DSP56600

16-bit Digital Signal Processor Family Manual

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

This document (and other documents) can be viewed on the World Wide Web at http://www.motorola-dsp.com.

This manual is one of a set of three documents. You need the following manuals to have complete product information: Family Manual, User's Manual, and Technical Data.

OnCE™ is a trademark of Motorola, Inc.

HP-UX™ is a trademark of Hewlett-Packard.

IBM™ is a trademark of International Business Machines.

Macintosh™ is a trademark of Apple Computer, Inc.

SBUS™, SUN-4™, and SunOS™ are trademarks of Sun Microsystems, Inc.

Windows™ is a trademark of Microsoft Corporation.

Order this document by **DSP56600FM/AD**

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not authorized for use as components in life support devices or systems intended for surgical implant into the body or intended to support or sustain life. Buyer agrees to notify Motorola of any such intended end use whereupon Motorola shall determine availability and suitability of its product or products for the use intended. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Employment Opportunity / Affirmative Action Employer.

TABLE OF CONTENTS

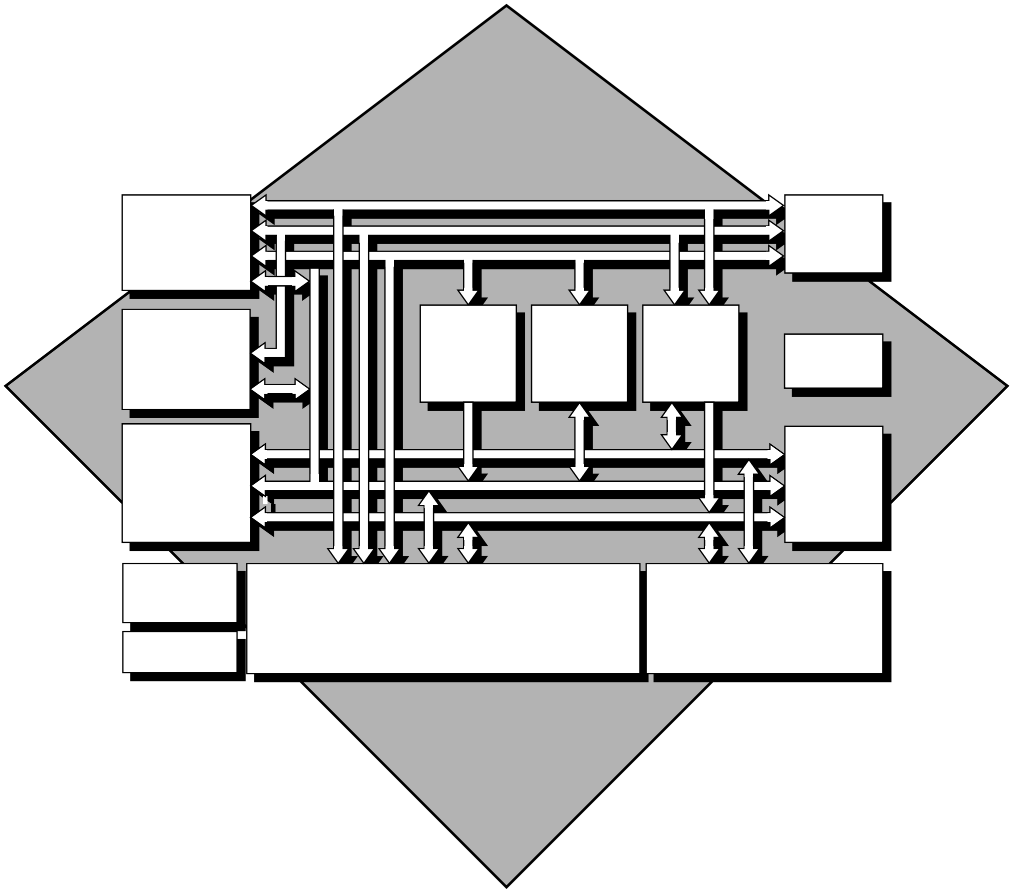
LIST OF FIGURES

LIST OF TABLES

LIST OF EXAMPLES

SECTION 1

OVERVIEW



1.1 OVERVIEW

The DSP56600 family of 16-bit high performance Digital Signal Processors (DSPs) is designed specifically for low-power digital handset cellular applications, and is capable of performing a wide variety of fixed-point DSP algorithms. The family architecture features a central processing module that is common to the various family members. A variety of highly integrated and cost-effective DSP devices can be built around this core, based upon a library of modules containing memories and peripherals. Current DSP applications require the very high execution speeds in a real time, Input/Output (I/O) intensive environment that this DSP can provide.

This *DSP56600 Family Manual* provides a description of the components that are common to all the DSP56600 family of DSPs. **Table 1-1** lists the documents needed to design properly with a member of the DSP56600 family.

Table 1-1 DSP56600 Family Product Literature

Document Name	Description	Order Number
DSP56600 Family Manual	Detailed description of the DSP56600-family architecture, the core processor, and instruction set	DSP56600FM/AD
DSP56603 User's Manual	Detailed description of the DSP56603 memory, peripherals, and interfaces	DSP56603UM/AD
DSP56603 Technical Data	Pin and package descriptions, electrical and timing specifications, and ordering information for the DSP56603	DSP56603/D

The central processor and instruction set are described in this manual. It is intended to be used with a family member’s *User’s Manua*l, such as those listed in **Table 1-1**. The *User’s Manual* presents the specifics of the device, including pin descriptions, operating modes, memory and peripherals. Packaging and timing information can be found in the device’s *Technical Data* sheet. As new chips are added to the DSP56600 family, more documents will be added to this list.

1.2 MANUAL ORGANIZATION

This manual describes the Central Processing Unit (CPU) of the DSP56600 family in detail. It is intended to be used with the appropriate DSP56600 family member user’s manual, which describes the CPU, programming models, and details of the instruction set. The appropriate DSP56600 family member *Technical Data* sheet provides timing, pinout, and packaging descriptions.

This manual is arranged in the following sections:

- **Section 1—Overview** introduces general DSP theory and discusses the features and benefits of the Motorola DSP56600 family of high performance, general purpose processors. A brief description of each section of the manual is also included.
- **Section 2—Central Architecture Overview** describes the central architecture of the DSP56600, which consists of the Data Arithmetic Logic Unit Data (ALU), Address Generation Unit (AGU), Program Control Unit (PCU), the Phase Lock Loop (PLL) based clock oscillator, JTAG Test Access Port, and On-Chip Emulation (OnCE™) circuitry. This section describes each subsystem and the buses interconnecting the major components in the DSP56600 central processing module. Detailed descriptions are provided in the subsequent sections.
- **Section 3—Data Arithmetic Logic Unit** describes in detail the Data ALU architecture and its programming model. An introduction to fractional and integer arithmetic is included, and other topics are discussed, such as unsigned and multiprecision arithmetic.
- **Section 4—Address Generation Unit** describes the AGU architecture, its programming model, addressing modes, and address modifiers.
- **Section 5—Program Control Unit** describes in detail the PCU architecture, its programming model, and hardware looping.
- **Section 6—Program Patch Logic** describes a method of fixing the program code, which is located in the on-chip ROM, without generating a new mask.
- **Section 7—Processing States** describes five processing states: Normal, Exception, Reset, Wait, and Stop.
- **Section 8—PLL and Clock Generator** describes the PLL and its functions.
- **Section 9—External Memory Interface (Port A)** describes the external memory port, its control register, and control signals.
- **Section 10—JTAG Port and OnCE Module** describes the JTAG Test Access Port, and the OnCE circuitry and its functions.
- **Section 11—Operating Modes and Memory Spaces** describes the DSP56600 operating mode pins which determine the reset vector address for start-up

after reset. The core memory map shows the partitions into program, X data, and Y data memory space.

- **Section 12—Development Tools** describes the hardware and software development tools that are available for the DSP56600 family.
- **Section 13—Additional Support** lists additional support resources and how to obtain them.
- **Appendix A—Instruction Set Details** provides a detailed description of each DSP56600 family instruction, its use, and its effect on the processor.
- **Appendix B—Instruction Timing** lists DSP56600 family instruction execution timings.
- **Appendix C—Benchmark Programs** lists DSP56600 family benchmark example programs and results.

In addition, an index, a table of contents, and lists of figures, tables, and examples are provided.

1.3 MANUAL CONVENTIONS

The following conventions are used in this manual:

- Bits within registers are always listed from Most Significant Bit (MSB) to Least Significant Bit (LSB).

Note: Other manuals may use the opposite convention, with bits listed from LSB to MSB.

- Bits within a register are indicated AA[n:0] when more than one bit is involved in a description. For purposes of description, the bits are presented as if they are contiguous within a register. However, this is not always the case. Refer to the programming model diagrams or to the programmer's sheets to see the exact location of bits within a register.
- When a bit is described as "set," its value is 1. When a bit is described as "cleared," its value is 0.
- Pins or signals that are asserted low (made active when pulled to ground) have an overbar over their name; for example, the $\overline{\text{MCS}}$ pin is asserted low.
- Hex values are indicated with a dollar sign ($) preceding the hex value as follows: $FFFB is the X memory address for the Interrupt Priority Register (IPR).

- Code examples are displayed in a monospaced font, as shown in **Example 1-1**.

Example 1-1 Sample Code Listing

```
    BFSET #$0007,X:PCC; Configure:                              line 1
                      ; MISO0, MOSI0, SCK0 for SPI master       line 2
                      ; ~SS0 as PC3 for GPIO                    line 3
```

- Pins or signals listed in code examples that are asserted low have a tilde in front of their names. In the previous example, line 3 refers to the $\overline{\text{SS0}}$ pin (shown as `~SS0`).
- The word “assert” means that a high true (active high) signal is pulled high to V_{CC} or that a low true (active low) signal is pulled low to ground. The word “deassert” means that a high true signal is pulled low to ground or that a low true signal is pulled high to V_{CC}. See **Table 1-2**.

Table 1-2 High True / Low True Signal Conventions

Signal/Symbol	Logic State	Signal State	Voltage
$\overline{\text{PIN}}$	True	Asserted	Ground[1]
$\overline{\text{PIN}}$	False	Deasserted	V_{CC}[2]
PIN	True	Asserted	V_{CC}
PIN	False	Deasserted	Ground

Note: 1. Ground is an acceptable low voltage level. See the appropriate data sheet for the range of acceptable low voltage levels (typically a TTL logic low).
2. V_{CC} is an acceptable high voltage level. See the appropriate data sheet for the range of acceptable high voltage levels (typically a TTL logic high).

- The word “reset” is used in three different contexts in this manual. There is a reset pin that is always written as “$\overline{\text{RESET}}$”, a reset instruction that is always written as “RESET”, and the word reset that refers to the reset function and is written in lower case with a leading capital letter as grammar dictates. The word “pin” is a generic term for any pin on the chip.

1.4 DSP FUNCTIONAL ADVANTAGES

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

- Filtering signals

- Convolution—mixing two signals
- Correlation—comparing two signals
- Rectifying, amplifying, and/or transforming a signal

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

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

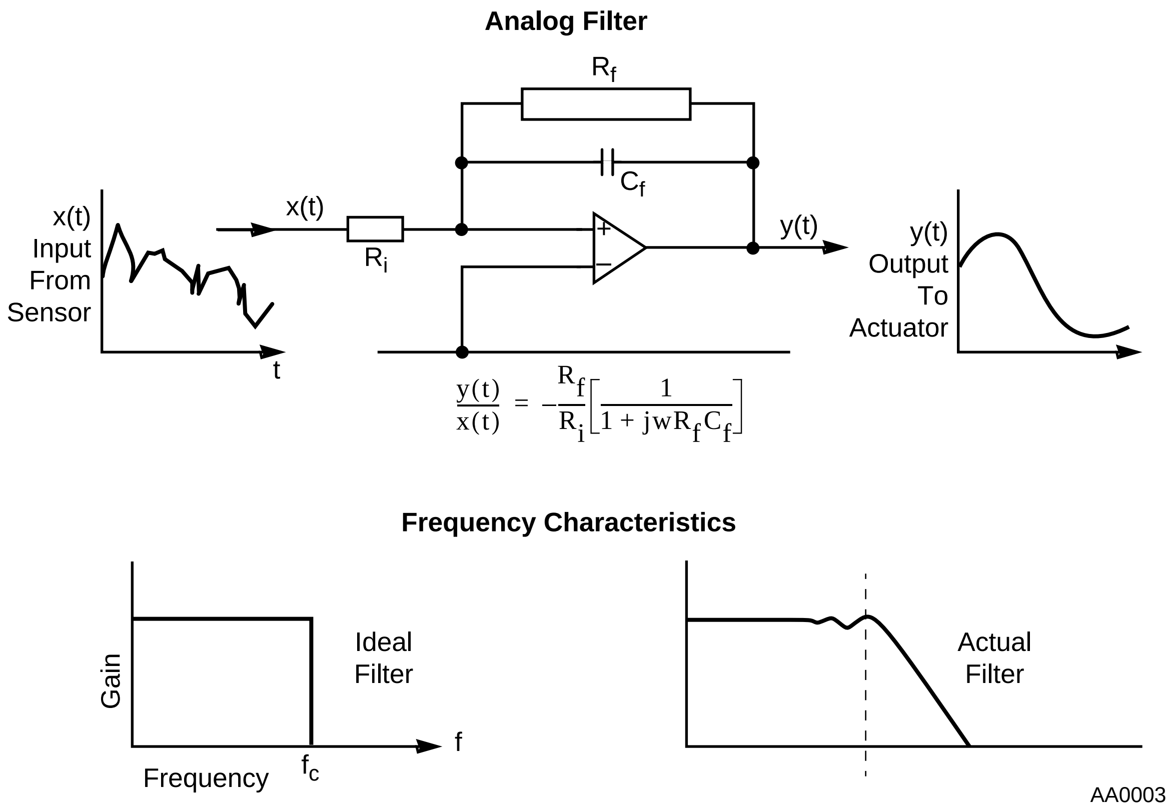


Figure 1-1 Analog Signal Processing

DSP Functional Advantages

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.

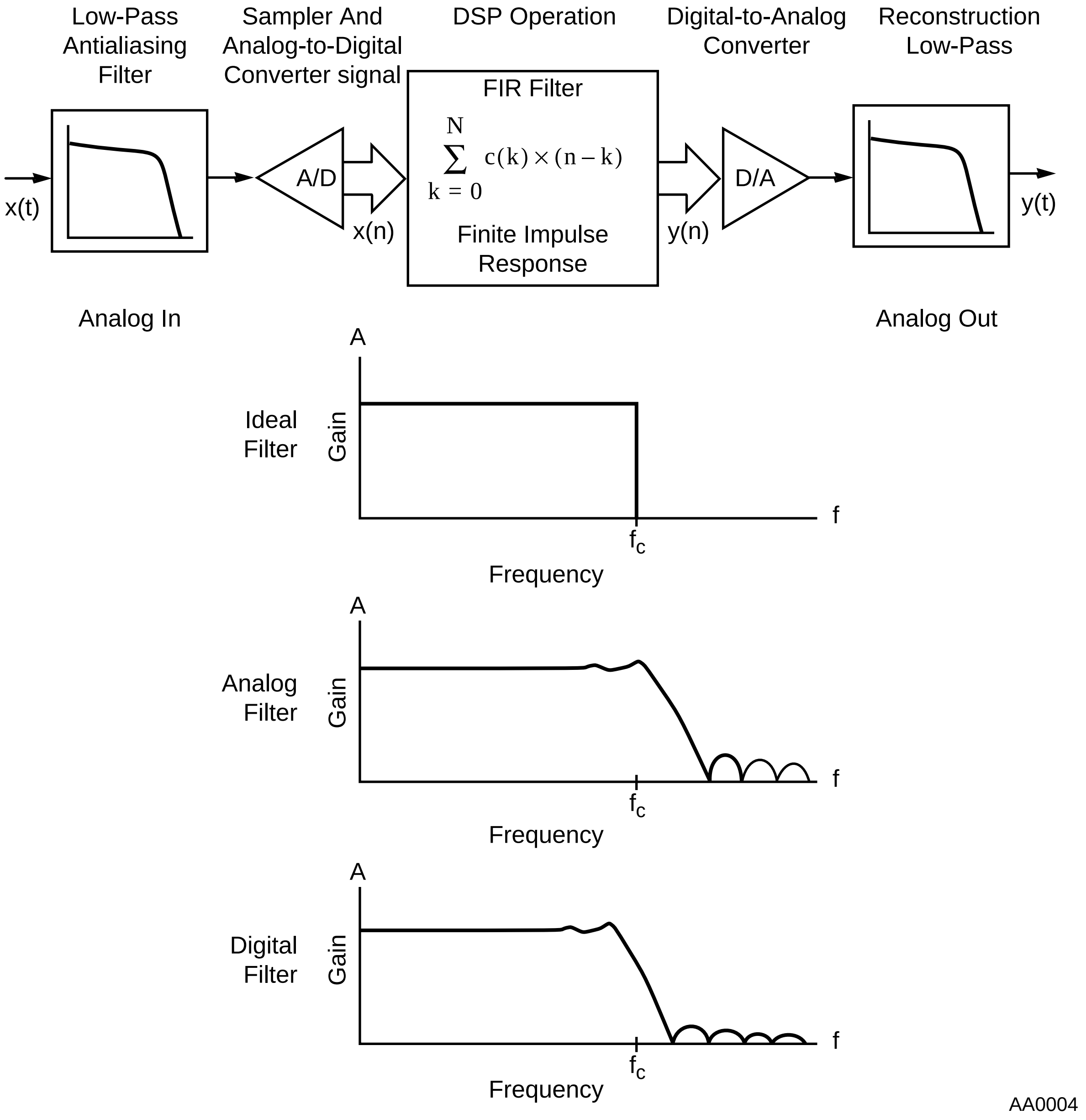


Figure 1-2 Digital Signal Processing

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

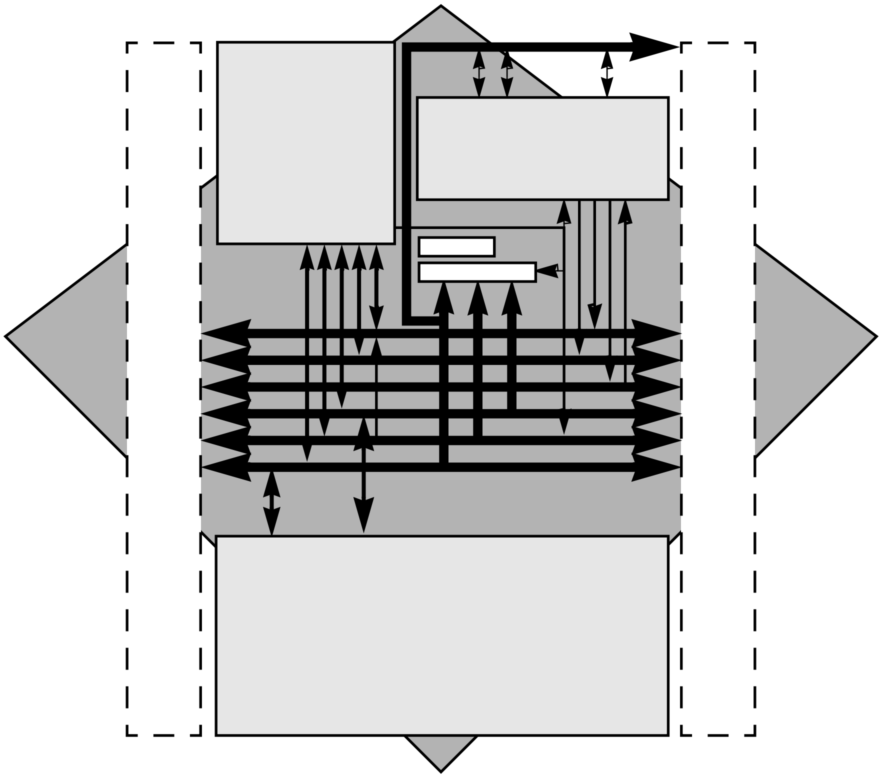
The filter that the DSP implements depends entirely upon the software. The DSP can directly implement any filter that can also be implemented using analog techniques. Also, adaptive filters can be easily implemented using DSP, whereas these filters are extremely difficult to implement using analog techniques.

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 DSPs, compared to analog-only circuits, include the following:

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

SECTION 2

CENTRAL ARCHITECTURE OVERVIEW



2.1 INTRODUCTION

This section describes the DSP56600 core, a member of Motorola's family of programmable CMOS Digital Signal Processors (DSPs). The design priorities for the DSP56600 core are:

- Low power dissipation
- Low cost
- High performance
- High integration

The DSP56600 core is based on the DSP56300 core, with the following modifications:

- All internal data and address buses, with the exception of the Program Data Bus, were reduced to 16 bits.
- All registers were reduced to 16 bits.
- The Direct Memory Access (DMA) module was removed.
- The instruction cache was removed.

An instruction decoding mechanism has been added to allow code developed for the DSP56300 chips to run on DSP56600 family members, and information on emulating DSP56600 members on DSP56300 chips is provided as an appendix to this document.

A significant new feature of the DSP56600 core is its instruction pipeline, which allows the core to execute instructions as rapidly as one instruction per clock cycle. Many modern DSP applications require extremely low power parts capable of very high execution speed in a real-time I/O intensive environment. The DSP56600 core, with its capability of executing one instruction per clock cycle, has the processing power to meet this demand. Power consumption is significantly reduced as compared with other chips, while still maintaining the rich instruction set of the DSP56300.

Lowered power consumption on DSP56600 family members is achieved in both active and standby modes. Power management units are included in all chip blocks in order to dynamically reduce each block's power consumption on a cycle by cycle basis. Power consumption scales down with clock frequency reduction, use of on-chip memory, use of on-chip peripherals, and use of Wait and Stop standby modes. External buses are driven only when required. On-chip memory expansion does not increase power dissipation significantly, because only memory modules being accessed consume power.

2.2 DSP56600 CORE FEATURES

The following lists some of the features of the DSP56600 core:

- 60 Million Instructions Per Second (MIPS) with a 60 MHz clock at 2.7 V
- Fully pipelined 16 × 16-bit parallel Multiplier-Accumulator (MAC)
- 40-bit parallel barrel shifter
- Highly parallel instruction set
- Position Independent Code (PIC) support
- Unique DSP addressing modes
- Nested hardware DO loops
- Fast auto-return interrupts
- On-chip 16-stage hardware stack with stack extension
- On-chip support for software patching and enhancements
- On-chip PLL
- On-Chip Emulation (OnCE) module
- Address tracing for debugging
- JTAG port compatible with the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)*

Low-power features of the DSP56600 core include the following:

- Very low power CMOS design
- Low power Wait standby mode
- Ultra-low power Stop mode
- Power management units for further power reduction
- Fully static logic, with operation frequency down to DC

2.3 DSP56600 CORE ARCHITECTURE

The DSP56600 core provides the following functional blocks:

- Data Arithmetic Logic Unit (Data ALU)

- Address Generation Unit (AGU)
- Program Control Unit (PCU)
- Program Patch Logic
- PLL and Clock Oscillator
- Expansion Port (Port A)
- JTAG Test Access Port and On-Chip Emulation (OnCE) module
- Memory

In addition, each member of the DSP56600 family provides its own set of on-chip peripherals for enhanced functionality.

