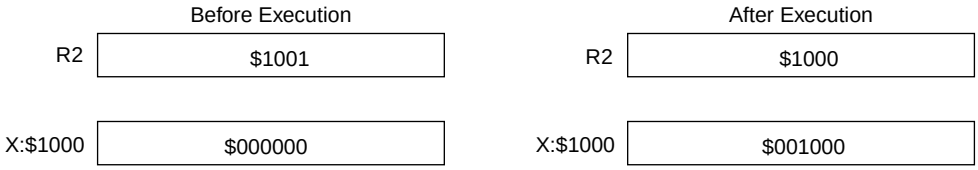
When a 24-bit source operand is moved into a 16-bit destination register, the 16 LS bits of the 24-bit source operand are stored in the 16-bit destination register. When a 16-bit source operand is moved into a 24-bit destination register, the 16 LS bits of the destination register are loaded with the contents of the 16-bit source operand, and the eight MS bits of the 24-bit destination register are zeroed.

Note: This parallel data move is considered to be a move-type instruction. Due to pipe-

lining, if an address register (R or N) is changed using a move-type instruction, the new contents of the destination address register will not be available for use during the following instruction (i.e., there is a single instruction cycle pipeline delay).

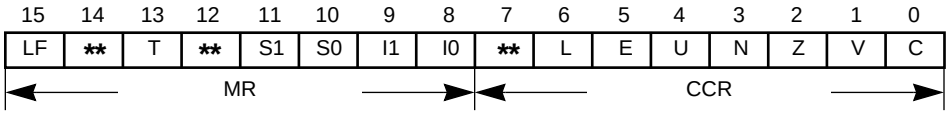
Example:

```
        :
ASL A   R2,X:–(R2)          ;A*2→A, save updated R2 in X:(R2)
        :
```



Explanation of Example: Prior to execution, the 16-bit R2 address register contains the value $1001, and the 24-bit X memory location X:$1000 contains the value $000000. The execution of the parallel move portion of the instruction, R2,X:–(R2), predecrements the R2 address register and then uses the R2 address register to move the updated contents of the R2 address register into the 24-bit X memory location X:$1000.

Condition Codes:



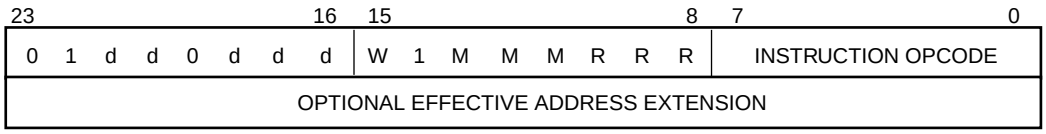
L — Set if data limiting has occurred during parallel move.

Note: The MOVE A,X:ea operation will result in a 24-bit positive or negative saturation constant being stored in the specified 24-bit X memory location if the signed integer portion of the A accumulator is in use.

Instruction Format:

(. . . . .) X:ea,D
(. . . . .) S,X:ea
(. . . . .) #xxxxxx,D

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate data	1	1	0	1	0	0

S,D	d	d	d	d	d	S S/L	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where “rrr”=Rn number
where “nnn”=Nn number

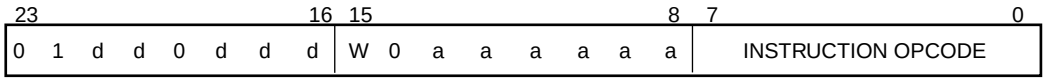
Timing: mv oscillator clock cycles

Memory: mv program words

Instruction Format:

(. . . . .) X:aa,D
(. . . . .) S,X:aa

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Register W		**Absolute Short Address a a a a a a**
Read S	0	0 0 0 0 0 0
Write D	1	•
		1 1 1 1 1 1

S,D	**d**	**d**	**d**	**d**	**d**	**S S/L**	**D Sign Ext**	**D Zero**
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where “rrr”=Rn number
where “nnn”=Nn number

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:	**Assembler Syntax:**
Class I	**Class I**
(. . . . .); X:ea→D1; S2→D2	(. . . . .) X:ea,D1 S2,D2
(. . . . .); S1→X:ea; S2→D2	(. . . . .) S1,X:ea S2,D2
(. . . . .); #xxxxxx→D1; S2→D2	(. . . . .) #xxxxxx,D1 S2,D2
Class II	**Class II**
(. . . . .); A→X:ea; X0→A	(. . . . .) A,X:ea X0,A
(. . . . .); B→X:ea; X0→B	(. . . . .) B,X:ea X0,B

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Class I: Move a one-word operand from/to X memory and move another word operand from an accumulator (S2) to an input register (D2). All memory addressing modes, including absolute addressing and 24-bit immediate data, may be used. The register to register move (S2,D2) allows a data ALU accumulator to be moved to a data ALU input register for use as a data ALU operand in the following instruction.

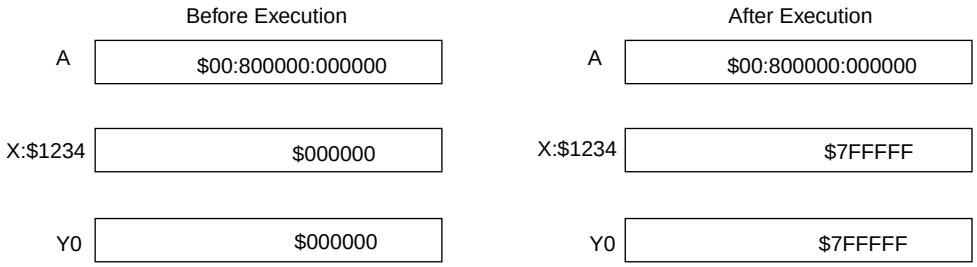
Class II: Move one-word operand from a data ALU accumulator to X memory and one-word operand from data ALU register X0 to a data ALU accumulator. One effective address is specified. All memory addressing modes, excluding long absolute addressing and long immediate data, may be used.

For both Class I and Class II X:R parallel data moves, if the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D1 in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D1. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D1. That is, **duplicate destinations are NOT allowed within the same instruction**.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S1 and/or S2 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**. Note that S1 and S2 may specify the same register.

Class I Example:

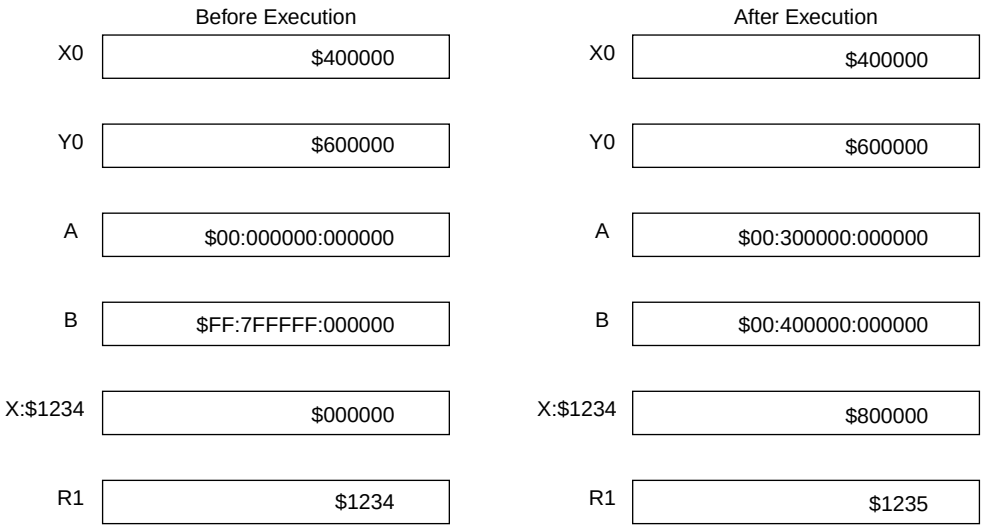
```
        :
CMPM Y0,A A,X:$1234 A,Y0        ;compare A,Y0 mag., save A, update Y0
        :
```



Explanation of the Class I Example: Prior to execution, the 56-bit A accumulator contains the value $00:800000:000000, the 24-bit X memory location X:$1234 contains the value $000000, and the 24-bit Y0 register contains the value $000000. The execution of the parallel move portion of the instruction, A,X:$1234 A,Y0, moves the 24-bit limited positive saturation constant $7FFFFF into both the X:$1234 memory location and the Y0 register since the signed portion of the A accumulator was in use.

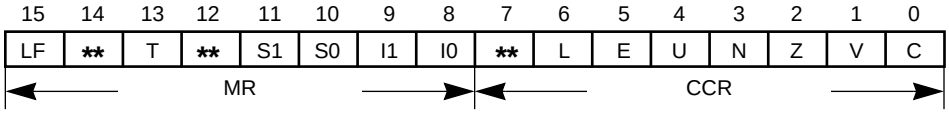
Class II Example:

```
        :
MAC X0,Y0,A   B,X:(R1)+   X0,B   ;multiply X0 and Y0 and accumulate in A
        :                        ;move B to X memory location pointed to
                                 ;by R1 and postincrement R1
                                 ;move X0 to B
```



Explanation of the Class II Example: Prior to execution, the 24-bit registers X0 and Y0 contain $400000 and $600000, respectively. The 56-bit accumulators A and B contain the values $00:000000:000000 and $FF:7FFFFF:000000, respectively. The 24-bit X memory location X:$1234 contains the value $000000, and the 16-bit R1 register contains the value $1234. Execution of the parallel move portion of the instruction (B,X:(R1)+X0,B) moves the 24-bit limited value of B ($800000) into the X:$1234 memory location and the X0 register ($400000) into accumulator B1 ($400000), sign extends B1 into B2 ($00), and zero fills B0 ($000000). It also increments R1 to $1235.

Condition Codes:

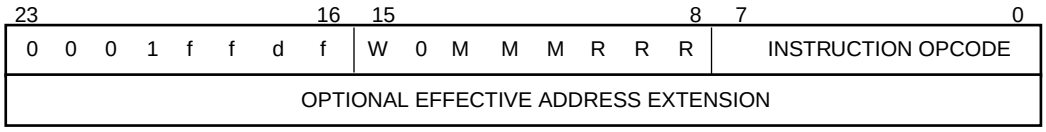


L — Set if data limiting has occurred during parallel move

Class I Instruction Format:

(. . . . .) X:ea,D1 S2,D2
(. . . . .) S1,X:ea S2, D2
(. . . . .) #xxxxxx, S2,D2

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate data	1	1	0	1	0	0

where ‘‘rrr’’ refers to an address register R0–R7

S1,D1	f	f	S1 S/L	D1 Sign Ext	D1 Zero	S2	d	S2 S/L	D2	f	D2 Sign Ext	D2 Zero
X0	0	0	no	no	no	A	0	yes	Y0	0	no	no
X1	0	0	no	no	no	B	1	yes	Y1	1	no	no
A	0	1	yes	A2	A0							
B	0	1	yes	B2	B0							

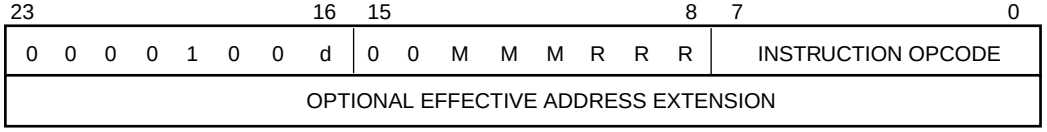
Timing: mv oscillator clock cycles

Memory: mv program words

Class II Instruction Format:

(. . . . .) A→X:ea X0→A
(. . . . .) B→X:ea X0→B

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	MMMRRR
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r

where ‘‘rrr’’ refers to an address register R0–R7

S D	S S/L	D Sign Ext	D Zero
X0	no	N/A	N/A
Y0	no	N/A	N/A
A	yes	A2	A0
B	yes	B2	B0

d	MOVE Opcode	
0	A→X:ea	X0→A
1	B→X:ea	X0→B

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:	Assembler Syntax:
(. . . . .); Y:ea→D	(. . . . .) Y:ea,D
(. . . . .); Y:aa→D	(. . . . .) Y:aa,D
(. . . . .); S→Y:ea	(. . . . .) S,Y:ea
(. . . . .); S→Y:aa	(. . . . .) S,Y:aa
(. . . . .); #xxxxxx→D	(. . . . .) #xxxxxx,D

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move the specified word operand from/to Y memory. All memory addressing modes, including absolute addressing and 24-bit immediate data, may be used. Absolute short addressing may also be used.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D. That is, **duplicate destinations are NOT allowed within the same instruction**.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S 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**.

When a 24-bit source operand is moved into a 16-bit destination register, the 16 LS bits of the 12-bit source operand are stored in the 16-bit destination register. When a 16-bit source operand is moved into a 24-bit destination register, the 16 LS bits of the destination register are loaded with the contents of the 16-bit source operand, and the eight MS bits of the 24-bit destination register are zeroed.

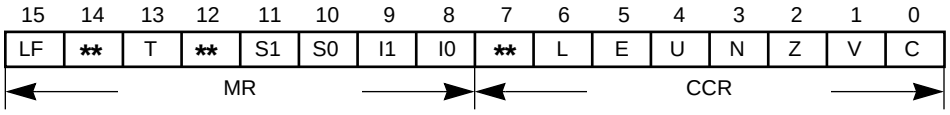
Note: This parallel data move is considered to be a move-type instruction. Due to pipelining, if an address register (R or N) is changed using a move-type instruction, the new contents of the destination address register will not be available for use during the following instruction (i.e., there is a single instruction cycle pipeline delay).

Example:

```
        :
EOR X0,B    #$123456,A      ;exclusive OR X0 and B, update A accumulator
        :
```

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $FF:FFFFFF:FFFFFF. The execution of the parallel move portion of the instruction, #$123456,A, moves the 24-bit immediate value $123456 into the 24-bit A1 register, then sign extends that value into the A2 portion of the accumulator, and zeros the lower 24-bit A0 portion of the accumulator.

Condition Codes:



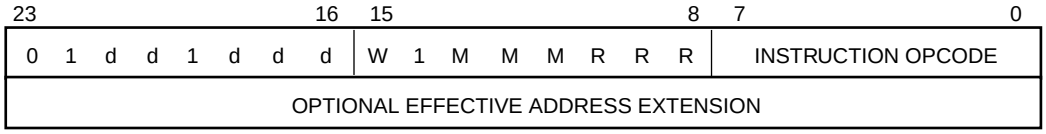
L — Set if data limiting has occurred during parallel move

Note: The MOVE A,Y:ea operation will result in a 24-bit positive or negative saturation constant being stored in the specified 24-bit Y memory location if the signed integer portion of the A accumulator is in use.

Instruction Format:

(. . . . .) Y:ea,D
(. . . . .) S,Y:ea
(. . . . .) #xxxxxx,D

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate data	1	1	0	1	0	0

where ‘‘rrr’’ refers to an address register R0–R7

S,D	d	d	d	d	d	S S/L	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where “rrr”=Rn number
where “nnn”=Nn number

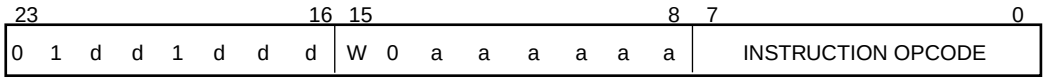
Timing: mv oscillator clock cycles

Memory: mv program words

Instruction Format:

(. . . . .) Y:aa,D
(. . . . .) S,Y:aa

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Register	**W**	**Absolute Short Address aaaaaa**
Read S	0	000000
Write D	1	•
		111111

S,D	**d**	**d**	**d**	**d**	**d**	**S S/L**	**D Sign Ext**	**D Zero**
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where “rrr”=Rn number
where “nnn”=Nn number

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:

Class I

(. . . . .); S1→D1; Y:ea→D2

(. . . . .); S1→D1; S2→Y:ea

(. . . . .); S1→D1; #xxxxxx→D2

Class II

(. . . . .); Y0 →A; A→Y:ea

(. . . . .); Y0→B; B→Y:ea

Assembler Syntax:

Class I

(. . . . .) S1,D1 Y:ea,D2

(. . . . .) S1,D1 S2,Y:ea

(. . . . .) S1,D1 #xxxxxx,D2

Class II

(. . . . .) Y0,A A,Y:ea

(. . . . .) Y0,B B,Y:ea

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Class I: Move a one-word operand from an accumulator (S1) to an input register (D1) and move another word operand from/to Y memory. All memory addressing modes, including absolute addressing and 24-bit immediate data, may be used. The register to register move (S1,D1) allows a data ALU accumulator to be moved to a data ALU input register for use as a data ALU operand in the following instruction.

Class II: Move one-word operand from a data ALU accumulator to Y memory and one-word operand from data ALU register Y0 to a data ALU accumulator. One effective address is specified. All memory addressing modes, excluding long absolute addressing and long immediate data, may be used. Class II move operations have been added to the R:Y parallel move (and a similar feature has been added to the X:R parallel move) as an added feature available in the first quarter of 1989.

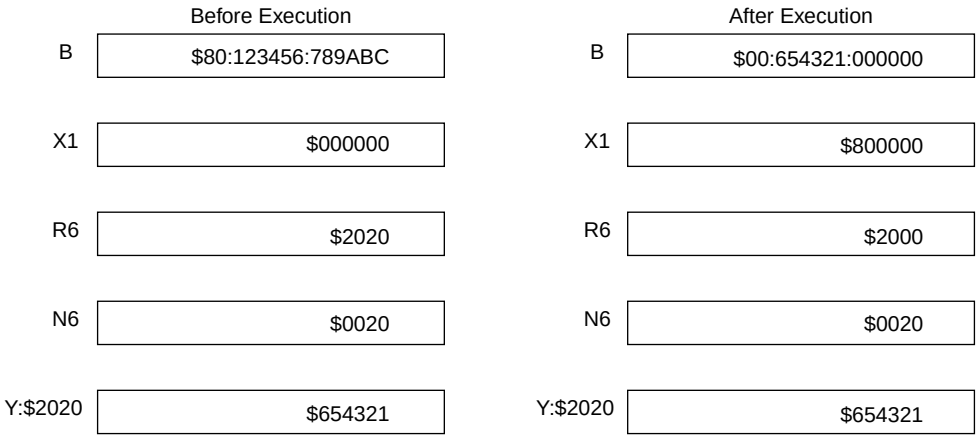
For both Class I and Class II R:Y parallel data moves, if the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D2 in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D2. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D2. That is, duplicate destinations are NOT allowed within the same instruction.

If the opcode-operand portion of the instruction specifies a given source or destination

register, that same register or portion of that register may be used as a source S1 and/or S2 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**. Note that S1 and S2 may specify the same register.

Class I Example:

```
        :
ADDL  B,A    B,X1   Y:(R6)–N6,B        ;2*A+B → A, update X1,B and R6
        :
```



Explanation of the Class I Example: Prior to execution, the 56-bit B accumulator contains the value $80:123456:789ABC, the 24-bit X1 register contains the value $000000, the 16-bit R6 address register contains the value $2020, the 16-bit N6 address offset register contains the value $0020 and the 24-bit Y memory location Y:$2020 contains the value $654321. The execution of the parallel move portion of the instruction, B,X1 Y:(R6)–N6,B, moves the 24-bit limited negative saturation constant $800000 into the X1 register since the signed integer portion of the B accumulator was in use, uses the value in the 16-bit R6 address register to move the 24-bit value in the Y memory location Y:$2020 into the 56-bit B accumulator with automatic sign extension of the upper portion of the accumulator (B2) and automatic zeroing of the lower portion of the accumulator (B0), and finally uses the contents of the 16-bit N6 address offset register to update the value in the 16-bit R6 address register. The contents of the N6 address offset register are not affected.

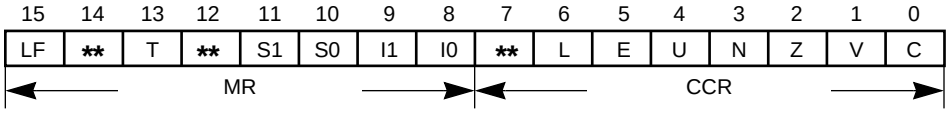
Class II Example:

```
        :
MAC X0,Y0,A    Y0,B     B,Y:(R1)+     ;multiply X0 and Y0 and accumulate in A
        :                             ;move B to Y memory location pointed to
                                      ;by R1 and postincrement R1
                                      ;move Y0 to B
```

	Before Execution	After Execution
X0	$400000	$400000
Y0	$600000	$600000
A	$00:000000:000000	$00:300000:000000
B	$00:800000:000000	$00:600000:000000
Y:$1234	$000000	$7FFFFF
R1	$1234	$1235

Explanation of the Class II Example: Prior to execution, the 24-bit registers, X0 and Y0, contain $400000 and $600000, respectively. The 56-bit accumulators A and B contain the values $00:000000:000000 and $00:800000:000000 (+1.0000), respectively. The 24-bit Y memory location Y:$1234 contains the value $000000, and the 16-bit R1 register contains the value $1234. Execution of the parallel move portion of the instruction (Y0,B B,Y:(R1)+) moves the Y0 register ($600000) into accumulator B1 ($600000), sign extends B1 into B2 ($00), and zero fills B0 ($000000). It also moves the 24-bit limited value of B ($7FFFFF) into the Y:$1234 memory location and increments R1 to $1235.

Condition Codes:

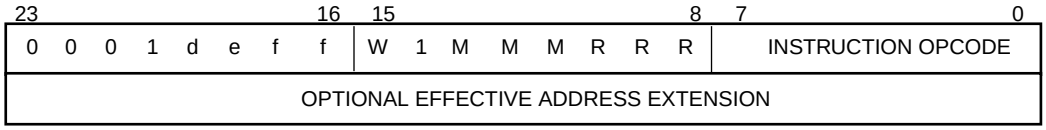


L — Set if data limiting has occurred during parallel move

Class I Instruction Format:

(. . . . .) S1,D1 Y:ea,D2
(. . . . .) S1,D1 S2,Y:ea
(. . . . .) S1,D1 #xxxxxx,D2

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S2	0	(Rn)-Nn	0	0	0	r	r	r
Write D2	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate data	1	1	0	1	0	0

where ‘‘rrr’’ refers to an address register R0–R7

S1	d	S1 S/L	D1	e	D1 Sign Ext	D1 Zero
A	0	yes	X0	0	no	no
B	0	yes	X1	1	no	no

S2,D2	f	f	S2 S/L	D2 Sign Ext	D2 Zero
Y0	0	0	no	no	no
Y1	0	1	no	no	no
A	1	0	yes	A2	A0
B	1	1	yes	B2	B0

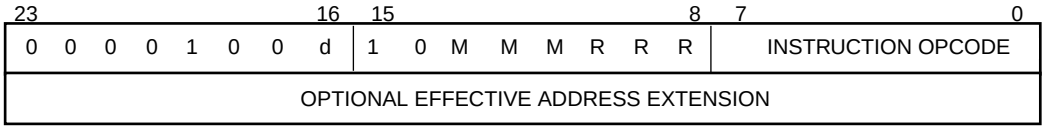
Timing: mv oscillator clock cycles

Memory: mv program words

Class II Instruction Format:

(. . . . .) Y0 → A A → Y:ea
(. . . . .) Y0 → B B → Y:ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	MMMRRR
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r

where “rrr” refers to an address register R0–R7

S, D	SRC S/L	DEST Sign Ext	DEST Zero
X0	no	N/A	N/A
Y0	no	N/A	N/A
A	yes	A2	A0
B	yes	B2	B0

d	MOVE Opcode	
0	Y0 → A	A → Y:ea
1	Y0 → B	B → Y:ea

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:	Assembler Syntax:
(. . . . .); X:ea → D1; Y:ea → D2	(. . . . .) L:ea,D
(. . . . .); X:aa → D1; Y:aa → D2	(. . . . .) L:aa,D
(. . . . .); S1 → X:ea; S2 → Y:ea	(. . . . .) S,L:ea
(. . . . .); S1 → X:aa; S2 → Y:aa	(. . . . .) S,L:aa

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move one 48-bit long-word operand from/to X and Y memory. Two data ALU registers are concatenated to form the 48-bit long-word operand. This allows efficient moving of both double-precision (high:low) and complex (real:imaginary) data from/to one effective address in L (X:Y) memory. The same effective address is used for both the X and Y memory spaces; thus, only one effective address is required. Note that the A, B, A10, and B10 operands reference a single 48-bit signed (double-precision) quantity while the X, Y, AB, and BA operands reference two separate (i.e., real and imaginary) 24-bit signed quantities. All memory alterable addressing modes may be used. Absolute short addressing may also be used.

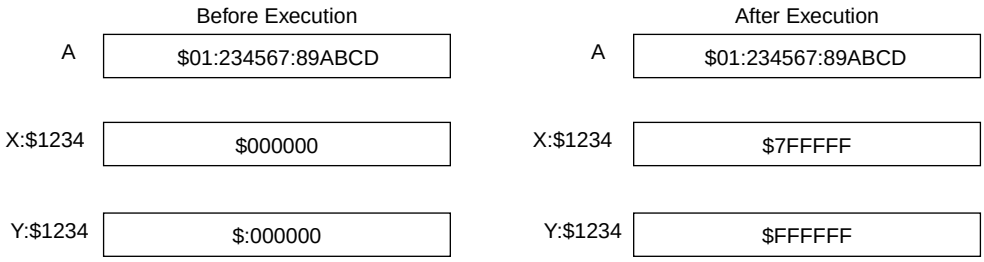
If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A, A10, AB, or BA as destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B, B10, AB, or BA as its destination D. That is, duplicate destinations are NOT allowed within the same instruction.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S 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.

Note: The operands A10, B10, X, Y, AB, and BA may be used only for a 48-bit long memory move as previously described. These operands may not be used in any other type of instruction or parallel move.

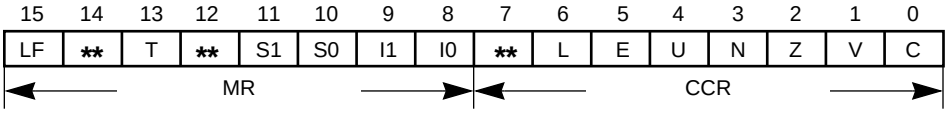
Example:

```
        :
CMP Y0,B    A,L:$1234       ;compare Y0 and B, save 48-bit A1:A0 value
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $01:234567:89ABCD, the 24-bit X memory location X:$1234 contains the value $000000, and the 24-bit Y memory location Y:$1234 contains the value $000000. The execution of the parallel move portion of the instruction, A,L:$1234, moves the 48-bit limited positive saturation constant $7FFFFF:FFFFFF into the specified long memory location by moving the MS 24 bits of the 48-bit limited positive saturation constant ($7FFFFF) into the 24-bit X memory location X:$1234 and by moving the LS 24 bits of the 48-bit limited positive saturation constant ($FFFFFF) into the 24-bit Y memory location Y:$1234 since the signed integer portion of the A accumulator was in use.

Condition Codes:



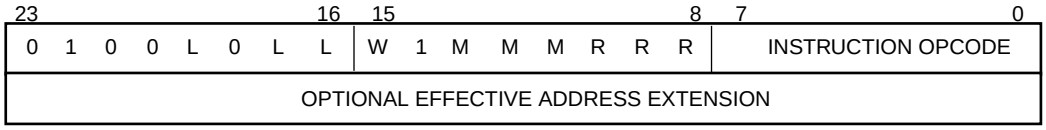
L — Set if data limiting has occurred during parallel move

Note: The MOVE A,L:ea operation will result in a 48-bit positive or negative saturation constant being stored in the specified 24-bit X and Y memory locations if the signed integer portion of the A accumulator is in use. The MOVE AB,L:ea operation will result in either one or two 24-bit positive and/or negative saturation constant(s) being stored in the specified 24-bit X and/or Y memory location(s) if the signed integer portion of the A and/or B accumulator(s) is in use.

Instruction Format:

(. . . . .) L:ea,D
(. . . . .) S,L:ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	0	r	r	r
		Absolute address	1	1	0	0	0	0

where ‘‘rrr’’ refers to an address register R0–R7

S	S1	S2	S S/L	D	D1	D2	D Sign Ext	D Zero	L	L	L
A10	A1	A0	no	A10	A1	A0	no	no	0	0	0
B10	B1	B0	no	B10	B1	B0	no	no	0	0	1
X	X1	X0	no	X	X1	X0	no	no	0	1	0
Y	Y1	Y0	no	Y	Y1	Y0	no	no	0	1	1
A	A1	A0	yes	A	A1	A0	A2	no	1	0	0
B	B1	B0	yes	B	B1	B0	B2	no	1	0	1
AB	A	B	yes	AB	A	B	A2,B2	A0,B0	1	1	0
BA	B	A	yes	BA	B	A	B2,A2	B0,A0	1	1	1

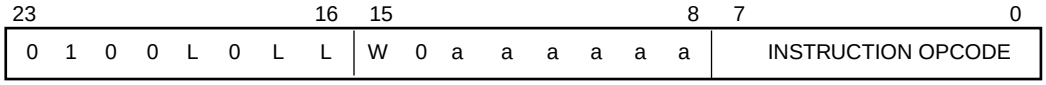
Timing: mv oscillator clock cycles

Memory: mv program words

Instruction Format:

(. . . . .) L:aa,D
(. . . . .) S,L:aa

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Register W		Absolute Short Address aaaaaa
Read S	0	000000
Write D	1	•
		•
		111111

S	S1	S2	S S/L	D	D1	D2	D Sign Ext	D Zero	L	L	L
A10	A1	A0	no	A10	A1	A0	no	no	0	0	0
B10	B1	B0	no	B10	B1	B0	no	no	0	0	1
X	X1	X0	no	X	X1	X0	no	no	0	1	0
Y	Y1	Y0	no	Y	Y1	Y0	no	no	0	1	1
A	A1	A0	yes	A	A1	A0	A2	no	1	0	0
B	B1	B0	yes	B	B1	B0	B2	no	1	0	1
AB	A	B	yes	AB	A	B	A2,B2	A0,B0	1	1	0
BA	B	A	yes	BA	B	A	B2,A2	B0,A0	1	1	1

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:	**Assembler Syntax:**
(. . . . .); X:<eax> → D1; Y:<eay> → D2	(. . . . .) X:<eax>,D1 Y:<eay>,D2
(. . . . .); X:<eax> → D1; S2 → Y:<eay>	(. . . . .) X:<eax>,D1 S2,Y:<eay>
(. . . . .); S1 → X:<eax>; Y:<eay> → D2	(. . . . .) S1,X:<eax> Y:<eay>,D2
(. . . . .); S1 → X:<eax>; S2 → Y:<eay>	(. . . . .) S1,X:<eax> S2,Y:<eay>

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

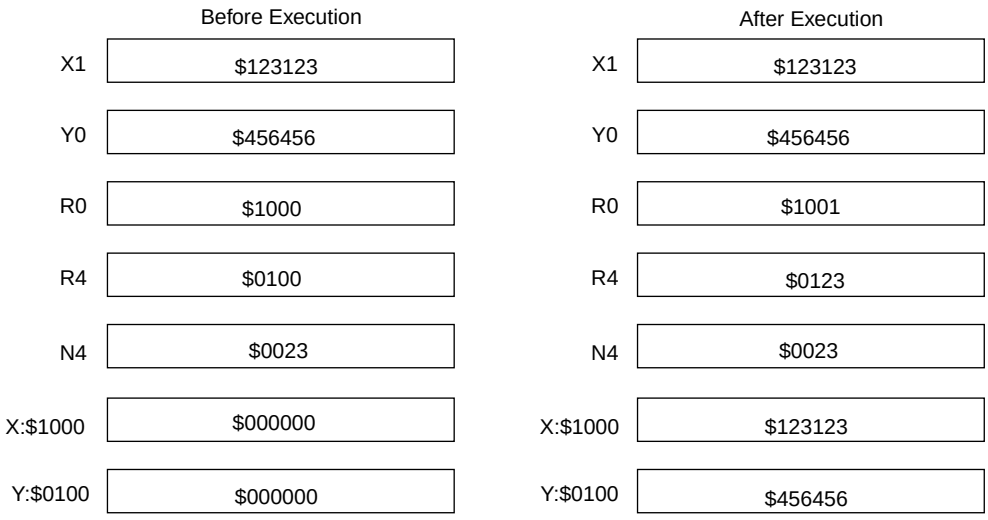
Description: Move a one-word operand from/to X memory and move another word operand from/to Y memory. Note that two independent effective addresses are specified (<eax> and <eay>) where one of the effective addresses uses the lower bank of address registers (R0–R3) while the other effective address uses the upper bank of address registers (R4–R7). All parallel addressing modes may be used.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D1 or D2 in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A as its destination D1 or D2. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B as its destination D1 or D2. That is, **duplicate destinations are NOT allowed within the same instruction**. D1 and D2 may not specify the same register.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S1 and/or S2 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**. Note that S1 and S2 may specify the same register.

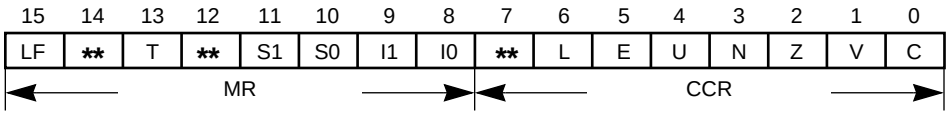
Example:

```
        :
MPYR X1,Y0,A    X1,X:(R0)+      Y0,Y:(R4)+N4      ;X1*Y0 → A,save X1 and Y0
        :
```



Explanation of Example: Prior to execution, the 24-bit X1 register contains the value $123123, the 24-bit Y0 register contains the value $456456, the 16-bit R0 address register contains the value $1000, the 16-bit R4 address register contains the value $0100, the 16-bit N4 address offset register contains the value $0023, the 24-bit X memory location X:$1000 contains the value $000000, and the 24-bit Y memory location Y:$0100 contains the value $000000. The execution of the parallel move portion of the instruction, X1,X:(R0)+ Y0,Y:(R4)+N4, moves the 24-bit value in the X1 register into the 24-bit X memory location X:$1000 using the 16-bit R0 address register, moves the 24-bit value in the Y0 register into the 24-bit Y memory location Y:$0100 using the 16-bit R4 address register, updates the 16-bit value in the R0 address register, and updates the 16-bit R4 address register using the 16-bit N4 address offset register. The contents of the N4 address offset register are not affected.

Condition Codes:



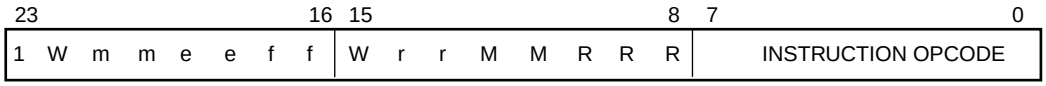
L — Set if data limiting has occurred during parallel move

Note: The MOVE A,X:<eax> B,Y:<eay> operation will result in one or two 24-bit positive and/or negative saturation constant(s) being stored in the specified 24-bit X and/or Y memory location(s) if the signed integer portion of the A and/or B accumulator(s) is in use.

Instruction Format:

(. . . . .) X:<eax>,D1 Y:<eay>,D2
(. . . . .) X:<eax>,D1 S2,Y:<eay>
(. . . . .) S1,X:<eax> Y:<eay>,D2
(. . . . .) S1,X:<eax> S2,Y:<eay>

Opcode:



Instruction Fields:

X:<eax>=6-bit X Effective Address=WMMRRR (R0–R3 or R4–R7)
X:<eay>=5-bit Y Effective Address=wmmrr (R4–R7 or R0–R3)

X Effective Addressing Mode	**M**	**M**	**R**	**R**	**R**
(Rn)+Nn	0	1	s	s	s
(Rn)-	1	0	s	s	s
(Rn)+	1	1	s	s	s
(Rn)	0	0	s	s	s

where “sss” refers to an address register R0–R7

Register	**W**	**S1, D1**	**e**	**e**	**S1 S/L**	**D1 Sign Ext**	**D1 Zero**	**Y Effective Addressing Mode**	**m**	**m**	**r**	**r**
Read S1	0	X0	0	0	no	no	no	(Rn) +Nn	0	1	t	t
Write D1	1	X1	0	1	no	no	no	(Rn) -	1	0	t	t
		A	1	0	yes	A2	A0	(Rn) +	1	1	t	t
		B	1	1	yes	B2	B0	(Rn)	0	0	t	t

where “tt” refers to an address register R4 - R7 or R0 - R3 which is in the **opposite** address register bank from the one used in the X effective address, previously described

Register	**W**	**S2, D2**	**f**	**f**	**S2 S/L**	**D2 Sign Ext**	**D2 Zero**
Read S2	0	Y0	0	0	no	no	no
Write D2	1	Y1	0	1	no	no	no
		A	1	0	yes	A2	A0
		B	1	1	yes	B2	B0

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:	Assembler Syntax:	
X:ea→D1	MOVE(C)	X:ea,D1
X:aa→D1	MOVE(C)	X:aa,D1
S1→X:ea	MOVE(C)	S1,X:ea
S1→X:aa	MOVE(C)	S1,X:aa
Y:ea→D1	MOVE(C)	Y:ea,D1
Y:aa→D1	MOVE(C)	Y:aa,D1
S1→Y:ea	MOVE(C)	S1,Y:ea
S1→Y:aa	MOVE(C)	S1,Y:aa
S1→D2	MOVE(C)	S1,D2
S2→D1	MOVE(C)	S2,D1
#xxxx→D1	MOVE(C)	#xxxx,D1
#xx→D1	MOVE(C)	#xx,D1

Description: Move the contents of the specified source **control register** S1 or S2 to the specified destination ormove the specified source to the specified destination **control register** D1 or D2. The control registers S1 and D1 are a subset of the S2 and D2 register set and consist of the address ALU modifier registers and theprogram controller registes. These registers may be moved to or from any other register or memory space. Al memory addressing modes, as well as an immediate short addressing mode, may be usedl

If the system stack register SSH is specified as a source operand, the system stack pointer (SP) is postdecremented by 1 after SSH has been read. If the system stack register SSH is specified as a destination operand, the system stack pointer (SP) is preincremented by 1 before SSH is written . This allows the system stack to be efficiently extended using software stack pointer operations.

When a 56-bit accumulator (A or B) is specified as a **source** operand, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 int he system status register (SR). If the data our of the shifter indicates that the accumulator extension register is in use and the data is to be moved into a 24-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constand to mini-

mize truncation error. If the data is to be moved into a 16-bit destination and the accumulator extension register is in use, the value is limited to a maximum positive or negative saturation constand whose LS 16 bits are then stored in the 16-bit destination register. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-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 condition code register is latched.

When a 56-bit accumulator (A or B) is specified as a **destination** operand, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 23) and appending the source operand with 24 LS zeros. Whenever a 16-bit source operand is to be moved into a 24-bit destination, the 16-bit value is stored in the LS 16 bits of the 24-bit destination, and the MS 8 bits of that destination are zeroed. Similarly, whenever a 16-bit source operand is to be moved into a 56-bit accumulator, the 16-bit value is moved into the LS 16 bits of the MSP portion of the accumulator (A1 or B1), the MS 8 bits of the MSP portion of that accumulator are zeroed, and the resulting 24-bit value is extended to 56 bits by sign extending the MS bit and appending the result with 24 LS zeros. Note that for 24-bit source operands both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1).

Note: Due to pipelining, if an address register (R, N, or M) is changed using a move-type instruction, the new contents of the destination address register will not be available for use during the following instruction (i.e., there is a single instruction cycle pipeline delay).

Restrictions: The following restrictions represent very unusual operations which probably would never be used but are listed only for completeness.

A MOVEC instruction used **within a DO loop** which specifies SSH as the **source** operand or **LA, LC, SR, SP, SSH, or SSL** as the **destination** operand cannot begin at the address LA – 2, LA – 1, or LA within that DO loop.

A MOVEC instruction which specifies **SSH** as the **source** operand or **LA, LC, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** a DO instruction.

A MOVEC instruction which specifies SSH as the **source** operand or **LA, LC, SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an ENDDO instruction.

A MOVEC instruction which specifies SSH as the **source** operand or **SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTI instruction.

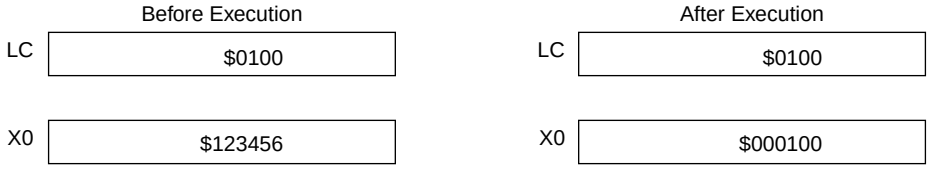
A MOVEC instruction which specifies SSH as the **source** operand or **SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTS instruction.

A MOVEC instruction which specified **SP** as the **destination** operand cannot be used **immediately before** a MOVEC, MOVEM, or MOVEP instruction which specifies **SSH or SSL** as the **source** operand.

A MOVEC SSH, SSH instruction is **illegal** and cannot be used.

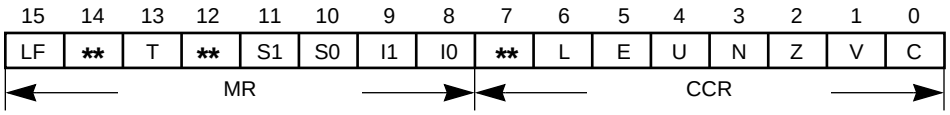
Example:

```
        :
MOVEC LC,X0          ;move LC into X0
        :
```



Explanation of Example: Prior to execution, the 16-bit loop counter (LC) register contains the value $0100, and the 24-bit X0 register contains the value $123456. The execution of the MOVEC LC,X0 instruction moves the contents of the 16-bit LC register into the 16 LS bits of the 24-bit X0 regiser and zeros the 8 MS bits of the X0 register.

Condition Codes:



For D1 or D2=SR operand:

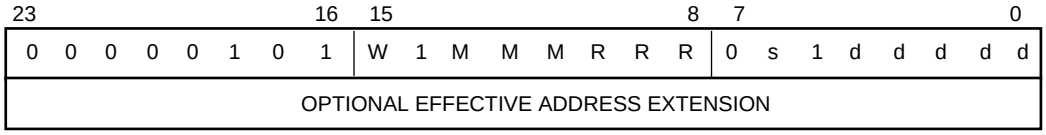
L — **Set according to bit 6 of the source operand**
E — **Set according to bit 5 of the source operand**
U — **Set according to bit 4 of the source operand**
N — **Set according to bit 3 of the source operand**
Z — **Set according to bit 2 of the source operand**

V — **Set according to bit 1 of the source operand**
C — **Set according to bit 0 of the source operand**

For D1 and D2≠SR operand:
L — Set if data limiting has occurred during the move
Instruction Format:

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

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate Data	1	1	0	1	0	0

where ‘‘rrr’’ refers to an address register R0–R7

Memory Space	s	S1, D1	d d d d d
X Memory	0	M0–M7	0 0 n n n
Y Memory	1	SR	1 1 0 0 1
		OMR	1 1 0 1 0
		SP	1 1 0 1 1
		SSH	1 1 1 0 0
		SSL	1 1 1 0 1
		LA	1 1 1 1 0
		LC	1 1 1 1 1

where “nnn” = Mn number (M0–M7)

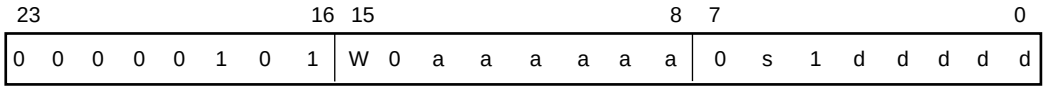
Timing: 2+mvc oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Register	W	Absolute Short Address aaaaaa
Read S	0	000000
Write D	1	•
		•
		111111

Memory Space	s	S1, D1	d d d d d
X Memory	0	M0–M7	0 0 n n n
Y Memory	1	SR	1 1 0 0 1
		OMR	1 1 0 1 0
		SP	1 1 0 1 1
		SSH	1 1 1 0 0
		SSL	1 1 1 0 1
		LA	1 1 1 1 0
		LC	1 1 1 1 1

where “nnn” = Mn number (M0–M7)

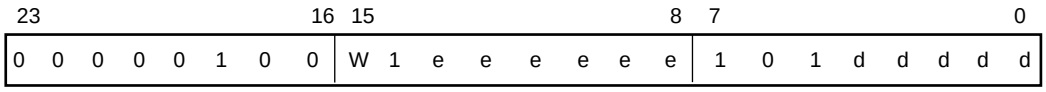
Timing: 2+mvc oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

Register	**W**
Read S1	0
Write D1	1

Memory Space	**s**
X Memory	0
Y Memory	1

S1, D1	**d d d d d**
M0–M7	0 0 n n n
SR	1 1 0 0 1
OMR	1 1 0 1 0
SP	1 1 0 1 1
SSH	1 1 1 0 0
SSL	1 1 1 0 1
LA	1 1 1 1 0
LC	1 1 1 1 1

where “nnn” = Mn number (M0–M7)

S2, D2	**e e e e e e**	**S2 S/L**	**D2 Sign Ext**	**D2 Zero**
X0	0 0 0 1 0 0	no	no	no
X1	0 0 0 1 0 1	no	no	no
Y0	0 0 0 1 1 0	no	no	no
Y1	0 0 0 1 1 1	no	no	no
A0	0 0 1 0 0 0	no	no	no
B0	0 0 1 0 0 1	no	no	no
A2	0 0 1 0 1 0	no	no	no
B2	0 0 1 0 1 1	no	no	no
A1	0 0 1 1 0 0	no	no	no
B1	0 0 1 1 0 1	no	no	no
A	0 0 1 1 1 0	yes	A2	A0
B	0 0 1 1 1 1	yes	B2	B0

S2, D2	**e e e e e e**
R0 - R7	0 1 0 n n n
N0 - N7	0 1 1 n n n
M0 - M7	1 0 0 n n n
SR	1 1 1 0 0 1
OMR	1 1 1 0 1 0
SP	1 1 1 0 1 1
SSH	1 1 1 1 0 0
SSL	1 1 1 1 0 1
LA	1 1 1 1 1 0
LC	1 1 1 1 1 1

where “nnn” = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

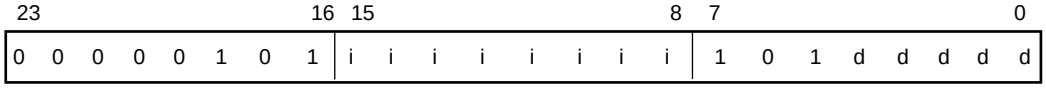
Timing: 2+mvc oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

MOVE(C) #xx,D1

Opcode:



Instruction Fields:

#xx=8-bit Immediate Short Data=i i i i i i i i

D1	**d d d d d**
M0–M7	0 0 n n n
SR	1 1 0 0 1
OMR	1 1 0 1 0
SP	1 1 0 1 1
SSH	1 1 1 0 0
SSL	1 1 1 0 1
LA	1 1 1 1 0
LC	1 1 1 1 1

where “nnn” = Mn number (M0–M7)

Timing: 2+mvc oscillator clock cycles

Memory: 1+ea program words

Operation:	Assembler Syntax:	
S→P:ea	MOVE(M)	S,P:ea
S→P:aa	MOVE(M)	S,P:aa
P:ea→D	MOVE(M)	P:ea,D
P:aa→D	MOVE(M)	P:aa,D

Description: Move the specified operand from/to the specified **program** (P) **memory location**. This is a powerful move instruction in that the source and destination registers S and D may be **any** register. All memory alterable addressing modes may be used as well as the absolute short addressing mode.

If the system stack register SSH is specified as a source operand, the system stadk pointer (SP) is postdecremented by 1 after SSH has been read. If the system stack register SSH is specified as a destination operand, the system stack pointer (SP) is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

When a 56-bit accumulator (A or B) is specified as a **source** operand S, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). 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 24-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. If a 24-bit source operand is to be moved into a 16-bit destination register D, the 8 MS bits of the 24-bit source operand are discarded, and the 16 LS bits are stored in the 16-bit destination register. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-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 condition code register is latched.

When a 56-bit accumulator (A) is specified as a **destination** operand D, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 24) and appending the source operand with 24 LS zeros. Whenever a 16-bit source operand S is to be moved into a 24-bit destination, the 16-bit source is loaded into the LS 16 bits of the destination operand, and the remaining 8 MS bits of the destination are zeroed. Note that for 24-bit source operands, both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1).

Note: Due to pipelining, if an adderss register (R, N, or M) is changed using a move-type instruction, the new contents of the destination address register will not be avaiolable for use during the following instruction (i.e., there is a single instruction cycle pipeline delay).

Restrictions: The following restrictions represent very unusual operations, which probably would never be used but are listed only for completeness.

A MOVEM instruction **used within a DO loop** which specifies **SSH** as the **source** operand or **LA, LC, SR, SP, SSH, or SSL** as the **destination** operand cannot begin at the address LA–2, LA–1, or LA within that DO loop.

A MOVEM instruction which specifies **SSH** as the **source** operand or **LA, LC, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** a DO instruction.

A MOVEM instruction which specifies **SSH** as the **source** operand or **LA, LC, SR, SSH, SL, or SP** as the **destination** operand cannot be used **immediately before** an ENDDO instruction.

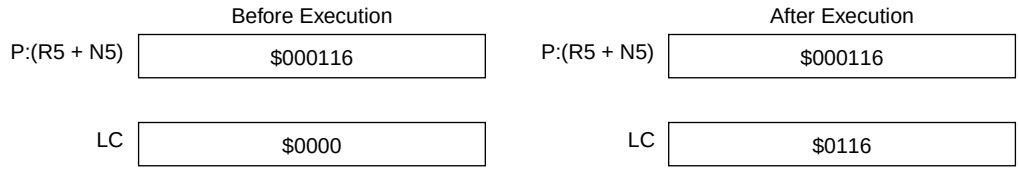
A MOVEM instruction which specifies **SSH** as the **source** operand or **SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTI instruction.

A MOVEM instruction which specifies **SSH** as the **source** operand or **SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTS instruction.

A MOVEM instruction which specifies **SP** as the **destination** operand cannot be used **immediately before** a MOVEC, MOVEM, or MOVEP instruction which specifies **SSH or SSL** as the **source** operand.

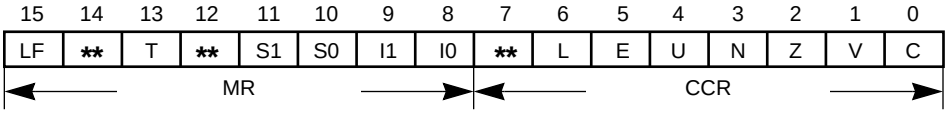
Example:

```
        :
MOVEM P:(R5+N5), LC          :move P:(R5+N5) into the loop counter (LC)
        :
```



Explanation of Example: Prior to execution, the 16-bit loop counter (LC) register contains the value $0000, and the 24-bit program (P) memory location P:(R5+N5) contains the value $000116. The execution of the MOVEM P:(R5+N5), LC instruction moves the 16 LS bits of the 24-bit program (P) memory location P:(R5+N5) into the 16-bit LC register.

Condition Codes:



For D=SR 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**

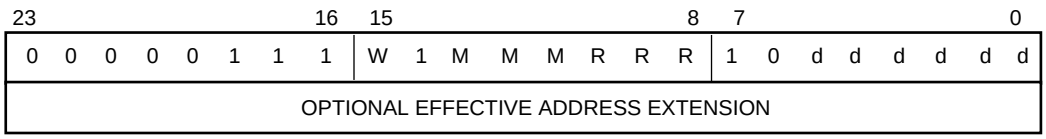
For D≠SR operand:

L — Set if data limiting has occurred during the move

Instruction Format:

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

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0

where ‘‘rrr’’ refers to an address register R0–R7

S,D	d d d d d d	S S/L	D Sign Ext	D Zero	S,D	d d d d d d
X0	0 0 0 1 0 0	no	no	no	R0 - R7	0 1 0 n n n
X1	0 0 0 1 0 1	no	no	no	N0 - N7	0 1 1 n n n
Y0	0 0 0 1 1 0	no	no	no	M0 - M7	1 0 0 n n n
Y1	0 0 0 1 1 1	no	no	no	SR	1 1 1 0 0 1
A0	0 0 1 0 0 0	no	no	no	OMR	1 1 1 0 1 0
B0	0 0 1 0 0 1	no	no	no	SP	1 1 1 0 1 1
A2	0 0 1 0 1 0	no	no	no	SSH	1 1 1 1 0 0
B2	0 0 1 0 1 1	no	no	no	SSL	1 1 1 1 0 1
A1	0 0 1 1 0 0	no	no	no	LA	1 1 1 1 1 0
B1	0 0 1 1 0 1	no	no	no	LC	1 1 1 1 1 1
A	0 0 1 1 1 0	yes	A2	A0		
B	0 0 1 1 1 1	yes	B2	B0		

where “nnn” = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

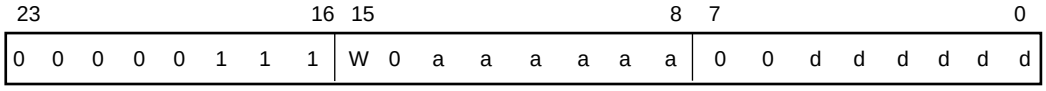
Timing: 2+mvm oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

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

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaa

Register	W	Absolute Short Address aaaaaa
Read S	0	000000
Write D	1	•
		•
		111111

S,D	d d d d d d	S S/L	D Sign Ext	D Zero
X0	0 0 0 1 0 0	no	no	no
X1	0 0 0 1 0 1	no	no	no
Y0	0 0 0 1 1 0	no	no	no
Y1	0 0 0 1 1 1	no	no	no
A0	0 0 1 0 0 0	no	no	no
B0	0 0 1 0 0 1	no	no	no
A2	0 0 1 0 1 0	no	no	no
B2	0 0 1 0 1 1	no	no	no
A1	0 0 1 1 0 0	no	no	no
B1	0 0 1 1 0 1	no	no	no
A	0 0 1 1 1 0	yes	A2	A0
B	0 0 1 1 1 1	yes	B2	B0

S,D	d d d d d d
R0 - R7	0 1 0 n n n
N0 - N7	0 1 1 n n n
M0 - M7	1 0 0 n n n
SR	1 1 1 0 0 1
OMR	1 1 1 0 1 0
SP	1 1 1 0 1 1
SSH	1 1 1 1 0 0
SSL	1 1 1 1 0 1
LA	1 1 1 1 1 0
LC	1 1 1 1 1 1

where “nnn” = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

Timing: 2+mvm oscillator clock cycles

Memory: 1+ea program words

Operation:	Assembler Syntax:	
X:pp → D	MOVEP	X:pp,D
X:pp → X:ea	MOVEP	X:pp,X:ea
X:pp → Y:ea	MOVEP	X:pp,Y:ea
X:pp → P:ea	MOVEP	X:pp,P:ea
S → X:pp	MOVEP	S,X:pp
#xxxxxx → X:pp	MOVEP	#xxxxxx,X:pp
X:ea → X:pp	MOVEP	X:ea,X:pp
Y:ea → X:pp	MOVEP	Y:ea,X:pp
P:ea → X:pp	MOVEP	P:ea,X:pp
Y:pp → D	MOVEP	Y:pp,D
Y:pp → X:ea	MOVEP	Y:pp,X:ea
Y:pp → Y:ea	MOVEP	Y:pp,Y:ea
Y:pp → P:ea	MOVEP	Y:pp,P:ea
S → Y:pp	MOVEP	S,Y:pp
#xxxxxx → Y:pp	MOVEP	#xxxxxx,Y:pp
X:ea → Y:pp	MOVEP	X:ea,Y:pp
Y:ea → Y:pp	MOVEP	Y:ea,Y:pp
P:ea → Y:pp	MOVEP	P:ea,Y:pp

Description: Move the specified operand from/to the specified **X or Y I/O peripheral**. The I/O short addressing mode is used for the I/O peripheral address. All memory addressing modes may be used for the X or Y memory effective address; all memory alterable addressing modes may be used for the P memory effective address.

If the system stack register SSH is specified as a source operand, the system stack pointer (SP) is postdecremented by 1 after SSH has been read. If the system stack register SSH is specified as a destination operand, the system stack pointer (SP) is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

When a 56-bit accumulator (A or B) is specified as a **source** operand S, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). 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 24-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. If a 24-bit source operand is to be moved into a 16-bit destination register D, the 8 MS bits of the 24-bit source operand are discarded, and the 16 LS bits are stored in the 16-bit destination register. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-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 condition code register is latched.

When a 56-bit accumulator (A or B) is specified as a **destination** operand D, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 23) and appending the source operand with 24 LS zeros. Whenever a 16-bit source operand S is to be moved into a 24-bit destination, the 16-bit source is loaded into the LS 16 bits of the destination operand, and the remaining 8 MS bits of the destination are zeroed. Note that for 24-bit source operands both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1).

Note: Due to pipelining, if an address register (R, N, or M) is changed using a move-type instruction, the new contents of the destination address register will not be available for use during the following instruction (i.e, there is a single instuction cycle pipeline delay).

Restrictions: The following restrictions represent very unusual operations, which probably would never be used but are listed only for completeness.

A MOVEP instruction used **within a DO loop** which specifies **SSH** as the **source** operand or **LA, LC, SR, SP, SSH, or SSL** as the **destination** operand cannot begin at the address LA–2, LA–1, or LA within that DO loop.

A MOVEP instruction which specifies **SSH** as the **source** operand or **LA, LC, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** a DO instruction.

A MOVEP instruction which specifies **SSH** as the **source** operand or **LA, LC, SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an ENDDO instruction.

A MOVEP instruction which specifies **SSH** as the **source** operand or **SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTI instruction.

A MOVEP instruction which specifies **SSH** as the **source** operand or **SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTS instruction.

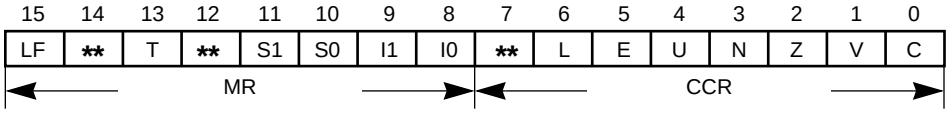
A MOVEP instruction which specifies **SP** as the **destination** operand cannot be used **immediately before** a MOVEC, MOVEM, or MOVEP instruction which specifies **SSH or SSL** as the **source** operand.

Example:

```
        :
MOVEP #1113,X:<<$FFFE        :initialize Bus Control Register wait states
        :
```

Explanation of Example: Prior to execution, the 16-bit, X memory-mapped, I/O bus control register (BCR) contains the value $FFFF. The execution of the MOVEP #$1113,X:<<$FFFE instruction moves the value $1113 into the 16-bit bus control register X:$FFFE, resulting in one wait state for all external X, external Y, and external program memory accesses and three wait states for all external I/O accesses.

Condition Codes:



For D=SR 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**

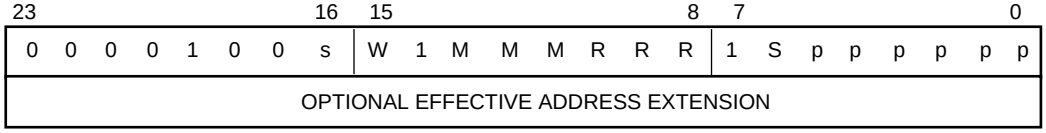
For D≠SR operand:

L — Set if data limiting has occurred during the move

Instruction Format (X: or Y: Reference):