To provide data exchange between these blocks, the following buses are implemented:

- Peripheral I/O Expansion Bus (PIO_EB) to peripherals
- Program Memory Expansion Bus (PM_EB) to Program ROM
- X Memory Expansion Bus (XM_EB) to X Memory
- Y Memory Expansion Bus (YM_EB) to Y Memory
- Global Data Bus (GDB) between Program Control Unit and other core structures
- Program Data Bus (PDB) for carrying program data throughout the core
- X Memory Data Bus (XDB) for carrying X data throughout the core
- Y Memory Data Bus (YDB) for carrying Y data throughout the core
- Program Address Bus (PAB) for carrying program memory addresses throughout the core
- X Memory Address Bus (XAB) for carrying X memory addresses throughout the core
- Y Memory Address Bus (YAB) for carrying Y memory addresses throughout the core

With the exception of the Program Data Bus (PDB), all internal buses on the DSP56600 family members are 16-bit buses. The PDB is a 24-bit bus.

The block diagram of one member of the DSP56600 family, the DSP56603, is shown in **Figure 2-1**. This diagram illustrates the core blocks of the DSP56600, and shows representative peripherals for a DSP56600 chip implementation.

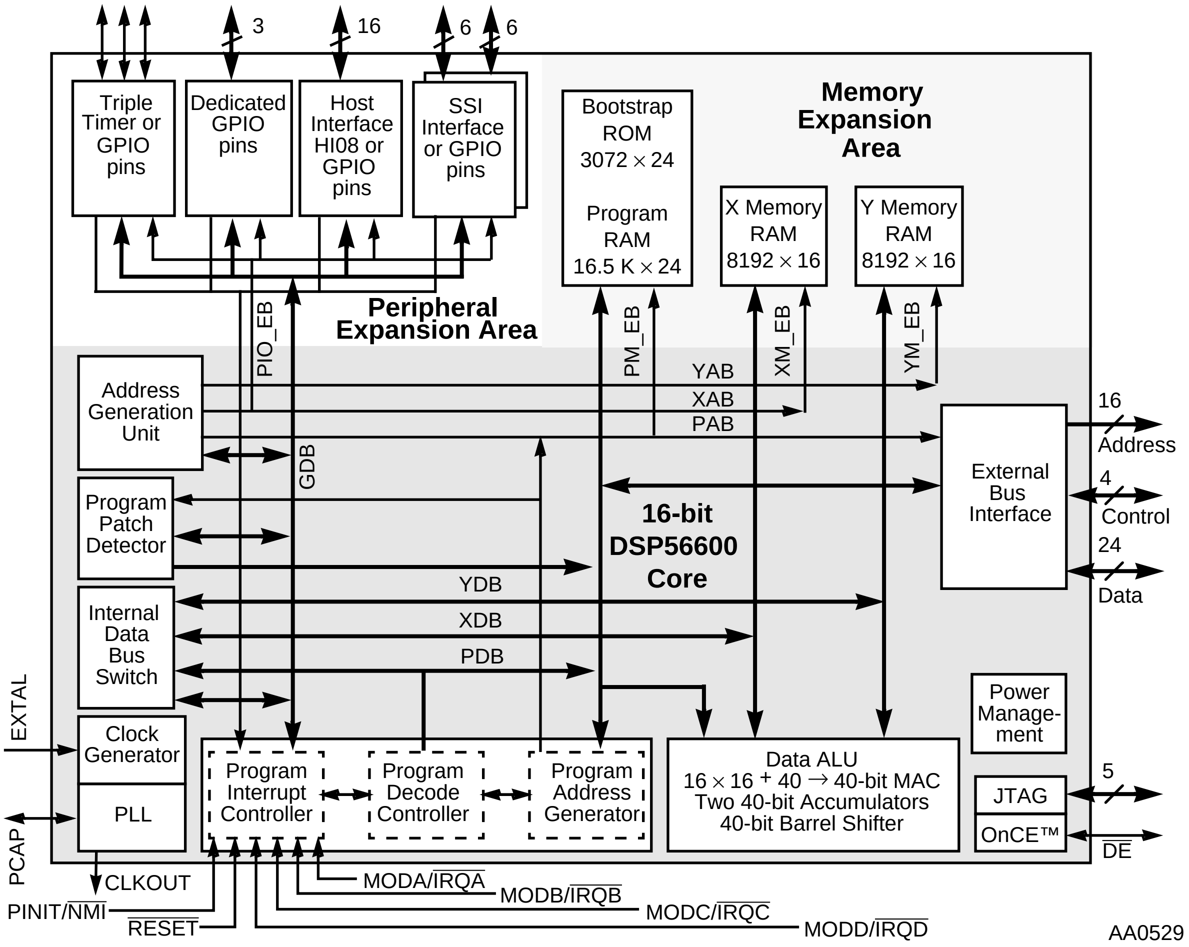


Figure 2-1 DSP56603 Block Diagram

2.3.1 Data Arithmetic Logic Unit

The Data Arithmetic Logic Unit (Data ALU) performs all the arithmetic and logical operations on data operands in the DSP56600 core. The components of the Data ALU are as follows:

- Four 16-bit input general purpose registers: X1, X0, Y1, and Y0
- A parallel, fully pipelined Multiplier-Accumulator unit (MAC)
- Six Data ALU registers (A2, A1, A0, B2, B1, and B0) that are concatenated into two general purpose, 40-bit accumulators, A and B

- An accumulator shifter that is an asynchronous parallel shifter with a 40-bit input and a 40-bit output
- A Bit Field Unit (BFU) with a 40-bit barrel shifter
- Two data bus shifter/limiter circuits

2.3.1.1 Data ALU Registers

The Data ALU registers can be read or written over the X Data Bus (XDB) and the Y Data Bus (YDB) as 16- or 32-bit operands. The source operands for the Data ALU, which can be 16, 32, or 40 bits, always originate from Data ALU registers. The results of all Data ALU operations are stored in an accumulator.

All the Data ALU operations are performed in 2 clock cycles in pipeline fashion so that a new instruction can be initiated in every clock, yielding an effective execution rate of one instruction per clock cycle. The destination of every arithmetic operation can be used as a source operand for the immediate following operation without penalty.

2.3.1.2 Multiplier-Accumulator (MAC)

The Multiplier-Accumulator (MAC) unit comprises the main arithmetic processing unit of the DSP56600 core and performs all of the calculations on data operands. In the case of arithmetic instructions, the unit accepts as many as three input operands and outputs one 40-bit result of the following form, Extension:Most Significant Product:Least Significant Product (EXT:MSP:LSP).

The multiplier executes 16-bit × 16-bit, parallel, fractional multiplies, between two's-complement signed, unsigned, or mixed operands. The 32-bit product is right-justified and added to the 40-bit contents of either the A or B accumulator. A 40-bit result can be stored as a 16-bit operand. The LSP can either be truncated or rounded into the MSP. Rounding is performed if specified.

2.3.2 Address Generation Unit

The Address Generation Unit (AGU) performs the effective address calculations using integer arithmetic necessary to address data operands in memory and contains the registers used to generate the addresses. It implements four types of arithmetic: linear, modulo, multiple wrap-around modulo, and reverse-carry. The AGU operates in parallel with other chip resources to minimize address-generation overhead.

The AGU is divided into two halves, each with its own Address Arithmetic Logic Unit (Address ALU). Each Address ALU has four sets of register triplets, and each register triplet is composed of an address register, an offset register, and a modifier register. The two Address ALUs are identical. Each contains a 16-bit full adder (called an offset adder).

A second full adder (called a modulo adder) adds the summed result of the first full adder to a modulo value that is stored in its respective modifier register. A third full adder (called a reverse-carry adder) is also provided.

The offset adder and the reverse-carry adder are in parallel and share common inputs. The only difference between them is that they 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 from its respective address register file during one instruction cycle. The contents of the associated modifier register specifies the type of arithmetic to be used in the address register update calculation. The modifier value is decoded in the Address ALU.

2.3.3 Program Control Unit

The Program Control Unit (PCU) performs instruction prefetch, instruction decoding, hardware DO loop control and exception processing. The PCU implements a seven-stage pipeline and controls the different processing states of the DSP56600 core. The PCU consists of three hardware blocks:

- Program Decode Controller (PDC)
- Program Address Generator (PAG)
- Program Interrupt Controller (PIC)

The PDC decodes the 24-bit instruction loaded into the instruction latch and generates all signals necessary for pipeline control. The PAG contains all the hardware needed for program address generation, system stack and loop control. The PIC arbitrates among all interrupt requests (internal interrupt, s as well as the five external requests IRQA, IRQB, IRQC, IRQD, and NMI), and generates the appropriate interrupt vector address.

The PCU implements its functions using the following registers:

- PC—Program Counter register
- SR—Status Register
- LA—Loop Address register

- LC—Loop Counter register
- VBA—Vector Base Address register
- SZ—Size register
- SP—Stack Pointer
- OMR—Operating Mode Register
- SC—Stack Counter register

The PCU also includes a hardware System Stack (SS).

2.3.4 Program Patch Logic

The Program Patch Logic (PPL) block provides the DSP56600 core user a way to fix the program code in the on-chip ROM without generating a new mask. Implementing the code correction is done by replacing a piece of ROM-based code with a patch program stored in RAM. The PPL consists of four Patch Address Registers (PAR1–PAR4) and four patch address comparators. Each PAR points to a starting location in the ROM code where the program flow is to be changed. The PC register in the PCU is compared to each PAR. When an address of a fetched instruction is identical to an address stored in one of the PARs, the Program Data Bus (PDB) is forced to a corresponding JMP instruction, replacing the instruction that otherwise would have been fetched from the ROM.

2.3.5 PLL and Clock Oscillator

The DSP56600 core features a Phase Lock Loop (PLL) clock oscillator in its central processing module. The PLL allows the processor to operate at a high internal clock frequency using a low frequency clock input, a feature that offers two immediate benefits:

- A lower frequency clock input reduces the overall electromagnetic interference generated by a system.
- The ability to oscillate at different frequencies reduces costs by eliminating the need to add additional oscillators to a system.

The clock generator in the DSP56600 core is composed of two main blocks: the PLL, which performs clock input division, frequency multiplication, and skew elimination; and the Clock Generator (CLKGEN), which performs low power division and clock pulse generation.

2.3.6 Expansion Port (Port A)

Port A is the memory expansion port is used for both program and data memory. It provides an easy to use, low part-count connection with fast or slow static memories and with I/O devices. 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 two clock cycles. External memory can be as large as 64 K × 24-bit program memory space, depending on chip configuration. An internal wait state generator can be programmed to insert as many as thirty-one wait states if access to slower memory or I/O device is required.

For power-sensitive applications and applications that do not require external memory, Port A can be fully disabled.

2.3.7 JTAG Test Access Port and On-Chip Emulator (OnCE)

The DSP56600 core provides a dedicated user-accessible Test Access Port (TAP) that is fully compatible with the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)* . Problems associated with testing high density circuit boards have led to development of this standard under the sponsorship of the Test Technology Committee of IEEE and the Joint Test Action Group (JTAG). The DSP56600 core implementation supports circuit-board test strategies based on this standard.

The test logic includes a Test Access Port (TAP) consisting of four dedicated signal pins, a 16-state controller, and three test data registers. A boundary scan register links all device signal pins into a single shift register. The test logic, implemented utilizing static logic design, is independent of the device system logic.

The On-Chip Emulation (OnCE) module provides a means of interacting with the DSP56600 core and its peripherals non-intrusively so that a user can examine registers, memory, or on-chip peripherals. This facilitates hardware and software development on the DSP56600 core processor. OnCE module functions are provided through the JTAG TAP pins.

2.3.8 On-Chip Memory

The memory space of the DSP56600 core is partitioned into program memory space, X data memory space, and Y data memory space. The data memory space is divided into X data memory and to Y data memory in order to work with the two Address ALUs and to feed two operands simultaneously to the Data ALU. Memory space typically includes internal RAM and ROM and can be expanded off-chip under software control.

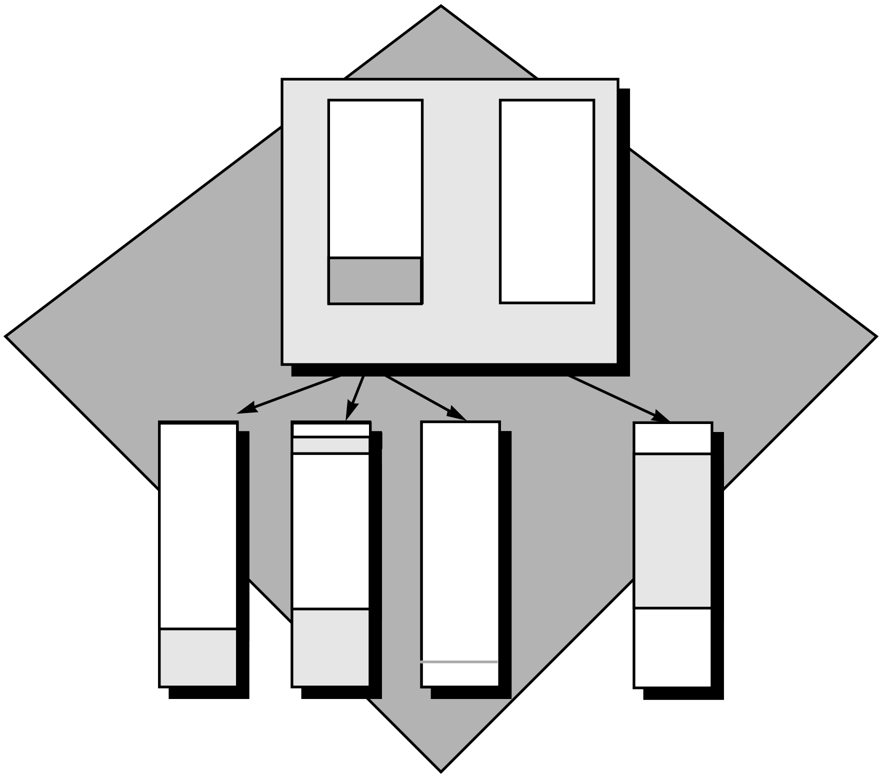
Both internal and external memory configuration is specific to each member of the DSP56600 family. For complete details of memory configuration, see the *User's Manual* for the particular DSP56600 family member.

2.3.9 Peripherals

Each member of the DSP56600 family can be configured with its own set of on-chip peripherals for communicating with external devices or memory, as well as for providing additional on-chip functionality. For complete details of on-chip peripherals, see the *User's Manual* for the particular DSP56600 family member.

SECTION 3

DATA ARITHMETIC LOGIC UNIT



3.1 INTRODUCTION

This section describes the architecture and the operation of the Data Arithmetic Logic Unit (Data ALU), the block where all the arithmetic and logical operations on data operands are performed. In addition, this section describes the arithmetic and rounding performed by the Data ALU, as well as its programming model.

3.2 DATA ALU ARCHITECTURE

The Data ALU performs all the arithmetic and logical operations on data operands in the DSP56600 core.

The Data ALU registers can be read or written over the X Data Bus (XDB) and the Y Data Bus (YDB) as 16- or 32-bit operands. The source operands for the Data ALU, which can be 16, 32, or 40 bits, always originate from Data ALU registers. The results of all Data ALU operations are stored in an accumulator.

All the Data ALU operations are performed in 2 clock cycles in pipeline fashion so that a new instruction can be initiated in every clock, yielding an effective execution rate of one instruction per clock cycle. The destination of every arithmetic operation can be used as a source operand for the immediate following operation without penalty.

The components of the Data ALU are as follows:

- Four 16-bit input registers
- A parallel, fully pipelined Multiplier-Accumulator unit (MAC)
- Two 32-bit accumulator registers
- Two 8-bit accumulator extension registers
- A Bit Field Unit (BFU) with a 40-bit barrel shifter
- An accumulator shifter
- Two data bus shifter/limiter circuits

Figure 3-1 provides a block diagram of the Data ALU.

Data ALU Architecture

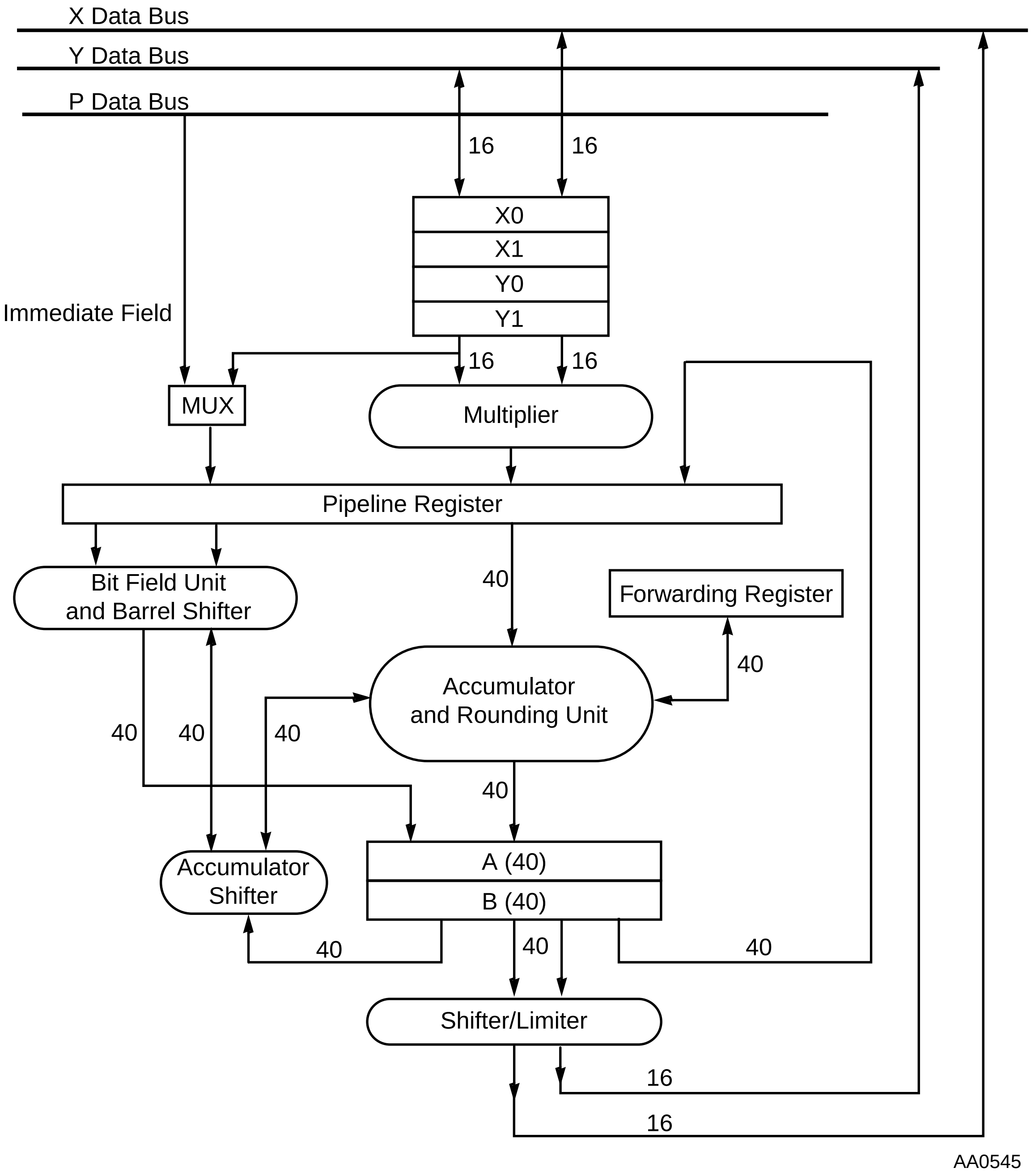


Figure 3-1 Data ALU Block Diagram

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

X1, X0, Y1, and Y0 are four 16-bit, general purpose data registers. They can be treated as four independent 16-bit registers or as two 32-bit registers called X and Y, formed by 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 or barrel shifter. They are used as Data ALU source operands, allowing new operands to be loaded for the next instruction while the register contents are used by the current instruction. The registers can also be read back out to the appropriate data bus.

3.2.2 MAC Unit

The Multiplier-Accumulator (MAC) unit comprises the main arithmetic processing unit of the DSP56600 core and performs all of the calculations on data operands. In the case of arithmetic instructions, the unit accepts as many as three input operands and outputs one 40-bit result of the following form, Extension:Most Significant Product:Least Significant Product (EXT:MSP:LSP). The operation of the MAC unit occurs independently of 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 input to the multiplier can only come from the X or Y registers. The multiplier executes 16-bit × 16-bit, parallel, fractional multiplies, between two’s-complement signed, unsigned, or mixed operands. The 32-bit product is right-justified and added to the 40-bit contents of either the A or B accumulator.

The 40-bit sum is stored back in the same accumulator. The MAC operation is fully pipelined and takes 2 clock cycles to complete. In the first clock cycle, the multiply is performed and the product is stored in the pipeline register. In the second clock cycle, the accumulator is added or subtracted. If a multiply without accumulation (MPY) is specified in the instruction, the MAC clears the accumulator and then adds the contents to the product. When a 40-bit result is to be stored as a 16-bit operand, the LSP can either be truncated or rounded into the MSP. Rounding is performed if specified in the DSP instruction (e.g., using the MACR instruction). The rounding performed is either convergent rounding (round-to-nearest-even) or two’s-complement rounding. The type of rounding is specified by the Rounding Mode bit (RM) in the Status Register (SR). The bit in the accumulator that is rounded is specified by the Scaling Mode bits (S0 and S1) in the SR.

It is possible to saturate the arithmetic unit's result going into the accumulator so that it fits into 32 bits (MSP and LSP). This process is referred to as arithmetic saturation. It is activated by the Arithmetic Saturation Mode (SM) bit in the SR. The purpose of this mode is to provide for algorithms that do not recognize or cannot take advantage of the Extension Accumulator (EXT). For more information, refer to **Arithmetic Saturation Mode** on page 3-14.

3.2.3 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, 40-bit accumulators, A and B. Each of these accumulators consists of three concatenated registers,A2:A1:A0 and B2:B1:B0, respectively. The 16-bit MSP is stored in A1 or B1; the 16-bit LSP is stored in A0 or B0. The 8-bit EXT is stored in A2 or B2.

Reading the A or B accumulators over the XDB and YDB buses 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 if limiting occurs. Only the value transferred over the XDB or YDB is limited. This process is commonly referred to as transfer saturation and should not be confused with the Arithmetic Saturation mode.

The overflow protection is performed after the contents of the accumulator have been shifted according to the scaling mode. Shifting and limiting are performed only when the entire 40-bit A or B register is specified as the source for a parallel data move over the XDB or YDB buses. When an individual register within an accumulator (A0, A1, A2, B0, B1, or B2 register) is specified as the source for a parallel data move, shifting and limiting are not performed. When the 8-bit wide accumulator extension register (A2 or B2) is specified as the source for a parallel data move, it is sign-extended to produce the full 16-bit wide word. The A and B accumulators serve as buffer registers between the arithmetic unit and the XDB and YDB buses. These registers are used as both Data ALU source and destination operands.