```
MOVEP   X:ea,X:pp
MOVEP   Y:ea,X:pp
MOVEP   #xxxxxx,X:pp
MOVEP   X:pp,X:ea
MOVEP   X:pp,Y:ea
MOVEP   X:ea,Y:pp
MOVEP   Y:ea,Y:pp
MOVEP   #xxxxxx,Y:pp
MOVEP   Y:pp,Y:ea
MOVEP   Y:pp,Y:ea
```

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR,
pp=6-bit I/O Short Address=pppppp

Memory Space	S	Effective Addressing Mode	M	M	M	R	R	R
X Memory	0	(Rn)-Nn	0	0	0	r	r	r
Y Memory	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
Peripheral Space	**s**	(Rn)+	0	1	1	r	r	r
X Memory	0	(Rn)	1	0	0	r	r	r
Y Memory	1	(Rn+Nn)	1	0	1	r	r	r
		–(Rn)	1	1	1	r	r	r
Peripheral	**W**	Absolute address	1	1	0	0	0	0
Read	0	Immediate data	1	1	0	1	0	0
Write	1							

where ‘‘rrr’’ refers to an address register R0–R7

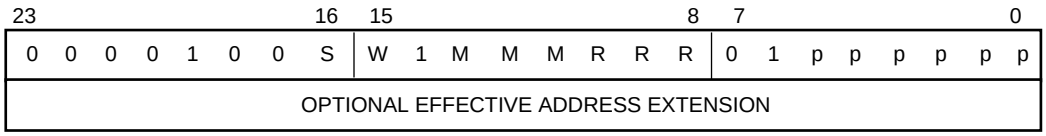
Timing: 4+mvp oscillator clock cycles

Memory: 1+ea program words

Instruction Format (P: Reference):

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

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR
pp=6-bit I/O Short Address=pppppp

Peripheral Space	S	Effective Addressing Mode	M	M	M	R	R	R
X Memory	0	(Rn)-Nn	0	0	0	r	r	r
Y Memory	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
Peripheral	**W**	(Rn)+	0	1	1	r	r	r
Read	0	(Rn)	1	0	0	r	r	r
Write	1	(Rn+Nn)	1	0	1	r	r	r
		–(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0

where ‘‘rrr’’ refers to an address register R0–R7

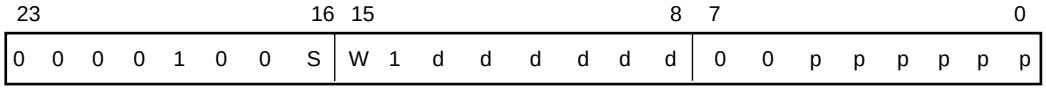
Timing: 4+mvp oscillator clock cycles

Memory: 1+ea program words

Instruction Format (Register Reference):

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

Opcode:



Instruction Fields:

pp=6-bit I/O Short Address=pppppp

Peripheral Space	**S**	**Peripheral**	**W**
X Memory	0	Read	0
Y Memory	1	Write	1

S,D	**d d d d d d**	**S S/L**	**D Sign Ext**	**D Zero**
X0	0 0 0 1 0 0	no	no	no
X1	0 0 0 1 0 1	no	no	no
Y0	0 0 0 1 1 0	no	no	no
Y1	0 0 0 1 1 1	no	no	no
A0	0 0 1 0 0 0	no	no	no
B0	0 0 1 0 0 1	no	no	no
A2	0 0 1 0 1 0	no	no	no
B2	0 0 1 0 1 1	no	no	no
A1	0 0 1 1 0 0	no	no	no
B1	0 0 1 1 0 1	no	no	no
A	0 0 1 1 1 0	yes	A2	A0
B	0 0 1 1 1 1	yes	B2	B0

S,D	**d d d d d d**
R0 - R7	0 1 0 n n n
N0 - N7	0 1 1 n n n
M0 - M7	1 0 0 n n n
SR	1 1 1 0 0 1
OMR	1 1 1 0 1 0
SP	1 1 1 0 1 1
SSH	1 1 1 1 0 0
SSL	1 1 1 1 0 1
LA	1 1 1 1 1 0
LC	1 1 1 1 1 1

where “nnn” = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

Timing: 4+mvp oscillator clock cycles

Memory: 1+ea program words

Operation:

±S1∗S2 → D (parallel move)

±S1∗S2 → D (parallel move)

Assembler Syntax:

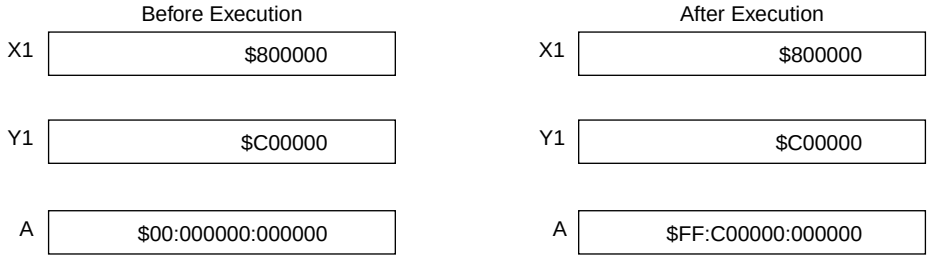
MPY (±)S1,S2,D (parallel move)

MPY (±)S2,S1,D (parallel move)

Description: Multiply the two signed 24-bit source operands S1 and S2 and store the resulting product in the specified 56-bit destination accumulator D. The ‘‘–’’ sign option is used to negate the specified product. The default sign option is ‘‘+’’.

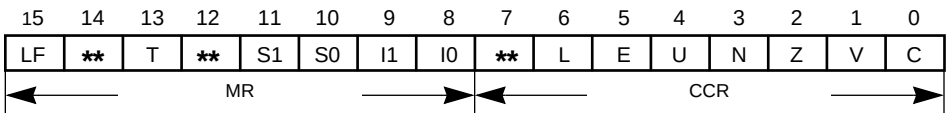
Example:

```
        :
MPY –X1,Y1,A #$543210,Y0        ;–(X1∗Y1) → A, update Y0
        :
```



Explanation of Example: Prior to execution, the 24-bit X1 register contains the value $800000 (–1.0), the 24-bit Y1 register contains the value $C00000, (–0.5), and the 56-bit A accumulator contains the value $00:000000:000000 (0.0). The execution of the MPY –X1,Y1,A instruction multiples the 24-bit signed value in the X1 register by the 24-bit signed value in the Y1 register, negates the 48-bit product, and stores the result in the 56-bit A accumulator (–X1∗Y1=–0.5=$FF:C00000:000000=A).

Condition Codes:



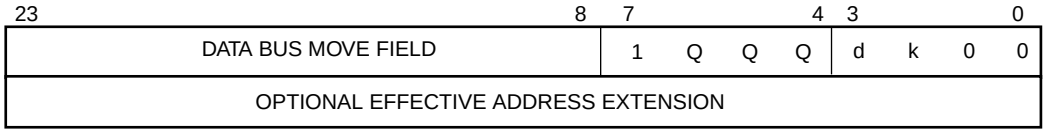
L — Set if data limiting has occurred during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Always cleared

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

MPY (±)S1,S2,D
MPY (±)S2,S1,D

Opcode:



Instruction Fields:

S1∗S2	Q Q Q	Sign	k	D d
X0 X0	0 0 0	+	0	A 0
Y0 Y0	0 0 1	–	1	B 1
X1 X0	0 1 0			
Y1 Y0	0 1 1			
X0 Y1	1 0 0			
Y0 X0	1 0 1			
X1 Y0	1 1 0			
Y1 X1	1 1 1			

Note: Only the indicated S1∗S2 combinations are valid. X1∗X1 and Y1∗Y1 are **not** valid.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

±S1∗S2+r → D (parallel move)

±S1∗S2+r → D (parallel move)

Assembler Syntax:

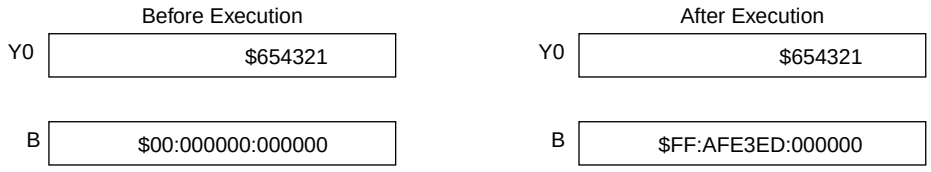
MPYR (±)S1,S2,D (parallel move)

MPYR (±)S2,S1,D (parallel move)

Description: Multiply the two signed 24-bit source operands S1 and S2, round the result using convergent rounding, and store it in the specified 56-bit destination accumulator D. The ‘‘–’’ sign option is used to negate the product prior to rounding. The default sign option is ‘‘+’’. The contribution of the LS bits of the result is rounded into the upper portion of the destination accumulator (A1 or B1) by adding a constant to the LS bits of the lower portion of the accumulator (A0 or B0). The value of the constant added is determined by the scaling mode bits S0 and S1 in the status register. Once the rounding has been completed, the LS bits of the destination accumulator D (A0 or B0) are loaded with zeros to maintain an unbiased accumulator value which may be reused by the next instruction. The upper portion of the accumulator (A1 or B1) contains the rounded result which may be read out to the data buses. Refer to the RND instruction for more complete information on the convergent rounding process.

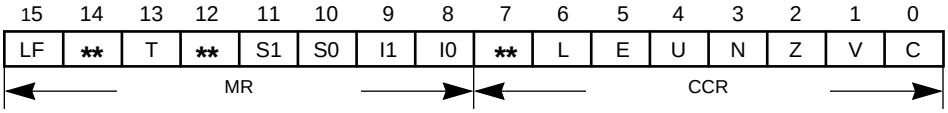
Example:

```
        :
MPYR –Y0,Y0,B (R3)–N3          ;square and negate Y0, update R3
        :
```



Explanation of Example: Prior to execution, the 24-bit Y0 register contains the value $654321 (0.791111112), and the 56-bit B accumulator contains the value $00:000000:000000 (0.0). The execution of the MPYR –Y0,Y0,B instruction squares the 24-bit signed value in the Y0 register, negates the resulting 48-bit product, rounds the result into B1, and zeros B0 (–Y0∗Y0=–0.625856790961748 approximately= $FF:AFE3EC:B76B7E, which is rounded to the value $FF:AFE3ED:000000= –0.625856757164002=B).

Condition Codes:



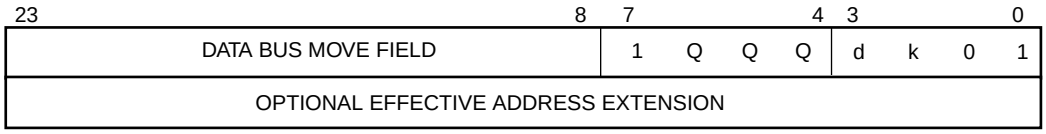
L — Set if data limiting has occurred during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Always cleared

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

MPYR (±)S1,S2,D
MPYR (±)S2,S1,D

Opcode:



Instruction Fields:

S1∗S2	**Q Q Q**	**Sign**	**k**	**D**	**d**
X0 X0	0 0 0	+	0	A	0
Y0 Y0	0 0 1	–	1	B	1
X1 X0	0 1 0				
Y1 Y0	0 1 1				
X0 Y1	1 0 0				
Y0 X0	1 0 1				
X1 Y0	1 1 0				
Y1 X1	1 1 1				

Note: Only the indicated S1∗S2 combinations are valid. X1∗X1 and Y1∗Y1 are **not** valid.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
0–D → D (parallel move)

Assembler Syntax:
NEG D (parallel move)

Description: Negate the destination operand D and store the result in the destination accumulator. This is a 56-bit, twos-complement operation.

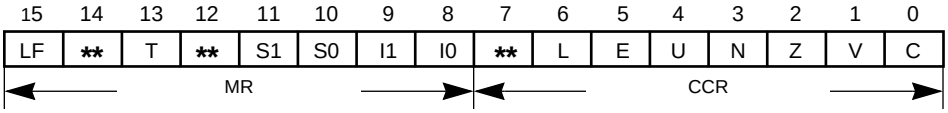
Example:

```
        :
NEG B X1,X:(R3)+  Y:(R6)–,A        ;0–B → B, update A,X1,R3,R6
        :
```

	Before Execution		After Execution
B	$00:123456:789ABC	B	$FF:EDCBA9:876544

Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $00:123456:789ABC. The NEG B instruction takes the twos complement of the value in the B accumulator and stores the 56-bit result back in the B accumulator.

Condition Codes:



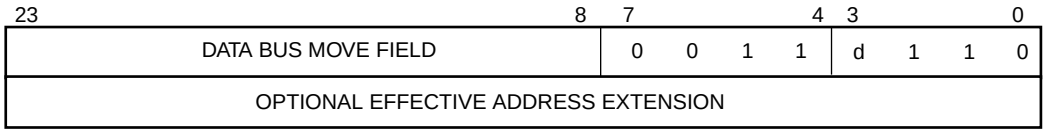
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

NEG D

Opcode:



Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
PC+1→PC

Assembler Syntax:
NOP

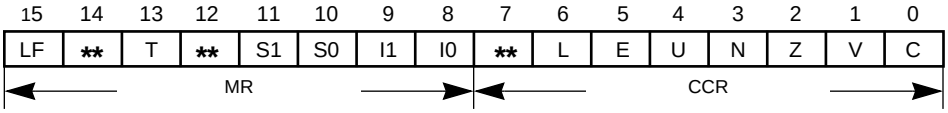
Description: Increment the program counter (PC). Pending pipeline actions, if any, are completed. Execution continues with the instruction following the NOP.

Example:

```
        :
NOP                     ;increment the program counter
        :
```

Explanation of Example: The NOP instruction increments the program counter and completes any pending pipeline actions.

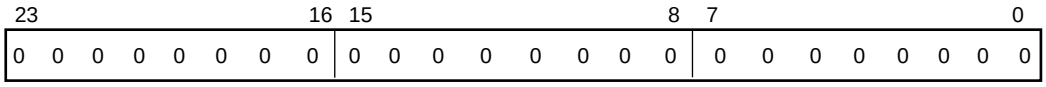
Condition Codes:



The condition codes are not affected by this instruction.

Instruction Format:
NOP

Opcode:



Instruction Fields:
None

Timing: 2 oscillator clock cycles

Memory: 1 program word

Operation:

If $\overline{E} \bullet U \bullet \overline{Z}$=1, then ASL D and Rn–1→Rn
else if E=1, then ASR D and Rn+1→Rn
else NOP

where $\overline{E}$ denotes the logical complement of E, and
where • denotes the logical AND operator

Assembler Syntax:

NORM Rn,D

Description: Perform one normalization iteration on the specified destination operand D, update the specified address register Rn based upon the results of that iteration, and store the result back in the destination accumulator. This is a 56-bit operation. If the accumulator extension is not in use, the accumulator is unnormalized, and the accumulator is not zero, the destination operand is arithmetically shifted one bit to the left, and the specified address register is decremented by 1. If the accumulator extension register is in use, the destination operand is arithmetically shifted one bit to the right, and the specified address register is incremented by 1. If the accumulator is normalized or zero, a NOP is executed and the specified address register is not affected. Since the operation of the NORM instruction depends on the E, U, and Z condition code register bits, these bits must correctly reflect the current state of the destination accumulator prior to executing the NORM instruction. Note that the L and V bits in the condition code register will be cleared unless they have been improperly set up prior to executing the NORM instruction.

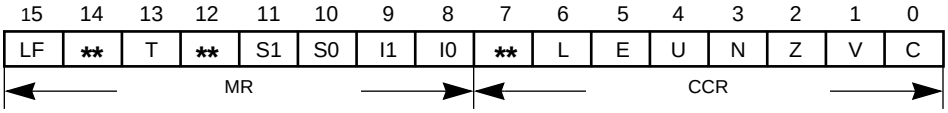
Example:

```
    :
REP #$2F              ;maximum number of iterations needed
NORM R3,A             ;perform 1 normalization iteration
    :
```

	Before Execution		After Execution
A	$00:000000:000001	A	$00:400000:000000
R3	$0000	R3	$FFD2

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:000000:000001, and the 16-bit R3 address register contains the value $0000. The repetition of the NORM R3,A instruction normalizes the value in the 56-bit accumulator and stores the resulting number of shifts performed during that normalization process in the R3 address register. A negative value reflects the number of left shifts performed; a positive value reflects the number of right shifts performed during the normalization process.

Condition Codes:



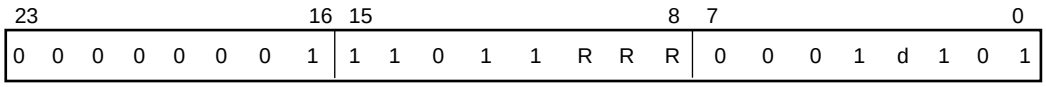
L — Set if overflow has occurred in A or B result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — **Set if bit 55 is changed as a result of a left shift**

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

NORM Rn,D

Opcode:



Instruction Fields:

D	**d**	**Rn**	**R R R**
A	0	Rn	n n n
B	1		

where “nnn” = Rn number

Timing: 2 oscillator clock cycles

Memory: 1 program word

Operation:

$\overline{D[47:24]}$ → D[47:24] (parallel move)

where "—" denotes the logical NOT operator

Assembler Syntax:

NOT D (parallel move)

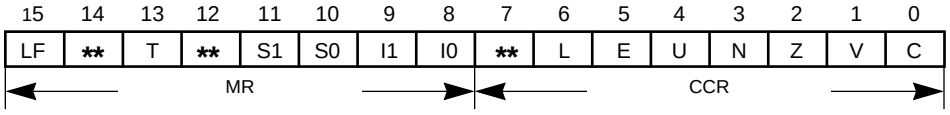
Description: Take the ones complement of bits 47–24 of the destination operand D and store the result back in bits 47–24 of the destination accumulator. This is a 24-bit operation. The remaining bits of D are not affected.

Example:

```
NOT A    AB,L:(R2)+          ;save A1,B1, take the ones complement of A1
```

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:123456:789ABC. The NOT A instruction takes the ones complement of bits 47–24 of the A accumulator (A1) and stores the result back in the A1 register. The remaining bits of the A accumulator are not affected.

Condition Codes:

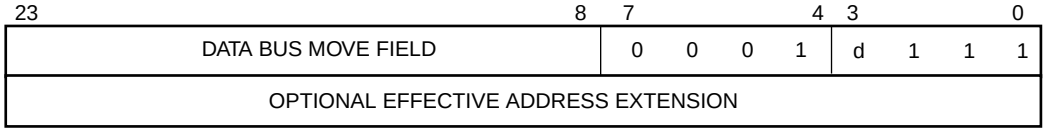


L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47-24 of A or B result are zero**
V — Always cleared

Instruction Format:

NOT D

Opcode:



Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

S+D[47:24] → D[47:24] (parallel move)
where + denotes the logical inclusive OR operator

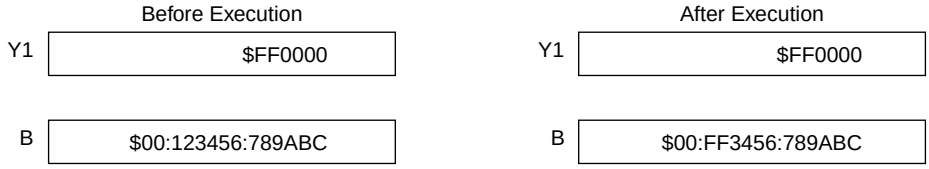
Assembler Syntax:

OR S,D (parallel move)

Description: Logically inclusive OR the source operand S with bits 47–24 of the destination operand D and store the result in bits 47–24 of the destination accumulator. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

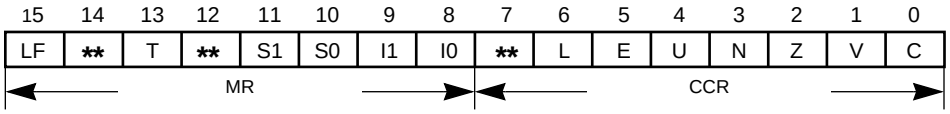
Example:

```
        :
OR Y1,B         BA,L:$1234              ;save A1,B1, OR Y1 with B
        :
```



Explanation of Example: Prior to execution, the 24-bit Y1 register contains the value $FF0000, and the 56-bit B accumulator contains the value $00:123456:789ABC. The OR Y1,B instruction logically ORs the 24-bit value in the Y1 register with bits 47–24 of the B accumulator (B1) and stores the result in the B accumulator with bits 55–48 and 23–0 unchanged.

Condition Codes:

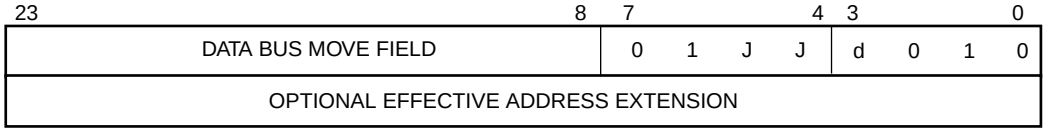


L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47-24 of A or B result are zero**
V — Always cleared

Instruction Format:

OR S.D

Opcode:



Instruction Fields:

S	**J J**	**D d**
X0	0 0	A 0
X1	1 0	B 1
Y0	0 1	
Y1	1 1	

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
#xx+D → D
where + denotes the logical inclusive OR operator

Assembler Syntax:
OR(I) #xx,D

Description: Logically OR the 8-bit immediate operand (#xx) with the contents of the destination control register D and store the result in the destination control register. The condition codes are affected only when the condition code register is specified as the destination operand.

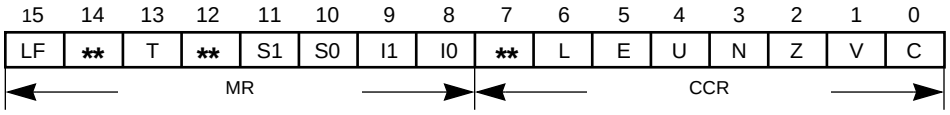
Restrictions: The ORI #xx,MR instruction cannot be used **immediately before** an ENDDO or RTI instruction and cannot be one of the **last three** instructions in a DO loop (at LA–2, LA–1, or LA).

Example:

```
        :
OR   #$8,MR                ;set scaling mode bit S1 to scale up
        :
```

Explanation of Example: Prior to execution, the 8-bit mode register (MR) contains the value $03. The OR #$8,MR instruction logically ORs the immediate 8-bit value $8 with the contents of the mode register and stores the result in the mode register.

Condition Codes:



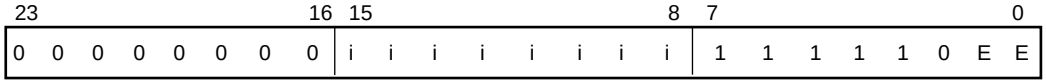
For CCR operand:
L — **Set if bit 6 of the immediate operand is set**
E — **Set if bit 5 of the immediate operand is set**
U — **Set if bit 4 of the immediate operand is set**
N — **Set if bit 3 of the immediate operand is set**
Z — **Set if bit 2 of the immediate operand is set**
V — **Set if bit 1 of the immediate operand is set**
C — **Set if bit 0 of the immediate operand is set**

For MR and OMR operands:
The condition codes are not affected using these operands.

Instruction Format:

OR(I) #xx,D

Opcode:



Instruction Fields:

#xx=8-bit Immediate Short Data = i i i i i i i i

D	**E E**
MR	0 0
CCR	0 1
OMR	1 0

Timing: 2 oscillator clock cycles

Memory: 1 program words

Operation:	Assembler Syntax:	
LC → TEMP; X:ea → LC Repeat next instruction until LC=1 TEMP → LC	REP	X:ea
LC → TEMP; X:aa → LC Repeat next instruction until LC=1 TEMP → LC	REP	X:aa
LC → TEMP; Y:ea → LC Repeat next instruction until LC=1 TEMP → LC	REP	Y:ea
LC → TEMP; Y:aa → LC Repeat next instruction until LC=1 TEMP → LC	REP	Y:aa
LC → TEMP; S → LC Repeat next instruction until LC=1 TEMP → LC	REP	S
LC → TEMP; #xxx → LC Repeat next instruction until LC=1 TEMP → LC	REP	#xxx

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 16-bit loop counter (LC) register. The single-word instruction is then executed the specified number of times, decrementing the loop counter (LC) after each execution until LC=1. 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** (sequential repeats are also not interruptible). The current loop counter (LC) value is stored in an internal temporary register. If LC is set equal to zero, the instruction is repeated 65,536 times. The instruction's effective address specifies the address of the value which is to be loaded into the loop counter (LC). All address register indirect addressing modes may be used. The absolute short and the immediate short addressing modes may also be used. The four MS bits of the 12-bit immediate value are zeroed to form the 16-bit value that is to be loaded into the loop counter (LC).

If the A or B accumulator is specified as a **source** operand, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). If the data out of the shifter indicates that the accumulator extension is in use, the value to be loaded into the loop counter (LC) register will be limited to a 24-bit maximum positive or negative saturation constant to minimize the error due to truncation. The LS 16 bits of the resulting 24-bit value are then stored in the 16-bit loop counter (LC) register.

If the system stack register SSH is specified as a source operand, the system stack pointer (SP) is postdecremented by 1 after SSH has been read.

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

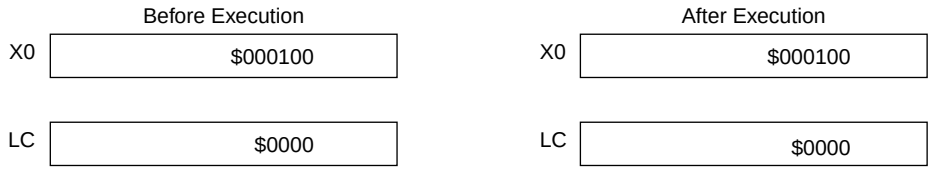
Immediately after REP

DO	JSSET
Jcc	REP
JCLR	RTI
JMP	RTS
JSET	STOP
JScc	SWI
JSCLR	WAIT
JSR	ENDDO

Also, a REP instruction cannot be the **last** instruction in a DO loop (at LA). The assembler will generate an error if any of the previous instructions are found immediately following a REP instruction.

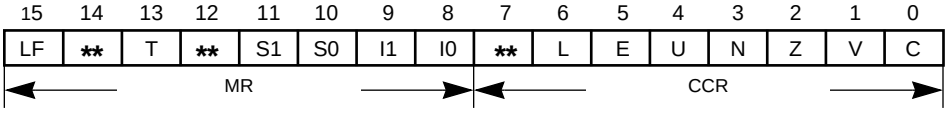
Example:

```
        :
        REP X0                                          ;repeat (X0) times
        MAC X1,Y1,A     X:(R1)+,X1    Y:(R4)+,Y1        ;X1*Y1+A → A, update X1,Y1
        :
```



Explanation of Example: Prior to execution, the 24-bit X0 register contains the value $000100, and the 16-bit loop counter (LC) register contains the value $0000. The execution of the REP X0 instruction takes the 24-bit value in the X0 register, truncates the MS 8 bits, and stores the 16 LS bits in the 16-bit loop counter (LC) register. Thus, the single-word MAC instruction immediately following the REP instruction is repeated $100 times.

Condition Codes:

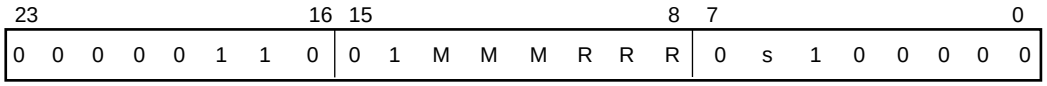


L — Set if data limiting occurred using A or B as source operands

Instruction Format:

REP X:ea
REP Y:ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR,

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r

Memory Space	s
X Memory	0
Y Memory	1

where “rrr” refers to an address register R0-R7

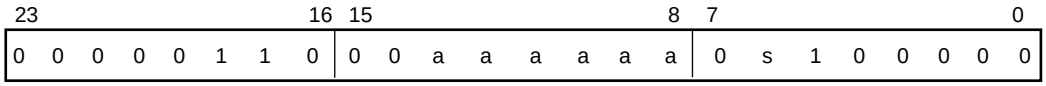
Timing: 4+mv oscillator clock cycles

Memory: 1 program word

Instruction Format:

REP X:ea
REP Y:ea

Opcode:



Instruction Fields:

ea=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory Space s	
000000	X Memory	0
•	Y Memory	1
•		
111111		

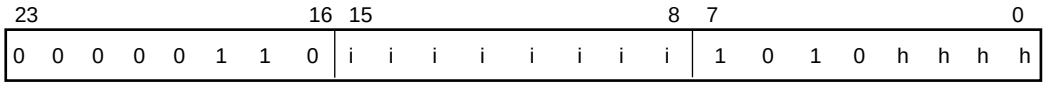
Timing: 4+mv oscillator clock cycles

Memory: 1 program word

Instruction Format:

REP #xxx

Opcode:



Instruction Fields:

#xxx=12-bit Immediate Short Data = hhhh i i i i i i i i

Immediate Short Data hhhh i i i i i i i i
000000000000
•
•
111111111111

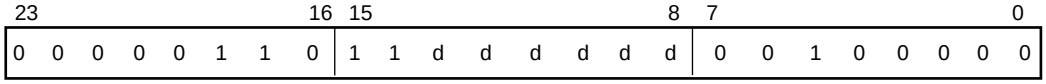
Timing: 4+mv oscillator clock cycles

Memory: 1 program word

Instruction Format:

REP S

Opcode:



Instruction Fields:

S	d d d d d d	S S/L	S	d d d d d d
X0	0 0 0 1 0 0	no	R0 - R7	0 1 0 n n n
X1	0 0 0 1 0 1	no	N0 - N7	0 1 1 n n n
Y0	0 0 0 1 1 0	no	M0 - M7	1 0 0 n n n
Y1	0 0 0 1 1 1	no	SR	1 1 1 0 0 1
A0	0 0 1 0 0 0	no	OMR	1 1 1 0 1 0
B0	0 0 1 0 0 1	no	SP	1 1 1 0 1 1
A2	0 0 1 0 1 0	no	SSH	1 1 1 1 0 0
B2	0 0 1 0 1 1	no	SSL	1 1 1 1 0 1
A1	0 0 1 1 0 0	no	LA	1 1 1 1 1 0
B1	0 0 1 1 0 1	no	LC	1 1 1 1 1 1
A	0 0 1 1 1 0	yes		
B	0 0 1 1 1 1	yes		

where “nnn” = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

Timing: 4 oscillator clock cycles

Memory: 1 program word

Operation:
Reset the interrupt priority register and all on-chip peripherals

Assembler Syntax:
RESET

Description: Reset the interrupt priority register and all on-chip peripherals. This is a **software reset** which is **NOT** equivalent to a hardware reset since only on-chip peripherals and the interrupt structure are affected. The processor state is not affected, and execution continues with the next instruction. All interrupt sources are disabled except for the trace, stack error, NMI, illegal instruction, and hardware reset interrupts.

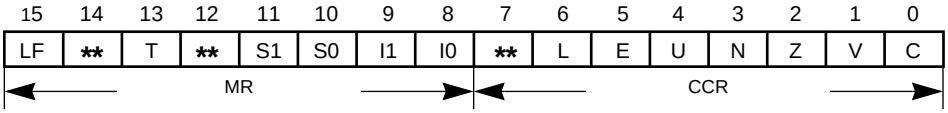
Restrictions: A RESET instruction cannot be the **last** instruction in a DO loop (at LA).

Example:

```
        :
RESET                   ;reset all on-chip peripherals and IPR
        :
```

Explanation of Example: The execution of the RESET instruction resets all on-chip peripherals and the interrupt priority register (IPR).

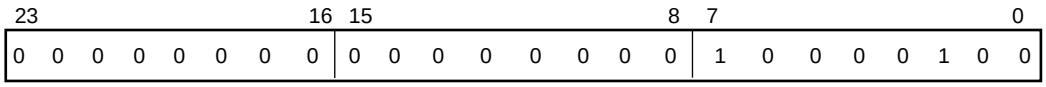
Condition Codes:



The condition codes are not affected by this instruction

Instruction Format:
RESET

Opcode:



Instruction Fields:
None

Timing: 4 oscillator clock cycles

Memory: 1 program word

Operation:
D+r → D (parallel move)

Assembler Syntax:
RND D (parallel move)

Description: Round the 56-bit value in the specified destination operand D and store the result in the MSP portion of the destination accumulator (A1 or B1). This instruction uses a convergent rounding technique. The contribution of the LS bits of the result (A0 and B0) is rounded into the upper portion of the result (A1 or B1) by adding a rounding constant to the LS bits of the result. The MSP portion of the destination accumulator contains the rounded result which may be read out to the data buses.

The value of the rounding constant added is determined by the scaling mode bits S0 and S1 in the system status register (SR). A ‘‘1’’ is added in the rounding position as shown below:

			Rounding	Rounding Constant				
S1	**S0**	**Scaling Mode**	**Position**	**55 - 25**	**24**	**23**	**22**	**21 - 0**
0	0	No Scaling	23	0. . . .0	0	1	0	0. . . .0
0	1	Scale Down	24	0. . . .0	1	0	0	0. . . .0
1	0	Scale Up	22	0. . . .0	0	0	1	0. . . .0

Normal or ‘‘standard’’ rounding consists of adding a rounding constant to a given number of LS bits of a value to produce a rounded result. The rounding constant depends on the scaling mode being used as previously shown. Unfortunately, when using a twos-complement data representation, this process introduces a positive bias in the statistical distribution of the roundoff error.

Convergent rounding differs from ‘‘standard’’ rounding in that convergent rounding attempts to remove the aforementioned positive bias by equally distributing the round-off error. The convergent rounding technique initially performs ‘‘standard’’ rounding as previously described. Again, the rounding constant depends on the scaling mode being used. Once ‘‘standard’’ rounding has been done, the convergent rounding method tests the result to determine if **all** bits **including and to the right** of the rounding position are **zero**. **If, and only if**, this **special condition** is true, the convergent rounding method will clear the bit immediately to the **left** of the rounding position. When this special condition is true, numbers which have a ‘‘1’’ in the bit immediately to the left of the rounding position are rounded **up**; numbers with a ‘‘0’’ in the bit immediately to the left of the rounding position are rounded **down**. Thus, these numbers are rounded **up** half the time and rounded **down** the rest of the time. Therefore, the **roundoff error averages out to zero**. The LS bits of the convergently rounded result are then cleared so that the rounded result may be immediately used by the next instruction.

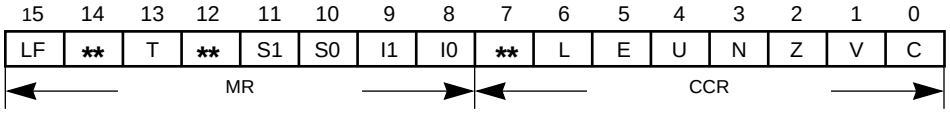
Example:

```
        :
    RND A #$123456,X1 B,Y1      ;round A accumulator into A1, zero A0
        :
```

	Before Execution		After Execution
Case I: A	$00:123456:789AB	A	$00:123456:000000
Case II: A	$00:123456:800000	A	$00:123456:000000
Case III: A	$00:123456:800000	A	$00:123456:000000

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:123456:789ABC for Case I, the value $00:123456:800000 for Case II, and the value $00:123455:800000 for Case III. The execution of the RND A instruction rounds the value in the A accumulator into the MSP portion of the A accumulator (A1), using convergent rounding, and then zeros the LSP portion of the A accumulator (A0). Note that Case II is the special case that distinguishes convergent rounding from standard or biased rounding.

Condition Codes:



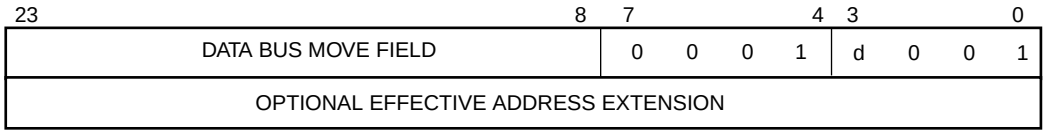
L — Set if data limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

RND D

Opcode:



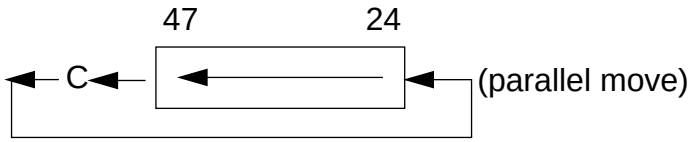
Instruction Fields:

D	**D**
A	0
B	1

Timing: 4 oscillator clock cycles

Memory: 1 program word

Operation:

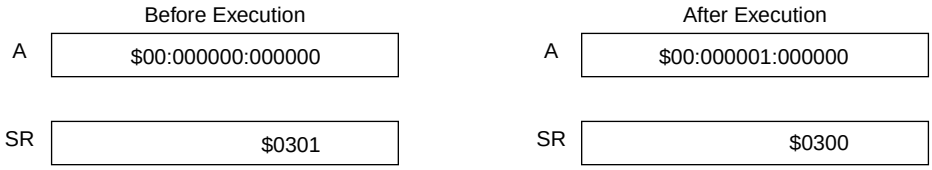


Assembler Syntax: ROL D (parallel move)

Description: Rotate bits 47–24 of the destination operand D one bit to the left and store the result in the destination accumulator. Prior to instruction execution, bit 47 of D is shifted into the carry bit C, and, prior to instruction execution, the value in the carry bit C is shifted into bit 24 of the destination accumulator D. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

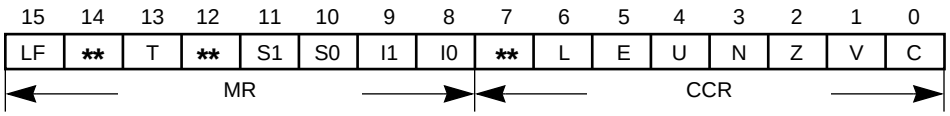
Example:

```
        :
ROL A   #314,N2          ;rotate A1 one left bit, update N2
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:000000:000000. The execution of the ROL A instruction shifts the 24-bit value in the A1 register one bit to the left, shifting bit 47 into the carry bit C, rotating the carry bit C into bit 24, and storing the result back in the A1 register.

Condition Codes:

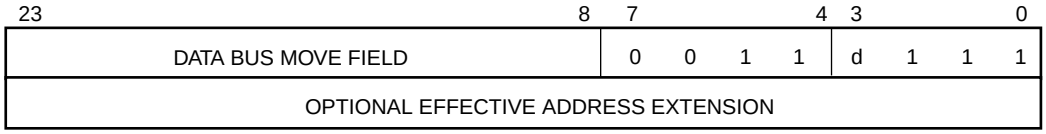


L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47–24 of A or B result are zero**
V — Always cleared
C — **Set if bit 47 of A or B was set prior to instruction execution**

Instruction Format:

ROL D

Opcode:



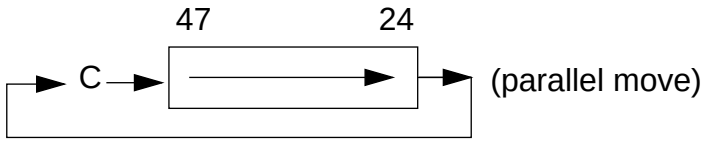
Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:

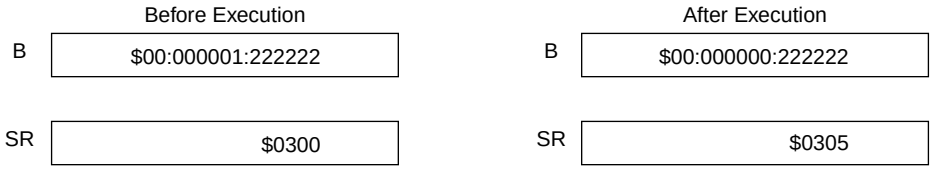


Assembler Syntax: ROR D (parallel move)

Description: Rotate bits 47–24 of the destination operand D one bit to the right and store the result in the destination accumulator. Prior to instruction execution, bit 24 of D is shifted into the carry bit C, and, prior to instruction execution, the value in the carry bit C is shifted into bit 47 of the destination accumulator D. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

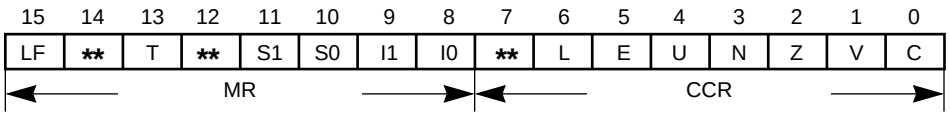
Example:

```
        :
ROR B   #$1234,R2       ;rotate B1 right one bit, update R2
        :
```



Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $00:000001:222222. The execution of the ROR B instruction shifts the 24-bit value in the B1 register one bit to the right, shifting bit 24 into the carry bit C, rotating the carry bit C into bit 47, and storing the result back in the B1 register.

Condition Codes:

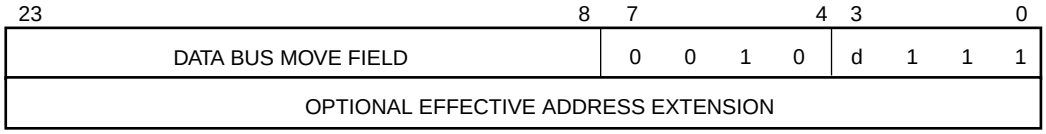


L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47–24 of A or B result are zero**
V — Always cleared
C — **Set if bit 47 of A or B was set prior to instruction execution**

Instruction Format:

ROR D

Opcode:



Instruction Fields:

D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
SSH → PC; SSL → SR; SP–1 → SP

Assembler Syntax:
RTI

Description: Pull the program counter (PC) and the status register (SR) from the system stack. The previous program counter and status register are lost.

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:

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

An RTI instruction cannot be the **last** instruction in a DO loop (at LA).

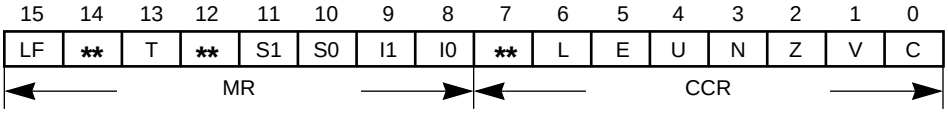
An RTI instruction cannot be repeated using the REP instruction.

Example:

```
        :
    RTI             ;pull PC and SR from system stack
        :
```

Explanation of Example: The RTI instruction pulls the 16-bit program counter (PC) and the 16-bit status register (SR) from the system stack and updates the system stack pointer (SP).

Condition Codes:

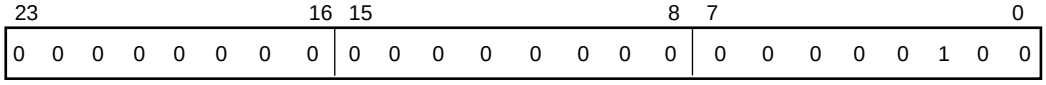


L — **Set according to the value pulled from the stack**
E — **Set according to the value pulled from the stack**
U — **Set according to the value pulled from the stack**
N — **Set according to the value pulled from the stack**
Z — **Set according to the value pulled from the stack**
V — **Set according to the value pulled from the stack**
C — **Set according to the value pulled from the stack**

Instruction Format:

RTI

Opcode:



Instruction Fields:

None

Timing: 4+rx oscillator clock cycles

Memory: 1 program word

Operation:
SSH → PC; SP–1 → SP

Assembler Syntax:
RTS

Description: Pull the program counter (PC) from the system stack. The previous program counter is lost. The status register (SR) is not affected.

Restrictions: Due to pipelining in the program controller and the fact that the RTS instruction accesses certain controller registers, the RTS instruction must not be immediately preceded by any of the following instructions:

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

An RTS instruction cannot be the **last** instruction in a DO loop (at LA).

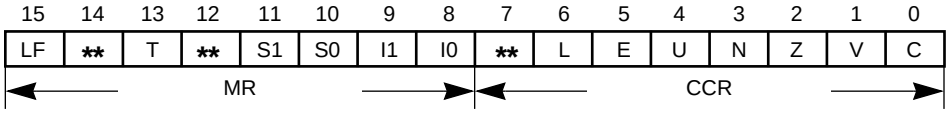
An RTS instruction cannot be repeated using the REP instruction.

Example:

```
        :
    RTS             ;pull PC from system stack
        :
```

Explanation of Example: The RTS instruction pulls the 16-bit program counter (PC) from the system stack and updates the system stack pointer (SP).

Condition Codes:

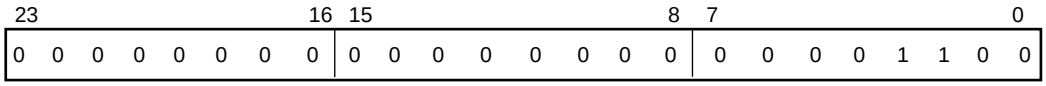


The condition codes are not affected by this instruction.

Instruction Format:

RTI

Opcode:



Instruction Fields:

None

Timing: 4+rx oscillator clock cycles

Memory: 1 program word

Operation:
D–S–C → D (parallel move)

Assembler Syntax:
SBC S,D (parallel move)

Description: Subtract the source operand S and the carry bit C of the condition code register from the destination operand D and store the result in the destination accumulator. Long words (48 bits) may be subtracted from the (56-bit) destination accumulator.

Note: The carry bit is set correctly for multiple-precision arithmetic using long-word operands if the extension register of the destination accumulator (A2 or B2) is the sign extension of bit 47 of the destination accumulator (A or B).

Example:

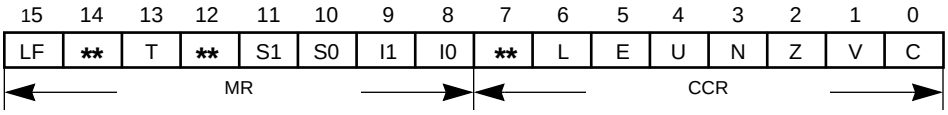
```
        :
MOVE L:<$0,X        ;get a 48-bit LS long-word operand in X
MOVE L:<$1,A        ;get other LS long word in A (sign ext.)
MOVE L:<$2,Y        ;get a 48-bit MS long-word operand in Y
SUB X,A L:<$3,B     ;sub. LS words; get other MS word in B
SBC YB A10,L:<$4    ;sub. MS words with carry; save LS dif.
MOVE B10,L:<$5      ;save MS difference
        :
```

	Before Execution	After Execution
A	$00:000000:000000	$00:800000:000000
X	$800000:000000	$800000:000000
B	$00:000000:000003	$00:000000:000001
Y	$000000:000001	$000000:000001

Explanation of Example: This example illustrates long-word double-precision (96-bit) subtraction using the SBC instruction. Prior to execution of the SUB and SBC instructions, the 96-bit value $000000:000001:800000:000000 is loaded into the Y and X registers (X:Y), respectively. The other double-precision 96-bit value $000000:000003:000000:000000 is loaded into the B and A accumulators (B:A), respectively. Since the 48-bit value loaded into the A accumulator is automatically sign extended to 56 bits and the other 48-bit long-word operand is internally sign extended to 56 bits during instruction execution, the carry bit will be set correctly after the execution

of the SUB X,A instruction. The SBC Y,B instruction then produces the correct MS 56-bit result. The actual 96-bit result is stored in memory using the A10 and B10 operands (instead of A and B) because shifting and limiting is not desired.

Condition Codes:



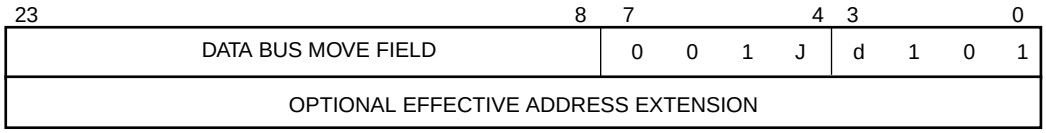
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

SBC S,D

Opcode:



Instruction Fields:

S,D	**J d**
X,A	0 0
X,B	0 1
Y,A	1 0
Y,B	1 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
Enter the stop processing state and stop the clock oscillator

Assembler Syntax:
STOP

Description: Enter the STOP processing state. All activity in the processor is suspended until the $\overline{\text{RESET}}$ or $\overline{\text{IRQA}}$ pin is asserted. The clock oscillator is gated off internally. The STOP processing state is a low-power standby state.

During the STOP state, port A is in an idle state with the control signals held inactive (i.e., $\overline{\text{RD}}$=$\overline{\text{WR}}$=V_{CC} etc.), the data pins (D0–D23) are high impedance, and the address pins (A1–A15) are unchanged from the previous instruction. If the bus grant was asserted when the STOP instruction was executed, port A will remain three-stated until the DSP exits the STOP state.

If the exit from the STOP state was caused by a low level on the $\overline{\text{RESET}}$ pin, then the processor will enter the reset processing state. The time to recover from the STOP state using $\overline{\text{RESET}}$ will depend on the oscillator used. Consult the DSP56001 Advance Information Data Sheet (ADI1290) for details.

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 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 65,536 clock cycles (or a three clock cycle delay if the stop delay bit in the OMR is set to one) plus 17T (see the DSP56001 Advance Information Data Sheet (ADI1290) for details). During this clock stabilization count delay, all peripherals and external interrupts are cleared and re-enabled/arbitrated at the start of the 17T period following the count interval. The processor will resume program execution at the instruction following the STOP instruction that caused the entry into the STOP state after the interrupt has 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 clock will not be gated off, and the internal delay counter will be started.

Restrictions:
A STOP instruction cannot be used in a fast interrupt routine.

A STOP instruction cannot be the **last** instruction in a DO loop (i.e., at LA).

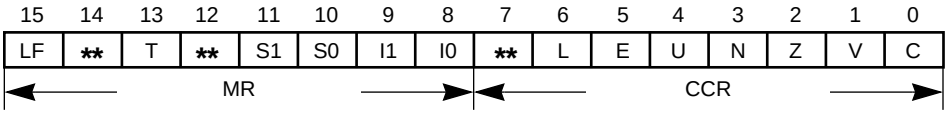
A STOP instruction cannot be repeated using the REP instruction.

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 state.

Condition Codes:

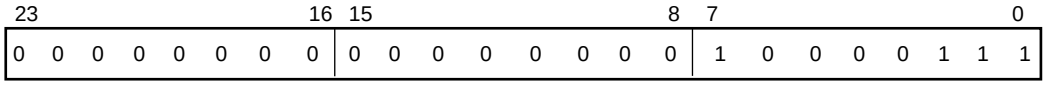


The condition codes are not affected by this instruction.

Instruction Format:

STOP

Opcode:



Instruction Fields:

None

Timing: The STOP instruction disables the internal clock oscillator and internal distribution of the external clock.

Memory: 1 program word

Operation:
D–S → D (parallel move)

Assembler Syntax:
SUB S,D (parallel move)

Description: Subtract the source operand S from the destination operand D and store the result in the destination operand D. Words (24 bits), long words (48 bits), and accumulators (56 bits) may be subtracted from the destination accumulator.

Note: The carry bit is set correctly using word or long-word source operands if the extension register of the destination accumulator (A2 or B2) is the sign extension of bit 47 of the destination accumulator (A or B). The carry bit is always set correctly using accumulator source operands.

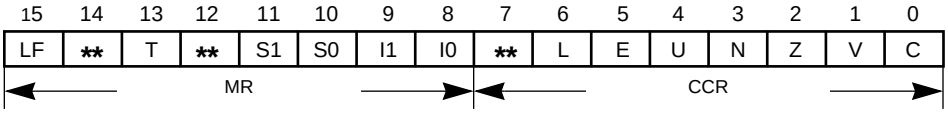
Example:

```
        :
SUB X1,A X:(R2)+N2,R0        ;24-bit subtract, load R0, update R2
        :
```

	Before Execution		After Execution
X1	$000003	X1	$000003
A	$00:000058:242424	A	$00:000055:242424

Explanation of Example: Prior to execution, the 24-bit X1 register contains the value $000003, and the 56-bit A accumulator contains the value $00:000058:242424. The SUB instruction automatically appends the 24-bit value in the X1 register with 24 LS zeros, sign extends the resulting 48-bit long word to 56 bits, and subtracts the result from the 56-bit A accumulator. Thus, 24-bit operands are subtracted from the MSP portion of A or B (A1 or B1) because all arithmetic instructions assume a fractional, twos complement data representation. Note that 24-bit operands can be subtracted from the LSP portion of A or B (A0 or B0) by loading the 24-bit operand into X0 or Y0, forming a 48-bit word by loading X1 or Y1 with the sign extension of X0 or Y0, and executing a SUB X,A or SUB Y,A instruction.

Condition Codes:



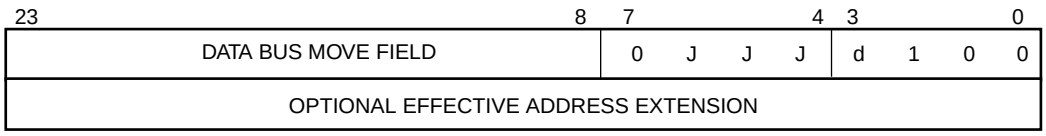
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

SUB S,D

Opcode:



Instruction Fields:

S,D	J J J d	S,D	J J J d	S,D	J J J d
B,A	0 0 1 0	X0,A	1 0 0 0	Y1,A	1 1 1 0
A,B	0 0 1 1	X0,B	1 0 0 1	Y1,B	1 1 1 1
X,A	0 1 0 0	Y0,A	1 0 1 0		
X,B	0 1 0 1	Y0,B	1 0 1 1		
Y,A	0 1 1 0	X1,A	1 1 0 0		
Y,B	0 1 1 1	X1,B	1 1 0 1		

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

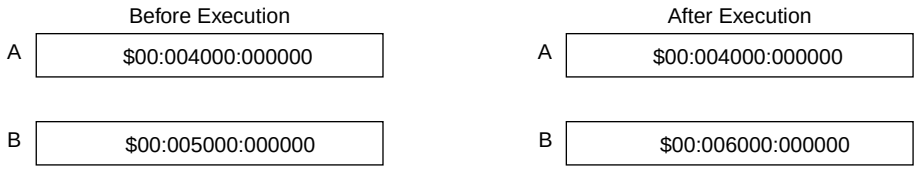
Operation:
2∗D–S → D (parallel move)

Assembler Syntax:
SUBL SD (parallel move)

Description: Subtract the source operand S from two times the destination operand D and store the result in the destination accumulator. The destination operand D is arithmetically shifted one bit to the left, and a zero is shifted into the LS bit of D prior to the subtraction operation. The carry bit is set correctly if the source operand does not overflow as a result of the left shift operation. The overflow bit may be set as a result of either the shifting or subtraction operation (or both). This instruction is useful for efficient divide and decimation in time (DIT) FFT algorithms.