Automatic sign extension of the 40-bit accumulators is provided when the A or B accumulator is written with a smaller operand. Sign extension can occur when writing to the A or B accumulator from the XDB or YDB bus, 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 MSP:LSP, the low-order portion of the accumulator register. The EXT portion is sign-extended from MSP. No sign extension is performed if an individual 16-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 detect overflows out of the data shifter so that the limiter can substitute one of several constants to minimize errors caused by the overflow.

3.2.4 Accumulator Shifter

The accumulator shifter is an asynchronous parallel shifter with a 40-bit input and a 40-bit output that is implemented immediately before the MAC accumulator input. The source accumulator shifting operations are as follows:

- No Shift (Unmodified)
- 16-bit Right Shift (Arithmetic) for DMAC
- Force to zero

3.2.5 Bit Field Unit (BFU)

The Bit Field Unit (BFU) contains a 40-bit parallel bidirectional shifter with a 40-bit input and a 40-bit output, mask generation unit, and logic unit. The BFU is used in the following operations:

- Multibit Left Shift (Arithmetic or Logical) for ASL, LSL
- Multibit Right Shift (Arithmetic or Logical) for ASR, LSR
- 1-bit Rotate (Right or Left) for ROR, ROL
- Bit Field Merge, Insert, and Extract for MERGE, INSERT, EXTRACT, and EXTRACTU
- Count Leading Bits for CLB
- Fast Normalization for NORMF
- Logical operations for AND, OR, EOR, and NOT

3.2.6 Data Shifter/Limiter

The data shifter/limiter circuits provide special post-processing on data read from the A and B accumulators out to the XDB or YDB buses. There are two independent shifter/limiter circuits, one for the XDB bus and one for the YDB bus. Each consists of a shifter followed by a limiting circuit.

3.2.6.1 Scaling

The data shifters in the shifters/limiters unit can perform the following data shift operations:

- Scale up—shift data one bit to the left
- Scale down—shift data one bit to the right
- No scaling—pass the data unshifted

Each data shifter has a 16-bit output with overflow indication. 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 (FFTs) to be implemented in a regular fashion. The data shifters are controlled by the Scaling Mode bits (S0 and S1, bits 11 and 10) in the SR.

3.2.6.2 Limiting

In the DSP56600 core, the Data ALU accumulators A and B have eight extension bits. Limiting occurs 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 DSP56600 core 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 32-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., the signed integer portion of the accumulator is not in use), the data limiter is disabled, and the operand is not modified. If the contents of the selected source accumulator cannot be represented without overflow in the destination operand size, the data limiter substitutes a limited data value having maximum magnitude (saturated) and having the same sign as the source accumulator contents:

- $7FFF for 16-bit positive numbers
- $7FFF FFFF for 32-bit positive numbers
- $8000 for 16-bit negative numbers
- $8000 0000 for 32-bit negative numbers

This process is called transfer saturation. The value in the accumulator register is not shifted or limited and can be reused within the Data ALU. When limiting does occur, a flag is set and latched in the SR.

3.3 DATA ALU ARITHMETIC AND ROUNDING

The following paragraphs describe the Data ALU data representation, rounding modes, and arithmetic methods.

3.3.1 Data Representation

The DSP56600 core uses a fractional data representation for all Data ALU operations. **Figure 3-2** shows the bit weighting of words, long words, and accumulator operands for this representation. The decimal points are all aligned and are left-justified.

The most negative number that can be represented is –1.0. The internal representation is $8000 for words and $80000000 for long words.

The most positive word is $7FFF or $1 – 2^{–15}$ and the most positive long word is $7FFFFFFF or $1 – 2^{–31}$. 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 –256.

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 (A2 or B2). The least significant accumulator register (A0 or B0) 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 3-2**).

Data ALU Arithmetic and Rounding

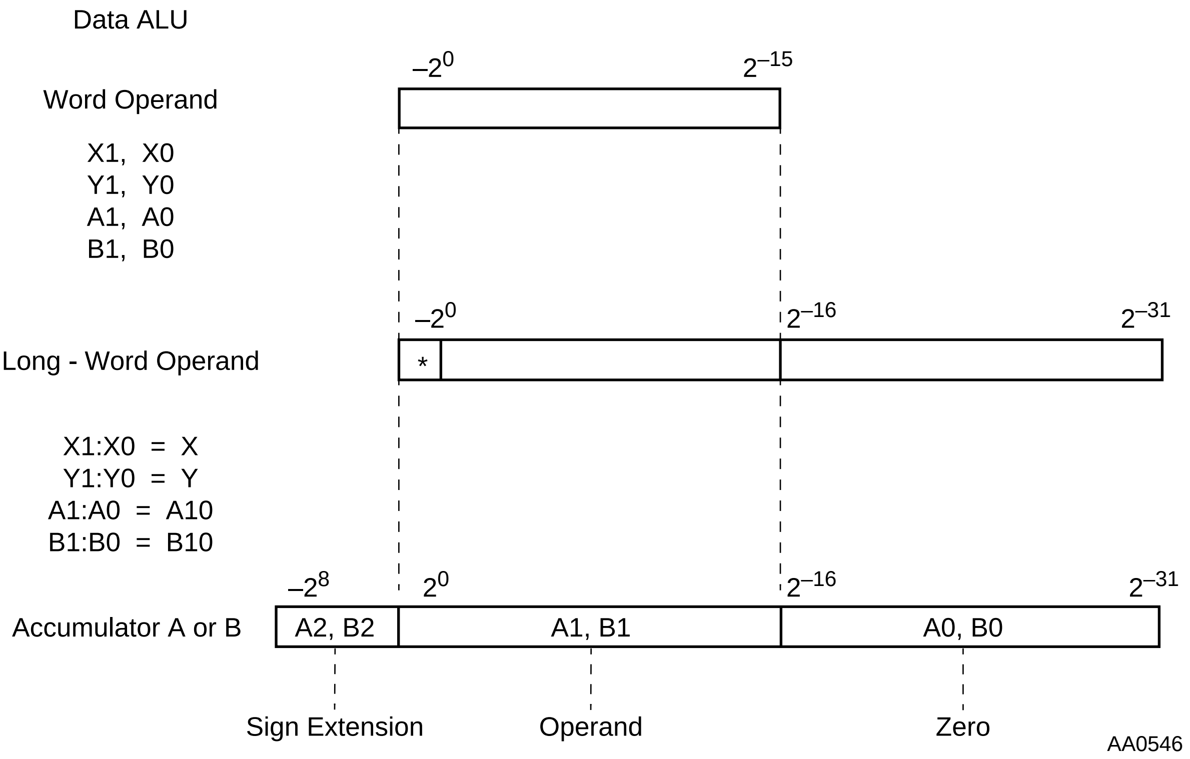


Figure 3-2 Bit Weighting and Alignment of Operands

The number representation for integers is between ± 2 $^{(N-1)}$. 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 when the numbers are multiplied or divided. An example of two numbers multiplied together is given in **Figure 3-3**.

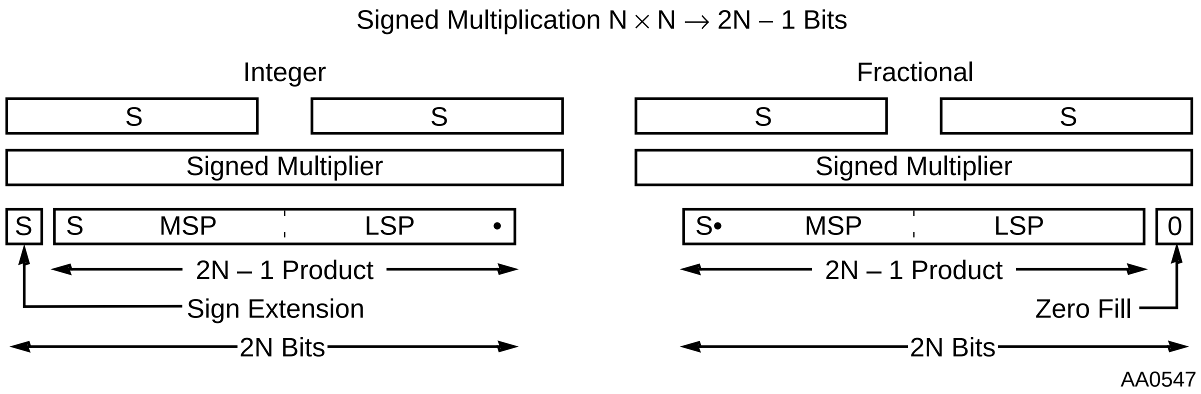


Figure 3-3 Integer/Fractional Multiplication

The key difference is in the alignment of the 2N – 1 bit product. In fractional multiplication, the 2N – 1 significant product bits should be left-aligned, and a 0 filled in the LSB to maintain fractional representation. In integer multiplication, the 2N – 1 significant product bits should be right-aligned, and the sign bit duplicated to maintain integer representation. Since the DSP56600 core incorporates a fractional array multiplier, it always aligns the 2N – 1 significant product bits to the left.

Note: The DSP56600 core always aligns the 2N – 1 significant product bits to the left in fractional multiplication.

3.3.2 Rounding Modes

The DSP56600 core's Data ALU performs rounding of the accumulator register to single precision if requested in the instruction. The upper portion of the accumulator is rounded according to the contents of the lower portion of the accumulator. The boundary between the lower portion and the upper portion is determined by the Scaling Mode bits (S0 and S1) in the SR. Two types of rounding are implemented: convergent rounding and two's-complement rounding. The type of rounding is selected by the Rounding Mode (RM) bit in the MR portion of the SR.

3.3.2.1 Convergent Rounding

Convergent rounding (also called round-to-nearest even number) is the default rounding mode. The traditional rounding method rounds up any value greater than one-half and rounds down any value less than one-half. The question arises as to which way one-half should be rounded. If it is always rounded one way, the results are eventually biased in that direction. Convergent rounding solves the problem by rounding down if the number is even (LSB = 0) and rounding up if the number is odd (LSB = 1). **Figure 3-4** shows the four cases for rounding a number in the A1 (or B1) register. If scaling is set in the SR , the rounding position is updated to reflect the alignment of the result when it is put on the data bus. However, the contents of the register are not scaled.

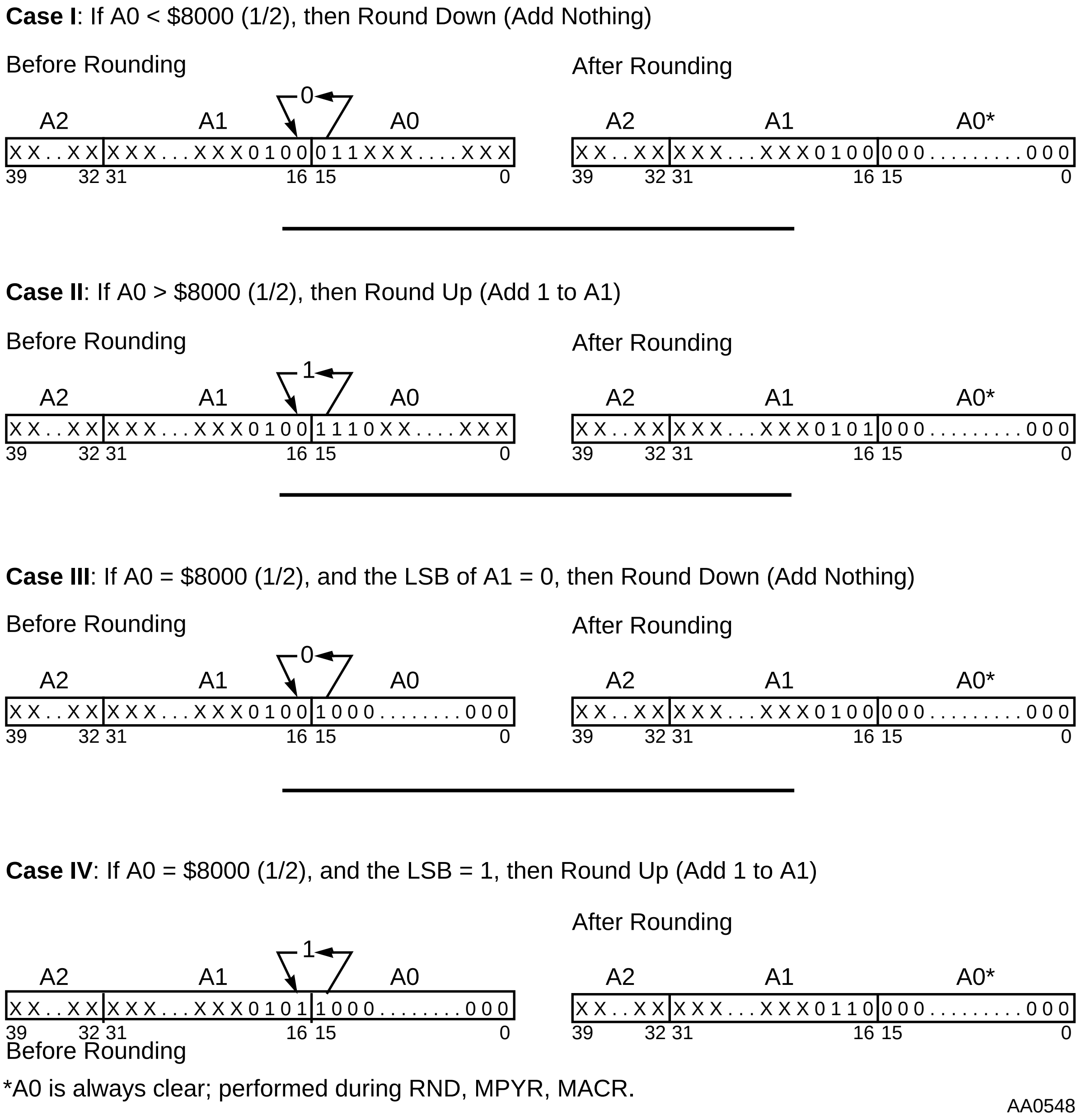

Case I: If A0 < $8000 (1/2), then Round Down (Add Nothing)
Before Rounding
After Rounding
0
A2
A1
A0
A0*
X X . . X X | X X X . . . X X X 0 1 0 0 | 0 1 1 X X X . . . . X X X
X X . . X X | X X X . . . X X X 0 1 0 0 | 0 0 0 . . . . 0 0 0
39 32 31 16 15 0
Case II: If A0 > $8000 (1/2), then Round Up (Add 1 to A1)
Before Rounding
After Rounding
1
X X . . X X | X X X . . . X X X 0 1 0 0 | 1 1 1 0 X X . . . . X X X
X X . . X X | X X X . . . X X X 0 1 0 1 | 0 0 0 . . . . 0 0 0
Case III: If A0 = $8000 (1/2), and the LSB of A1 = 0, then Round Down (Add Nothing)
Before Rounding
After Rounding
0
X X . . X X | X X X . . . X X X 0 1 0 0 | 1 0 0 0 . . . . 0 0 0
X X . . X X | X X X . . . X X X 0 1 0 0 | 0 0 0 . . . . 0 0 0
Case IV: If A0 = $8000 (1/2), and the LSB = 1, then Round Up (Add 1 to A1)
After Rounding
1
X X . . X X | X X X . . . X X X 0 1 0 1 | 1 0 0 0 . . . . 0 0 0
X X . . X X | X X X . . . X X X 0 1 1 0 | 0 0 0 . . . . 0 0 0
Before Rounding
*A0 is always clear; performed during RND, MPYR, MACR.
AA0548

Figure 3-4 Convergent Rounding (No Scaling)

3.3.2.2 Two's-Complement Rounding

When two's-complement rounding is selected by setting the Rounding Mode (RM) bit in the SR, all values greater than or equal to one-half are rounded up and all values less than one-half are rounded down. Therefore, a small positive bias is introduced. **Figure 3-5** shows the four cases for rounding a number in the A1 (or B1) register. If scaling is set in the SR, the rounding position is updated to reflect the alignment of the result when it is put on the data bus. However, the contents of the register are not scaled.

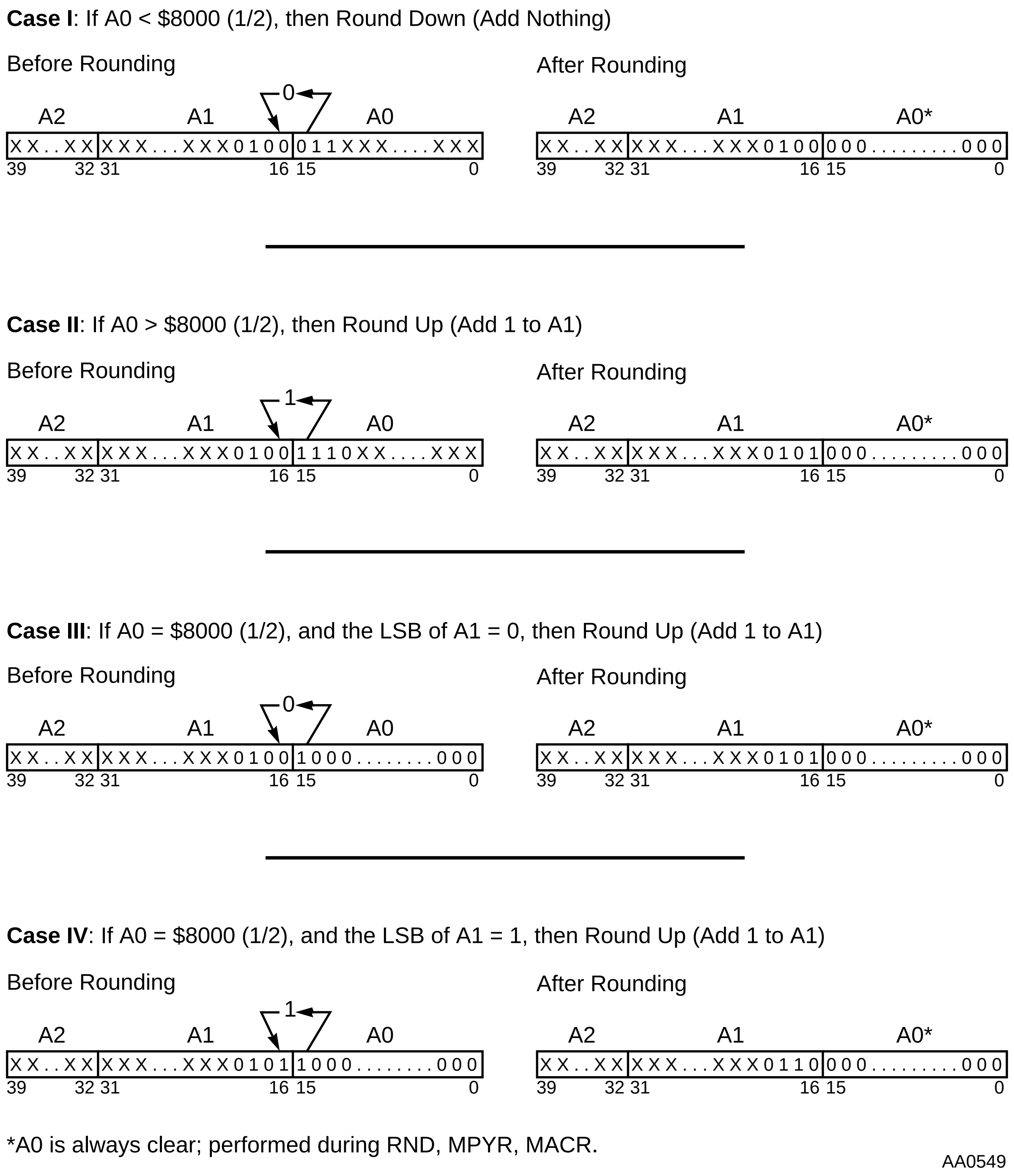


Figure 3-5 Two's-Complement Rounding (No Scaling)

3.3.3 Arithmetic Saturation Mode

By setting the Arithmetic Saturation Mode (SM) bit in the SR, the arithmetic unit's result is limited to 32 bits (MSP and LSP). The highest dynamic range of the machine is then limited to 32 bits. The purpose of this bit is to provide a saturation mode for algorithms that do not recognize or cannot take advantage of the extension accumulator.

The arithmetic saturation logic operates by checking three bits of the 40-bit result after rounding: two bits of the extension byte (EXT[7] and EXT[0]) and one bit on the MSP (MSP[15]). The result obtained in the accumulator when the SM bit is set to 1 is shown in **Table 3-1**:

Table 3-1 Actions of the Arithmetic Saturation Mode (SM = 1)

EXT[7]	EXT[0]	MSP[15]	Result in Accumulator
0	0	0	unchanged
0	0	1	$00 7FFF FFFF
0	1	0	$00 7FFF FFFF
0	1	1	$00 7FFF FFFF
1	0	0	$FF 8000 0000
1	0	1	$FF 8000 0000
1	1	0	$FF 8000 0000
1	1	1	unchanged

The two saturation constants $00 7FFF FFFF and $FF 8000 0000 are not affected by the scaling mode. In the same way, the rounding of the saturation constant during execution of the MPYR, MACR, and RND instructions is independent of the scaling mode: $00 7FFF FFFF is rounded to $00 7FFF 0000 and $FF 8000 0000 to $FF 8000 0000.

When in Arithmetic Saturation mode, the Overflow bit (V bit) in the SR is set if the Data ALU result is not representable in the 32-bit accumulator; that is, arithmetic saturation has occurred. This also implies that the Limiting bit (L bit) in the SR is set when an arithmetic saturation occurs.

Note: The Arithmetic Saturation mode is *always* disabled during the execution of the following instructions: TFR, Tcc, DMACsu, DMACuu, MACsu, MACuu, MPYsu, MPYuu, CMPU, and all BIT FIELD UNIT operations. If the result of these instructions should be saturated, a MOVE A,A (or B,B) instruction must be added following the original instruction (provided no scaling is set). However, the V bit of the SR is never set by the arithmetic saturation of the accumulator during the MOVE A,A (or B,B). Only the L bit is set.

3.3.4 Multi-Precision Arithmetic Support

A set of Data ALU operations is provided in order to facilitate multi-precision multiplications. When these instructions are used, the multiplier accepts some combinations of signed two's-complement format and unsigned format. **Table 3-2** shows these instructions.

Table 3-2 Acceptable Signed and Unsigned Two's-Complement Multiplication

Instruction	Description
MPY/MAC su	Multiplication and multiply-accumulate with signed times unsigned operands
MPY/MAC uu	Multiplication and multiply-accumulate with unsigned times unsigned operands
DMACss	Multiplication with signed times signed operands and 16-bit arithmetic right shift of the accumulator before accumulation
DMACsu	Multiplication with signed times unsigned operands and 16-bit arithmetic right shift of the accumulator before accumulation
DMACuu	Multiplication with unsigned times unsigned operands and 16-bit arithmetic right shift of the accumulator before accumulation

Figure 3-6 shows how the DMAC instruction is implemented inside the Data ALU.

Data ALU Arithmetic and Rounding

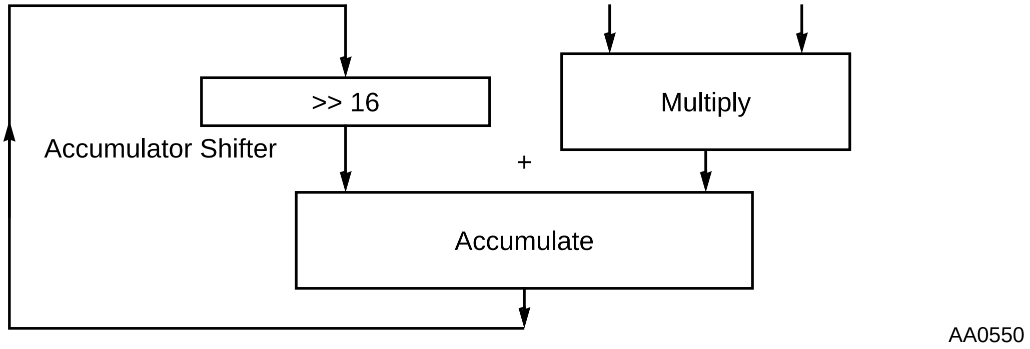


Figure 3-6 DMAC Implementation

Figure 3-7 illustrates the use of these instructions in the case of a double-precision multiplication. The signed × signed operation is used to multiply or multiply-accumulate the two upper signed portions of two signed double-precision numbers. The unsigned × signed operation is used to multiply or multiply-accumulate the upper signed portion of one double-precision number with the lower unsigned portion of the other double-precision number. The unsigned × unsigned operation is used to multiply or multiply-accumulate the lower, unsigned portion of one double-precision number with the lower unsigned portion of the other double-precision number.

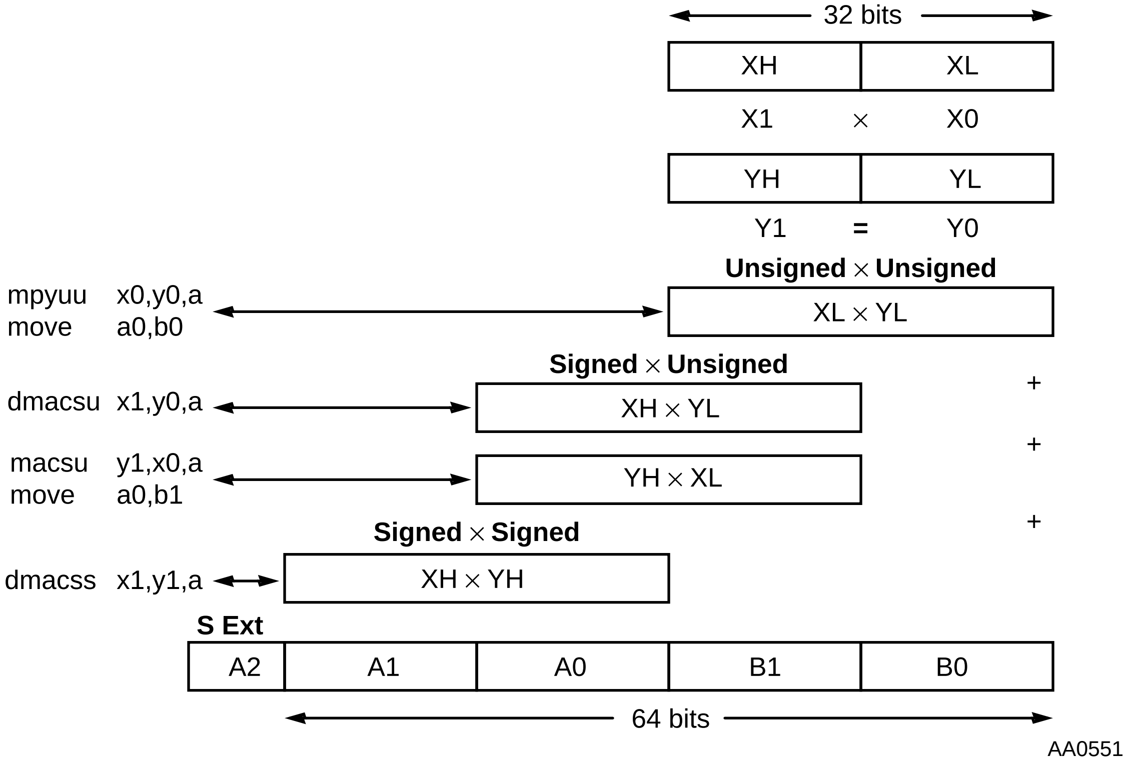


Figure 3-7 Double Precision Multiplication Using DMAC

3.3.5 Block Floating Point FFT Support

The Block Floating Point FFT operation requires the early detection of data growth between FFT butterfly passes. If data growth is detected, suitable down scaling must be applied to ensure that no overflow occurs during the next butterfly calculation pass. The total scaling applied is the block exponent of the FFT output. The Block Floating Point FFT algorithm is described in the Motorola application note, *Implementation of Fast Fourier Transforms on Motorola's Digital Signal Processors (APR4/D).*

Data growth detection is implemented as a status bit in the SR. The FFT Scaling Bit S (Bit 7) of the SR is set upon moving a result from accumulator A or B to the XDB or YDB bus (during an accumulator to memory or accumulator to register move) and remains set until explicitly cleared—that is, the S bit is a "sticky" bit.

3.4 DATA ALU PROGRAMMING MODEL

The Data ALU features 16-bit input/output data registers that can be concatenated to accommodate 32-bit data and two 40-bit accumulators, which are segmented into three 16-bit pieces that can be transferred over the buses. **Figure 3-8** illustrates how the registers in the programming model are grouped.

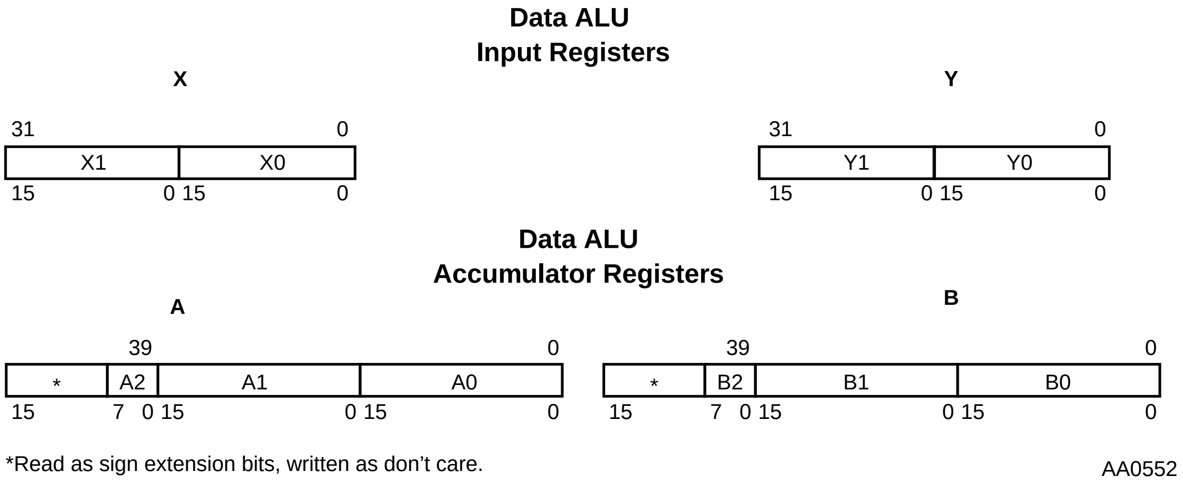


Figure 3-8 DSP56600 Core Programming Model

The Data ALU is fully pipelined and every instruction takes 2 clock cycles to complete. However, a new instruction can be started on every clock cycle and a new result is produced on every clock cycle, thus yielding an effective execution rate of one instruction per clock cycle. There are no pipeline dependencies when using the result of the Data ALU as source operand for the immediate following Data ALU instruction. Nevertheless, Data ALU operations can produce pipeline conflicts as described in the following paragraphs.

3.4.1 Pipeline Conflicts—Arithmetic Stall

Since every Data ALU instruction takes 2 clock cycles to complete, an interlock condition occurs when trying to read an accumulator (or parts of an accumulator) while the preceding instruction was a Data ALU instruction that specified that same accumulator as the destination. This interlock condition, named arithmetic stall, is detected in hardware and an idle cycle (NOP instruction) is inserted, thereby guaranteeing correctness. The user can optimize code by inserting a useful instruction before the read instruction. **Example 3-1** describes the cases in which the pipelined nature of the Data ALU generates arithmetic stall cases.

Example 3-1 Pipeline Conflicts—Arithmetic Stall

```
;the following example illustrates a one-clock pipeline delay when
;trying to read an accumulator as source for move:
mac     x0,y0,a        ;data ALU operation
move    a1,x:(r0)+     ;one clock delay is added to
                       ;allow mac to complete

;the following example illustrates a one-clock pipeline delay when
;trying to read an accumulator as source for bset:
tfr     a,b            ;data ALU operation
bset    #3,b           ;one clock delay is added to
                       ;allow tfr to complete

;the following example illustrates a way to find useful usage of
;the pipeline delay clock:
mac     x0,y0,a        ;data ALU operation
mac     x1,y1,b        ;insert a useful instruction
move    a,x:(r0)+      ;read accumulator A without
                       ;any time penalty
```

3.4.2 Pipeline Conflicts—Status Stall

A second interlock condition, named status stall, occurs when trying to read the SR while the preceding or the second preceding instruction was a Data ALU instruction or an accumulator read (which updates the S and L condition codes in the SR). The hardware inserts two or one idle cycles (NOP instruction) accordingly, thereby guaranteeing correctness. Note that “read status register” implies a MOVE status register, Bit Manipulation instructions (for example, the BSET instruction) on an SR bit, or Program Control instructions (such as the BSCLR instruction) that test for a bit in the SR. **Example 3-2** describes the cases in which the pipelined nature of the Data ALU generates stall interlock cases.

Example 3-2 Pipeline Conflicts—Status Stall

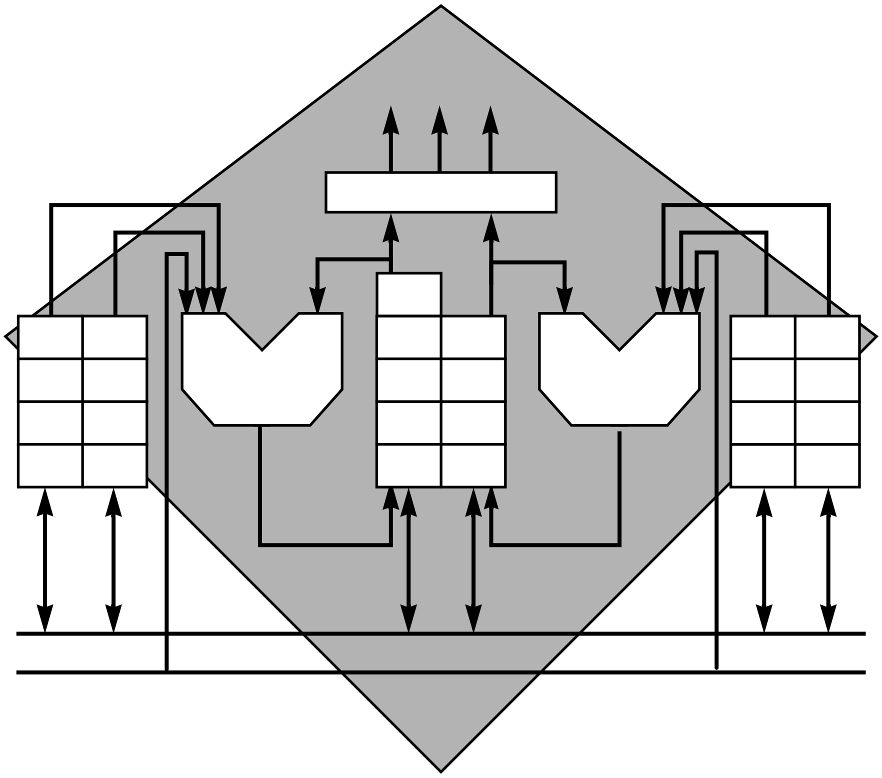
```
following example illustrates a two-clock pipeline delay when
;trying to read the status register as source for move:
mac    x0,y0,a        ;data ALU operation
move   sr,x:(r0)+     ;TWO clock delay is added to
                      ;allow mac to update SR
;following example illustrates a one-clock pipeline delay when
;trying to read the status register as source for bit
;manipulation instruction:
move   a,x:(r0)+      ;read full accumulator
nop
btst   #5,sr          ;ONE clock delay is added (and
                      ;not two) due to the previous nop
;following example illustrates a one-clock pipeline delay when
;trying to read the status register as source for program control

;instruction:
insert x0,y1,a        ;data ALU operation
bsclr  #5,sr,$f00f    ;ONE clock delay is added (and not
                      ;two) since bsclr is a two word
                      ;instruction
```

Note: A special case of interlock occurs when using a 16-bit logic instruction and writing concurrently to the EXT or the LSP of the same accumulator. The hardware inserts one idle cycle (NOP instruction), thereby the correctness is guaranteed. For example:

```
or x1,a       y1,a0
```

SECTION 4

ADDRESS GENERATION UNIT



4.1 INTRODUCTION

The Address Generation Unit (AGU) is one of the three execution units on the DSP56600 core. The AGU performs the effective address calculations using integer arithmetic necessary to address data operands in memory and contains the registers used to generate the addresses. It implements four types of arithmetic: linear, modulo, multiple wrap-around modulo, and reverse-carry. The AGU operates in parallel with other chip resources to minimize address-generation overhead.

4.2 AGU ARCHITECTURE

The AGU is divided into two halves, each with its own Address Arithmetic Logic Unit (Address ALU). Each Address ALU has four sets of register triplets, and each register triplet is composed of an address register, an offset register, and a modifier register. The two Address ALUs are identical. Each contains a 16-bit full adder (called offset adder), which can perform the following additions:

- Plus one
- Minus one
- The contents of the respective offset register N
- Minus 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 perform the following additions:

- Plus one
- Minus one
- The offset N (stored in the respective offset register)
- Minus N to the selected address register with the carry propagating in the reverse direction—that is, from the Most Significant Bit (MSB) to the Least Significant Bit (LSB)

The offset adder and the reverse-carry adder operate 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. **Figure 4-1** shows a block diagram of the AGU.

AGU Architecture

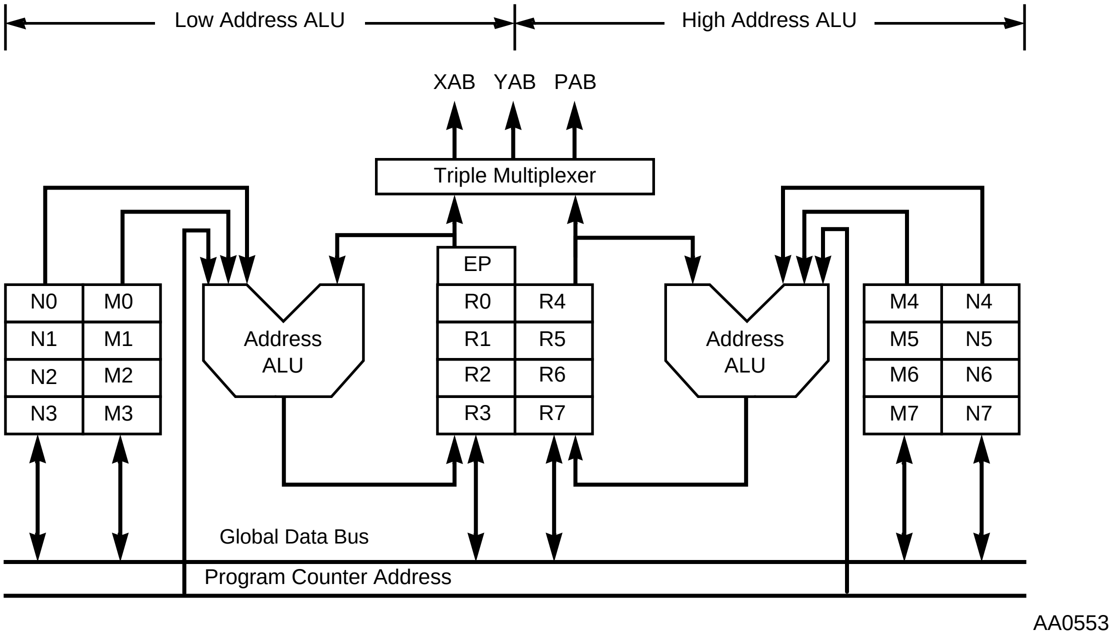


Figure 4-1 AGU Block Diagram

Each Address ALU can update one address register from its respective address register file during one instruction cycle. The contents of the associated modifier register specifies the type of arithmetic to be used in the address register update calculation. The modifier value is decoded in the Address ALU.

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

The registers are the address registers R0–R3 on the Low Address ALU and R4–R7 on the High Address ALU, the offset registers N0–N3 on the Low Address ALU and N4–N7 on the High Address ALU, and the modifier registers M0–M3 on the Low Address ALU and M4–M7 on the High Address ALU. In this section, these registers are referred to as Rn for any address register, Nn for any offset register, and Mn for any modifier register. The Rn, Nn, and Mn registers are register triplets—that is, only registers within a triplet can modify the other registers within that triplet. For example, only N2 and M2 can be used to update R2. The eight triplets are as follows:

- Low Address ALU register triplets
  - R0:N0:M0
  - R1:N1:M1
  - R2:N2:M2
  - R3:N3:M3
- High Address ALU register triplets
  - R4:N4:M4
  - R5:N5:M5
  - R6:N6:M6
  - R7:N7:M7

Each register can be read or written by the Global Data Bus (GDB).

The address output multiplexers select the source for the XAB, YAB, and PAB buses. These multiplexers allow the XAB, YAB, or PAB outputs to originate from the R0–R3 or R4–R7 registers.

4.3 PROGRAMMING MODEL

The programmer's view of the AGU is eight sets of three registers, as shown in **Figure 4-2**. These registers can be used as temporary data registers and indirect memory pointers. Automatic updating is available when using address register indirect addressing. The address registers can be programmed for linear addressing, modulo addressing (regular or multiple wrap-around), and bit-reverse addressing.

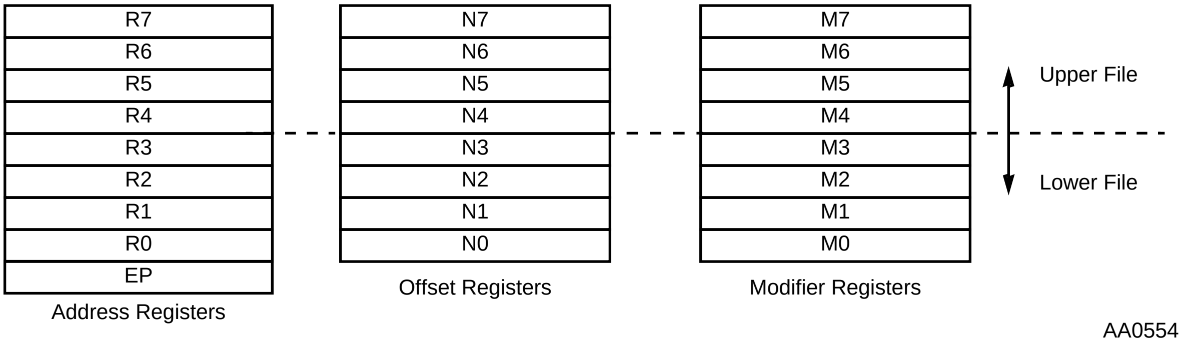


Figure 4-2 AGU Programming Model

4.3.1 Address Register Files

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 calculating the effective address of an operand. When supporting parallel X and Y data memory moves, the address registers must be programmed as two separate files, R0–R3 and R4–R7. The contents of an address register can point directly to data or can be offset.

In addition, an address register can be pre-updated or post-updated according to the addressing mode selected. If an address register is updated, a modifier register (Mn) is 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.

4.3.2 Stack Extension Pointer

The contents of the 16-bit stack Extension Pointer (EP) register are used to point to the stack extension in data memory whenever the stack extension is enabled and move operations to or from the on-chip hardware stack are needed. The EP register is a read/write register and can be referenced implicitly (e.g., by the DO, JSR, or RTI instructions) or directly (e.g., by the MOVEC instruction). The EP register is not initialized during hardware reset, and must be set (using a MOVEC instruction) prior to enabling the stack extension. For more information of the stack extension mode of operation, see **Stack Extension Pointer (EP)** on page 5-9.

4.3.3 Offset Register Files

The eight 16-bit offset registers, N0–N7, can contain offset values used to increment or decrement address registers in address register update calculations. These registers can also 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 has its own offset register associated with it.

4.3.4 Modifier Register Files

The eight 16-bit modifier registers, M0–M7, define the type of address arithmetic performed for addressing mode calculations. These registers can also 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 has its own modifier register 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.

4.4 ADDRESSING MODES

The DSP56600 core provides four different addressing modes: Register Direct, Address Register Indirect, PC Relative, and Special, as listed in **Table 4-1**.

Table 4-1 Addressing Modes Summary

Addressing Modes	Uses Mn Modi-fier	Operand Reference									Assembler Syntax
		S	C	D	A	P	X	Y	L	X Y	
Register Direct											
Data or Control Register	No		√	√							
Address Register Rn	No				√						
Address Modifier Register Mn	No				√						
Address Offset Register Nn	No				√						
Address Register Indirect											
No Update	No					√	√	√	√	√	(Rn)
Post-increment by 1	Yes					√	√	√	√	√	(Rn) +
Post-decrement by 1	Yes					√	√	√	√	√	(Rn) –
Postincrement by Offset Nn	Yes					√	√	√	√	√	(Rn) + Nn
Post-decrement by Offset Nn	Yes					√	√	√	√		(Rn) – Nn
Indexed by Offset Nn	Yes					√	√	√	√		(Rn + Nn)
Pre-decrement by 1	Yes					√	√	√	√		– (Rn)
Short/Long Displacement	Yes						√	√	√		(Rn + displ)
PC Relative											
Short/Long Displacement PC Relative	No					√					(PC + displ)
Address Register	No					√					(PC + Rn)

Table 4-1 Addressing Modes Summary (Continued)

Addressing Modes	Uses Mn Modi-fier	Operand Reference									Assembler Syntax
		S	C	D	A	P	X	Y	L	XY	
Special											
Short/Long Immediate Data	No					√					
Absolute Address	No					√	√	√	√		
Absolute Short Address	No						√	√	√		
Short Jump Address	No					√					
I/O Short Address	No						√	√			
Implicit	No	√	√			√					

Note: Use this key to the Operand Reference columns:

S = System Stack Reference
C = Program Control Unit Register Reference
D = Data ALU Register Reference
A = Address ALU Memory Reference
P = Program Memory Reference

X = X Memory reference
Y = Y Memory Reference
L = L Memory reference
XY = XY Memory Reference

4.4.1 Register Direct Modes

The Register Direct addressing modes specify that the operand is in one (or more) of the ten Data ALU registers, twenty-four address registers, or seven control registers.