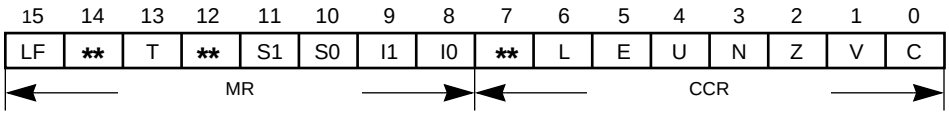
Example:

```
        :
SUBL    A,B     Y:(R5+N5),R7    ;2∗B–A → B, load R7, no R5 update
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:004000:000000, and the 56-bit B accumulator contains the value $00:005000:000000. The SUBL A,B instruction subtracts the value in the A accumulator from two times the value in the B accumulator and stores the 56-bit result in the B accumulator.

Condition Codes:



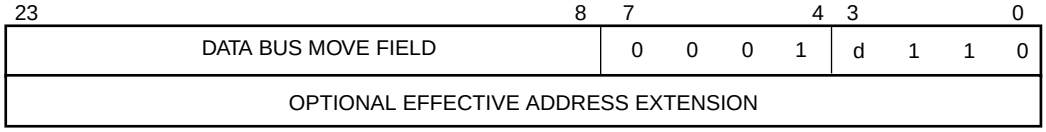
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — **Set if overflow has occurred in A or B result or if the MS bit of the destination operand is changed as a result of the instruction's left shift**
C — Set if a carry (or borrow) occurs from bit 55 of A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

SUBL S,D

Opcode:



Instruction Fields:

S,D	d
B,A	0
A,B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
D/2–S → D (parallel move)

Assembler Syntax:
SUBR S,D (parallel move)

Description: Subtract the source operand S from one-half the destination operand D and store the result in the destination accumulator. The destination operand D is arithmetically shifted one bit to the right while the MS bit of D is held constant prior to the subtraction operation. In contrast to the SUBL instruction, the carry bit is always set correctly, and the overflow bit can only be set by the subtraction operation, and not by an overflow due to the initial shifting operation. This instruction is useful for efficient divide and decimation in time (DIT) FFT algorithms.

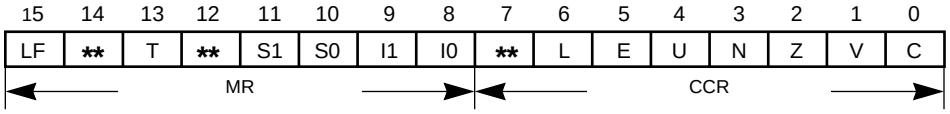
Example:

```
        :
SUBR B,A N5,Y:–(R5)          ;A/2–B → A, update R5, save N5
        :
```

	Before Execution	After Execution
A	$80:000000:2468AC	$C0:000000:000000
B	$00:000000:123456	$00:000000:123456

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $80:000000:2468AC, and the 56-bit B accumulator contains the value $00:000000:123456. The SUBR B,A instruction subtracts the value in the B accumulator from one-half the value in the A accumulator and stores the 56-bit result in the A accumulator.

Condition Codes:



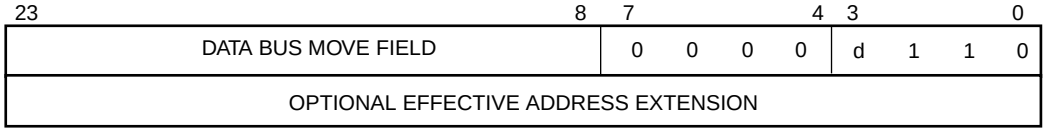
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

SUBR S,D

Opcode:



Instruction Fields:

S,D	d
B,A	0
A,B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
Begin SWI exception processing

Assembler Syntax:
SWI

Description: Suspend normal instruction execution and begin SWI exception processing. The interrupt priority level (I1,I0) is set to 3 in the status register (SR) if a long interrupt service routine is used.

Restrictions:
An SWI instruction cannot be used in a fast interrupt routine.

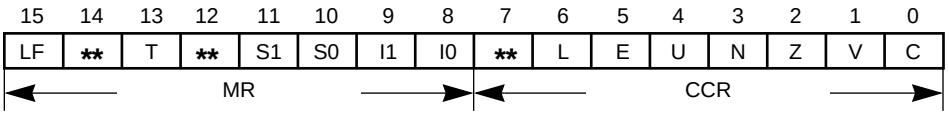
An SWI instruction cannot be repeated using the REP instruction.

Example:

```
        :
SWI                 ;begin SWI exception processing
        :
```

Explanation of Example: The SWI instruction suspends normal instruction execution and initiates SWI exception processing.

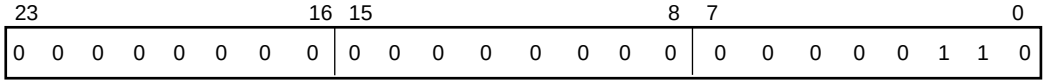
Condition Codes:



The condition codes are not affected by this instruction.

Instruction Format:
SWI

Opcode:



Instruction Fields:
None

Timing: 8 oscillator clock cycles

Memory: 1 program word

Operation:

If cc, then S1 → D1

If cc, then S1 → D1 and S2 → D2

Assembler Syntax:

Tcc S1,D1

Tcc S1,D1 S2,D2

Description: Transfer data from the specified source register S1 to the specified destination accumulator D1 if the specified condition is true. If a second source register S2 and a second destination register D2 are also specified, transfer data from address register S2 to address register D2 if the specified condition is true. If the specified condition is false, a NOP is executed. The term ‘‘cc’’ may specify the following conditions:

	‘‘cc’’ Mnemonic	**Condition**
CC (HS)	— carry clear (higher or same)	C=0
CS (LO)	— carry set (lower)	C=1
EC	— extension clear	E=0
EQ	— equal	Z=1
ES	— extension set	E=1
GE	— greater than or equal	N ⊕ V=0
GT	— greater than	Z+(N ⊕ V)=0
LC	— limit clear	L=0
LE	— less than or equal	Z+(N ⊕ V)=1
LS	— limit set	L=1
LT	— less than	N ⊕ V=1
MI	— minus	N=1
NE	— not equal	Z=0
NR	— normalized	Z+($\overline{U}$•$\overline{E}$)=1
PL	— plus	N=0
NN	— not normalized	Z+($\overline{U}$•$\overline{E}$)=0

where
$\overline{U}$ denotes the logical complement of U,
+ denotes the logical OR operator,
• denotes the logical AND operator, and
⊕ denotes the logical Exclusive OR operator

When used after the CMP or CMPM instructions, the Tcc instruction can perform many useful functions such as a ‘‘maximum value,’’ ‘‘minimum value,’’ ‘‘maximum absolute value,’’ or ‘‘minimum absolute value’’ function. The desired value is stored in the destination accumulator D1. If address register S2 is used as an address pointer into an array of data, the address of the desired value is stored in the address register D2. The Tcc

instruction may be used after any instruction and allows efficient searching and sorting algorithms.

The Tcc instruction uses the internal data ALU paths and internal address ALU paths. The Tcc instruction does not affect the condition code bits.

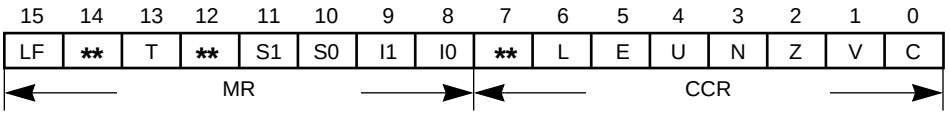
Note: This instruction is considered to be a move-type instruction. Due to pipelining, if an address register (R0–R7) is changed using a move-type instruction, the new contents of the destination address register will not be available for use during the following instruction (i.e., there is a single instruction cycle pipeline delay).

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 24-bit X0 register are transferred to the 56-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:

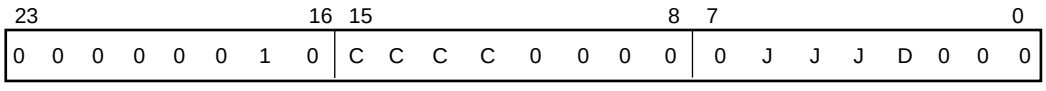


The condition codes are not affected by this instruction.

Instruction Format:

Tcc S1,D1

Opcode:



Instruction Fields:

cc=4=bit condition code=CCCC

S1,D1	J J J D	Mnemonic	C C C C	Mnemonic	C C C C
B,A	0 0 0 0	CC (HS)	0 0 0 0	CS (LO)	1 0 0 0
A,B	0 0 0 1	GE	0 0 0 1	LT	1 0 0 1
X0,A	1 0 0 0	NE	0 0 1 0	EQ	1 0 1 0
X0,B	1 0 0 1	PL	0 0 1 1	MI	1 0 1 1
X1,A	1 1 0 0	NN	0 1 0 0	NR	1 1 0 0
X1,B	1 1 0 1	EC	0 1 0 1	ES	1 1 0 1
Y0,A	1 0 1 0	LC	0 1 1 0	LS	1 1 1 0
Y0,B	1 0 1 1	GT	0 1 1 1	LE	1 1 1 1
Y1,A	1 1 1 0				
Y1,B	1 1 1 1				

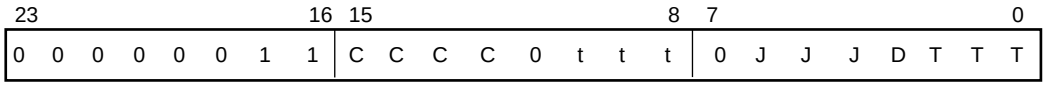
Timing: 2 oscillator clock cycles

Memory: 1 program word

Instruction Format:

Tcc S1,D1 S2,D2

Opcode:



Instruction Fields:

cc=4=bit condition code=CCCC

S1,D1	J J J D	S2	t t t	Mnemonic	C C C C	Mnemonic	C C C C
B,A	0 0 0 0	Rn	n n n	CC (HS)	0 0 0 0	CS (LO)	1 0 0 0
A,B	0 0 0 1			GE	0 0 0 1	LT	1 0 0 1
X0,A	1 0 0 0			NE	0 0 1 0	EQ	1 0 1 0
X0,B	1 0 0 1			PL	0 0 1 1	MI	1 0 1 1
X1,A	1 1 0 0	**D2**	**T T T**	NN	0 1 0 0	NR	1 1 0 0
X1,B	1 1 0 1	Rn	n n n	EC	0 1 0 1	ES	1 1 0 1
Y0,A	1 0 1 0			LC	0 1 1 0	LS	1 1 1 0
Y0,B	1 0 1 1			GT	0 1 1 1	LE	1 1 1 1
Y1,A	1 1 1 0						
Y1,B	1 1 1 1						

where ‘‘nnn’’=Rn number (R0–R7)

Timing: 2 oscillator clock cycles

Memory: 1 program word

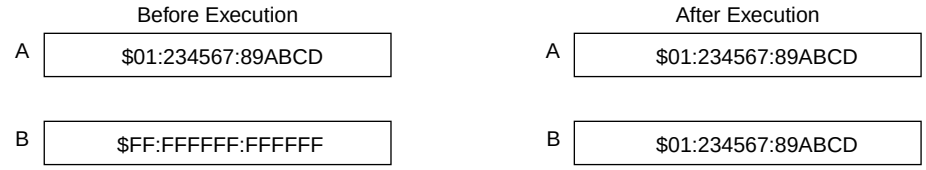
Operation:
S→D (parallel move)

Assembler Syntax:
TFR S,D (parallel move)

Description: Transfer data from the specified source data ALU register S to the specified destination data ALU accumulator D. TFR uses the internal data ALU data paths; thus, data does not pass through the data shifter/limiters. This allows the full 56-bit contents of one of the accumulators to be transferred into the other accumulator **without** data shifting and/or limiting. Moreover, since TFR uses the internal data ALU data paths, parallel moves are possible. The TFR instruction only affects the L condition code bit which can be set by data limiting associated with the instruction's **parallel move** operations.

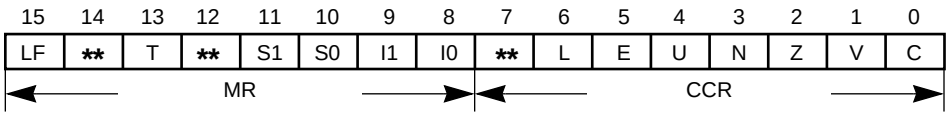
Example:

```
        :
TFR A,B  A,X1     Y:(R4+N4),Y0      ;move A to B and X1, update Y0
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $01:234567:89ABCD, and the 56-bit B accumulator contains the value $ff:FFFFFF:FFFFFF. The execution of the TFR A,B instruction moves the 56-bit value in the A accumulator into the 56-bit B accumulator using the internal data ALU data paths without any data shifting and/or limiting. The value in the B accumulator **would** have been limited if a MOVE A,B instruction had been used. Note, however, that the **parallel move** portion of the TFR instruction **does** use the data shifter/limiters. Thus, the value stored in the 24-bit X1 register (not shown) **would** have been limited in this example. This example illustrates a **triple** move instruction.

Condition Codes:

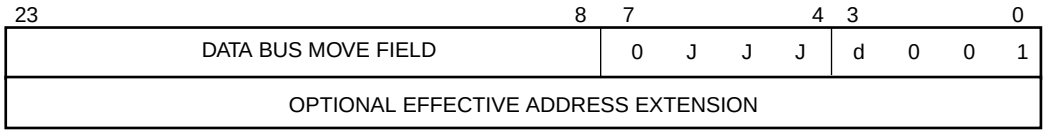


L — Set if data limiting has occurred during parallel move

Instruction Format:

TFR S,D

Opcode:



Instruction Fields:

S,D	J	J	J	D
B,A	0	0	0	0
A,B	0	0	0	1
X0,A	1	0	0	0
X0,B	1	0	0	1
X1,A	1	1	0	0
X1,B	1	1	0	1
Y0,A	1	0	1	0
Y0,B	1	0	1	1
Y1,A	1	1	1	0
Y1,B	1	1	1	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
S–0 (parallel move)

Assembler Syntax:
TST S (parallel move)

Description: Compare the specified source accumulator S with zero and set the condition codes accordingly. No result is stored although the condition codes are updated.

Example:

```
        :
TST A   #$345678,B          ;set CCR bits for value in A, update B
        :
```

	Before Execution	After Execution
A	$01:020304:000000	$01:020304:000000
CCR	$0300	$0330

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $01:020304:000000, and the 16-bit condition code register contains the value $0300. The execution of the TST A instruction compares the value in the A register with zero and updates the condition code register accordingly. The contents of the A accumulator are not affected.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	**	T	**	S1	S0	I1	I0	**	L	E	U	N	Z	V	C
←			MR				→	←			CCR			→	

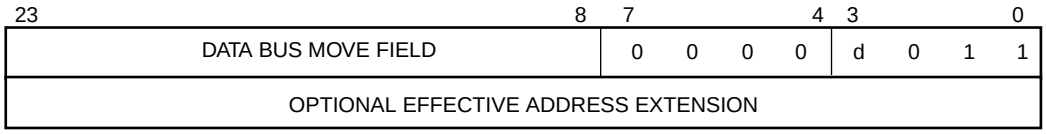
L — Set if data limiting has occurred during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Always cleared

Note: The definition of the E and U bits varies according to the scaling mode being used. Refer to A.4 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

TST S

Opcode:



Instruction Fields:

S	d
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

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. If WAIT is executed when an interrupt is pending, the interrupt will be processed; the effect will be the same as if the processor never entered the WAIT state and three NOPs followed the WAIT instruction. When an unmasked interrupt or external (hardware) processor RESET occurs, the processor leaves the WAIT state and begins exeption processing of the unmasked interrupt or RESET condition. The BR/BG circuits remain active during the WAIT state. The WAIT state is a low-power standby state. The processor always leaves the WAIT state in the T2 clock phase (see the DSP56001 Advance Information Data Sheet (ADI1290)). Therefore, multiple processors may be synchronized by having them all enter the WAIT state and then interrupting them with a common interrupt.

Restrictions: A WAIT instruction cannot be used in a fast interrupt routine.

A WAIT instruction cannot be the **last** instruction in a DO loop (at LA).

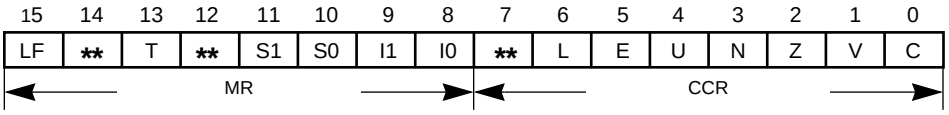
A WAIT instruction cannot be repeated using the REP instruction.

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.

Condition Codes:

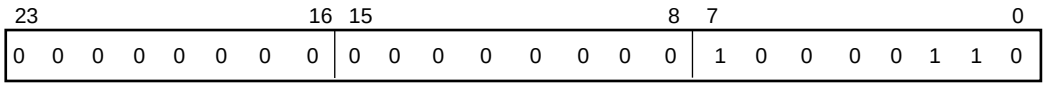


The condition codes are not affected by this instruction.

Instruction Format:

WAIT

Opcode:



Instruction Fields:

None

Timing: The WAIT instruction takes a minumum of 16 cycles to execute when an internal interrupt is pending during the execution of the WAIT instruction

Memory: 1 program word

A.7 INSTRUCTION TIMING

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

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

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

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

All tables are based on the following assumptions:

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

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

3. There is no contention for **instruction** fetches. Thus, external program instruction fetches are assumed not to have to contend with external data memory accesses.

4. There are no wait states for **instruction** fetches done sequentially (as for non-change-of-flow instructions), but they are taken into account for change-of-flow instructions which flush the pipeline such as JMP, Jcc, RTI, etc.

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

Example 1: Arithmetic Instruction with Two Parallel Moves

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

MACR –X0,X0,A X1,X:(R6)– Y0,Y:(R0)+

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

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

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

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

2. Evaluate the ‘‘mv’’ term using Table A-7.

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

3. Evaluate the ‘‘ea’’ term using Table A-13.

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

4. Evaluate the ‘‘axy’’ term using Table A-14.

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

5. Compute final results.

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

```
MACR –X0,X0,A      X1,X:(R6)–      Y0,Y:(R0)+
```

will require

```
(1+mv)
=(1+0)
=1                  instruction program word
```

and will execute in

```
=(2+mv)
=(2+ea+axy)
=(2+ea+wy)
=(2+0+1)            oscillator clock cycles.
=    3
```

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

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

Example 2: Jump Instruction

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

JLC (R2+N2)

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

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

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

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

2. Evaluate the ‘‘jx’’ term using Table A-11.

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

3. Evaluate the ‘‘ea’’ term using Table A-13.

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

4. Evaluate the ‘‘ap’’ term using Table A-14.

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

5. Compute final results.

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

JLC (R2+N2)

will require

=(1+ea)
=(1+0)
= 1 instruction program word

and will execute in

=(4+jx)
=(4+ea+(2 ∗ ap))
=(4+ea+(2 ∗ wp))
=(4+2+(2 ∗ 4)) oscillator clock cycles.
= 14

Example 3: RTI Instruction

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

RTI

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

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

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

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

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

2. Evaluate the ‘‘rx’’ term using Table A-12.

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

3. Evaluate the ‘‘ap’’ term using Table A-14.

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

4. Compute final results.

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

RTI

will require

1 instruction program word

and will execute in

(4+rx)
=(4+(2 ∗ ap))
=(4+(2 ∗ 0))
= 4 oscillator clock cycles

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

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

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

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

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

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

Table A-7 Parallel Data Move Timing

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

Note 1: The ax or ay term does not apply to MOVE IMMEDIATE DATA.
Note 2: The ea term does not apply to ABSOLUTE ADDRESS and IMMEDIATE DATA.

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

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

Note 1: The ax or ay term does not apply to MOVE IMMEDIATE DATA.
Note 2: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ ap" term should be added, and to each two-word instruction, a "+ (2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

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

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

Note 1: The ax or ay term does not apply to MOVE IMMEDIATE DATA.
Note 2: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ ap" term should be added, and to each two-word instruction, a "+ (2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

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

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

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

Note 1: Bxxx = BCHG, BCLR, or BSET.
Note 2: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ ap" term should be added, and to each two-word instruction, a "+ (2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

Table A-11 Jump Instruction Timing Summary

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

Note 1: Jbit = JCLR, JSCLR, JSET , and JSSET
Note 2: Jxxx = Jcc, JMP, JScc, and JSR

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

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

Table A-12 RTI/RTS Timing Summary

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

The term ‘‘2 ∗ ap’’ come from the two instruction fetches done by the RTI/RTS instruction to refill the pipeline.

Table A-13 Addressing Mode Timing Summary

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

Table A-14 Memory Access Timing Summary

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

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

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

A.8 INSTRUCTION SEQUENCE RESTRICTIONS

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

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

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

A.8.1 Restrictions Near the End of DO Loops

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

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

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

---

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

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

A.8.2 Other DO Restrictions

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

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

A.8.3 ENDDO Restrictions

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

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

A.8.4 RTI and RTS Restrictions

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

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

A.8.5 SP and SSH/SSL Manipulation Restrictions

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

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

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

Also, the instruction MOVEC SSH,SSH is illegal.

A.8.6 R, N, and M Register Restrictions

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

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

A.8.7 Fast Interrupt Routines

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

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

A.8.8 REP Restrictions

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

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

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

A.9 INSTRUCTION ENCODING

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

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

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

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

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

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

A.9.1 Partial Encodings for Use in Instruction Encoding

Table A-15 Single-Bit Register Encodings

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

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

Table A-16 Single-Bit Special Register Encodings

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

Table A-17 Double-Bit Register Encodings

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

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

Table A-18 Triple-Bit Register Encodings

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

* Reserved

Where: DDD : 8 accumulators in data ALU
LLL: 8 extended-precision registers in data ALU; LLL field is encoded as L0LL
FFF: 8 address modifier registers in address ALU
NNN: 8 address offset registers in address ALU
TTT: 8 address registers in address
FFF: 8 program controller registers

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

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

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

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

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

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

Where: eeeee = source
ddddd = destination

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

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

Table A-22 Write Control Encoding

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

Table A-23 Memory Space Bit Encoding

S	Operation
0	X Memory
1	Y Memory

Table A-24 Program Controller Register Encoding

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

Table A-25 Condition Code and Address Encoding

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

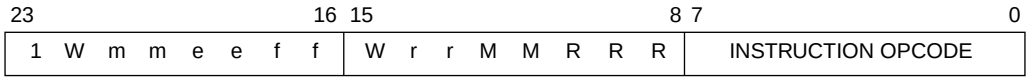
Table A-26 Effective Addressing Mode Encoding

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

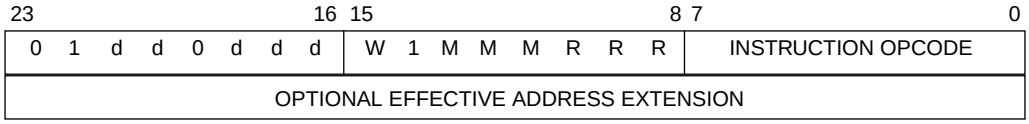
RRR = three unencoded bits R0, R!, R2
MMM = three unencoded bits M0, m1, m2
NOTES:
(1) R2 is 0 for low register bank and 1 for the high register bank.
(2) M2 is 0 for all post update modes and 1 otherwise.
(3) M1 is 0 for update by register offset and 1 for update by one.
(4) M0 is 0 for minus and 1 for plus.
(5) For X and Y moves, rr is a subfield or rrr with equations: r2: = $\overline{R2}$.
(6) For rr field, r1 is bit 14; r0 is bit 13.
(7) For X and Y moves, mm is a subfield of mmm with equations: M2: = ($\overline{M1 \vee M0}$) m2: = ($\overline{m1 \vee m0}$).
(8) For mm field, m1 is bit 21; m0 is bit 20. For MM field, M1 is bit 12; M0 is bit 11.

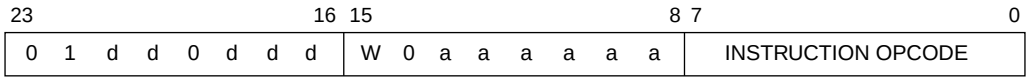
A.9.2 Instruction Encoding for the Parallel Move Portion of an Instruction

X: Y: Parallel Data Move

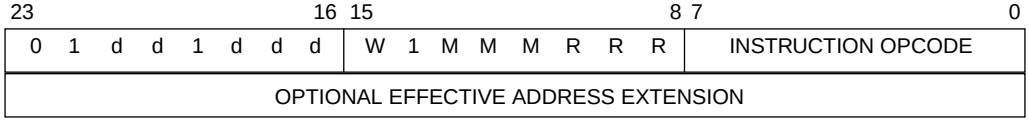


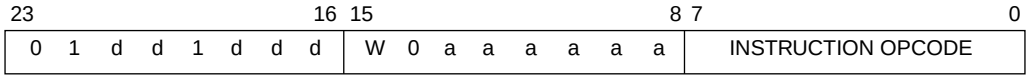
X: Parallel Data Move



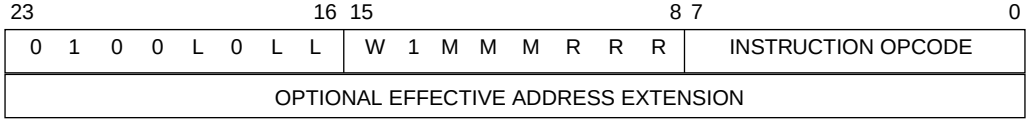


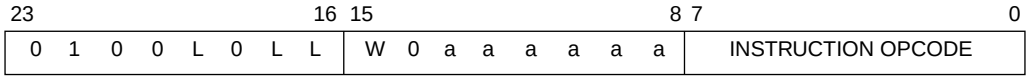
Y: Parallel Data Move



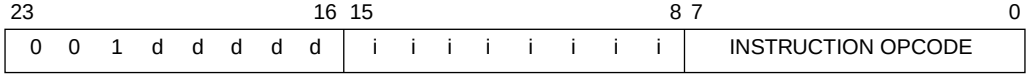


L: Parallel Data Move

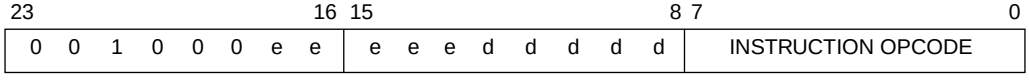




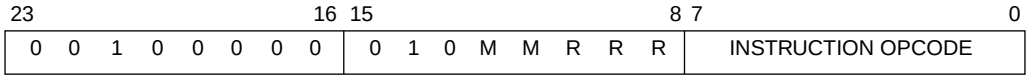
I: Immediate Short Parallel Data Move



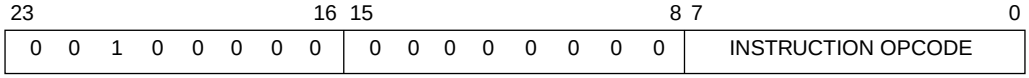
R: Register to Register Parallel Data Move



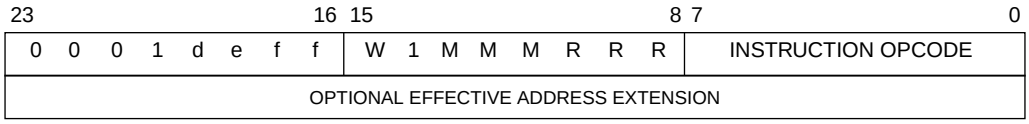
U: Address Register Update Parallel Data Move



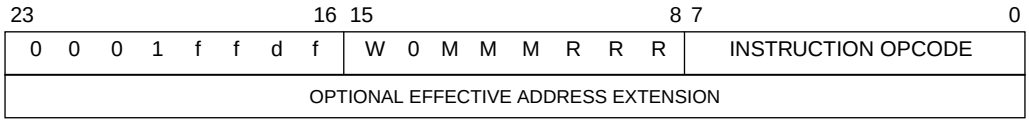
Parallel Data Move NOP



R:Y Parallel Data Move



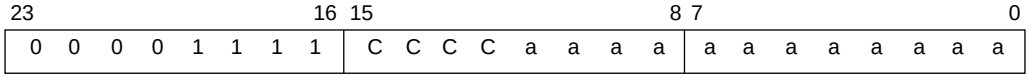
X:R Parallel Data Move



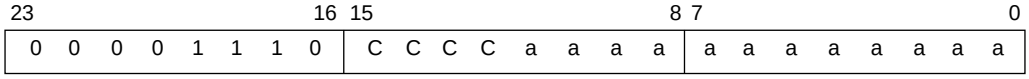
A.9.3 Instruction Encoding for the Parallel Move Portion of an Instruction

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

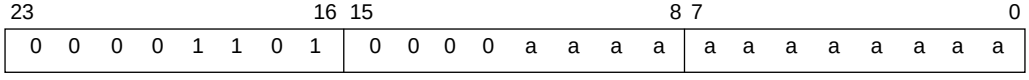
JScc xxx



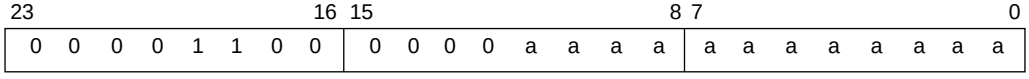
Jcc xxx



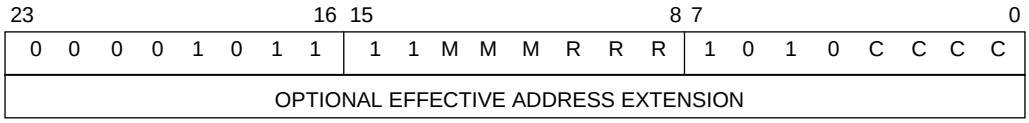
JSR xxx



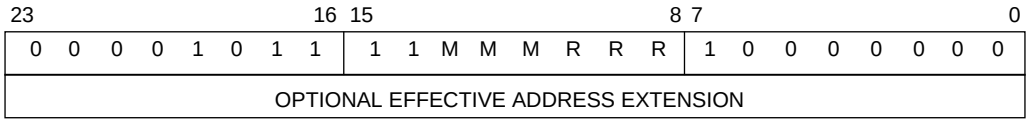
JMP xxx



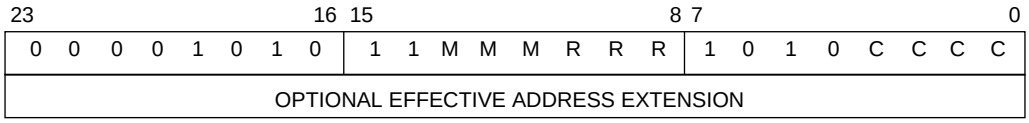
JScc ea



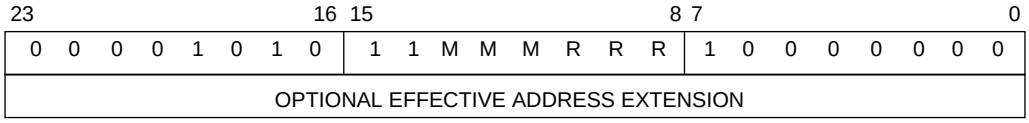
JSR ea



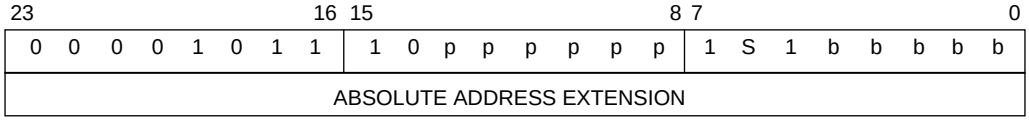
Jcc ea



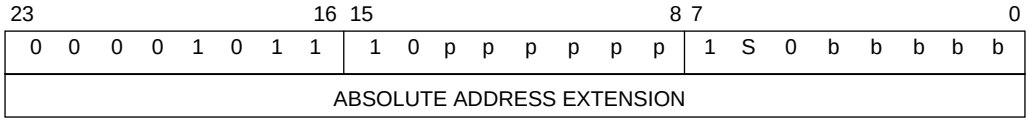
JMP ea



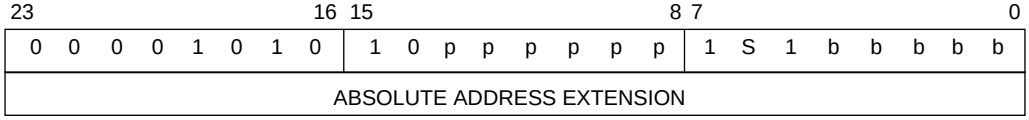
JSSET #n,X:pp,xxxx
JSSET #n,Y:pp,xxxx



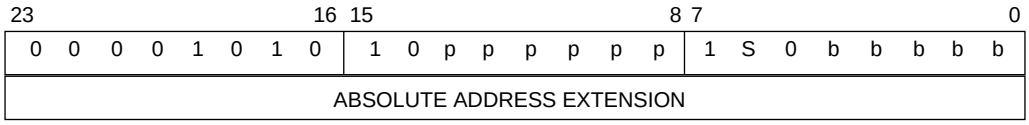
JSCLR #n,X:pp,xxxx
JSCLR #n,Y:pp,xxxx



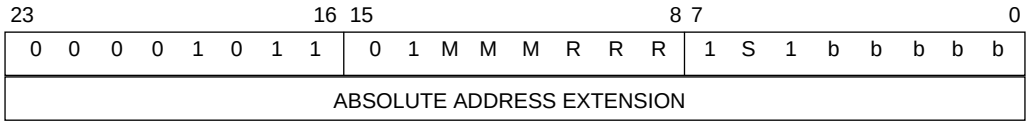
JSET #n,X:pp,xxxx
JSET #n,Y:pp,xxxx



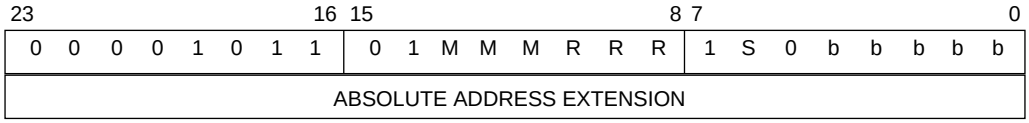
JCLR #n,X:pp,xxxx
JCLR #n,Y:pp,xxxx



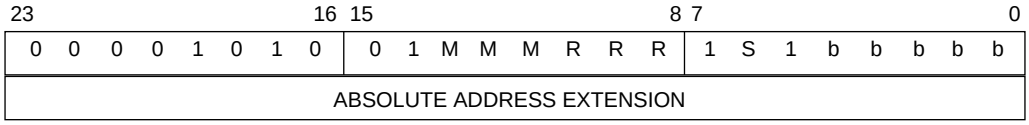
JSSET #n,X:ea,xxxx
JSSET #n,Y:ea,xxxx



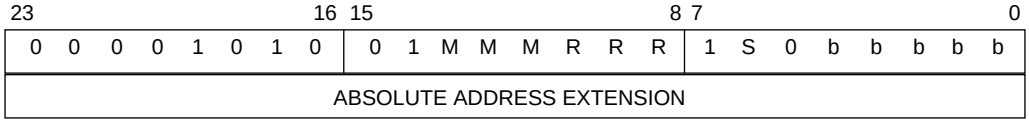
JSCLR #n,X:ea,xxxx
JSCLR #n,Y:ea,xxxx



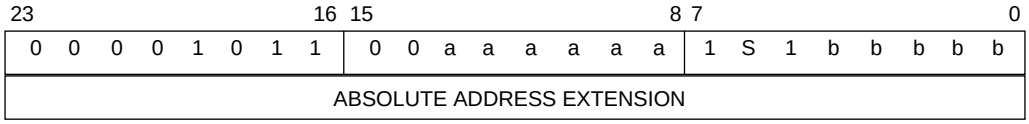
JSET #n,X:ea,xxxx
JSET #n,Y:ea,xxxx



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

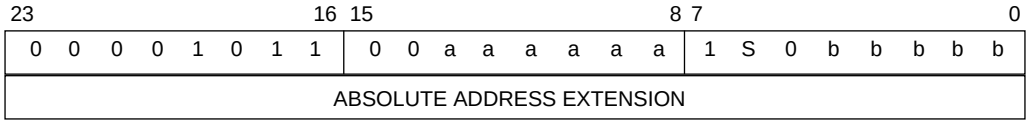


JSSET #n,X:aa,xxxx
JSSET #n,Y:aa,xxxx

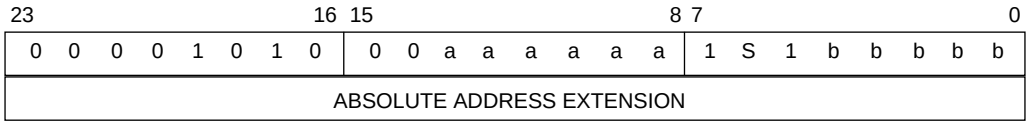


JSCLR #n,X:aa,xxxx

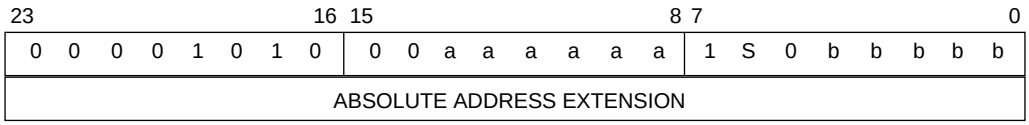
JSCLR #n,Y:aa,xxxx



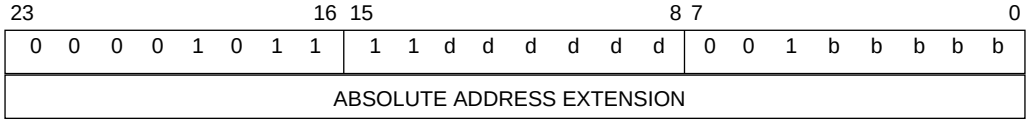
JSET #n,X:aa,xxxx
JSET #n,Y:aa,xxxx



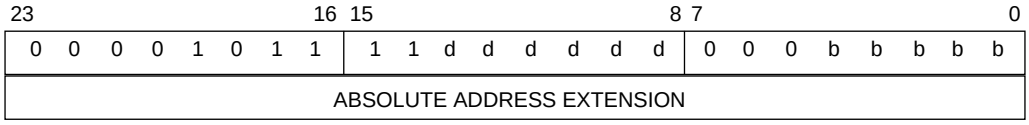
JCLR #n,X:aa,xxxx
JCLR #n,Y:aa,xxxx



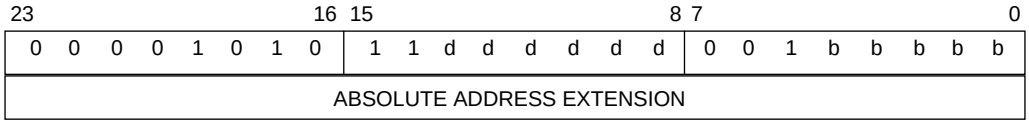
JSSET #n,S,xxxx



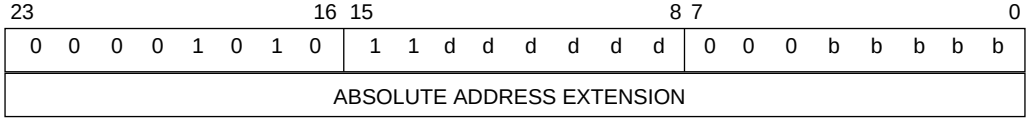
JSCLR #n,S,xxxx



JSET #n,S,xxxx

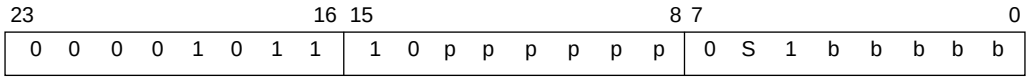


JCLR #n,S,xxxx

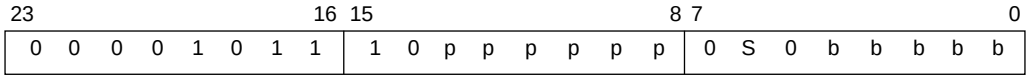


BTST #n,X:pp

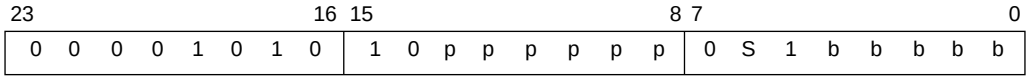
BTST #n,Y:pp



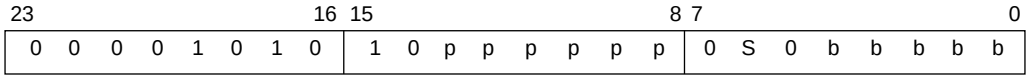
BCHG #n,X:pp
BCHG #n,Y:pp



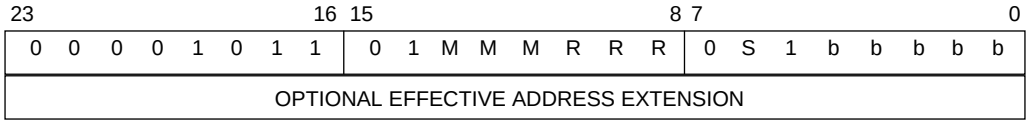
BSET #n,X:pp
BSET #n,Y:pp



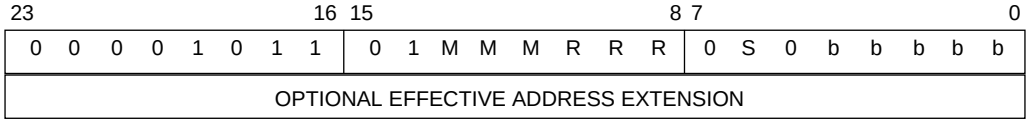
BCLR #n,X:pp
BCLR #n,Y:pp



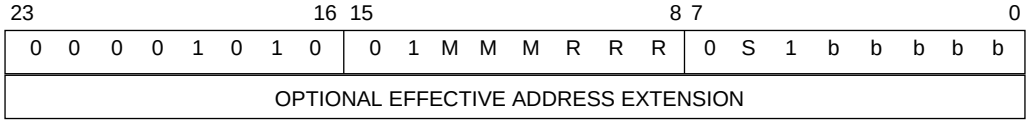
BTST #n,X:ea
BTST #n,Y:ea



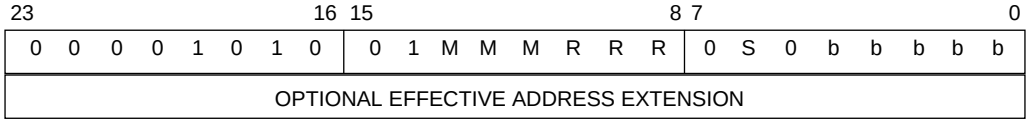
BCHG #n,X:ea
BCHG #n,Y:ea



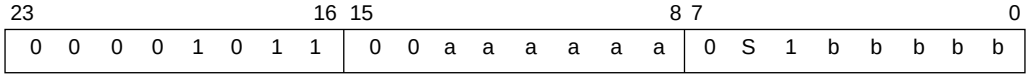
BSET #n,X:ea
BSET #n,Y:ea



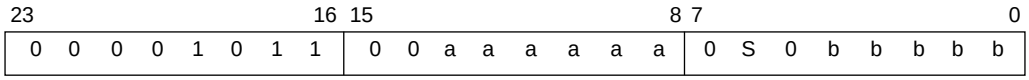
BCLR #n,X:ea
BCLR #n,Y:ea



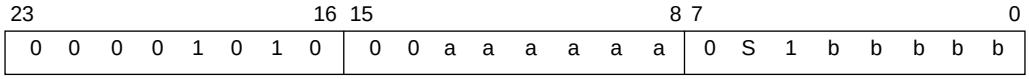
BTST #n,X:aa
BTST #n,Y:aa



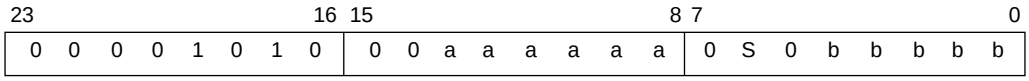
BCHG #n,X:aa
BCHG #n,Y:aa



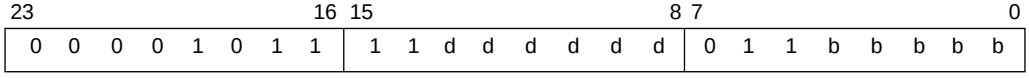
BSET #n,X:aa
BSET #n,Y:aa



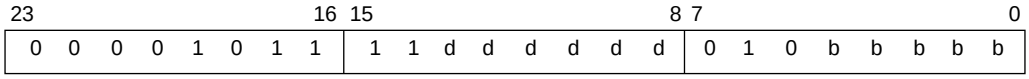
BCLR #n,X:aa
BCLR #n,Y:aa



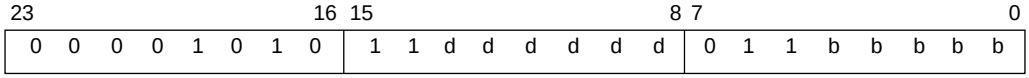
BTST #n,D



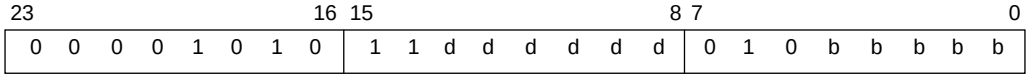
BCHG #n,D



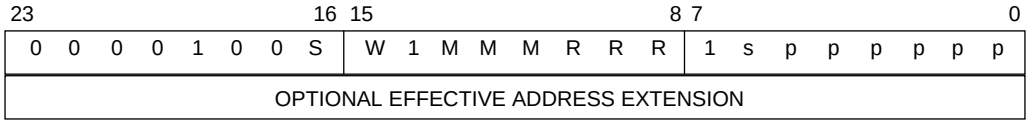
BSET #n,D



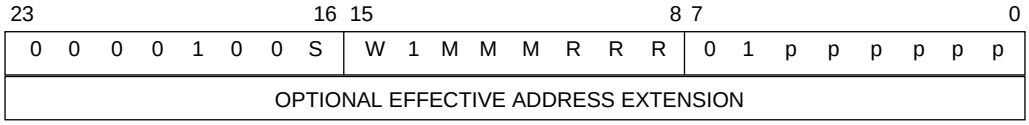
BCLR #n,D



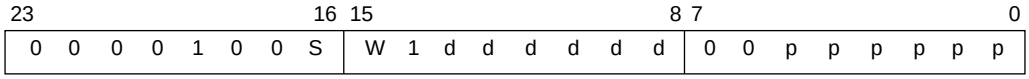
MOVEP X:ea,X:pp
MOVEP Y:ea,X:pp
MOVEP #xxxxxx,X:pp
MOVEP X:pp,X:ea
MOVEP X:pp,Y:ea
MOVEP X:ea,Y:pp
MOVEP Y:ea,Y:pp
MOVEP #xxxxxx,Y:pp
MOVEP Y:pp,X:ea
MOVEP Y:pp,Y:ea



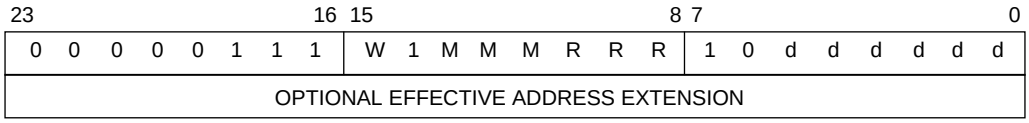
MOVEP P:ea,X:pp
MOVEP X:pp,P:ea
MOVEP P:ea,Y:pp
MOVEP Y:pp,P:ea



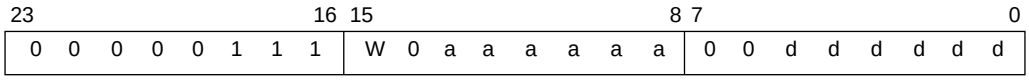
MOVEP S,X:pp
MOVEP X:pp,D
MOVEP S,Y:pp
MOVEP Y:pp,D



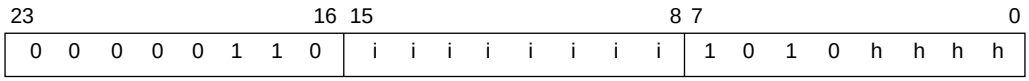
MOVE(M) S,P:ea
MOVE(M) P:ea,D



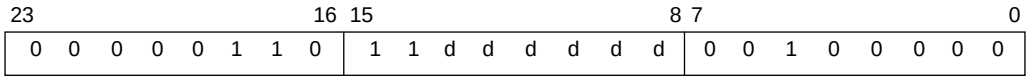
MOVE(M) S,P:aa
MOVE(M) P:aa,D



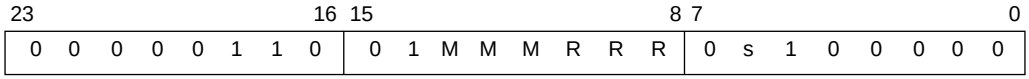
REP #xxx



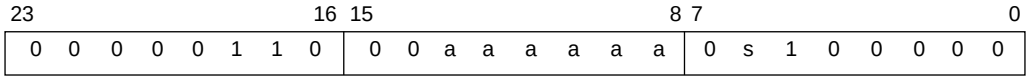
REP S



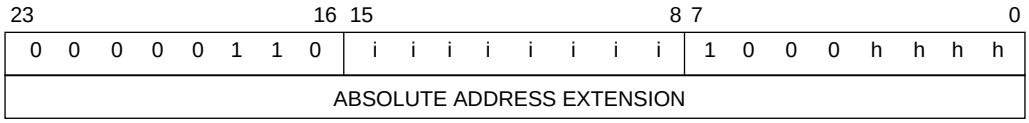
REP X:ea
REP Y:ea



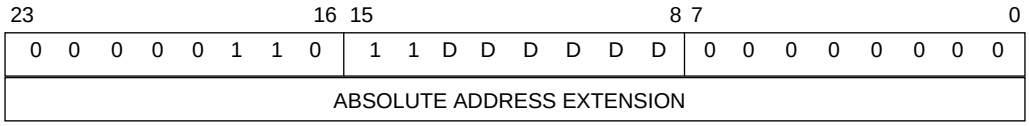
REP X:aa
REP Y:aa



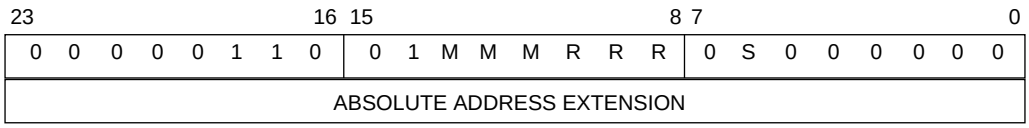
DO #xxx,expr



DO S,expr

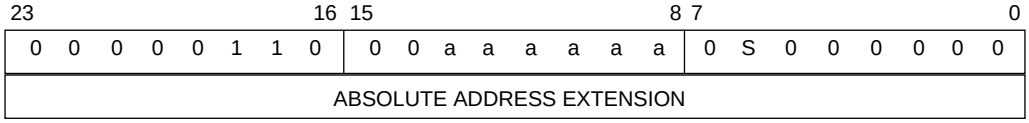


DO X:ea,expr
DO Y:ea,expr

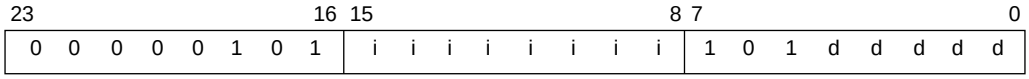


DO X:aa,expr

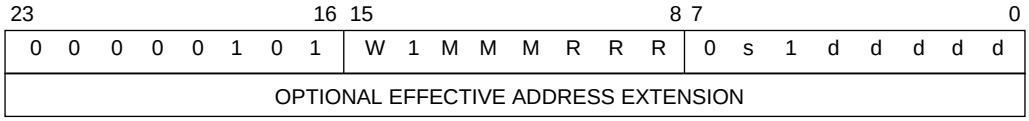
DO Y:aa,expr



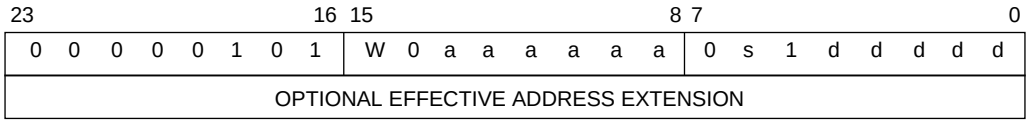
MOVE(C) #xx,D1



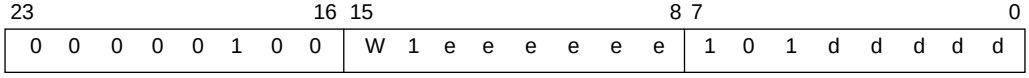
MOVE(C) X:ea,D1
MOVE(C) S1,X:ea
MOVE(C) Y:ea,D1
MOVE(C) S1,Y:ea
MOVE(C) #xxxx,D1



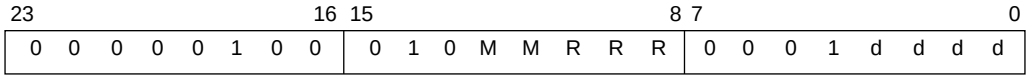
MOVE(C) X:aa,D1
MOVE(C) S1,X:aa
MOVE(C) Y:aa,D1
MOVE(C) S1,Y:aa



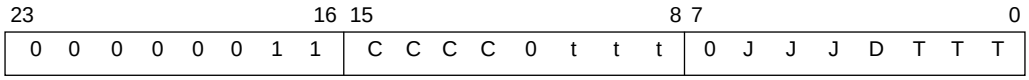
MOVE(C) S1,D2
MOVE(C) S2,D1



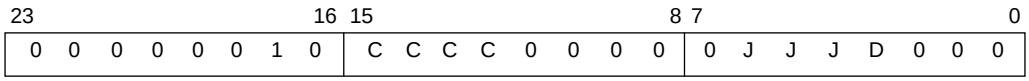
LUA ea,D



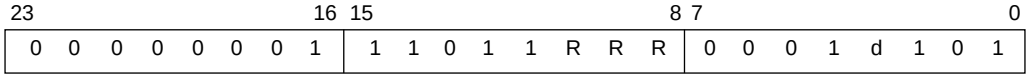
Tcc S1,D1 S2,D2



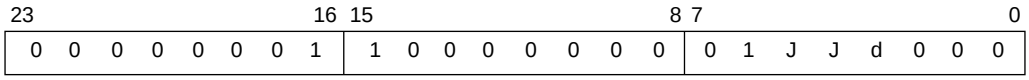
Tcc S1,D1



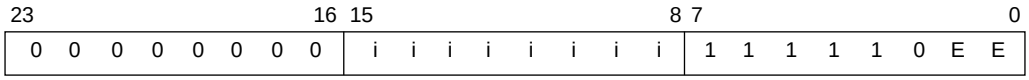
NORM Rn,D



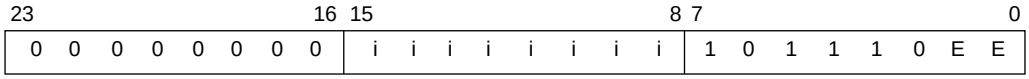
DIV S,D



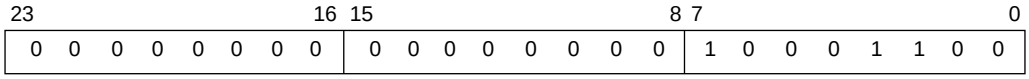
OR(I) #xx,D



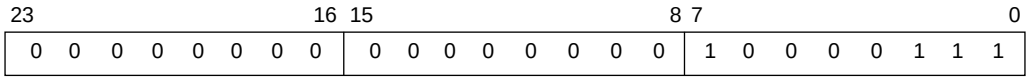
AND(I) #xx,D



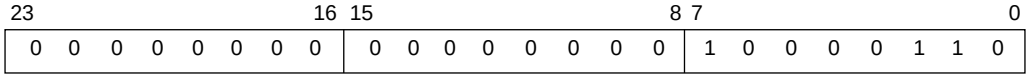
ENDDO



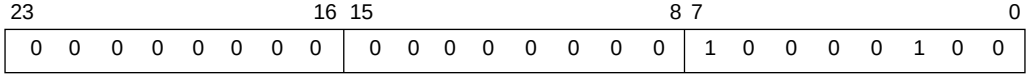
STOP



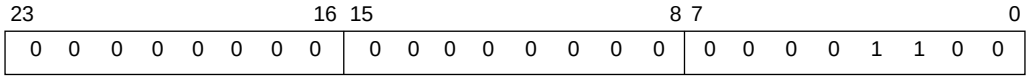
WAIT



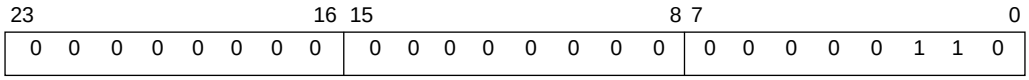
RESET



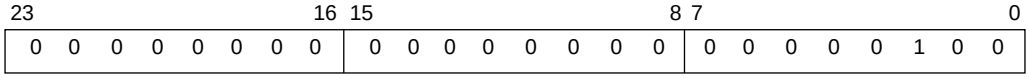
RTS



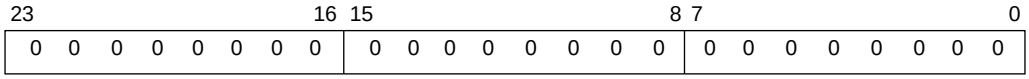
SWI



RTI



NOP



A.9.4 Parallel Instruction Encoding of the Operation Code

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

Multiply Instruction Encoding

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

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

Table A-27 Operation Code K0-2 Decode

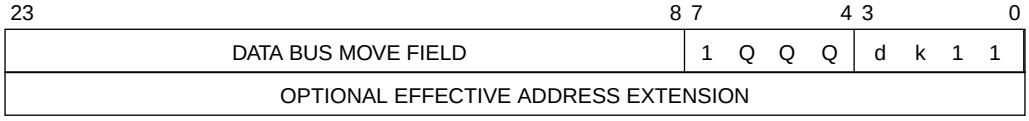
Code	k2	k1	k0
0	positive	mpy only	don't round
1	negative	mpy and acc	round

Table A-28 Operation Code QQQ Decode

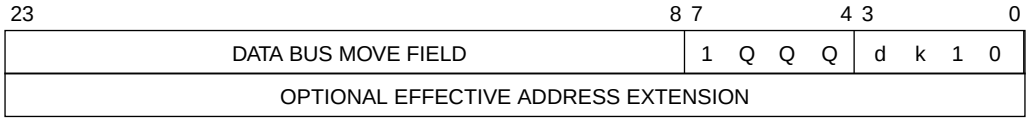
Q	Q	Q	S1	S2
0	0	0	X0	X0
0	0	1	Y0	Y0
0	1	0	X1	X0
0	1	1	Y1	Y0
1	0	0	X0	Y1
1	0	1	Y0	X0
1	1	0	X1	Y0
1	1	1	Y1	X1

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

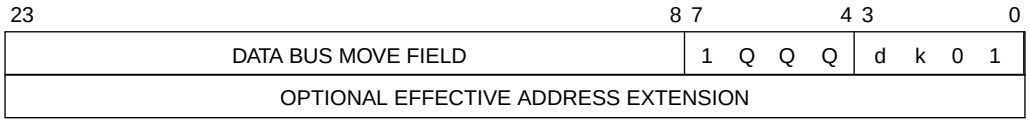
MACR (±) S1,S2,D
MACR (±) S2,S1,D



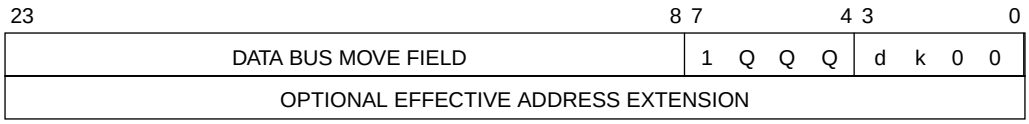
MAC (±) S1,S2,D
MAC (±) S2,S1,D



MPYR (±) S1,S2,D
MPYR (±) S2,S1,D



MPY (±) S1,S2,D
MPY (±) S2,S1,D



Nonmultiply Instruction Encoding

The 8-bit operation code for instructions allowing parallel moves contains two 3-bit fields defining which instruction the operation code represents and one bit defining the destination accumulator register.