4.4.1.1 Data or Control Register Direct

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

4.4.1.2 Address Register Direct

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

4.4.2 Address Register Indirect Modes

The Address Register Indirect modes specify that the address register is used to point to a memory location. The term indirect is used because the register contents are not the operand itself, but rather the operand address. These addressing modes specify that an operand is in memory and specify the effective address of that operand.

4.4.2.1 No Update (Rn)

The operand address is in the address register. The contents of the address register are unchanged by executing the instruction.

4.4.2.2 Post-Increment By 1 (Rn) +

The operand address is in the address register. After the operand address is used, it is incremented by 1 and stored in the same address register. The type of arithmetic used to calculate is determined by the Mn register. The Nn register is ignored.

4.4.2.3 Post-Decrement By 1 (Rn) –

The operand address is in the address register. After the operand address is used, it is decremented by 1 and stored in the same address register. The type of arithmetic used to calculate is determined by the Mn register. The Nn register is ignored.

4.4.2.4 Post-Increment By Offset Nn (Rn) + Nn

The operand address is in the address register. After the operand address is used, it is incremented by the contents of the Nn register and stored in the same address register. The type of arithmetic used to calculate is determined by the Mn register. The contents of the Nn register are unchanged.

4.4.2.5 Post-Decrement By Offset Nn (Rn) – Nn

The operand address is in the address register. After the operand address is used, it is decremented by the contents of the Nn register and stored in the same address register. The type of arithmetic used to calculate is determined by the Mn register. The contents of the Nn register are unchanged.

4.4.2.6 Indexed By Offset Nn (Rn + Nn)

The operand address is the sum of the contents of the address register and the contents of the address offset register, Nn. The type of arithmetic used to calculate is determined by the Mn register. The contents of the Rn and Nn registers are unchanged.

4.4.2.7 Pre-Decrement By 1 (Rn)

The operand address is the contents of the address register decremented by 1. The contents of Rn are decremented and stored in the same address register. The type of arithmetic used to calculate is determined by the Mn register. The Nn register is ignored.

4.4.2.8 Short Displacement (Rn + Short Displacement)

The operand address is the sum of the contents of the address register Rn and a short displacement occupying seven bits in the instruction word. The displacement is first sign-extended to sixteen bits and then added to Rn to obtain the operand address. The contents of the Rn register are unchanged. The type of arithmetic used to calculate is determined by the Mn register. The Nn register is ignored. This reference is classified as a memory reference.

4.4.2.9 Long Displacement (Rn + Long Displacement)

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

4.4.3 PC Relative Modes

In the PC relative addressing modes, the operand address is obtained by adding a displacement, represented in two's complement format, to the value of the Program Counter (PC). The PC points to the address of the instruction's opcode word. The Nn and Mn registers are ignored, and the arithmetic used is always linear.

4.4.3.1 Short Displacement PC Relative

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

4.4.3.2 Long Displacement PC Relative

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

4.4.3.3 Address Register PC Relative

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

4.4.4 Special Address Modes

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

4.4.4.1 Immediate Data

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

4.4.4.2 Immediate Short Data

The 8-bit or 12-bit operand is part of the instruction operation word. An 8-bit operand is used for immediate move to register, ANDI, and ORI instructions. It is zero-extended. A 12-bit operand is used for DO and REP instructions. It is also zero-extended. This reference is classified as a program reference.

4.4.4.3 Absolute Address

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

4.4.4.4 Absolute Short Address

The operand address occupies six bits in the instruction operation word and it is zero-extended. This reference is classified as a memory reference.

4.4.4.5 Short Jump Address

The operand occupies twelve bits in the instruction operation word. The address is zero-extended to sixteen bits. This reference is classified as a program reference.

4.4.4.6 I/O Short Address

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

4.4.4.7 Implicit Reference

Some instructions make implicit reference to the Program Counter (PC), System Stack (SSH, SSL), Loop Address register (LA), Loop Counter (LC), or Status Register (SR). These registers are implied by the instruction and their use is defined by the individual instruction descriptions. See **Appendix A, Instruction Set Details**, for more information.

4.5 ADDRESS MODIFIER TYPES

The DSP56600 core Address ALU supports linear, reverse-carry, modulo, and multiple wrap-around modulo arithmetic types for all address register indirect modes. These arithmetic types easily allow the creation of data structures in memory for First-In, First-Out (FIFO) queues, delay lines, circular buffers, stacks, and bit-reversed Fast Fourier Transform (FFT) buffers.

Data is manipulated by updating address registers (pointers) rather than moving large blocks of data. The contents of the address modifier register define the type of arithmetic to be performed for addressing mode calculations. For modulo arithmetic, the address modifier register also specifies the modulus. All address register indirect modes can be used with any address modifier. Each address register has its own modifier register associated with it.

The following address modifier types are available:

- Linear addressing
- Reverse-carry addressing
- Modulo addressing
- Multiple wrap-around modulo addressing

Linear addressing is useful for general-purpose addressing. Reverse-carry addressing is useful for 2^k-point FFT addressing. Modulo addressing is useful for creating circular buffers for FIFO queues, delay lines and sample buffers. Multiple wrap-around addressing is useful for decimation, interpolation, and waveform generation since the multiple wrap-around capability can be used for argument reduction. **Table 4-2** lists the address modifier types.

Table 4-2 Address Modifier Type Encoding Summary

Modifier Mn	Address Calculation Arithmetic
$0000	Reverse-Carry (Bit-Reverse)
$0001	Modulo 2
$0002	Modulo 3
:	:
$7FFE	Modulo 32767 (2^{15}-1)

Table 4-2 Address Modifier Type Encoding Summary (Continued)

Modifier Mn	Address Calculation Arithmetic
$7FFF	Modulo 32768 (2^{15})
$8001	Multiple Wrap-Around Modulo 2
$8003	Multiple Wrap-Around Modulo 4
$8007	Multiple Wrap-Around Modulo 8
:	:
$9FFF	Multiple Wrap-Around Modulo 2^{13}
$BFFF	Multiple Wrap-Around Modulo 2^{14}
$FFFF	Linear (Modulo 2^{16})
All other combinations are reserved	

4.5.1 Linear Modifier (Mn = $FFFF)

Address modification is performed using normal 16-bit linear (modulo 65,536) arithmetic. A 16-bit offset, Nn, and ±1 can be used in the address calculations. The range of values can be considered as signed (Nn from –37,268 to +37,267) or unsigned (Nn from 0 to +65,536), since there is no arithmetic difference between these two data representations.

4.5.2 Reverse-Carry Modifier (Mn = $0000)

Reverse carry is selected by setting the modifier register to 0. Address modification is performed in hardware by propagating the carry in the reverse direction—that is, from the MSB to the LSB. Reverse carry is equivalent to bit reversing the contents of Rn (redefining the MSB as the LSB, the next MSB as bit 1, and so on) 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 a 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 + 32 K (that is, Nn=2^{15}), which allows bit-reverse addressing for FFTs up to 65,536 points.

4.5.3 Modulo Modifier (Mn = Modulus – 1)

Address modification is performed modulo M, where M ranges from 2 to +32,768. Modulo M arithmetic causes the address register value to remain within an address range of size M, defined by a lower and upper address boundary.

The value m = M – 1 is stored in the modifier register. 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 + 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 is a space between sequential circular buffers of $(2^k) - M$.

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 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 + M – 1), it wraps 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 wraps around through the base address + 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 $Nn = P \times 2^k$, 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, jumps linearly to the same relative address in a new buffer, which is P blocks forward in memory. Similarly, for (Rn) – Nn, the pointer jumps 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 automatically wraps around the address pointer by the required amount. This type address modification is useful for creating circular buffers for FIFO queues, delay lines, and sample buffers up to 32,767 words long, as well as for decimation, interpolation, and waveform generation. The special case of (Rn) ± Nn modulo M with $Nn = P \times 2^k$ is useful for performing the same algorithm on multiple blocks of data in memory, for example, when performing parallel Infinite Impulse Response (IIR) filtering.

4.5.4 Multiple Wrap-Around Modulo Modifier

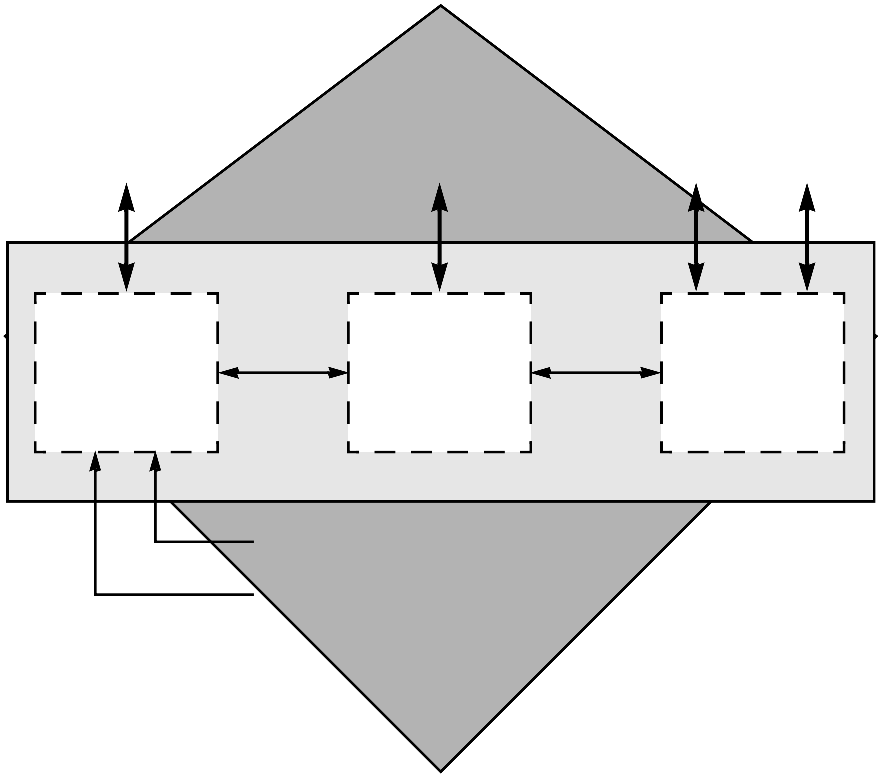
The multiple wrap-around addressing mode is selected by setting Bit 15 of the Mn register to 1, as shown in **Table 4-2** on page 4-13. The address modification is performed modulo M, where M is a power of 2 in the range from 2^1 to 2^{14}. Modulo M arithmetic causes the address register value to remain within an address range of size M defined by a lower and upper address boundary. The value M – 1 is stored in the Mn register's least significant 15 bits (bits 14–0), while bit 15 is set to 1. The lower boundary (base address) value must have 0s in the k LSBs, where $2^k = M$, and therefore must be a multiple of 2^k. The upper boundary is the lower boundary plus the modulo size minus one (base address + M – 1)

The address pointer is not required to start at the lower address boundary and may begin anywhere within the defined modulo address range (between the lower and upper boundaries). If the address register pointer increments past the upper boundary of the buffer (base address + M – 1), it wraps around to the base address. If the address decrements past the lower boundary (base address), it wraps around to the base address + M – 1. If an offset Nn is used in the address calculations, it is not required to be less than or equal to M for proper modulo addressing since multiple wrap around is supported for (Rn) + Nn, (Rn) – Nn and (Rn + Nn) address updates.

Multiple wraparound cannot occur with (Rn)+, (Rn)–, and –(Rn) addressing modes.

SECTION 5

PROGRAM CONTROL UNIT



5.1 INTRODUCTION

This section describes the Program Control Unit (PCU) hardware and its programming model. The instruction pipeline description is also included, since understanding the pipeline is particularly important in understanding the DSP56600 core. Note that the pipelined operation remains essentially hidden from the user, thus easing programmability.

5.2 PCU OVERVIEW

The PCU performs instruction prefetch, instruction decoding, hardware DO loop control and exception processing. Its programmer's model consists of the following registers:

- Program Counter (PC) register—16-bit, read/write
- Status Register (SR)—16-bit, read/write
- Loop Address (LA) register—16-bit, read/write
- Loop Counter (LC) register—16-bit, read/write
- Vector Base Address (VBA) register—16-bit, read/write
- Size (SZ) register—16-bit, read/write
- Stack Pointer (SP) register—6-bit, read/write
- Operating Mode Register (OMR)—16-bit, read/write
- Stack Counter (SC) register—5-bit, read/write

The PCU also includes a hardware System Stack (SS). In addition to the standard program flow-control resources (e.g., interrupts and jumps), the PCU supports hardware DO looping and REPEAT mechanism.

The SS is a 16-level by 32-bit separate internal memory used to automatically store the PC and SR registers during subroutine calls and long interrupts. For hardware loops, the SS stores the LC and LA registers as well as the PC and SR registers. All other data and control registers can be stored in the SS via software control. Each location in the SS is addressable as two 16-bit registers, the System Stack High (SSH) and System Stack Low (SSL) registers, which are pointed to by the four LSBs of the six-bit Stack Pointer (SP) register.

The PCU implements a seven-stage pipeline and controls the five processing states of the DSP56600 core: Normal, Exception, Reset, Wait, and Stop.

5.3 PCU ARCHITECTURE

The PCU consists of three hardware blocks:

- Program Decode Controller (PDC)
- Program Address Generator (PAG)
- Program Interrupt Controller (PIC)

The PDC decodes the 24-bit instruction loaded into the instruction latch and generates all signals necessary for pipeline control.

The PAG contains all the hardware needed for program address generation, system stack and loop control.

The PIC arbitrates among all interrupt requests (internal interrupts, as well as the five external requests IRQA, IRQB, IRQC, IRQD, and NMI), and generates the appropriate interrupt vector address.

Figure 5-1 shows a block diagram of the PCU.

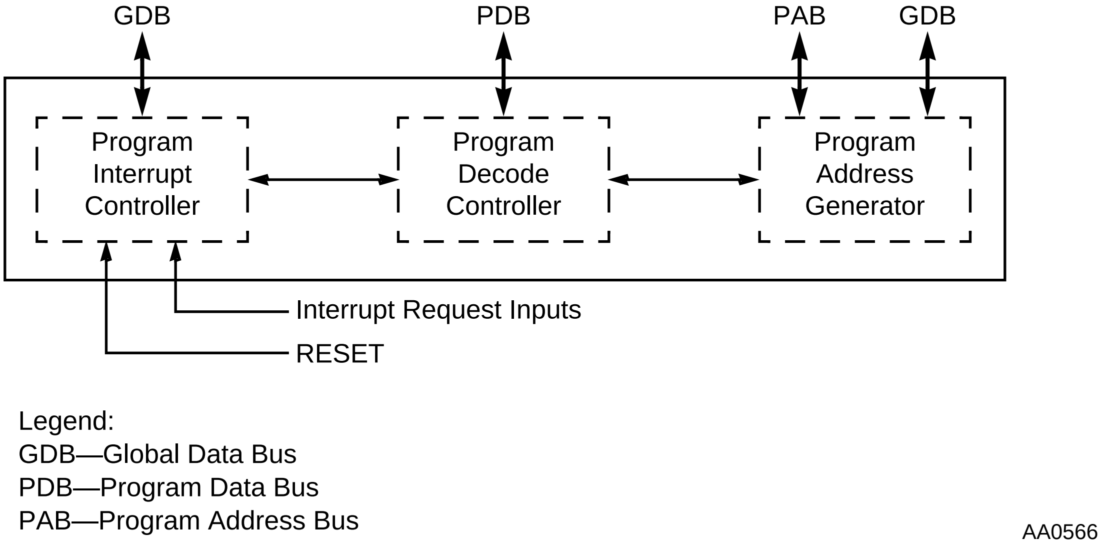


Figure 5-1 Program Control Unit Architecture

5.3.1 Instruction Pipeline

The PCU implements a seven-stage pipelined architecture in which concurrent stages of this pipeline occur. These seven stages consist of two prefetch stages, one decode stage, two address generation stages and two execute stages, as illustrated in **Figure 5-2** and described in **Table 5-1**. Although composed of many stages, the pipelined operation remains essentially hidden from the user, thus easing programmability. This is achieved by means of interlock hardware that is present in every execution unit of the processor. Because of this feature, programs written for the DSP56000 family devices will execute correctly on the DSP56600 core without any need for modification. Modification of the program may reduce the occurrence of interlocks and improve execution speed.

Table 5-1 Seven-Stage Pipeline

Pipeline Stage	Description of Pipeline Stage
PreFetch-I	• Address generation for Program Fetch • Increment PC register
PreFetch-II	• Instruction word read from memory
Decode	• Instruction Decode
AddressGen-I	• Address generation for Data Load/Store operations
AddressGen-II	• Address pointer update
Execute-I	• Read source operands to Multiplier and Adder • Read source register for memory store operations • Multiply • Write destination register for memory load operations
Execute-II	• Read source operands for Adder if written by previous ALU operation • Add • Write Adder results to the Adder destination operand • Write Multiplier results to the Multiplier destination operands

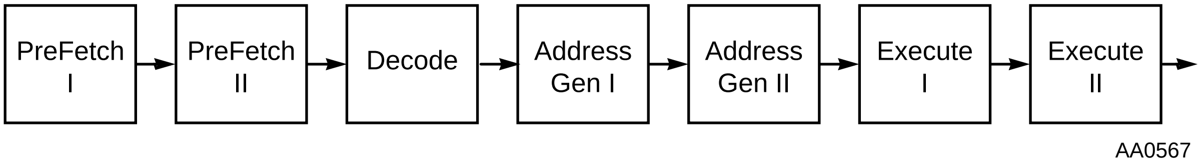


Figure 5-2 Seven-Stage Pipeline

5.3.2 Clock Oscillator

The DSP56600 core uses a two-phase clock for instruction execution. Therefore, the clock runs at the same rate as the instruction execution. The clock can be provided by connecting an external crystal between XTAL and EXTAL, or by an external oscillator connected to EXTAL. The PLL can be used in order to determine the internal frequency related to the external. For more information, see **Section 8, PLL and Clock Generator.**

5.4 PROGRAMMING MODEL

The PCU features the 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, SR, and SS. All registers are read/write to facilitate system debugging. **Figure 5-3** shows the PCU programming model with the registers and the SS. The following paragraphs describe each register.

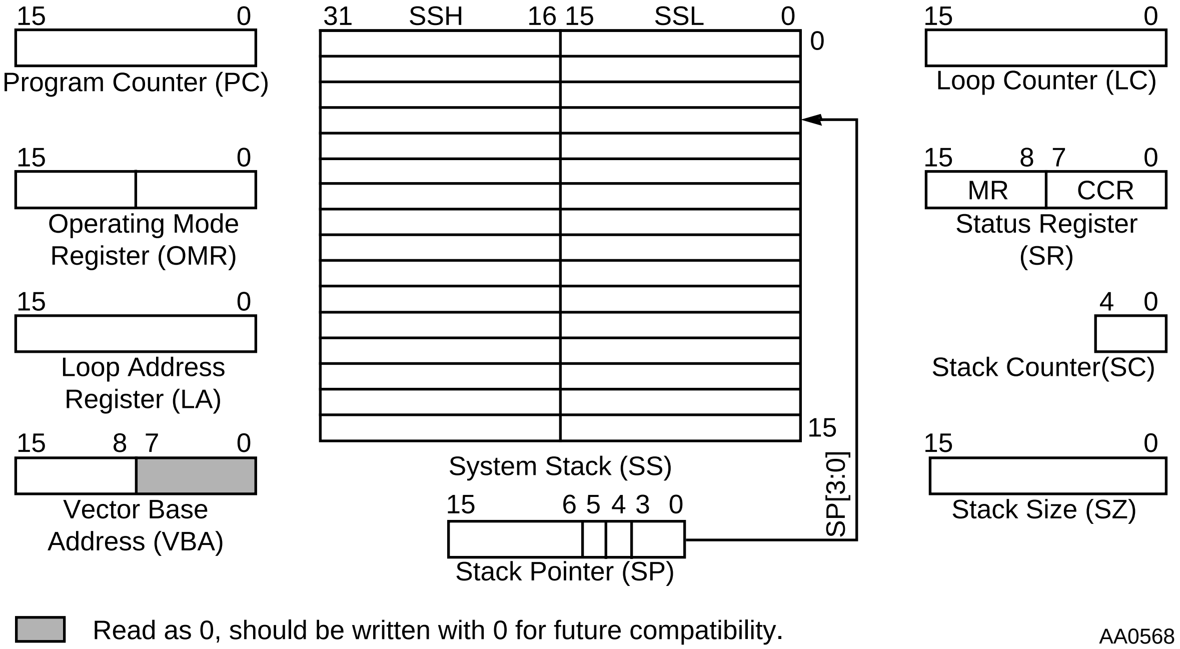


Figure 5-3 PCU Programming Model

5.4.1 Program Counter (PC)

The Program Counter (PC) register is a special-purpose 16-bit address register that contains the address of instruction words in the 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. The PC is stacked when hardware loops are initialized, when a JSR is performed, or when a long interrupt occurs.

5.4.2 Vector Base Address Register (VBA)

The Vector Base Address (VBA) register is a 16-bit register. The lower eight bits (bits 7-0) are read-only and are always read as 0. The VBA is used as a base address of the interrupt vector and interrupt vector plus one. When executing a fast or long interrupt, the vector address bits 7-0 are driven from the Program Interrupt Control unit, while bits 15-8 are driven from the VBA. The VBA register is a read/write register that is referenced implicitly by interrupt processing or directly by the MOVEC instruction. The VBA is cleared during hardware reset.

5.4.3 Loop Counter Register (LC)

The Loop Counter (LC) register is a special read/write 16-bit counter that specifies the number of times a hardware program loop is to be repeated, in the range of 0 to $(2^{16} - 1)$. This register is stacked into the SSL by a DO instruction and unstacked by end-of-loop processing or by execution of an ENDDO and BRKcc instructions. The LC is also used in the REP instruction to specify the number of times an instruction is to be repeated.

5.4.4 Loop Address Register (LA)

The Loop Address (LA) register is a 16-bit register whose contents indicate the location of the last instruction word in a hardware 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 and BRKcc instructions. The LA register, a read/write register, is written by a DO instruction and read by the SS when stacking the register.

5.4.5 System Stack (SS)

The System Stack (SS) is a separate 16 × 32-bit internal memory divided into two banks: System Stack High (SSH) and System Stack Low (SSL), 16 bits wide each. The SS is used for the following main tasks:

- Storing return address and status for subroutine calls
- Storing LA, LC, PC and SR for the hardware DO loops
- Storing calling routine variables for subroutine calls

When a subroutine is called (e.g., using the JSR instruction), the return address (PC) is automatically stored in the SSH and the chip status (SR) is automatically stored in the SSL.

When a return from subroutine is initiated by using the RTS instruction, the contents of the top location in the SSH are pulled and loaded into the PC and the SR is not affected. When a return is initiated using the RTI instruction, the contents of the top locations in the SS are pulled and loaded into the PC and SR (from SSH and SSL respectively).

The SS is also used to implement no-overhead nested hardware DO loops. When a hardware do-loop is initiated (e.g., by using the DO instruction), the previous contents of the Loop Counter (LC) register is automatically stored in the SSL, the previous contents of the Loop Address (LA) register is automatically stored in the SSH and the Stack Pointer (SP) is incremented. The address of the loop's first instruction (PC) is also stored in the SSH and the chip status register (SR) is stored in the SSL.

The SS can be extended in the data memory by means of control hardware that monitors the accesses to the SS. This extension is enabled by Stack Extension Enable (SEN) bit in the OMR. If this bit is cleared, the extension of the system stack is disabled and the amount of nesting is determined by the limited level of the hardware stack (limited to fifteen locations—one location is unusable when the stack extension is disabled). As many as fifteen long interrupts, seven DO loops, fifteen JSRs, or combinations of these can be accommodated by the SS when its extension in data memory is disabled. When the SS limit is exceeded, either in the Extended or in the Non-extended mode, a Nonmaskable stack error Interrupt (NMI) occurs.

By enabling the stack extension, the limits on the level of nesting of subroutines or DO loops can be set to any desired value. A stack extension algorithm is applied to all accesses to the stack.

If an explicit push operation (such as a move to SSH) or an implicit push operation (such as a JSR) is performed, then the stack is examined by the stack extension control logic after that push has finished. If the on-chip hardware stack is full, then the least recently used word is moved into data memory to the location specified by the stack Extension Pointer (EP).

If an explicit pop operation (such as a move from SSH) or an implicit pop operation (such as a RTS) is performed, then the stack is examined by the stack extension control logic after that pop has finished. If the on-chip hardware stack is empty, then the stack is loaded from the location (in data memory) specified by the stack Extension Pointer (EP).

5.4.6 Stack Extension Pointer (EP)

The stack Extension Pointer (EP) register is a 16-bit register whose contents point to the stack extension in data memory whenever the stack extension is enabled and move operations to or from the on-chip hardware stack are needed. The EP register is located in the Address Generation Unit (AGU). For more details, see **Stack Extension Pointer** on page 4-6

5.4.7 Stack Size Register (SZ)

The Stack Size (SZ) register is a 16-bit register that determines the number of stack levels that the software requires in the Extended mode. The Extended Stack Overflow flag is generated upon comparing the value in SP to the value in SZ. The SZ register is not initialized during hardware reset, and must be set (using a MOVEC instruction) prior to enabling the stack extension.

5.4.8 Stack Counter Register (SC)

The Stack Counter (SC) register is a 5-bit register that monitors how many entries of the hardware stack are in use. The SC register is a read/write register and is referenced implicitly by some instructions (DO, JSR, RTI, etc.) or directly by the MOVEC instruction. The SC register is cleared during hardware reset.

Note: During normal operation, the SC register should not be written. If a task switch is needed, writing a value greater than fourteen or smaller than two will automatically activate the stack extension control hardware. For proper operation, do not write values greater than sixteen to the SC.

5.4.9 Stack Pointer Register (SP)

The Stack Pointer (SP) register is a 16-bit register that indicates the location of the top of the SS. The status of the SS is also indicated in the SP register as underflow, empty, full, and overflow. The SP register is referenced implicitly by some instructions (such as DO, JSR, RTI) or directly by the MOVEC instruction. The SP register format, shown in **Figure 5-4**, is described in the following paragraphs. The SP register is implemented as a 4-bit counter that addresses (selects) a 16-locations stack. The possible SP values are shown in **Table 5-2**.

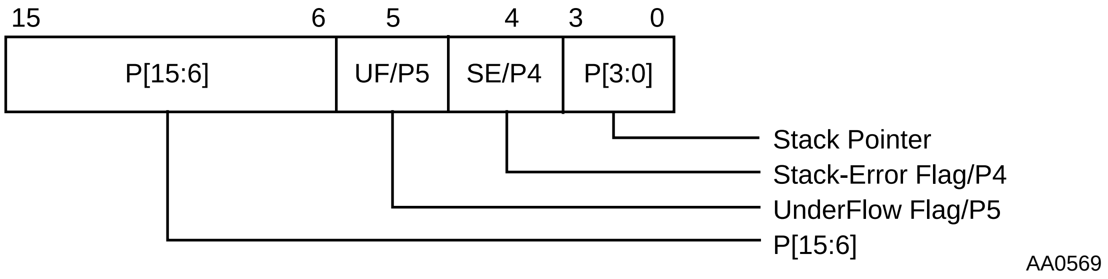


Figure 5-4 Stack Pointer (SP) Register Format

5.4.9.1 Stack Pointer (Bits 0–3)

The Stack Pointer (P) bits point 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 decrementing SP.

5.4.9.2 Stack Error Flag/P4 Bit (Bit 4)

The Stack Error Flag/P4 (SE/P4) bit is a dual function bit. In the Extended mode it acts as Bit 4 of the Stack Pointer, as part of a 16-bit up/down counter. In the Non-extended mode, it serves as the Stack Error (SE) flag that indicates that a stack error has occurred. The transition of the stack error flag from zero to one in the Non-extended mode causes a priority level-3 stack error exception.

When the non-extended 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 SP will read 010000 (or 010001 if an implied double push occurs).

Any implied pull operation with SP equal to 0 causes a stack error exception, and the SP reads $003F (or $003E if an implied double pull occurs). During such case, the Stack Error bit is set as shown in**Table 5-2**.

The stack error flag is a "sticky bit" that, once set, remains set until cleared by the user. The overflow/underflow bit remains latched until the first move to SP is executed.

Table 5-2 SP Register Values in the Non-Extended Mode

UF	SE	P3	P2	P1	P0	Description
1	1	1	1	1	0	Stack Underflow condition after double pull
1	1	1	1	1	1	Stack Underflow condition
0	0	0	0	0	0	Stack Empty (RESET); Pull causes underflow
0	0	0	0	0	1	Stack Location 1
.	.	.	.	.	.	Stack Locations 2–13
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

5.4.9.3 Underflow Flag / P5 Bit (Bit 5)

The Underflow Flag / P5 (UF/P5) bit is a dual function bit. In the Extended mode it acts as Bit 5 of the Stack Pointer, as part of a 16-bit up/down counter. In the Non-extended mode, the underflow flag is set when a stack underflow occurs. The stack underflow flag is a "sticky bit"; that is, once the stack error flag is set, the underflow flag will not change state until explicitly written by a move instruction. The combination of "underflow = 1" and "stack error = 0" is an illegal combination and does not occur unless forced by the user. See **Stack Error Flag/P4 Bit (Bit 4)** on page 5-10 for additional information.

5.4.10 Status Register (SR)

The Status Register (SR) is a 16-bit register that consists of an 8-bit Condition Code Register (CCR) and an 8-bit Mode Register (MR). 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 5-5**.

Each bit in the SR is mask-programmable, and can be programmed to one of the following configurations:

- Read/write bit with the functionality as described in the following paragraphs
- Read as zero bit

The CCR is a special purpose control register that defines the results of previous arithmetic computations. The CCR bits are affected by Data Arithmetic Logic Unit (Data ALU) operations, parallel move operations, and by instructions that directly reference the CCR (such as the ORI and ANDI instructions) or instructions that specify the SR as its destination (such as the MOVEC instruction). Parallel move operations only affect the S and L bits of the CCR. During processor reset all CCR bits are cleared.

The MR is a special purpose control register defining the current system state of the processor. The bits in the MR are affected by processor reset, exception processing, DO, DO FOREVER, ENDDO (end current DO loop), BRKcc, RTI (return from interrupt), and TRAP instructions, and by instructions that directly reference the MR, such as the ANDI and ORI instructions, or any instruction that specifies the SR as its destination, such as the MOVEC instruction. During processor reset the interrupt mask bits of the MR are set, while all the other bits are cleared.

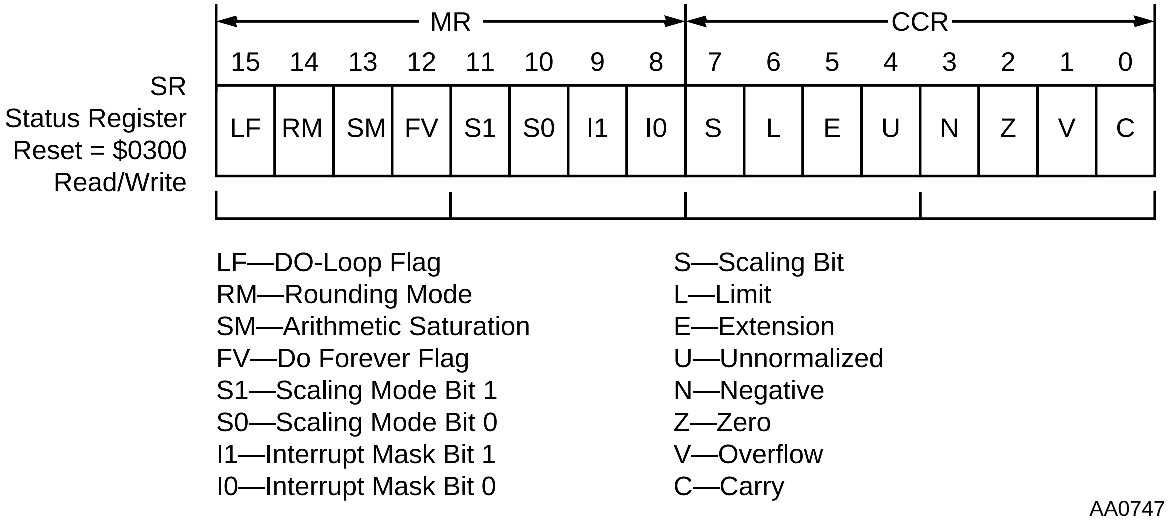


Figure 5-5 Status Register (SR) Format

5.4.10.1 Carry (C)—Bit 0

The Carry (C) bit is set when a carry is generated out of the MSB of the result in an addition operation. This bit is also set when a borrow is generated in a subtraction operation. Otherwise, this bit is cleared. The carry or borrow is generated from Bit 39 of the result. The carry bit is also affected by bit manipulation, rotate, and shift instructions.

5.4.10.2 Overflow (V)—Bit 1

The Overflow (V) bit is set when an arithmetic overflow occurs in the 40-bit result; otherwise, this bit is cleared. This bit indicates that the result cannot be represented in the accumulator register because the register has overflowed. In Arithmetic Saturation mode, an arithmetic overflow occurs if the Data ALU result is not representable in the accumulator without the extension part, that is, the 32-bit accumulator.

5.4.10.3 Zero (Z)—Bit 2

The Zero (Z) bit is set when the result equals zero. Otherwise, this bit is cleared.

5.4.10.4 Negative (N)—Bit 3

The Negative (N) bit is set when the MSB of the result is set. Otherwise, this bit is cleared.

5.4.10.5 Unnormalized (U)—Bit 4

The Unnormalized (U) bit is set when the two MSBs of the Most Significant Portion (MSP) of the result are identical. Otherwise, this bit is cleared. The MSP portion of the A or B accumulators is defined by the Scaling mode. The U bit is computed as described in **Table 5-3**.

Table 5-3 Unnormalized Bit Definition

S1	S0	Scaling Mode	U Bit Computation
0	0	No Scaling	$U = \overline{(\text{Bit 31xor Bit 30})}$
0	1	Scale Down	$U = \overline{(\text{Bit 32xor Bit 31})}$
1	0	Scale Up	$U = \overline{(\text{Bit 30xor Bit 29})}$

5.4.10.6 Extension (E)—Bit 5

The Extension (E) bit is cleared when the bits of the integer portion of the 40-bit result are all 1s or all 0s. Otherwise, this bit is set. The integer portion is defined by the Scaling mode, as described in **Table 5-4**. If the E bit is cleared, then the low-order fraction portion contains all the significant bits and 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.

Table 5-4 Extension Bit Definition

S1	S0	Scaling Mode	Integer Portion
0	0	No Scaling	Bits 39,38.............32,31
0	1	Scale Down	Bits 39,38.............33,32
1	0	Scale Up	Bits 39,38.............31,30

5.4.10.7 Limit (L)—Bit 6

The Limit (L) bit is set when the overflow bit is set or if the data shifter/limiter circuits perform a limiting operation. In Arithmetic Saturation mode, the limit bit is also set when an arithmetic saturation occurs in the Data ALU result; 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 (a “sticky” bit). The L bit is affected by data movement operations that read the A or B accumulator registers.

5.4.10.8 Scaling (S)—Bit 7

The Scaling bit (S) is set upon moving a result from accumulator A or B to the XDB or YDB buses (during an accumulator to memory or accumulator to register move) and will remain set until explicitly cleared; that is, the S bit is a “sticky” bit. The logical equations of this bit are dependent on the Scaling mode. The scaling bit is set when the absolute value in the accumulator before scaling, was greater or equal to 0.25 or less than 0.75. This bit is cleared during hardware reset. **Table 5-5** shows how these bits are defined.

Table 5-5 Scaling Bits Definition

S0	S1	Scaling Mode	S Equation
0	0	No scaling	S = (A30 XOR A29) OR (B30 XOR B29) OR S (previous)
0	1	Scale down	S = (A31 XOR A30) OR (B31 XOR B30) OR S (previous)
1	0	Scale up	S = (A29 XOR A28) OR (B29 XOR B28) OR S (previous)
1	1	Reserved	S = Undefined

5.4.10.9 Interrupt Mask (I0–I1)—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. **Table 5-6** shows how these bits are defined.

Table 5-6 Interrupt Mask Bits Definition

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

5.4.10.10 Scaling Mode (S0–S1)—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 MAC unit. 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. **Table 5-7** shows how these bits are defined.

Table 5-7 Scaling Mode Bits Definition

S1	S0	Rounding Bit	Scaling Mode
0	0	15	No Scaling
0	1	16	Scale down (1-bit Arithmetic Right Shift)
1	0	14	Scale Up (1-bit Arithmetic Left Shift)
1	1	—	Reserved

5.4.10.11 DO-Forever flag (FV)—Bit 12

The DO-Forever flag (FV) bit is set when a DO FOREVER instruction is performed. The FV flag, like LF flag, is restored from stack when terminating a DO FOREVER program loop. Stacking and restoring the FV flag when initiating and exiting a DO FOREVER program loop, respectively, allow the nesting of program loops. At the start of a long interrupt service routine, the SR (including the FV bit) is pushed on the SS and the FV is cleared. When returning from the long interrupt with an RTI instruction, the SS is pulled and the FV bit is restored. The FV is cleared during a processor reset.

5.4.10.12 Arithmetic Saturation Mode (SM)—Bit 13

The Arithmetic Saturation Mode (SM) bit, when set, selects automatic saturation on 32 bits for the results going to the accumulator. This saturation is done by a special circuit inside the MAC unit. The purpose of this bit is to provide an arithmetic saturation mode for algorithms that do not recognize or cannot take advantage of the extension accumulator. This bit is cleared during processor reset.

5.4.10.13 Rounding Mode (RM)—Bit 14

The Rounding Mode (RM) bit selects the type of rounding performed by the Data ALU during arithmetic operations. When the bit is cleared, convergent rounding is selected. When the bit is set, two's-complement rounding is selected. At the start of a long interrupt service routine, the SR (including the RM bit) is pushed on the system stack and the RM bit is cleared. This bit is cleared during processor reset.

5.4.10.14 DO-Loop Flag (LF)—Bit 15

The DO-Loop Flag (LF) bit, set when a program loop is in progress, enables the detection of the end of a program loop. The LF bit is restored from stack when terminating a program loop. Stacking and restoring the LF bit 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 system stack and the LF is cleared. When returning from the long interrupt with an RTI instruction, the system stack is pulled and the LF bit is restored. This bit is cleared during a processor reset.

5.4.11 Operating Mode Register (OMR)

The Operating Mode Register (OMR) is a 16-bit register, partitioned into two bytes. The least significant byte of OMR (bits 7–0) is the Chip Operating Mode byte (COM), which is used to determine the operating mode of the chip. This byte is only affected by processor reset and by instructions directly referencing the OMR: ANDI, ORI, or other instructions that specify OMR as a destination, such as the MOVEC instruction. During processor reset, the chip operating mode bits (MD, MC, MB, and MA) are loaded from the external mode select pins MODD, MODC, MODB, and MODA, respectively.

Each bit in the OMR is mask-programmable. They can be programmed to one of the following configurations:

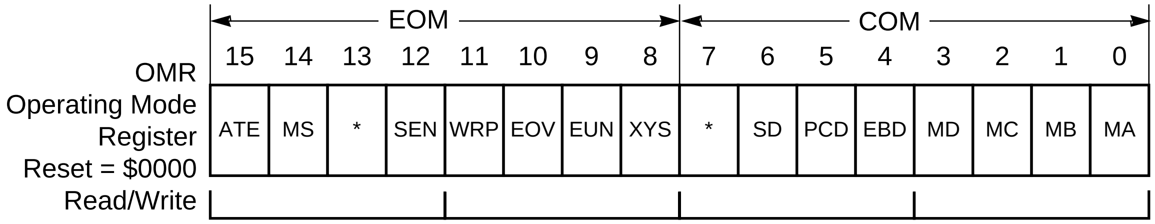
- Read / write bit with the functionality as described in the following paragraphs
- Read as zero bit

Some of the reserved bits, as described later, are also outputs of the DSP56600 core, with derivative-dependent functionality. These outputs are also mask-programmed to one of the following states:

- Connected to the OMR bit, reflecting its state
- Connected to GND (forced to 0)

The most significant byte of OMR (bits 15–8) is the Extended Chip Operating Mode byte (EOM), which is used to determine the operating mode of the chip. This byte is only affected by processor reset, by instructions that directly reference the OMR (such as ANDI and ORI instructions), or by other instructions that specify the OMR as a destination (such as the MOVEC instruction).

Figure 5-6 shows the format of the OMR. The bits in the OMR follow this format on all implementations of DSP56600-family chips. However, not all bits may be used on all DSP56600-family chips. For a list of the bits within the OMR on a specific chip, see the appropriate *User's Manual*, which also provides a detailed description of bit functionality.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ATE	MS	*	SEN	WRP	EOV	EUN	XYS	*	SD	PCD	EBD	MD	MC	MB	MA

EOM

ATE—Address Trace Enable
MS—Memory Switch Mode Enable
SEN—Stack Extension Enable
WRP—Extended Stack Wrap Flag
EOV—Extended Stack Overflow Flag
EUN—Extended Stack Underflow Flag
XYS—Stack Extension Space Select

COM

SD—Stop Delay
PCD—PC Relative Logic Disable
EBD—External Bus Disable
MA—Operating Mode A
MB—Operating Mode B
MC—Operating Mode C
MD—Operating Mode D

* Indicates reserved bits, written as 0 for future compatibility

AA0748

Figure 5-6 Operating Mode Register (OMR) Format

Programming Model

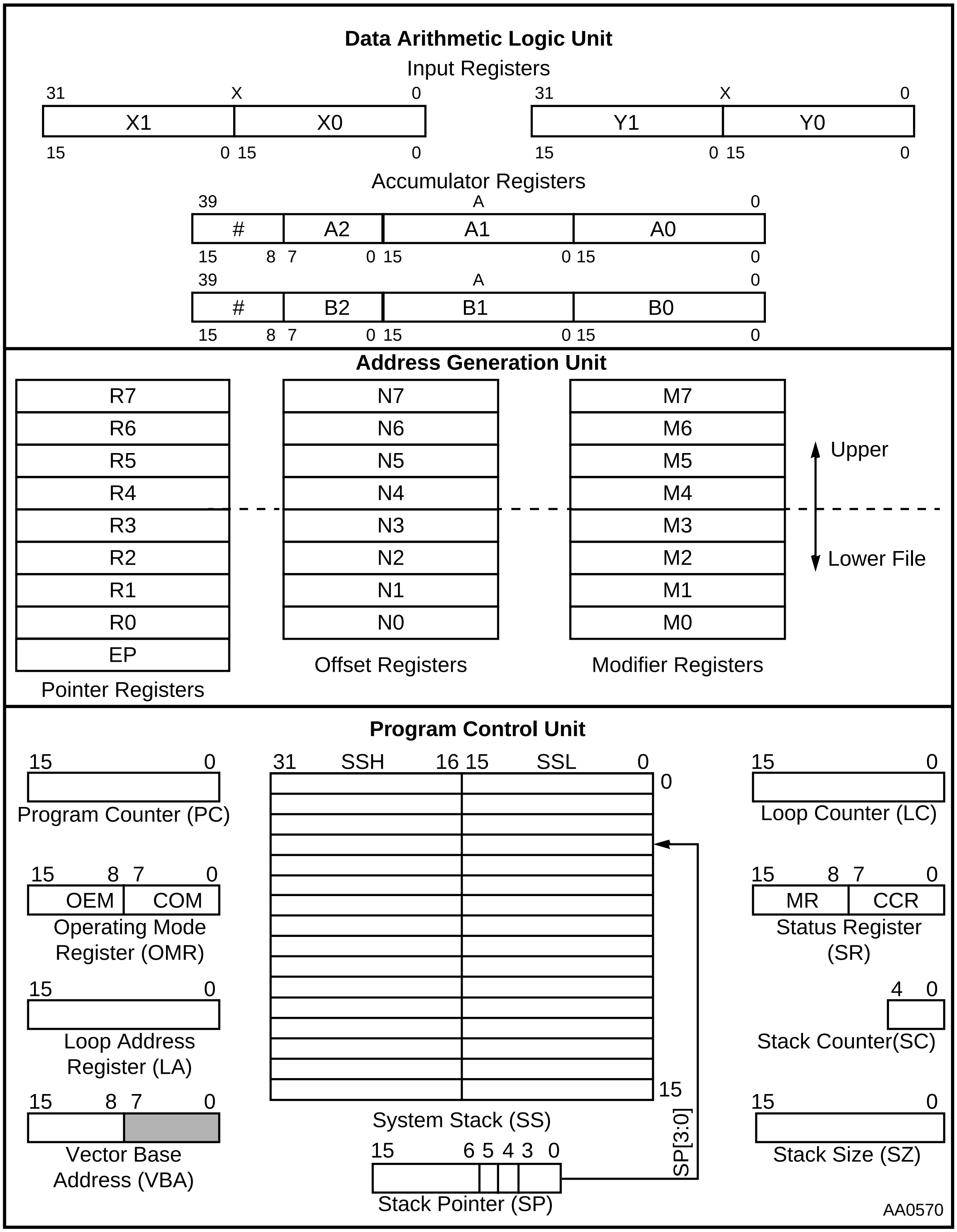
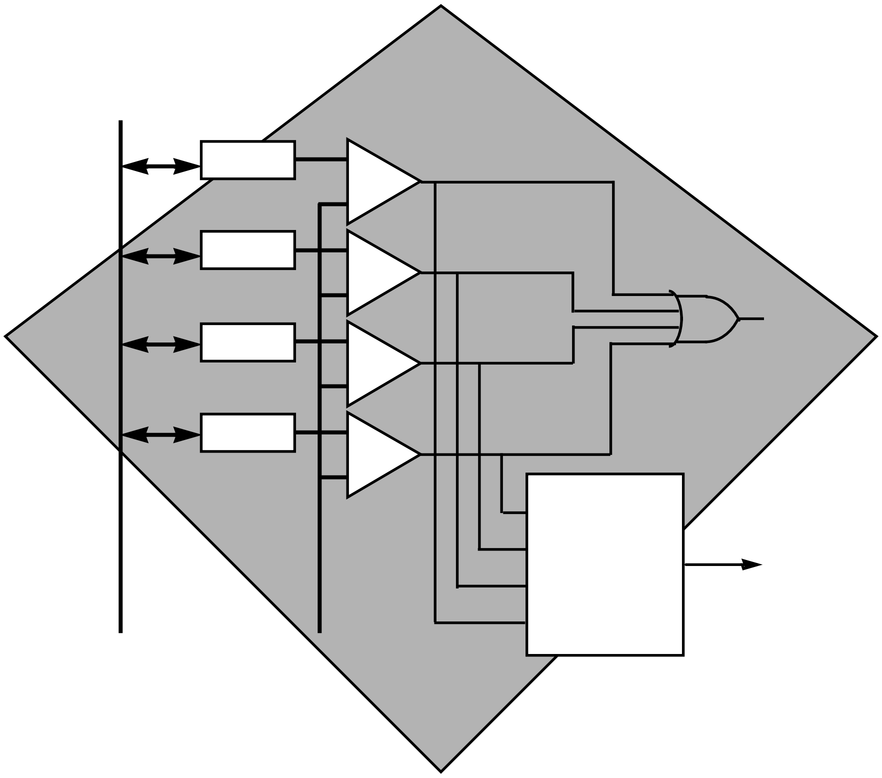


Figure 5-7 Central Processor Programming Model

SECTION 6

PROGRAM PATCH LOGIC



6.1 INTRODUCTION

This section describes the Program Patch Logic (PPL) hardware and its programming model. The PPL provides the DSP56600 core user a way to fix the program code in the on-chip ROM without generating a new mask.