The 8-bit operation code = **0JJJ Dkkk** where JJJ=1/2 instruction number
kkk=1/2 instruction number
D=0 → A
D=1 → B

Table A-29 Nonmultiply Instruction Encoding

JJJ	D = 0 Src Oper	D = 1 Src Oper	kkk							
			000	001	010	011	100	101	110	111
000	B	A	MOVE[1]	TFR	ADDR	TST	*	CMP	SUBR	CMPM
001	B	A	ADD	RND	ADDL	CLR	SUB	*	SUBL	NOT
010[2]	B	A	—	—	ASR	LSR	—	—	ABS	ROR
011[2]	B	A	—	—	ASL	LSL	—	—	NEG	ROL
010[2]	X1X0	X1X0	ADD	ADC	—	—	SUB	SBC		
011[2]	Y1Y0	Y1Y0	ADD	ADC	—	—	SUB	SBC		
100	X0_0	X0_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
101	Y0_0	Y0_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
110	X1_0	X1_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
111	Y1_0	Y1_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM

NOTES:
***** = Reserved
1 = Special Case #1 (See Table A - 30)
2 = Special Case #2 (See Table A - 31)

Table A-30 Special Case #1

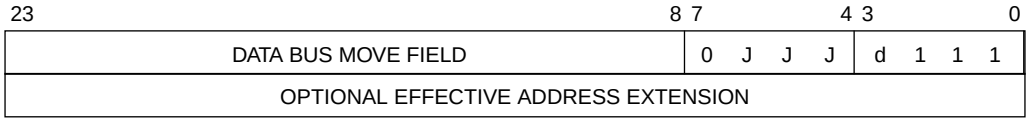
O	P	E	R	C	O	D	E	Operation
0	0	0	0	0	0	0	0	MOVE
0	0	0	0	1	0	0	0	Reserved

For JJJ=010 and 011, k1 qualifies source register selection:

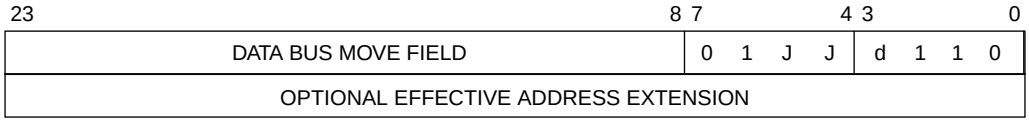
Table A-31 Special Case #2

0	J	J	J	d	k	k	k	Operation
0	0	1	0	x	x	0	x	Selects X1X0
0	0	1	1	x	x	0	x	Selects Y1Y0
0	0	1	x	x	x	1	x	Selects A/B

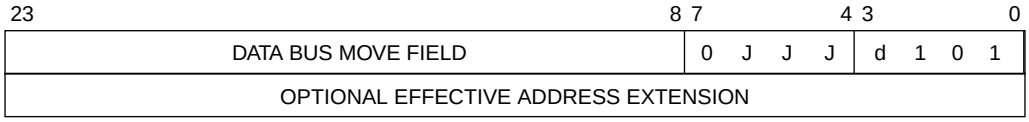
CMPM S1,S2



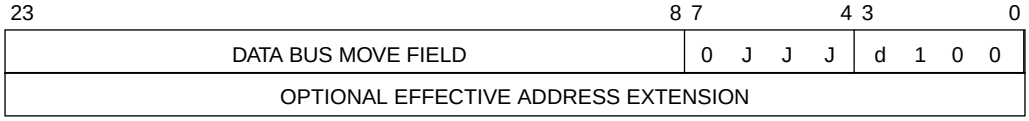
AND S,D



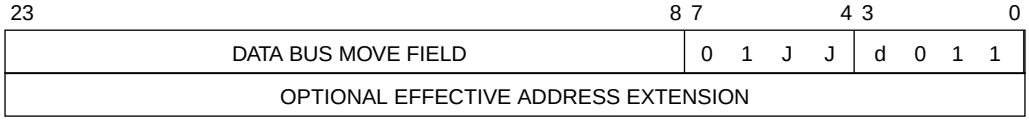
CMP S1,S2



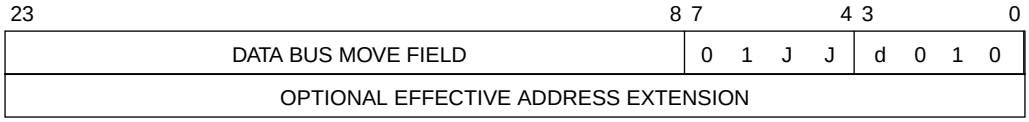
SUB S,D



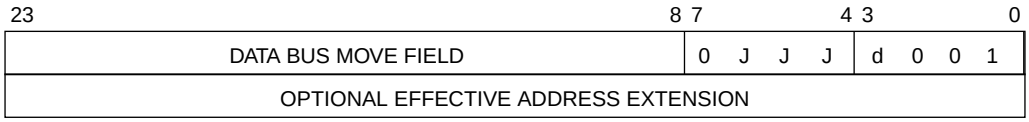
EOR S,D



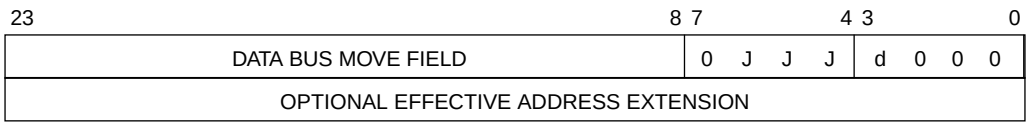
OR S,D



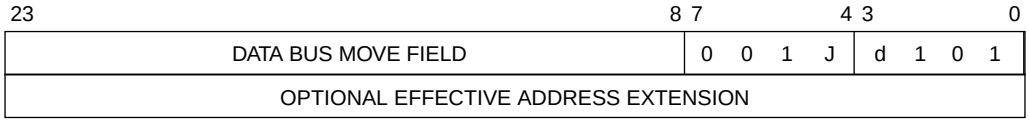
TFR S,D



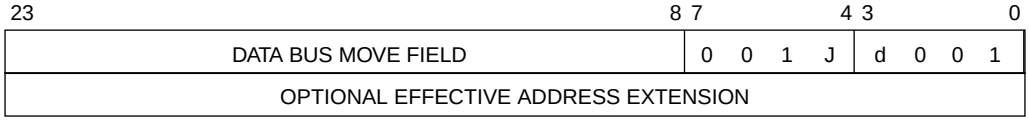
ADD S,D



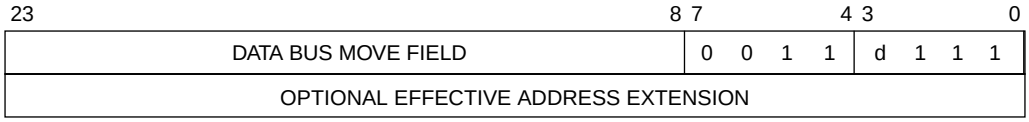
SBC S,D



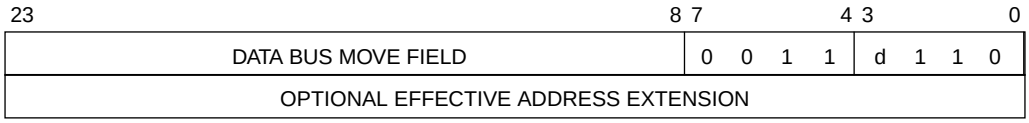
ADC S,D



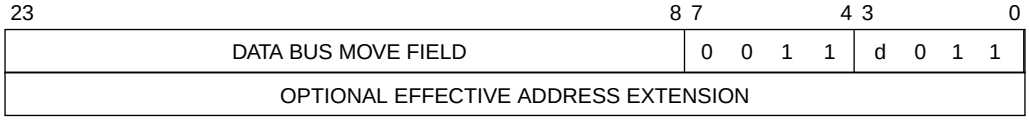
ROL D



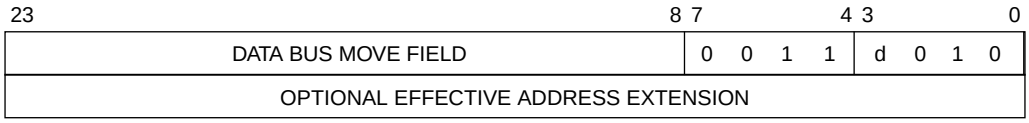
NEG D



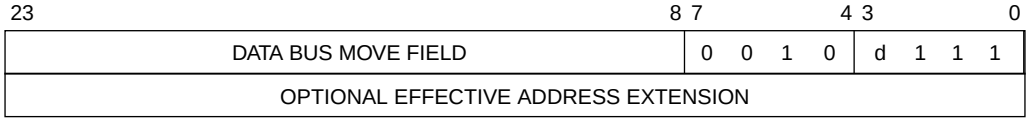
LSL D



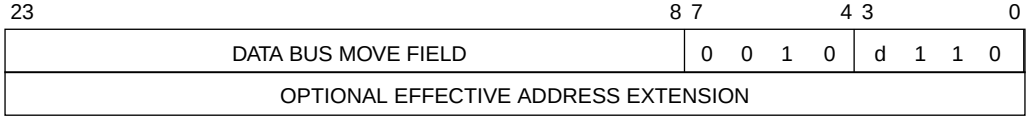
ASL D



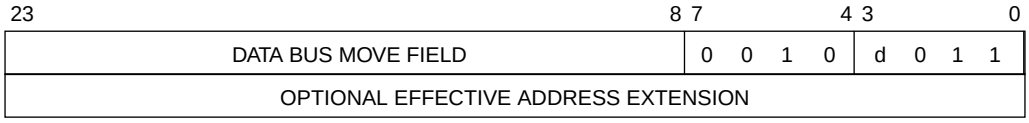
ROR D



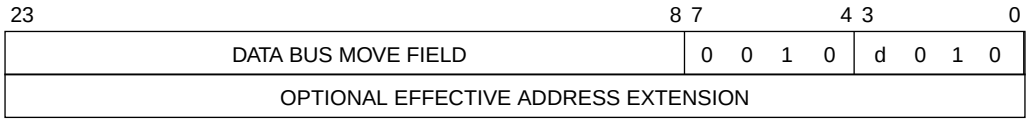
ABS D



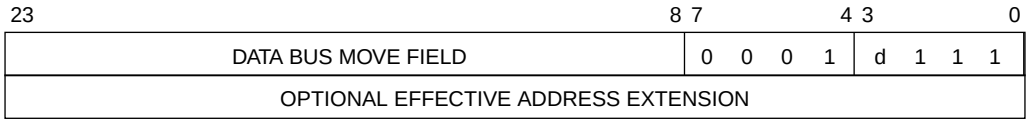
LSR D



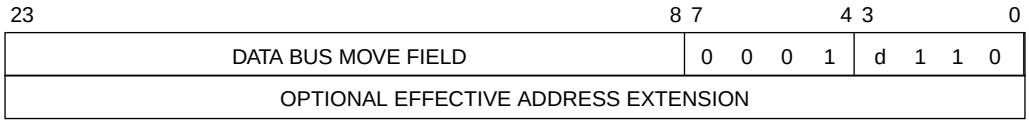
ASR D



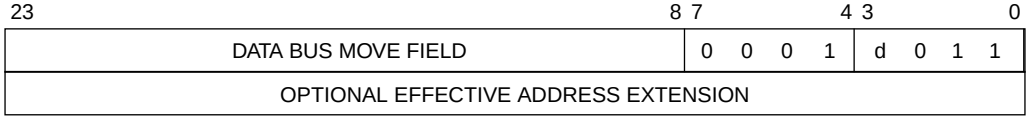
NOT D



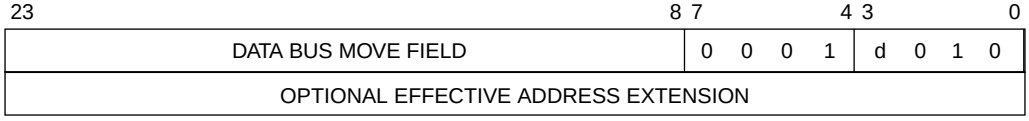
SUBL S,D



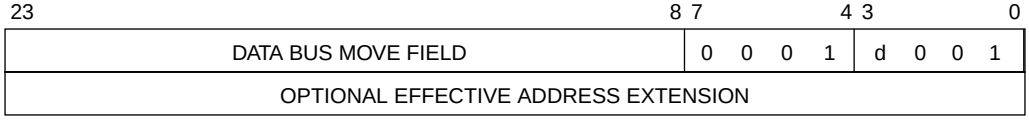
CLR D



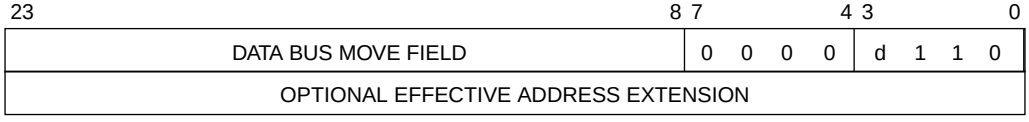
ADDL S,D



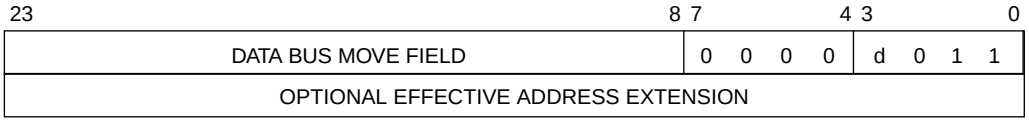
RND D



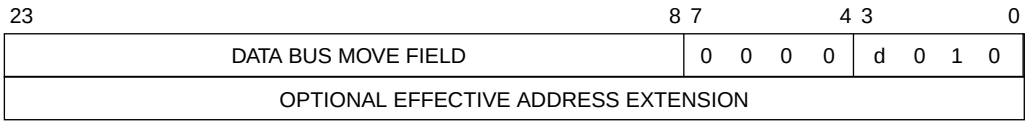
SUBR S,D



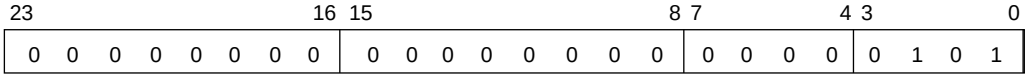
TST D



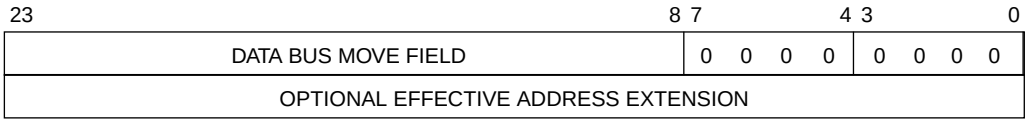
ADDR S,D



ILLEGAL



MOVE S,D



APPENDIX B
BENCHMARK PROGRAMS

Table B-1 and Table B-2 provide benchmark numbers for 18 common DSP programs. The two tables are identical except that Table B-1 is for the 20.5-MHz DSP56001 and Table B-2 is for the 27-MHz DSP56001. The following four code examples (Figures B-1 to B-4) are representative of the benchmark programs shown in Tables B-1 and B-2. The code for these and other programs is free and available through the Dr. BuB electronic bulletin board. Figure B-1 is the code for the 20-tap FIR filter shown in Tables B-1 and B-2. Figure B-2 is the code for an FFT using a triple nested DO LOOP. Although this code is easier to understand and very compact, it is not as fast as the code used for the benchmarks shown in Tables B-1 and B-2, which are highly optimized using the symmetry of the FFT and the parallelism of the DSP. Figure B-3 is the code for the 8-pole cascaded canonic biquad IIR filter, which uses four coefficients (see Tables B-1 and B-2). Figure B-4 is the code for a 2N delayed least mean square (LMS) FIR adaptive filter, which is useful for echo cancelation and other adaptive filtering applications.

Table B-1 20.5-MHz Benchmark Results for the DSP56001R20

Benchmark Program	Sample Rate (Hz) or Execution Time	Memory Size (Words)	Number of Clock Cycles
20 - Tap FIR Filter	379.6 kHz	50	54
64 - Tap FIR Filter	144.4 kHz	138	142
67 - Tap FIR Filter	138.5 kHz	144	148
8 - Pole Cascaded Canonic Biquad IIR Filter (4x)	410.0 kHz	40	50
8 - Pole Cascaded Canonic Biquad IIR Filter (5x)	353.5 kHz	45	58
8 - Pole Cascaded Transpose Biquad IIR Filter	292.9 kHz	48	70
Dot Product	585.4 ns	10	12
Matrix Multiply 2x2 times 2x2	2.049 µs	33	42
Matrix Multiply 3x3 times 3x1	1.659 µs	29	34
M - to - M FFT 64 Point	129.5 µs	489	2655
M - to - M FFT 256 Point	645.1 µs	1641	13255
M - to - M FFT 1024 Point	3.231 ms	6793	66240
P - to - M FFT 64 Point	121.9 µs	704	2499
P - to - M FFT 256 Point	458.2 µs	2048	9394
P - to - M FFT 1024 Point	1.958 ms	7424	40144

Table B-2 27-MHz Benchmark Results for the DSP56001R27

Benchmark Program	Sample Rate (Hz) or Execution Time	Memory Size (Words)	Number of Clock Cycles
20 - Tap FIR Filter	500.0 kHz	50	54
64 - Tap FIR Filter	190.1 kHz	138	142
67 - Tap FIR Filter	182.4 kHz	144	148
8 - Pole Cascaded Canonic Biquad IIR Filter (4x)	540.0 kHz	40	50
8 - Pole Cascaded Canonic Biquad IIR Filter (5x)	465.5 kHz	45	58
8 - Pole Cascaded Transpose Biquad IIR Filter	385.7 kHz	48	70
Dot Product	444.4 ns	10	12
Matrix Multiply 2x2 times 2x2	1.556 µs	33	42
Matrix Multiply 3x3 times 3x1	1.259 µs	29	34
M-to-M FFT 64 Point	98.33 µs	489	2655
M-to-M FFT 256 Point	489.8 µs	1641	13255
M-to-M FFT 1024 Point	2.453 ms	6793	66240
P-to-M FFT 64 Point	92.56 µs	704	2499
P-to-M FFT 256 Point	347.9 µs	2048	9394
P-to-M FFT 1024 Point	1.489 ms	7424	40144

```
    page 132,66,0,6
    opt    rc
;*********************************************************
;Motorola Austin DSP Operation      June 30, 1988
;*********************************************************
;DSP56000/1
;20 - tap FIR filter
;File name: 1-56.asm
;*********************************************************************************************************************
;    Maximum sample rate: 379.6 kHz at 20.5 MHz/500.0 kHz at 27.0 MHz
;    Memory Size: Prog: 4+6 words; Data: 2x20 words
;    Number of clock cycles: 54 (27 instruction cycles)
;    Clock Frequency: 20.5 MHz/27.0 MHz
;    Instruction cycle time: 97.6 ns/74.1 ns
;*********************************************************************************************************************
;    This FIR filter reads the input sample
;    from the memory location Y:input
;    and writes the filtered output sample
;    to the memory location Y:output
;
;    The samples are stored in the X memory
;    The coefficients are stored in the Y memory
;*********************************************************************************************************************
```

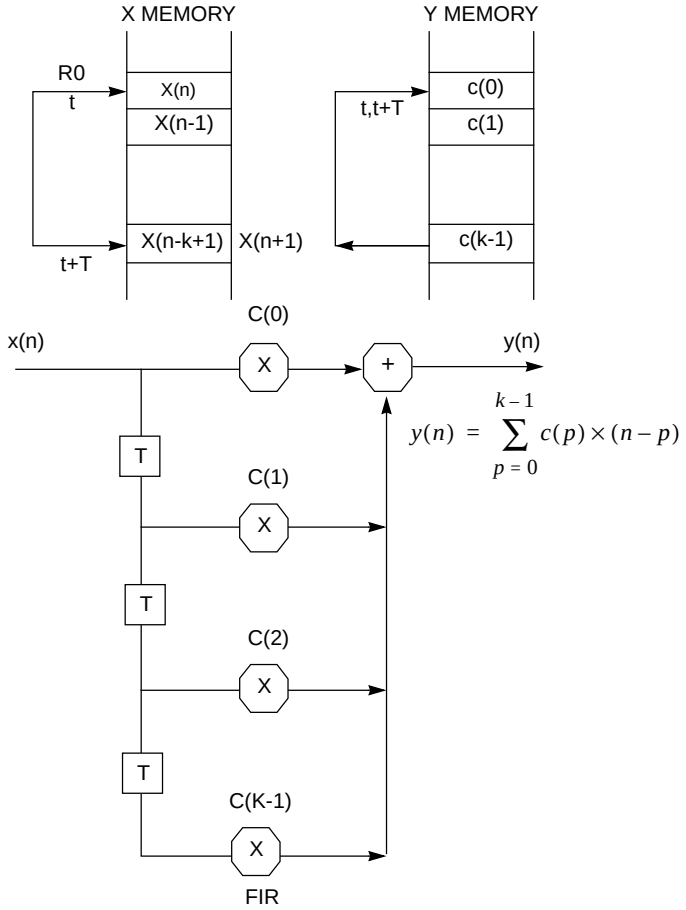


Figure B-1 20-Tap FIR Filter Example (Sheet 1 of 2)

```
;
;
;
;*****************************************************************************************************************
;
;               initialization
;****************************************
n               equ             20
start           equ             $40
wddr            equ             $0
cddr            equ             $0
input           equ             $ffe0
output          equ             $ffe1
;
                org             p:start
                move            #wddr,r0                        ;r0 ➡ samples
                move            #cddr,r4                        ;r1 ➡ coefficients
                move            #n-1,m0                         ;set modulo arithmetic
                move            m0,m4                           ;for the 2 circular buffers
;
                opt             cc
;               filter loop :8+(n-1) cycles
;*****************************************************************************************************************
                movep           y:input,x: (r0)                 ;input sample in memory
                clr             a               x:(r0)+,x0      y: (r4)+,y0

                rep             #n-1
                mac             x0,y0,a         x:(r0)+,x0      y:(r4)+,y0
                macr            x0,x0,a         (r0)-

                movep           a,y:output                      ;output filtered sample
;*****************************************************************************************************************
        end
```

Figure B-1 20-Tap FIR Filter Example (Sheet 2 of 2)

```
;This program originally available on the Motorola DSP bulletin board.
;It is provided under a DISCLAIMER OF WARRANTY available from
;Motorola DSP Operation, 6501 William Cannon Drive, Austin, TX, 78735
;
;Radix-2, In-Place, Decimation-In-Time FFT (smallest code size).
;
;Last Update 30 Sep 86        Version 1.1
;
fftr2a      macro       points,data,coef
fftr2a      ident       1,1
;
;Radix-2 Decimation-In-Time In-Place FFT Routine
;
;     Complex input and output data
;          Real data in X memory
;          Imaginary data in Y memory
;      Normally ordered input data
;      Bit reversed output data
;           Coefficient lookup table
;          -Cosine values in X memory
;          -Sine values in Y memory
;
;Macro Call — ffr2a      points,data,coef
;
;            points                  number of points (2-32768, power of 2)
;            data                    start of data buffer
;            coef                    start of sine/cosine table
;
;Alters Data ALU Registers
;            x1          x0          y1          y0
;            a2          a1          a0          a
;            b2          b1          b0          b
;
;Alters Address Registers
;            r0          n0          m0
;            r1          n1          m1
;                        n2
;
;            r4          n4          m4
;            r5          n5          m5
;            r6          n6          m6
;
‘Alters Program Control Registers
;            pc          sr
;
;Uses 6 locations on System Stack
;
```

Figure B-2 Radix 2, In-Place, Decimation-In-Time FFT (Sheet 1 of 2)

```
;Latest Revision — September 30, 1986
;
            move        #points/2,n0            ;initialize butterflies per group
            move        #1,n2                   ;initialize groups per pass
            move        #points/4,n6            ;initialize C pointer offset
            move        #-1,m0                  ;initialize A and B address modifiers
            move        m0,m1                   ;for linear addressing
            move        m0,m4
            move        m0,m5
            move        #0,m6                   ;initialize C address modifier for
                                                ;reverse carry (bit-reversed) addressing
;
;Perform all FFT passes with triple nested DO loop
;
            do          #@cvi (@log(points)/@log(2)+0.5),_end_pass
            move        #data,r0                ;initialize A input pointer
            move        r0,r4                   ;initialize A output pointer
            lua         (r0)+n0,r1              ;initialize B input pointer
            move        #coef,r6                ;initialize C input pointer
            lua         (r1)-,r5                ;initialize B output pointer
            move        n0,n1                   ;initialize pointer offsets
            move        n0,n4
            move        n0,n5

            do          n2,_end_grp
            move        x:(r1),X1       y:(r6),y0               ;lookup -sine and
                                                                ; -cosine values
            move        x:(r5),a        y:(r0),b                ;preload data
            move        x:(r6)+n6,x0                            ;update C pointer

            do          n0,_end_bfy
            mac         x1,y0,b         y:(r1)+,y1              ;Radx 2 DIT
                                                                ;butterfly kernel
            macr        -x0,y1,b        a,x:(r5)+       y:(r0),a
            subl        b,a             x:(r0),b        b,y:(r4)
            mac         -x1,x0,b        x:(r0)+,a       a,y:(r5)
            macr        -y1,y0,b        x:(r1),x1
            subl        b,a             b,x:(r4)+       y:(r0),b
_end_bfy
            move        a,x:(r5)+n5         y:(r1)+n1,y1        ;update A and B pointers
            move        x:(r0)+n0,x1        y:(r4)+n4,y1
_end_grp
            move        n0,b1                   ;divide butterflies per group by two
            lsr         b       n2,a1           ;multiply groups per pass by two
            lsl         a       b1,n0
            move        a1,n2
_end_pass
            endm
```

Figure B-2 Radix 2, In-Place, Decimation-In-Time FFT (Sheet 2 of 2)

```
     page 132,66,0,6
     opt     rc
;**************************************************************
;Motorola Austin DSP Operation         June 30, 1988
***************************************************************
;DSP56000/1
;8-pole 4-multiply cascaded canonic IIR filter
;File name: 4-56.asm
;*****************************************************************************************************************************
;     Maximum sample rate: 410.0 kHz at 20.5 MHz/540.0 kHz at 27.0 MHz
;     Memory Size: Prog: 6+10 words; Data: 4(2+4) words
;     Number of clock cycles: 50 (25 instruction cycles)
;     Clock Frequency: 20.5 MHz/27.0 MHz
;     Instruction cycle time: 97.5 ns/74.1 ns
;*****************************************************************************************************************************
;     This IIR filter reads the input sample
;     from the memory location Y:input
;     and writes the filtered output sample
;     to the memory location Y:output
;
;     The samples are stored in the X memory
;     The coefficients are stored in the Y memory
;
;
;     The equations of the filter are:
;          w(n)=            x(n)-ai1*w(n-1)-ai2*w(n-2)
;           y(n)=            w(n)+bi1*w(n-1)+bi2*w(n-2)
;
```

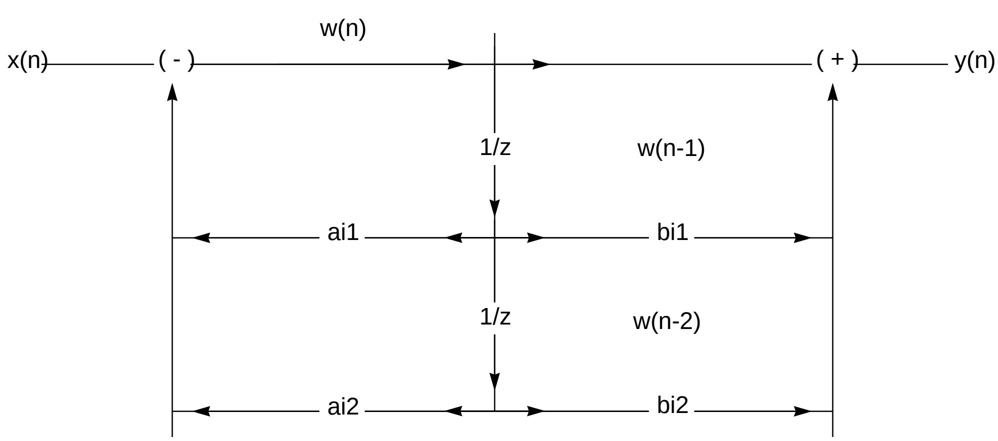


Figure B-3 8-Pole 4-Multiply Cascaded Canonic IIR Filter (Sheet 1 of 2)

```
;    All coefficients are divided by 2:
;    w(n)/2=x(n)/2-ai1/2*w(n-1)-ai2/2*w(n-2)
;    y(n)/2=w(n)/2+bi1/2*w(n-1)+bi2/2*w(n-2)
;
;        X Memory Organization                  Y Memory Organization
;                 |          |                          | b1N/2 |    Coef. + 4*nsec - 1
;                 |          |                          | b2N/2 |
;                 |          |                          | a1N/2 |
;                 |          |                          | a2N/2 |
;                 | wN(n-1)  |   Data + 2*nsec - 1      |   •   |
;                 | wN(n-2)  |                          |   •   |
;                 |    •     |                          | b11/2 |
;                 |    •     |                          | b21/2 |
;                 | w1(n-1)  |                          | a11/2 |
;     R0 ➡        | w1(n-2)  |   Data            R4 ➡   | a21/2 |    Coef.
;
;
;*****************************************************************************************************************
***
;
;    initialization
;************************************
nsec        equ         4
start       equ         $40
data        equ         0
coef        equ         0
input       equ         $ffe0
output      equ         $ffe1
igain       equ         0.5
            ori         #$08,mr             ;set scaling mode
            move        #data,r0            ;point to filter states
            move        #coef,r4            ;point to filter coefficients
            move        #2*nsec - 1,m0
            move        #4*nsec - 1,m4
            move        #igain,y1           ;y1=initial gain

            opt         cc
;                   filter loop: 4*nsec + 9
;*********************************************************************

            movep           y:input,y0                          ;get sample
            mpy         y0,y1,a     x:(r0) +,x0     y:(r4)+,y0  ;x0=1st section w(n-2),y0=ai2/2
;
            do              #nsec,end_cell                      ;do each section
            mac         -x0,y0,a    x:(r0) -,x1     y:(r4) +,y0 ;x1=w(n-1),y0=ai1/2
            macr        -x1,y0,a    x1,x:(r0) +     y:(r4) +,y0 ;push w(n-1) to w(n-2),y0=bi2/2
            mac         x0,y0,a     a,x:(r0)+       y:(r4) +,y0 ;push w(n) to w(n-1),y0=bi1/2
            mac         x1,y0,a     x:(r0) +,x0     y:(r4) +,y0 ;next iter:x0=w(n-2),y0=ai2/2
end_cell
            rnd         a                                       ;round result
            movep       a,y:output                              ;output sample
;*****************************************************************************************
            end
```

Figure B-3 8-Pole 4-Multiply Cascaded Canonic IIR Filter (Sheet 2 of 2)

```
  page 132,60,1,1
;newlms2n.asm
;    New Implementation of the delayed LMS on the DSP56000 Revision C
;Memory map:
;    Initial X                                                H
;    x(n)     x(n-1)     x(n-2)     x(n-3)     x(n-4)         hx     h0     h1     h2     h3
;    ]                                                       ]      ]
;    r0                                                      r5     r4
;hx is an unused value to make the calculations faster.
;
  opt     cc
ntaps   equ     4
input   equ     $FFC0
output  equ     $FFC1

  org     x:$0
state   ds      5
  org     y:$0
coef    ds      5
;
  org     p:$40
  move    #state,r0               ;start of X
  move    #2,n0
  move    #ntaps,m0               ;mod 5
  move    #coef +1,r4             ;coefficients
  move    #ntaps,m4               ;mod 5
  move    #coef,r5                ;coefficients
  move    m4,m5                   ;mod 5
_smploop                          ;                                  Prog    Icyc
  movep                   y:input,a ;get input sample                word
  move            a,x:(r0)        ;save input sample                  1       1
;error signal is in y1
;FIR sum in a=a+h(k) old*x(n-k)
;h(k)new in b=h(k)old + error*x(n-k-1)

  cir     a       x:(r0)+,x0              ;x0=x(n)                    1       1
  move            x:(r0)+,x1  y:(r4)+,y0  ;x1=x(n-1),y0=h(0)          1       1
  do      #taps/2,_lms                    ;                           2       3
  mac     x0,y0,a y0,b        b,y:(r5)+   ;a=h(0)*x(n),b=h(0)         1       1
  macr    x1,y1,b x:(r0)+,x0  y:(r4)+,y0  ;b=h(0)+e*x(n-1)=h(0)new    1       1
                                          ;x0=x(n-2) y0=h(1)
  mac     x1,y0,a y0,b        b,y:(r5)+   ;a=a+h(1)*x(n-1) b=h(1)     1       1
  macr    x0,y1,b x:(r0)+,x1  y:(r4)+,y0  ;b=h(1)+e*x(n-2)            1       1
;                                         ;x1=x(n-3) y0=H(2)
_lms
  move                        b,y:(r5)+   ;save last new c( )         1       1
  move            (r0) -n0                ;pointer update             1       1

;(Get d(n), subtract fir output (reg a), multiply by “u”, put
;the result in y1. This section is application dependent.)
  movep           a,y:output              ;output fir if desired
  jmp             _smploop
  end                                     ;
;                                                    Totals:  11     2N+8
```

Figure B-4 LMS FIR Adaptive Filter

APPENDIX C
ADDITIONAL SUPPORT

User support from the conception of a design through completion is available from Motorola and third-party companies as shown in the following list:

	Motorola	Third Party
Design	Data Sheets Application Notes Application Bulletins Software Examples Simulator	Data Acquisition Packages Filter Design Packages Operating System Software
Prototyping	Assembler Linker C Compiler Simulator Application Development System (ADS) In-Circuit Emulator Cable for ADS	Logic Analyzer with DSP56000/DSP56001 ROM Packages In-Circuit Emulators Data Acquisition Cards DSP Development System Cards Operating System Software Debug Software
Design Verification	Application Development System (ADS) In-Circuit Emulator Simulator	Data Acquisition Packages Logic Analyzer with DSP56000/DSP56001 ROM Packages Data Acquisition Cards DSP Development System Cards Application-Specific Development Tools Debug Software

The following is a partial list of the support available for the DSP56000/DSP56001. Additional information can be obtained through Dr. BuB or the appropriate support telephone service.