6.2 PROGRAM PATCH LOGIC ARCHITECTURE

Implementing the code correction is done by replacing a piece of ROM-based code with a patch program stored in RAM. The PPL consists of four Patch Address Registers (PAR1–PAR4) and four patch address comparators. Each PAR points to a starting location in the ROM code where the program flow is to be changed. The Program Counter (PC) register in the Program Control Unit (PCU) is compared to each PAR. When an address of a fetched instruction is identical to an address stored in one of the PARs, the Program Data Bus (PDB) is forced to the corresponding JMP instruction. **Figure 6-1** shows a block diagram of the patch detector.

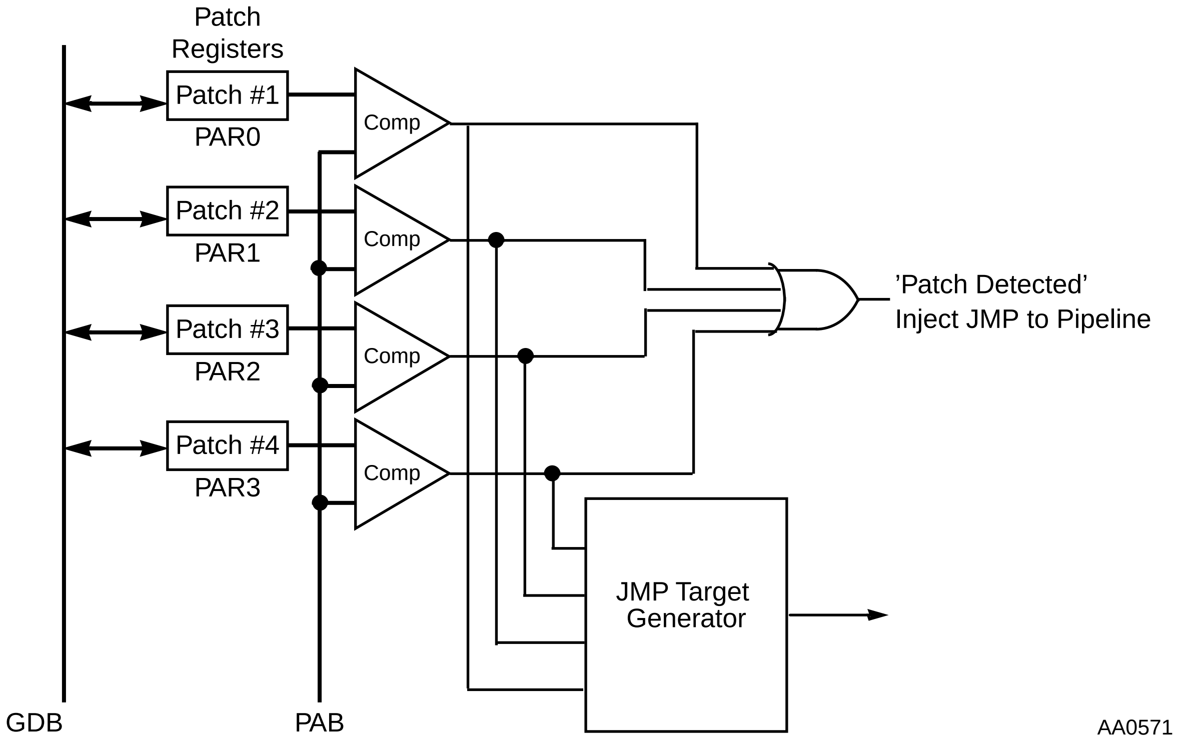


Figure 6-1 Patch Detector Block Diagram

6.3 PROGRAMMING MODEL

The programming model of the PPL is the four read/write PARs, each of which can be programmed to hold a program starting location in the on-chip ROM that should be replaced with a piece of corrective code. Only internal program space addresses from P:$0000 to P:$0100 are allowed. **Figure 6-2** shows this programming model.

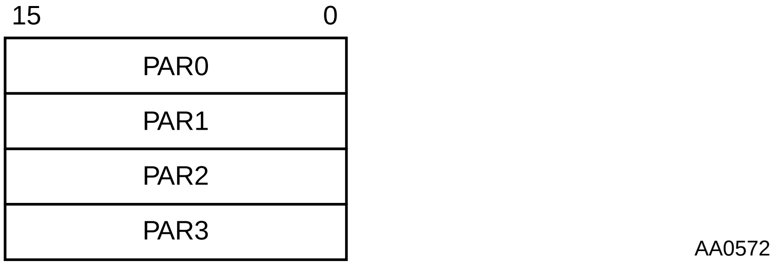


Figure 6-2 Program Patch Logic Register File

The PAR contents are compared with the Program Address Bus (PAB) used to initiate the program fetch. When the address in the PAB is equal to the contents of one of the PARs, a PATCH DETECTED signal is generated that injects a JMP instruction into the pipeline, replacing the instruction that otherwise would have been fetched from the ROM. The JMP target address is determined according to the identity of the comparator that generated the PATCH DETECTED signal. The JMP target can be any one of the first 4096 locations in the program memory space. The specific target address is mask-programmable. The user should download the correct piece of code to the target location. Comparison of each PAR to the PAB register is done only if the PAB register has been written since reset. This avoids false patch detections.

6.4 PPL OPERATION

For correct PPL operation, use the following procedure:

1. Download the correct code into the internal Program RAM. The start address of this code should correspond to one of the four pre-defined JMP target addresses. End each segment of corrected code with a JMP instruction back to the main program.
2. Initialize the PARs with the starting address of the code that is to be replaced.

3. When the PPL detects that the address on the PAB corresponds to the contents of one of the four PARs, a JMP to the Program RAM is executed.

Figure 6-3 shows the process of switching an instruction located in the ROM by another instruction that is located in the RAM. The JMP target address shown in the figure is only an example. The real address is mask-programmable.

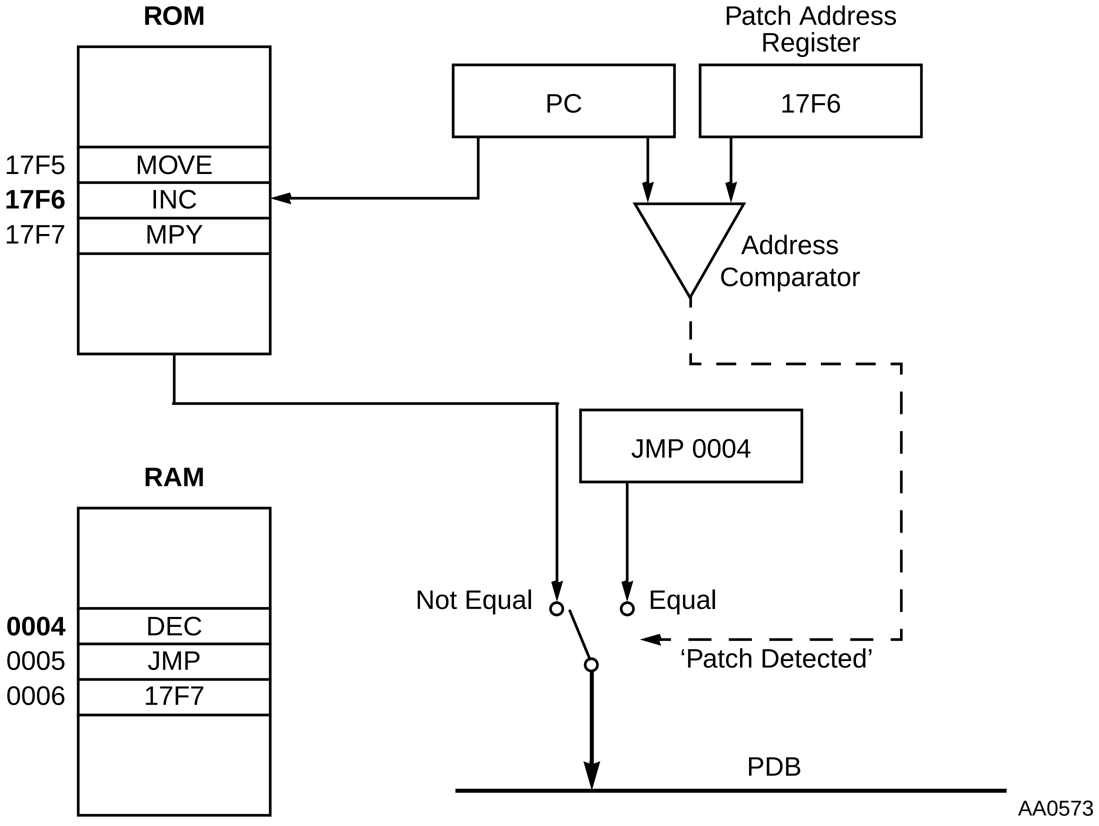
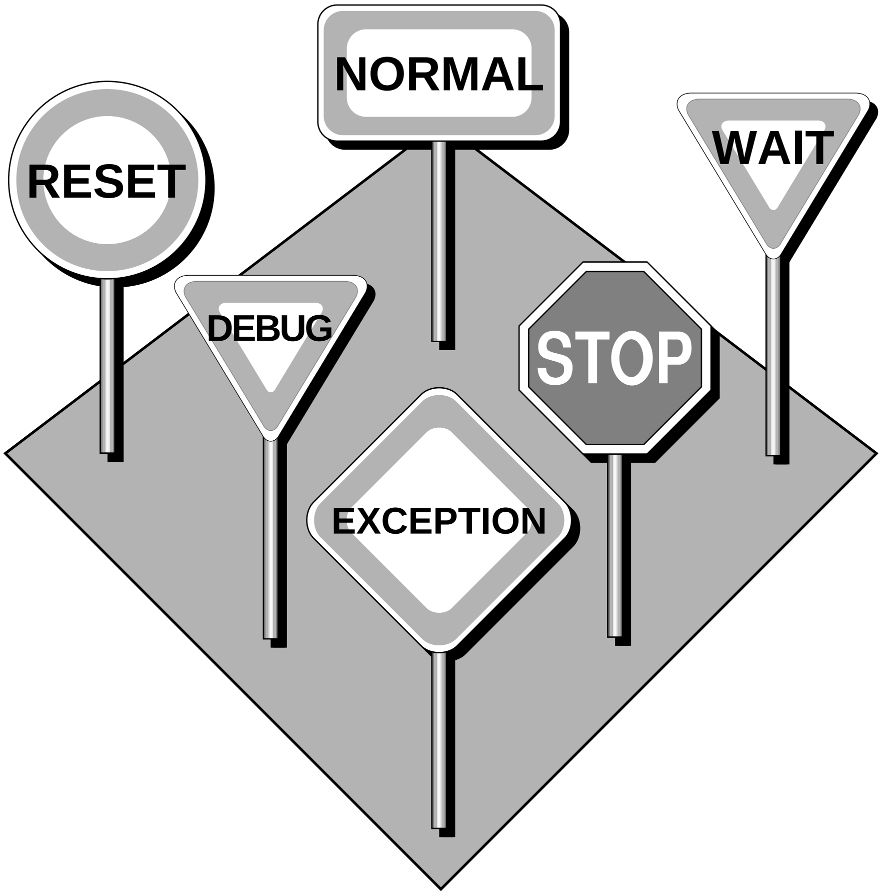


Figure 6-3 Patch Code Implementation

SECTION 7

PROCESSING STATES



7.1 INTRODUCTION

This section describes the processing states in the DSP56600 core. The DSP56600 core is always in one of five processing states:

- Normal
- Exception
- Reset
- Wait
- Stop

These states are described in the following paragraphs.

7.2 NORMAL PROCESSING STATE

The Normal processing state is associated with instruction execution. Instruction execution in the DSP56600 core is performed using a seven-stage pipeline, allowing most instructions to execute at a rate of one instruction every clock cycle. However, certain instructions require additional time to execute. These include:

- Double-word instructions
- Instructions using an addressing mode that requires more than one cycle for the address calculation
- Instructions causing a change of flow

Instruction pipelining allows overlapping of instruction execution so that a pipeline stage of a given instruction occurs concurrently with other pipeline stages of other instructions. Only one word is fetched per cycle, so that in the case of double-word instructions, the second word of an instruction is fetched before the next instruction is fetched.

The pipeline consists of seven stages: Fetch 1, Fetch 2, Decode, Address Generation 1, Address Generation 2, Execute1, and Execute 2. The abbreviations n1 and n2 refer to the first and second instructions, respectively. The third instruction (n3), which contains an instruction extension word (n3e), takes 2 clock cycles to execute. The extension word is either an absolute address or immediate data. Although it takes 7 clock cycles for the pipeline to fill and the first instruction to execute, further instructions are usually completed on each clock cycle. **Table 7-1** describes the DSP56600 core pipeline.

Table 7-1 Instruction Pipeline

Operation	Instruction Cycle										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
PreFetch 1	n1	n2	n3	n3e	n4	n5	n6	n7	n8	n9	n10
PreFetch 2		n1	n2	n3	n3e	n4	n5	n6	n7	n8	n9
Decode			n1	n2	n3	n3e	n4	n5	n6	n7	n8
Address Gen 1				n1	n2	n3	n3e	n4	n5	n6	n7
Address Gen 2					n1	n2	n3	n3e	n4	n5	n6
Execute 1						n1	n2	n3	n3e	n4	n5
Execute 2							n1	n2	n3	n3e	n4

Each instruction requires a minimum of 7 clock cycles to be fetched, decoded, and executed. This means that there is a delay of 7 clock cycles on power-up to fill the pipeline. A new instruction can begin immediately following the previous instruction. Two-word instructions require a minimum of 8 clock cycles to execute (7 cycles for the first instruction word to move through the pipeline and execute, and one more cycle for the second word to execute). For a complete description of the execution timing of the various instructions, addressing modes, etc., see **Appendix B, Instruction Timing**.

7.3 EXCEPTION PROCESSING STATE

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 to the DSP56600 core; some of these sources can generate more than one interrupt. An interrupt vector scheme with 128 vectors of predefined priorities is used to provide fast interrupt service. The following list outlines how interrupts are processed by the DSP56600 core:

1. A hardware interrupt is synchronized with the DSP56600 core 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.
2. All pending interrupts (external and internal) are arbitrated to select which interrupt is processed. The arbiter automatically ignores any interrupts with an Interrupt Priority Level (IPL) lower than the interrupt mask level in the Status Register (SR) and selects the remaining interrupt with the highest IPL.
3. The interrupt controller then freezes the Program Counter (PC) register and fetches two instructions at the two interrupt vector addresses associated with the selected interrupt.
4. The interrupt controller inserts the two instructions into the instruction stream and releases the PC register, which is used for the next instruction fetch. The next interrupt arbitration then begins.

If neither of the two instructions is a JSR instruction (e.g., JSCLR), the state of the machine is not saved on the stack, and a fast interrupt is executed. A long interrupt is executed if one of the interrupt instructions fetched is a JSR instruction. The PC register is immediately released, the SR and PC registers are saved in the stack, and the JSR instruction controls where the next instruction is fetched from.

Note: Any of the Jump To Subroutine instructions can be used as the JSR needed to make a long interrupt, such as JScc, JSSET, and so forth.

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.

7.3.1 Interrupt Sources

Exceptions can originate from any of 128 vector addresses, and from one of two sources: core and peripherals. **Table 7-2** lists the core-originating sources. The corresponding interrupt starting address for each interrupt source is shown. These addresses are located in the 256 locations of program memory pointed to by the Vector Base Address (VBA) register in the Program Control Unit (PCU).

The 128 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.

Table 7-2 Interrupt Sources

Interrupt Starting Address	IPL	Interrupt Source
VBA:$00	—	Reserved for Future Interrupt Source
VBA:$02	3	Stack Error
VBA:$04	3	Illegal Instruction
VBA:$06	3	Debug Request Interrupt
VBA:$08	3	Trap
VBA:$0A	3	Non-Maskable Interrupt ($\overline{\text{NMI}}$)
VBA:$0C	3	Reserved
VBA:$0E	3	Reserved
VBA:$10	0–2	IRQA
VBA:$12	0–2	IRQB
VBA:$14	0–2	IRQC
VBA:$16	0–2	IRQD
VBA:$18	0–2	Reserved
VBA:$1A	0–2	Reserved
VBA:$1C	0–2	Reserved
VBA:$1E	0–2	Reserved
VBA:$20	0–2	Reserved
VBA:$22	0–2	Reserved
VBA:$24	0–2	Peripheral Interrupt Request 1
VBA:$26	0–2	Peripheral Interrupt Request 2
. . .	. . .	. . .
VBA:$FE	0–2	Peripheral Interrupt Request 110

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 DSP56600 core is said to be vectored. A vectored interrupt structure has low overhead execution.

Peripheral-originating sources are described in the appropriate *User's Manual*, and are listed in **Table 7-2** as Peripheral Interrupt Requests. If it is known that certain interrupts will not be used, those interrupt vector locations can be used for program or data storage, but this is not recommended.

7.3.1.1 Hardware Interrupt Sources

There are two types of hardware interrupts to the DSP56600 core: internal and external. The internal interrupts include these on-chip sources:

- Stack Error
- Illegal Instruction
- Debug Request
- Trap
- Peripheral Interrupt

Each internal interrupt source is serviced if it is not masked. When serviced, the interrupt request is cleared. Each maskable internal hardware source has independent enable control.

The external hardware interrupts include $\overline{\text{NMI}}$, $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, $\overline{\text{IRQC}}$, and $\overline{\text{IRQD}}$. The $\overline{\text{NMI}}$ interrupt is an edge-triggered non-maskable interrupt that can be used for software development, watch-dog, power fail detect, and so forth. The $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, $\overline{\text{IRQC}}$, and $\overline{\text{IRQD}}$ interrupts can be programmed to be level-sensitive or edge-triggered. Since the level-sensitive interrupts are not cleared automatically when they are serviced, they must be cleared by other means to prevent multiple interrupts, usually by external hardware that detects the acknowledge of the core to the interrupt request. The edge-sensitive interrupts are latched as pending on the high-to-low transition of the interrupt input and are automatically cleared when the interrupt is serviced. $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, $\overline{\text{IRQC}}$, and $\overline{\text{IRQD}}$ can be programmed to one of three priority levels: 0, 1, or 2, all of which are maskable. Additionally, these interrupts have independent enable control.

When the $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, $\overline{\text{IRQC}}$, and $\overline{\text{IRQD}}$ interrupts are disabled in the Interrupt Priority Register (IPR), a pending request is 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 remains in the Reset state as long as the interrupt is disabled. If the interrupt is defined as level-sensitive, its edge-detection latch remains in the Reset state. If the level-sensitive interrupt is disabled while the interrupt is pending, the pending interrupt is cancelled. However, if the interrupt has been fetched, it normally cannot be cancelled.

Note: On all external, level-sensitive interrupt sources, the interrupt should be serviced (i.e., clear the source for the interrupt) either by the instruction at the vector location (if it is a fast interrupt) or by a long interrupt.

7.3.1.2 Software Interrupt Sources

There are two software interrupt sources, Illegal Instruction Interrupt (III) and TRAP.

The III is a nonmaskable interrupt (IPL 3) that is serviced immediately following the execution of the illegal instruction (any undefined operation code) or the attempted execution of an illegal instruction.

TRAP is a nonmaskable interrupt (IPL 3) that is serviced immediately following the TRAP or TRAPcc (condition true) instruction execution.

7.3.2 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 nonmaskable IPL 3 interrupts are:

- Stack Error
- Illegal Instruction
- Debug
- Request
- TRAP
- NMI pin
- Peripheral NMI

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 7-3**). 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 7-3 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

There are two Interrupt Priority Registers in the DSP56600 core, IPR-C and IPR-P. The IPR-C register is dedicated for DSP56600 core interrupt sources. The IPR-P register is dedicated for the specific chip peripherals interrupt sources. These control registers are mapped on the internal X I/O memory space at X:$FFFF for IPR-C and X:$FFFE for IPR-P. **Figure 7-1** shows the IPR-C register, and **Figure 7-2** shows the IPR-P register.

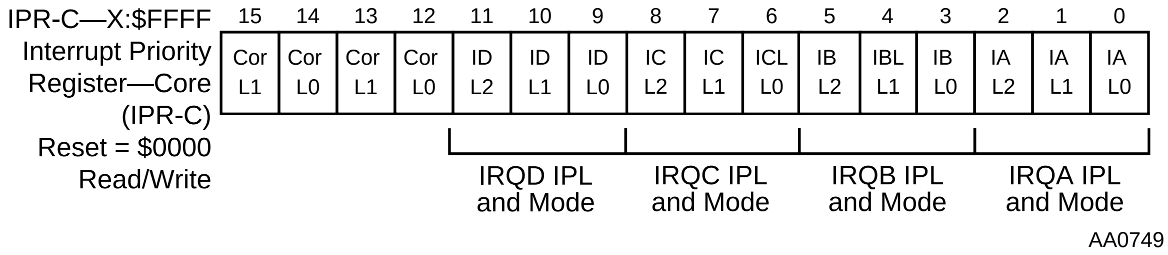


Figure 7-1 Interrupt Priority Register C (IPR-C) Format

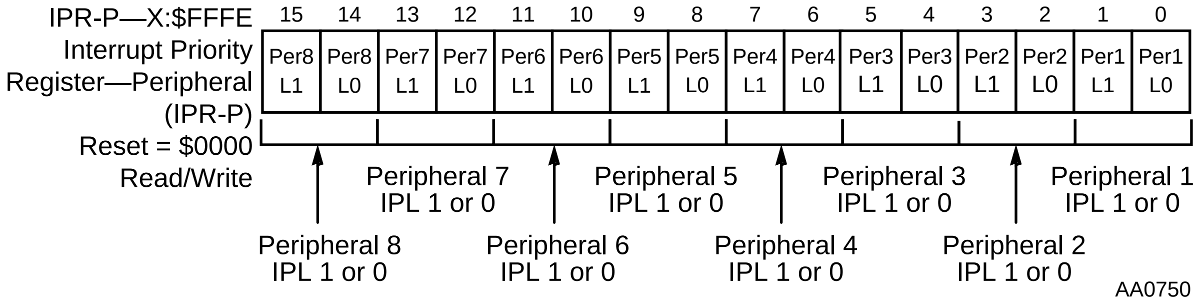


Figure 7-2 Interrupt Priority Register P (IPR-P) Format

The IPL for each interrupting source is software programmable. 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 registers shown. These two read/write registers specify the IPL for each of the interrupting devices. In addition, the IPR-C register specifies the trigger mode of each external interrupt source and is used to enable or disable the individual external interrupts. These registers are cleared on hardware $\overline{\text{RESET}}$ or by the RESET instruction.