Motorola DSP Product Support

- DSP56000CLASx Design-In Software Package which includes:
  - Relocatable Macro Assembler
  - Linker
  - Simulator (simulates single or multiple DSP56000/DSP56001s)
  - Librarian
- DSP56KCCx Full Kernighan and Ritchie C Compiler
- DSP320to56001 Translator Software
- DSP56000/DSP56001 Applications Development System (ADS)
- Support Integrated Circuits
- DSP Bulletin Board (Dr. BuB)
- Motorola DSP Newsletter
- Motorola Field Application Engineers (FAEs)
  - See your local telephone directory for the Motorola Semiconductor Sector sales office telephone number.
- Design Hotline
- Applications Assistance
- Marketing Information
- Third-Party Support Information
- University Support Information

DSP56000CLASx Assembler/Simulator

The macro cross assembler and simulator run on:

1. IBM™ PCs (386 or better)
2. Macintosh™ under MAC OS 4.1 or later
3. SUN-4™ under UNIX™ BSD 4.2

IBM is a trademark of International Business Machines.
Macintosh is a trademark of Apple Computer, Inc.
SUN-4 is a trademark of SUN Microsystems, Inc.
UNIX is a registered trademark of AT&T Bell Labratories.

Macro Cross Assembler Features:

- Production of relocatable object modules compatible with linker program when in relocatable mode
- Production of absolute files compatible with simulator program when in absolute mode
- Supports full instruction set, memory spaces, and parallel data transfer fields of the DSP56000/DSP56001
- Modular programming features: local labels, sections, and external definition/reference directives
- Nested macro processing capability with support for macro libraries
- Complex expression evaluation including boolean operators
- Built-in functions for data conversion, string comparison, and common transcendental math functions
- Directives to define circular and bit-reversed buffers
- Extensive error checking and reporting

Simulator Features:

- Simulation of all DSP56001 (default) or DSP56000
- Simulation of multiple DSP56000/DSP560001s
- Linkable object code modules:
  - –Nondisplay simulator library
  - –Display simulator library
- C language source code for:
  - –Screen management functions
  - –Terminal I/O functions
  - –Simulation examples
- Single stepping through object programs
- Up to 99 conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Instruction, clock cycle, and histogram counters
- Session and/or command logging for later reference
- ASCII input/output files for peripherals
- Help-file and help-line display of simulator commands
- Loading and saving of files to/from simulator memory
- Macro command definition and execution
- Display enable/disable of registers and memory
- Hexadecimal/decimal/binary calculator

C Language Compiler

DSP56KCCx Language Compiler Features:

- Full Kernighan and Ritchie C
- Structures/Unions
- Floating Point
- In-line assembler language code compatibility
- Full Function preprocessor for:
  - — Macro definition/expansion
  - — File Inclusion
  - — Conditional compilation
- Full error detection and reporting

DSP320to56001 Translator

DSP320to56001 Translator Features:

- Translates any TMS32010 linked object code to DSP56001 source assembler code
- Two modes of operation:
  - — Translates to DSP56001 source assembler code for optimization and assembly using DSP56000CLASx
  - — Translates and runs “as is” directly and immediately on the DSP56000ADSx
- C language DSP320to56001 source code is provided in addition to IBM PC/XT/AT object code to allow:
  - — User modification for TMS32020 or TMS320C25 translation
  - — User compilation to accommodate different host platforms

DSP56000ADSx Application Development System

DSP56000ADS Application Development System Hardware Features:

- Full-speed 20.48 MHz operation (upgradeable to 27 MHz)
- Multiple application development module (ADM) support with programmable ADM addresses
- 8K/32Kx24 user-configurable RAM for DSP56000/DSP56001 code development
- 1Kx24 monitor ROM expandable to 4Kx24
- 96-pin Euro-card connector making all DSP56001 pins accessible
- In-circuit emulation capabilities when used with the DSP56KEMULTRCABL cable
- Separate berg pin connectors for alternate accessing of serial or host/DMA ports
- ADM can be used in stand-alone configuration
- No external power supply needed when connected to a host platform

DSP56000ADSx Application Development System Software Features:

- Single/multiple stepping through DSP56000/DSP56001 object programs
- Up to 99 conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Session and/or command logging for later reference
- Loading and saving files to/from ADM memory
- Macro command definition and execution
- Display enable/disable of registers and memory
- Debug commands supporting multiple ADMs
- Hexadecimal/decimal/binary calculator
- Host operating system commands from within ADS user interface program
- Multiple OS I/O file access from DSP56000/DSP56001 object programs
- Fully compatible with the DSP56000CLASx design-in software package
- On-line help screens for each command and DSP56000/DSP56001 register

Support Integrated Circuits:

- 8Kx24 Static RAM
- DSP56ADC16 16-bit, 100-kHz analog-to-digital converter

Dr. BuB Electronic Bulletin Board

Dr. BuB is an electronic bulletin board which provides free source code for a large variety of topics that can be used to develop applications with Motorola DSP products. The software library contains files including FFTs, FIR filters, IIR filters, lattice filters, matrix algebra routines, companding routines, floating-point routines, a software debug monitor, and others. In addition, the latest product information and documentation (including information on new products and improvements to existing products) is posted. Questions about Motorola DSP products posted on Dr. BuB are answered promptly. The following phone numbers provide access to Dr. BuB:

(212A – 300/1200 Baud) . . . . . (512) 891-DSP1
(V.22 – 1200 Baud) . . . . . (512) 891-DSP2
(V.22bis – 2400 Baud) . . . . . (512) 891-DSP3
Format: 7 data bits, even parity, 1 stop bit
User ID=guest

The following is a partial list of the software available on Dr. BuB.

Document ID	Version	Synopsis	Size
Codec Routines:			
loglin.asm	1.0	Companded CODEC to linear PCM data conversion	4572
loglin.hlp		Help for loglin.asm	1479
loglint.asm	1.0	Test program for loglin.asm	2184
loglint.hlp		Help for loglint.asm	1993
linlog.asm	1.1	Linear PCM to companded CODEC data conversion	4847
linlog.hlp		Help for linlog.asm	1714
Fast Fourier Transforms:			
sincos.asm	1.2	Sine-Cosine Table Generator for FFTs	1185
sincos.hlp		Help for sincos.asm	887
sinewave.asm	1.1	Full-Cycle Sine wave Table Generator Macro	1029
sinewave.hlp		Help for sinewave.asm	1395
fftr2a.asm	1.1	Radix 2, In-Place, DIT FFT (smallest)	3386
fftr2a.hlp		Help for fftr2a.asm	2693
fftr2at.asm	1.1	Test Program for FFTs (fftr2a.asm)	999
fftr2at.hlp		Help for fftr2at.asm	563
fftr2b.asm	1.1	Radix 2, In-Place, DIT FFT (faster)	4290
fftr2b.hlp		Help for fftr2b.asm	3680
fftr2c.asm	1.2	Radix 2, In-Place, DIT FFT (even faster)	5991
fftr2c.hlp		Help for fftr2c.asm	3231
fftr2d.asm	1.0	Radix 2, In-Place, DIT FFT 3727 (using DSP56001 sine-cosine ROM tables)	3727
fftr2d.hlp		Help for fftr2d.asm	3457
fftr2dt.asm	1.0	Test program for fftr2d.asm	1287
fftr2dt.hlp		Help for fftr2dt.asm	614

Document ID	Version	Synopsis	Size
fftr2e.asm	1.0	1024 Point, Non-In-Place, FFT (3.39ms)	8976
fftr2e.hlp		Help for fftr2e.asm	5011
fftr2et.asm	1.0	Test program for fftr2e.asm	984
fftr2et.hlp		Help for fftr2et.asm	408
dct1.asm	1.2	Discrete Cosine Transform using FFT	5471
dct1.hlp	1.1	Help file for dct1.asm	970
fftr2cc.asm	1.0	Radix 2, In-place Decimation-in-time complex FFT macro	6524
fftr2cc.hlp	1.0	Help file for fftr2cc.asm	3533
fftr2cn.asm	1.0	Radix 2, Decimation-in-time complex FFT macro with normally ordered input/output	6584
fftr2cn.hlp	1.0	Help file for fftr2cn.asm	2468
fftr2en.asm	1.0	1024 point, not-in-place, complex FFT macro with normally ordered input/output	9723
fftr2en.hlp	1.0	Help file for fftr2en.asm	4886
dhit1.asm	1.0	Routine to compute Hilbert transform in the frequency domain	1851
Filters:			
fir.asm	1.0	Direct Form FIR Filter	545
fir.hlp		Help for fir.asm	2161
firt.asm	1.0	Test program for fir.asm	1164
iir1.asm	1.0	Direct Form Second Order All-Pole IIR Filter	656
iir1.hlp		Help for iir1.asm	1786
iir1t.asm	1.0	Test program for iir1.asm	1157
iir2.asm	1.0	Direct Form Second Order All-Pole IIR Filter with Scaling	801
iir2.hlp		Help for iir2.asm	2286

Document ID	Version	Synopsis	Size
iir2t.asm	1.0	Test program for iir2.asm	1311
iir3.asm	1.0	Direct Form Arbitrary Order All-Pole IIR Filter	776
iir3.hlp		Help for iir3.asm	2605
iir3t.asm	1.0	Test program for iir3.asm	1309
iir4.asm	1.0	Second Order Direct Canonic IIR Filter (Biquad IIR Filter)	713
iir4.hlp		Help for iir4.asm	2255
iir4t.asm	1.0	Test program for iir4.asm	1202
iir5.asm	1.0	Second Order Direct Canonic IIR Filter with Scaling (Biquad IIR Filter)	842
iir5.hlp		Help for iir5.asm	2803
iir5t.asm	1.0	Test program for iir5.asm	1289
iir6.asm	1.0	Arbitrary-Order Direct Canonic IIR Filter	923
iir6.hlp		Help for iir6.asm	3020
iir6t.asm	1.0	Test program for iir6.asm	1377
iir7.asm	1.0	Cascaded Biquad IIR Filters	900
iir7.hlp		Help for iir7.asm	3947
iir7t.asm	1.0	Test program for iir7.asm	1432
lms.hlp	1.0	LMS Adaptive Filter Algorithm	5818
transiir.asm	1.0	Implements the transposed IIR filter	1981
transiir.hlp	1.0	Help file for transiir.asm	974
Floating-Point Routines:			
fpdef.hlp	2.0	Storage format and arithmetic representation definition	10600
fpcalls.hlp	2.1	Subroutine calling conventions	11876
fplist.asm	2.0	Test file that lists all subroutines	1601
fprevs.hlp	2.0	Latest revisions of floating-point lib	1799

Document ID	Version	Synopsis	Size
fpinit.asm	2.0	Library initialization subroutine	2329
fpadd.asm	2.0	Floating-point add	3860
fpsub.asm	2.1	Floating-point subtract	3072
fpcmp.asm	2.1	Floating-point compare	2605
fpmpy.asm	2.0	Floating-point multiply	2250
fpmac.asm	2.1	Floating-point multiply-accumulate	2712
fpdiv.asm	2.0	Floating-point divide	3835
fpsqrt.asm	2.0	Floating-point square root	2873
fpneg.asm	2.0	Floating-point negate	2026
fpabs.asm	2.0	Floating-point absolute value	1953
fpscale.asm	2.0	Floating-point scaling	2127
fpfix.asm	2.0	Floating- to fixed-point conversion	3953
fpfloat.asm	2.0	Fixed- to floating-point conversion	2053
fpceil.asm	2.0	Floating-point CEIL subroutine	1771
durbin.asm	1.0	Solution for LPC coefficients	5615
durbin.hlp	1.0	Help file for DURBIN.ASM	2904
fpfrac.asm	2.0	Floating-point FRACTION subroutine	1862
Functions:			
log2.asm	1.0	Log base 2 by polynomial approximation	1118
log2.hlp		Help for log2.asm	719
log2t.asm	1.0	Test program for log2.asm	1018
log2nrm.asm	1.0	Normalizing base 2 logarithm macro	2262
log2nrm.hlp		Help for log2nrm.asm	676
log2nrmt.asm	1.0	Test program for log2nrm.asm	1084
exp2.asm	1.0	Exponential base 2 by polynomial approximation	926
exp2.hlp		Help for exp2.asm	759
exp2t.asm	1.0	Test program for exp2.asm	1019
sqrt1.asm	1.0	Square Root by polynomial approximation, 7 bit accuracy	991
sqrt1.hlp		Help for sqrt1.asm	779

sqrt1t.asm	1.0	Test program for sqrt1.asm	1065
sqrt2.asm	1.0	Square Root by polynomial approximation, 10 bit accuracy	899
sqrt2.hlp		Help for sqrt2.asm	776
sqrt2t.asm	1.0	Test program for sqrt2.asm	1031
sqrt3.asm	1.0	Full precision Square Root Macro	1388
sqrt3.hlp		Help for sqrt3.asm	794
sqrt3t.asm	1.0	Test program for sqrt3.asm	1053
tli.asm	1.1	Linear table lookup/interpolation routine for function generation	3253
tli.hlp	1.1	Help for tli.asm	1510
bingray.asm	1.0	Binary to Gray code conversion macro	601
bingrayt.asm	1.0	Test program for bingray.asm	991
rand1.asm	1.1	Pseudo Random Sequence Generator	2446
rand1.hlp		Help for rand1.asm	704

Lattice Filters:

latfir1.asm	1.0	Lattice FIR Filter Macro	1156
latfir1.hlp		Help for latfir1.asm	6327
latfir1t.asm	1.0	Test program for latfir1.asm	1424
latfir2.asm	1.0	Lattice FIR Filter Macro (modified modulo count)	1174
latfir2.hlp		Help for latfir2.asm	1295
latfir2t.asm	1.0	Test program for latfir2.asm	1423
latiir.asm	1.0	Lattice IIR Filter Macro	1257
latiir.hlp		Help for latiir.asm	6402
latiirt.asm	1.0	Test program for latiir.asm	1407

Document ID	Version	Synopsis	Size
latgen.asm	1.0	Generalized Lattice FIR/IIR Filter Macro	1334
latgen.hlp		Help for latgen.asm	5485
latgent.asm	1.0	Test program for latgen.asm	1269
latnrm.asm	1.0	Normalized Lattice IIR Filter Macro	1407
latnrm.hlp		Help for latnrm.asm	7475
latnrmt.asm	1.0	Test program for latnrm.asm	1595
Matrix Operations:			
matmul1.asm	1.0	[1x3][3x3]=[1x3] Matrix Multiplication	1817
matmul1.hlp		Help for matmul1.asm	527
matmul2.asm	1.0	General Matrix Multiplication, C=AB	2650
matmul2.hlp		Help for matmul2.asm	780
matmul3.asm	1.0	General Matrix Multiply-Accumulate, C=AB+Q	2815
matmul3.hlp	1.0	Help for matmul3.asm	865
Reed-Solomon Encoder:			
readme.rs	1.0	Instructions for Reed-Solomon coding	5200
rscd.asm	1.0	Reed-Solomon coder for DSP56000 simulator	5822
newc.c	1.0	Reed-Solomon coder coded in C	4075
table1.asm	1.0	Include file for R-S coder	7971
table2.asm	1.0	Include file for R-S coder	4011
Sorting Routines:			
sort1.asm	1.0	Array Sort by Straight Selection	1312
sort1.hlp		Help for sort1.asm	1908
sort1t.asm	1.0	Test program for sort1.asm	689
sort2.asm	1.1	Array Sort by Heapsort Method	2183
sort2.hlp		Help for sort2.asm	2004
sort2t.asm	1.0	Test program for sort2.asm	700

Document ID	Version	Synopsis	Size
Speech:			
lgsol1.asm	2.0	Leroux-Gueguen solution for PARCOR (LPC) coefficients	4861
lgsol1.hlp		Help for lgsol1.asm	3971
durbin1.asm	1.2	Durbin Solution for PARCOR (LPC) coefficients	6360
durbin1.hlp		Help for durbin1.asm	3616
Standard I/O Equates:			
ioequ.asm	1.1	Motorola Standard I/O Equate File	8774
ioequlc.asm	1.1	Lower Case Version of ioequ.asm	8788
intequ.asm	1.0	Standard Interrupt Equate File	1082
intequlc.asm	1.0	Lower Case Version of intequ.asm	1082

Motorola DSP News

The Motorola **DSP News** is a quarterly newsletter providing information on new products, application briefs, questions and answers, DSP product information, third-party product news, etc. This newsletter is free and is available upon request by calling the marketing information phone number listed below.

Motorola Field Application Engineers

Information and assistance for DSP applications is available through the local Motorola field office. See your local telephone directory for telephone numbers.

Design Help Line – 1-800-521-6274

This is the Motorola number for information about any Motorola product.

Applications Assistance – (512) 891-3230

Design assistance for specific DSP applications is available by calling this number.

Sales Information

Sales information, including brochures, application notes, manuals, price quotes, etc., for Motorola DSP-related products is available by calling your local Motorola field office or dealer.

Third-Party Support Information – (512) 891-3098

Information about third-party manufacturers who use and support Motorola DSP products is available by calling this number. Third-party support includes:

Filter design software
Logic analyzer support
Boards for VME, IBM-PC/XT/AT, MACII boards
Development systems
Data conversion cards
Operating system software
Debug software

Additional information is available on Dr. BuB and in **DSP News**.

University Support – (512) 891-3098

Information concerning university support programs and university discounts for all Motorola DSP products is available by calling this number.

Training Courses – (602) 897-3665

There are two DSP56000 Family training courses available:

1. Introduction to the DSP56000/DSP56001 (MTTA5) is a 4.5-hour audio-tape course on the DSP56000/DSP56001 architecture and programming.

2. Introduction to the DSP56000/DSP56001 (MTT31) is a four-day instructor-led course and laboratory which covers the details of the DSP56000/DSP56001 architecture and programming.

Additional information is available by writing to:

Motorola SPS Training and Technical Operations
Mail Drop EL524
P. O. Box 21007
Phoenix, Arizona 85036

or by calling the number above. A technical training catalog is available which describes these courses and gives the current training schedule and prices.

Reference Books and Manuals

A list of DSP-related books is included here as an aid for the engineer who is new to the field of DSP. This is a partial list of DSP references intended to help the new user find useful information in some of the many areas of DSP applications. Many of the books could be included in several categories but are not repeated.

General DSP:

ADVANCED TOPICS IN SIGNAL PROCESSING
Jae S. Lim and Alan V. Oppenheim
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

APPLICATIONS OF DIGITAL SIGNAL PROCESSING
A. V. Oppenheim
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978

DISCRETE-TIME SIGNAL PROCESSING
A. V. Oppenheim and R. W. Schafer
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989

DIGITAL PROCESSING OF SIGNALS THEORY AND PRACTICE
Maurice Bellanger
New York, NY: John Wiley and Sons, 1984

DIGITAL SIGNAL PROCESSING
Alan V. Oppenheim and Ronald W. Schafer
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975

DIGITAL SIGNAL PROCESSING: A SYSTEM DESIGN APPROACH
David J. DeFatta, Joseph G. Lucas, and William S. Hodgkiss
New York, NY: John Wiley and Sons, 1988

FOUNDATIONS OF DIGITAL SIGNAL PROCESSING AND DATA ANALYSIS
J. A. Cadzow
New York, NY: MacMillan Publishing Company, 1987

HANDBOOK OF DIGITAL SIGNAL PROCESSING
D. F. Elliott
San Diego, CA: Academic Press, Inc., 1987

INTRODUCTION TO DIGITAL SIGNAL PROCESSING
John G. Proakis and Dimitris G. Manolakis
New York, NY: Macmillan Publishing Company, 1988

MULTIRATE DIGITAL SIGNAL PROCESSING
R. E. Crochiere and L. R. Rabiner
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983

SIGNAL PROCESSING ALGORITHMS
S. Stearns and R. Davis
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

SIGNAL PROCESSING HANDBOOK
C.H. Chen
New York, NY: Marcel Dekker, Inc., 1988

SIGNAL PROCESSING – THE MODERN APPROACH
James V. Candy
New York, NY: McGraw-Hill Company, Inc., 1988

THEORY AND APPLICATION OF DIGITAL SIGNAL PROCESSING
Rabiner, Lawrence R., Gold and Bernard
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975

Digital Audio and Filters:

ADAPTIVE FILTER AND EQUALIZERS
B. Mulgrew and C. Cowan
Higham, MA: Kluwer Academic Publishers, 1988

ADAPTIVE SIGNAL PROCESSING
B. Widrow and S. D. Stearns
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985

ART OF DIGITAL AUDIO, THE
John Watkinson
Stoneham. MA: Focal Press, 1988

DESIGNING DIGITAL FILTERS
Charles S. Williams
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986

DIGITAL AUDIO SIGNAL PROCESSING AN ANTHOLOGY
John Strawn
William Kaufmann, Inc., 1985

DIGITAL CODING OF WAVEFORMS
N. S. Jayant and Peter Noll
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL FILTERS: ANALYSIS AND DESIGN
Andreas Antoniou
New York, NY: McGraw-Hill Company, Inc., 1979

DIGITAL FILTERS AND SIGNAL PROCESSING
Leland B. Jackson
Higham, MA: Kluwer Academic Publishers, 1986

DIGITAL SIGNAL PROCESSING
Richard A. Roberts and Clifford T. Mullis
New York, NY: Addison-Welsey Publishing Company, Inc., 1987

INTRODUCTION TO DIGITAL SIGNAL PROCESSING
Roman Kuc
New York, NY: McGraw-Hill Company, Inc., 1988

INTRODUCTION TO ADAPTIVE FILTERS
Simon Haykin
New York, NY: MacMillan Publishing Company, 1984

MUSICAL APPLICATIONS OF MICROPROCESSORS (Second Edition)
H. Chamberlin
Hasbrouck Heights, NJ: Hayden Book Co., 1985

Controls:

ADAPTIVE CONTROL
K. Astrom and B. Wittenmark
New York, NY: Addison-Welsey Publishing Company, Inc., 1989

ADAPTIVE FILTERING PREDICTION & CONTROL
G. Goodwin and K. Sin
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

AUTOMATIC CONTROL SYSTEMS
B. C. Kuo
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987

COMPUTER CONTROLLED SYSTEMS: THEORY & DESIGN
K. Astrom and B. Wittenmark
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL CONTROL SYSTEMS
B. C. Kuo
New York, NY: Holt, Reinholt, and Winston, Inc., 1980

DIGITAL CONTROL SYSTEM ANALYSIS & DESIGN
C. Phillips and H. Nagle
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

ISSUES IN THE IMPLEMENTATION OF DIGITAL FEEDBACK COMPENSATORS
P. Moroney
Cambridge, MA: The MIT Press, 1983

Graphics:

CGM AND CGI

D. B. Arnold and P. R. Bono
New York, NY: Springer-Verlag, 1988

COMPUTER GRAPHICS (Second Edition)
D. Hearn and M. Pauline Baker
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986

FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS
J. D. Foley and A. Van Dam
Reading MA: Addison-Wesley Publishing Company Inc., 1984

GEOMETRIC MODELING
Michael E. Morteson
New York, NY: John Wiley and Sons, Inc.

GKS THEORY AND PRACTICE
P. R. Bono and I. Herman (Eds.)
New York, NY: Springer-Verlag, 1987

ILLUMINATION AND COLOR IN COMPUTER GENERATED IMAGERY
Roy Hall
New York, NY: Springer-Verlag

POSTSCRIPT LANGUAGE PROGRAM DESIGN
Glenn C. Reid - Adobe Systems, Inc.
Reading MA: Addison-Wesley Publishing Company, Inc., 1988

MICROCOMPUTER DISPLAYS, GRAPHICS, AND ANIMATION
Bruce A. Artwick
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985

PRINCIPLES OF INTERACTIVE COMPUTER GRAPHICS
William M. Newman and Roger F. Sproull
New York, NY: McGraw-Hill Company, Inc., 1979

PROCEDURAL ELEMENTS FOR COMPUTER GRAPHICS
David F. Rogers
New York, NY: McGraw-Hill Company, Inc., 1985

RENDERMAN INTERFACE, THE
Pixar
San Rafael, CA. 94901

Image Processing:

DIGITAL IMAGE PROCESSING
William K. Pratt
New York, NY: John Wiley and Sons, 1978

DIGITAL IMAGE PROCESSING (Second Edition)
Rafael C. Gonzales and Paul Wintz
Reading, MA: Addison-Wesley Publishing Company, Inc., 1977

DIGITAL IMAGE PROCESSING TECHNIQUES
M. P. Ekstrom
New York, NY: Academic Press, Inc., 1984

DIGITAL PICTURE PROCESSING
Azriel Rosenfeld and Avinash C. Kak
New York, NY: Academic Press, Inc., 1982

SCIENCE OF FRACTAL IMAGES, THE
M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, H. O. Peitgen,
D. Saupe, and R. F. Voss
New York, NY: Springer-Verlag

????Motorola DSP Manuals:

MOTOROLA DSP56000 LINKER/LIBRARIAN REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP56000 MACRO ASSEMBLER REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP56000 SIMULATOR REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP56000/DSP56001 USER'S MANUAL
Motorola, Inc.,1990.

Numerical Methods:

ALGORITHMS (THE CONSTRUCTION, PROOF, AND ANALYSIS OF PROGRAMS)
P. Berliout and P. Bizard
New York, NY: John Wiley and Sons, 1986

MATRIX COMPUTATIONS
G. H. Golub and C. F. Van Loan
John Hopkins Press, 1983

NUMERICAL RECIPES IN C - THE ART OF SCIENTIFIC PROGRAMMING
William H. Press, Brian P. Flannery,
Saul A. Teukolsky, and William T. Vetterling
Cambridge University Press, 1988

NUMBER THEORY IN SCIENCE AND COMMUNICATION
Manfred R. Schroeder
New York, NY: Springer-Verlag, 1986

Pattern Recognition:

PATTERN CLASSIFICATION AND SCENE ANALYSIS
R. O. Duda and P. E. Hart
New York, NY: John Wiley and Sons, 1973

CLASSIFICATION ALGORITHMS
Mike James
New York, NY: Wiley-Interscience, 1985
Spectral Analysis:

Spectral Analysis:

STATISTICAL SPECTRAL ANALYSIS, A NONPROBABILISTIC THEORY
William A. Gardner
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

THE FAST FOURIER TRANSFORM AND ITS APPLICATIONS
E. Oran Brigham
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

THE FAST FOURIER TRANSFORM AND ITS APPLICATIONS
R. N. Bracewell
New York, NY: McGraw-Hill Company, Inc., 1986

Speech:

ADAPTIVE FILTERS – STRUCTURES, ALGORITHMS, AND APPLICATIONS
Michael L. Honig and David G. Messerschmitt
Higham, MA: Kluwer Academic Publishers, 1984

DIGITAL CODING OF WAVEFORMS
N. S. Jayant and P. Noll
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL PROCESSING OF SPEECH SIGNALS
Lawrence R. Rabiner and R. W. Schafer
Englwood Cliffs, NJ: Prentice-Hall, Inc., 1978

LINEAR PREDICTION OF SPEECH

J. D. Markel and A. H. Gray, Jr.
New York, NY: Springer-Verlag, 1976

SPEECH ANALYSIS, SYNTHESIS, AND PERCEPTION
J. L. Flanagan
New York, NY: Springer-Verlag, 1972

SPEECH COMMUNICATION – HUMAN AND MACHINE
D. O'Shaughnessy
Reading, MA: Addison-Wesley Publishing Company, Inc., 1987

Telecommunications:

DIGITAL COMMUNICATION
Edward A. Lee and David G. Messerschmitt
Higham, MA: Kluwer Academic Publishers, 1988

DIGITAL COMMUNICATIONS
John G. Proakis
New York, NY: McGraw-Hill Publishing Co., 1983

Order this document by DSP56000UM/AD