Table 7-4 defines the IPL bits used in both the IPR-C and IPR-P registers. **Table 7-5** defines the external interrupt trigger mode bits used for $\overline{\text{IRQA}}$–$\overline{\text{IRQD}}$ in the IPR-C register.

Table 7-4 Interrupt Priority Level Bits

xxL1	xxL0	Enabled	IPL
0	0	No	—
0	1	Yes	0
1	0	Yes	1
1	1	Yes	2

Table 7-5 External Interrupt Trigger Mode Bits

IxL2	Trigger Mode
0	Level
1	Negative Edge

In addition to the maskable peripheral interrupts, a peripheral-driven Non-Maskable Interrupt (NMI) source is available on some members of the DSP56600 family. On each of the DSP56600 family chips, the peripheral assigned to the bits within the IPR-P register can vary. Consult the appropriate *User's Manual* for more information.

7.3.3 Exception Priorities within an IPL

If more than one exception is pending when an instruction is executed, the interrupt with the highest priority level is serviced first. 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 7-6**.

Table 7-6 Exception Priorities Within an IPL

Priority	**Exception**
Level 3 (Nonmaskable)	
Highest	Hardware $\overline{\text{RESET}}$
	Stack Error
	Illegal Instruction
	Debug Request Interrupt
	Trap
	Non-Maskable Interrupt ($\overline{\text{NMI}}$)
Lowest	Non-Maskable Peripheral Interrupt
Levels 0, 1, 2 (Maskable)	
Highest	$\overline{\text{IRQA}}$ (External Interrupt)
	$\overline{\text{IRQB}}$ (External Interrupt)
	$\overline{\text{IRQC}}$ (External Interrupt)
	$\overline{\text{IRQD}}$ (External Interrupt)
Lowest	Peripheral interrupt sources

7.3.4 Instructions Preceding the Interrupt Instruction Fetch

Every instruction that takes more than one cycle to execute is aborted when it is fetched in the cycle preceding the fetch of the first interrupt instruction word. 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-one-cycle instruction, that instruction will complete normally before the start of the interrupt routine. 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.

7.3.5 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 unrestricted one-word instructions. Fast interrupt routines are never interruptible.

Note: 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 plus one.

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 register (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 of the same or lower levels (except hardware $\overline{\text{RESET}}$, Illegal Instruction, stack error and TRAP that can always interrupt).

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

7.3.6 Interrupt Arbitration

External interrupts are internally synchronized with the processor clock 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—that is, 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 7-6**, 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. 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 is not accepted for the next interrupt arbitration until that time.

7.3.7 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 register, and the interrupt instruction fetch address + 1 is used for the subsequent instruction fetch. While the interrupt instructions are being fetched, the PC register is inhibited from being updated. After the two interrupt words have been fetched, the PC register is used for subsequent instruction fetches.

7.3.8 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 registers from the System Stack (SS) and return to normal program execution. Reset is a special exception, which normally contains 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. A fast interrupt routine can contain either two single-word instructions or one double-word instruction. **Table 7-7**shows the effect of a fast interrupt routine on the instruction pipeline. The fast interrupt executes only two instructions (ii1 and ii2) and then automatically resumes execution of the main program.

Table 7-7 Fast Interrupt Pipeline

Operation	Instruction Cycle											
	1	2	3	4	5	6	7	8	9	10	11	12
PreFetch 1	n1	n2	ii1	ii2	n3	n4						
PreFetch 2		n1	n2	ii1	ii2	n3	n4					
Decode			n1	n2	ii1	ii2	n3	n4				
Address Gen 1				n1	n2	ii1	ii2	n3	n4			

Table 7-7 Fast Interrupt Pipeline

Operation	Instruction Cycle											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Address Gen 2					n1	n2	ii1	ii2	n3	n4		
Execute 1						n1	n2	ii1	ii2	n3	n4	
Execute 2							n1	n2	ii1	ii2	n3	n4
n = normal instruction word ii = interrupt instruction word												

Table 7-8 shows the effect of a long interrupt routine on the instruction pipeline. A short JSR (ii1) is used to call the long interrupt routine, which includes the sr1, sr2, sr3 and rti instructions. Instructions ii2, n3, sr5 and sr6 are not decoded or executed.

Table 7-8 Long Interrupt Pipeline

Operation	Instruction Cycle															
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
PreFetch 1	n1	n2	ii1	ii2	n3	sr1	sr2	sr3	rti	sr5	sr6	n3	n4	n5	n6	n7
PreFetch 2		n1	n2	jsr	ii2	n3	sr1	sr2	sr3	rti	sr5	sr6	n3	n4	n5	n6
Decode			n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—	—	n3	n4	n5
Address Gen 1				n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—	—	n3	n4
Address Gen 2					n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—	—	n3
Execute 1						n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—	—
Execute 2							n1	n2	jsr	—	—	sr1	sr2	sr3	rti	—
n = normal instruction word ii = interrupt instruction word sr = service routine word																

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

1. The processor status is not saved.
2. The fast interrupt routine may modify the status of the normal instruction stream (e.g., use the DO instruction), but such instructions should not be used in order to assure proper operation.
3. The PC register, which contains the address of the next instruction to be executed in normal processing, remains unchanged during a fast interrupt routine.
4. The fast interrupt returns without an RTI.
5. Normal instruction fetching resumes using the PC register following the completion of the fast interrupt routine.
6. A fast interrupt is not interruptible.
7. A JSR instruction within the fast interrupt routine forms a long interrupt routine.

Execution of a long interrupt routine always conforms 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 registers are stacked. The interrupt mask bits of the SR are updated to mask interrupts of the same or lower priority. The Loop flag and Scaling mode bits are cleared.
3. The interrupt service routine can be interrupted—that is, nested interrupts are supported.
4. The long interrupt routine, which can be any length, should be terminated by an RTI, which restores the PC and SR registers from the stack.

Either one of the two instructions of the fast interrupt can be the JSR instruction that forms the long interrupt.

A 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 register is decremented to 1. During the execution of the repeated instruction, no interrupts are serviced. When the LC register finally decrements to 1, the fetches are reinitiated, and pending interrupts can be serviced.

7.4 RESET PROCESSING STATE

The reset processing state is entered when the external RESET pin is asserted (a hardware reset). Upon entering the reset state:

1. Internal peripheral devices are reset.
2. The modifier registers (M0–M7) are set to $FFFF.
3. The IPR-P and IPR-C registers are cleared.
4. The Bus Control Register (BCR) is set to its initial value as described in **Bus Control Register** on page 9-5. This initial value causes thirty-one wait states (the maximum number of wait states available) to be added to every external memory access.
5. The SP register is cleared.
6. The Scaling mode, Loop flag and Condition Code bits of the SR are cleared, and the interrupt mask bits of the SR are set.
7. The PLL Control registers are initialized as described in **PLL Architecture** on page 8-6.
8. The Vector Base Address (VBA) register is cleared.

The DSP56600 core remains in the reset state until RESET is deasserted. Upon leaving the reset state, the chip operating mode bits of the OMR are loaded from the external mode select pins (MODA, MODB, MODC, and MODD), and program execution begins at the program memory address as described in **Chip Operating Modes** on page 11-3.

7.5 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. All internal processing is halted until an unmasked interrupt occurs, the DSP is reset, or $\overline{\text{DE}}$ is asserted. If exit from Wait state is caused by asserting $\overline{\text{DE}}$, the processor enters the Debug mode.

7.6 STOP PROCESSING STATE

The Stop processing state is the lowest power consumption mode and is entered by the execution of the STOP instruction. In the Stop mode, the clock oscillator activity depends on the Stop Processing State bit (PSTP) in PLL Control Register 1 (PCTL1). If this bit is cleared when the core enters Stop mode, the clock oscillator is turned off. If the bit is se when the core enters Stop mode, the VCO remains active and the global clock to the entire chip is gated off.

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

1. A low level is applied to the $\overline{\text{IRQA}}$ pin ($\overline{\text{IRQA}}$ asserted).
2. A low level is applied to the $\overline{\text{RESET}}$ pin ($\overline{\text{RESET}}$ asserted).
3. A low level is applied to the $\overline{\text{DE}}$ pin.

Any of these actions gates on the oscillator. After a clock stabilization delay, clocks to the processor and peripherals are re-enabled.

When the clocks to the processor and peripherals are re-enabled, the processor enters the reset processing state if the exit from Stop state was caused by assertion of the $\overline{\text{RESET}}$ pin (a low level on the pin).

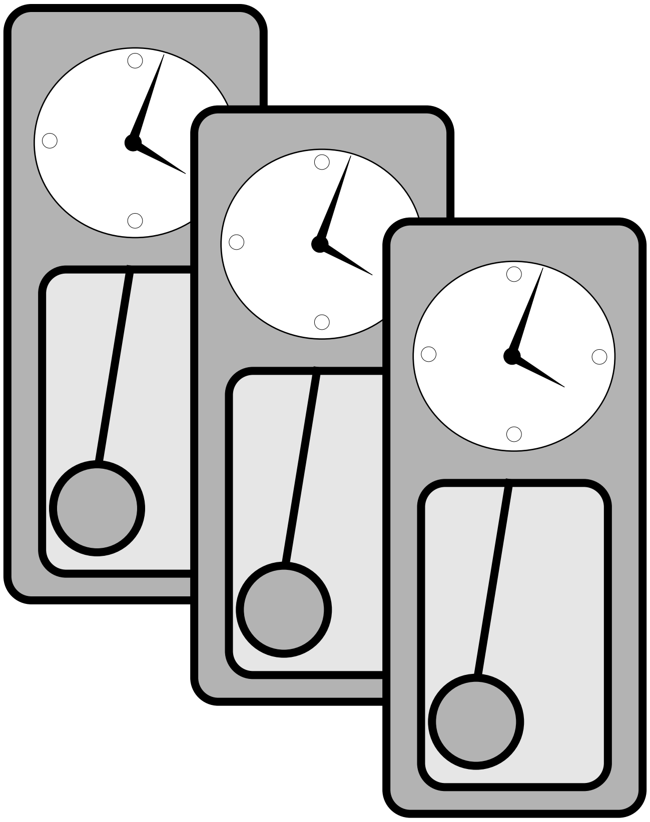
If the exit from Stop state was caused by asserting the $\overline{\text{IRQA}}$ pin, then the processor services the highest priority pending interrupt. If no interrupt is pending (e.g., if the $\overline{\text{IRQA}}$ pin was deasserted before interrupts were arbitrated) or if no interrupt is enabled, then the processor resumes execution at the instruction following the STOP instruction that caused the entry into the Stop state.

If the exit from Stop state was caused by a low level on the $\overline{\text{DE}}$ pin, the processor enters the Debug mode.

For minimum power consumption during the Stop state at the cost of longer recovery time, the PSTP bit in the PCTL1 register should be cleared. To enable faster recovery when exiting the Stop state but at the cost of higher power consumption, the PSTP bit should be set. The PSTP bit is cleared by hardware reset.

SECTION 8

PLL AND CLOCK GENERATOR



8.1 INTRODUCTION

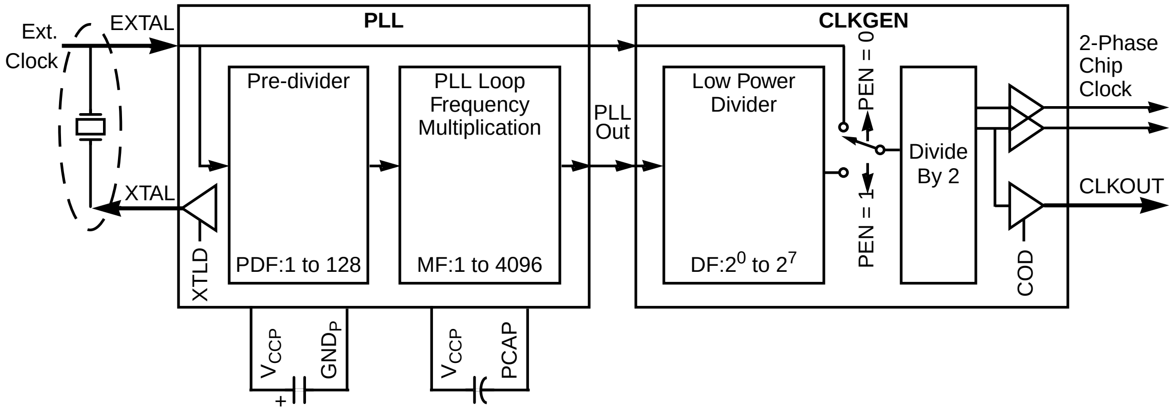
The DSP56600 core features a Phase Lock Loop (PLL) clock oscillator in its central processing module. The PLL allows the processor to operate at a high internal clock frequency using a low frequency clock input, a feature that offers two immediate benefits:

- Lower frequency clock input reduces the overall electromagnetic interference generated by a system.
- The ability to oscillate at different frequencies reduces costs by eliminating the need to add additional oscillators to a system.

The clock generator in the DSP56600 core is composed of two main blocks:

- Phase Lock Loop (PLL) that performs:
  - Clock input division
  - Frequency multiplication
  - Skew elimination
- Clock Generator (CLKGEN) that performs:
  - Low power division
  - Internal and external clock pulse generation

Figure 8-1 shows a block diagram of the clock generator.



Note: The clock source can be either an external source supplied to EXTAL, or a clock oscillator connected to EXTAL and XTAL.

AA0574

Figure 8-1 PLL and Clock Block Diagram

8.2 PLL PINS

The specific PLL pin configuration for each DSP56600 family manual is available in the respective device's *Technical Data* sheet. The following pins are dedicated to the PLL operation:

- PCAP
- CLKOUT
- PINIT
- PLOCK

The PCAP pin provides a connection for an off-chip capacitor for the PLL filter. One terminal of the capacitor is connected to PCAP, and the other terminal is connected to V_{CCP}. The value of this capacitor depends on the Multiplication Factor (MF) of the PLL.

The CLKOUT output pin provides a 50% duty cycle output clock synchronized to the internal processor clock when the PLL is enabled and locked. When the PLL is disabled, the output clock at CLKOUT is derived from, and has half the frequency of, EXTAL. This pin oscillates in all chip processing states except when the Clock Out Disable (COD) bit in the PCTL1 register is set, and during the Stop state. When the chip is in the Wait state, the CLKOUT pin continues to provide a signal.

During the assertion of hardware reset, the value at the PINIT input pin is written into the PLL Enable (PEN) bit of the PLL Control 1 (PCTL1) register. After hardware reset is deasserted, the PINIT pin is ignored by the PLL and can have a different function in the chip.

The PLOCK output originates from the Phase Detector. The chip asserts PLOCK when the PLL is enabled and locked. When the PLOCK output is deasserted by the chip, the PLL is enabled but is not locked. PLOCK is also asserted when the PLL is disabled. PLOCK is a reliable indicator of the PLL lock state only after exiting the hardware reset state.

8.3 CLOCK INPUT DIVISION

The PLL can divide the input frequency by any integer between 1 and 128. The combination of input division and output low-power division (see **Low Power Divide and Output Stage** on page 8-5) enables the user to generate almost every frequency value out of the PLL. The division factor can be modified by changing the value of the Predivider Factor bits (PD0–PD6) in the PLL Control registers PCTL0 and PCTL1. The output frequency of the predivider is

$$\frac{Fext}{PDF}$$

8.3.1 Frequency Multiplication

The PLL can multiply the input frequency by any integer between 1 and 4096. The Multiplication Factor can be modified by changing the value of the Multiplication Factor (MF) bits in PLL Control register 0 (PCTL0). The output frequency of the PLL is

$$\frac{Fext \cdot MF \cdot 2}{PDF}$$

8.3.2 Skew Elimination

The PLL is capable of eliminating the skew between the external clock entering the chip (EXTAL) and the internal clock phases and CLKOUT pin, making it useful for tighter synchronous timings. Skew elimination is active only when the PLL is enabled, the Multiplication Factor is less than five, and the Predivider Factor (PDF) is set to 1. When the PLL is disabled, when the Multiplication Factor is greater than four, or when the PDF is greater than 1, clock skew may exist.

Skew elimination is assured only if the input frequency (EXTAL) is greater than a minimum frequency specified in a device's *Technical Data* sheet (typically 15 MHz).

8.3.3 Low Power Divide and Output Stage

The Clock Generator has a divider connected to the output of the PLL. The output frequency of the PLL can be divided by a factor of 2^n (where $0 \leq n \leq 7$). The division factor can be modified by changing the value of the Division Factor bits DF[2:0] in the PLL Control register 1 (PCTL1). This divider permits reducing or restoring the chip operating frequency without losing the PLL lock.

The output stage of the Clock Generator generates the clock signals to the core and the chip peripherals, and drives the CLKOUT pin. The output stage divides the frequency by 2. The input source to the output stage is selected between:

- EXTAL itself (PEN = 0, PLL disabled), which causes the chip frequency to be

$$\frac{Fext}{2}$$

- Low Power Divider output (PEN = 1, PLL enabled), which causes the chip frequency to be

$$\frac{Fext \cdot MF}{PDF \cdot DF}$$

8.4 PLL ARCHITECTURE

The PLL block diagram is shown in **Figure 8-2**. The components of the PLL are described in the following sections.

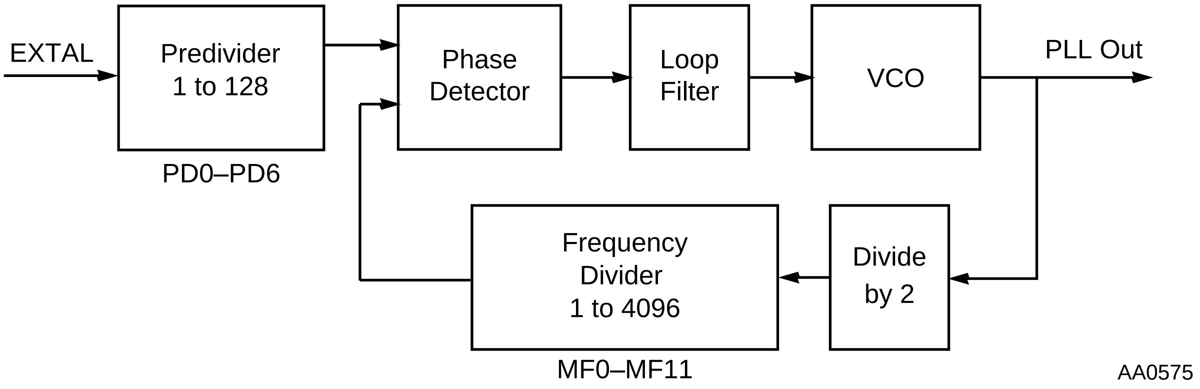


Figure 8-2 PLL Block Diagram

8.4.1 Frequency Predivider

Clock input frequency division is accomplished by means of a frequency divider of the input frequency. The programmable Division Factor ranges from 1 to 128.

8.4.2 Phase Detector and Charge Pump Loop Filter

The Phase Detector (PD) detects any phase difference between the external clock (EXTAL) and an internal clock phase from the frequency multiplier. At the point where there is negligible phase difference and the frequency of the two inputs is identical, the PLL is in the locked state. The charge pump loop filter receives signals from the PD, and either increases or decreases the phase based on the PD signals. An external capacitor is connected to the PCAP pin and determines the PLL operation. The value of this capacitor depends on the Multiplication Factor (MF) of the PLL. See **Section 2, Specifications** in the device's *Technical Data* sheet for a formula to use to determine the proper value for the PLL capacitor. After the PLL locks on to the proper phase and frequency, it reverts to the Narrow Bandwidth mode, which is useful for tracking small changes due to frequency drift of the EXTAL clock.

8.4.3 PLL Control Register 0 (PCTL0)

The PLL Control register 0 (PCTL0) is an X I/O-mapped 16-bit read/write register used to direct the operation of the on-chip PLL. **Figure 8-3** shows the programming model for the PCTL0 register.

PCTL0—X:$FFFD	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PLL Control Register 0 Reset = $0000 Read/Write	PD 3	PD 2	PD 1	PD 0	MF 11	MF 10	MF 9	MF 8	MF 7	MF 6	MF 5	MF 4	MF 3	MF 2	MF 1	MF 0

* Indicates reserved bits, read as 0 and should be written with 0 for future compatibility

AA0751

Figure 8-3 PLL Control Register 0 (PCTL0) Format

8.4.3.1 Multiplication Factor Bits (MF0–MF11)—Bits 0–11

The Multiplication Factor bits MF0–MF11 define the Multiplication Factor (MF) that is applied to the PLL input frequency. The MF can be any integer from 1 to 4096. The VCO oscillates at a frequency of

$$\frac{Fext \cdot MF \cdot 2}{PDF}$$

where PDF is the division factor of the Predivider. **Table 8-1** shows how to program the MF0–MF11 bits.

Table 8-1 Multiplication Factor Bits MF0–MF11

MF11–MF0	Multiplication Factor MF
$000	1
$001	2
$002	3
• • •	• • •
$FFE	4095
$FFF	4096

The MF must be chosen to ensure that the resulting VCO output frequency is in the range specified in the chip’s *Technical Data* sheet. Any time a new value is written into the MF0–MF11 bits, the PLL loses the lock condition. After a time delay, the PLL relocks. (This time is specified in the *Technical Data* sheet.) The Multiplication Factor bits (MF0–MF11) are set to a predetermined value during hardware reset; the value is implementation-dependent and can be found in the *User’s Manual*.

The Multiplication Factor Bits (MF0–MF11) in the PLL Control register 0 (PCTL0) define the MF that is applied to the PLL input frequency. The MF0–MF11 bits are set to a predetermined value during hardware reset. For example, in the DSP56603 this value is $000, which corresponds to an MF of 1.

8.4.3.2 Predivider Factor Bits (PD0–PD3)—Bits 12–15

The Predivider Factor bits PD0–PD3 define the least significant part of the Predivider Factor (PDF) that is applied to the PLL input frequency. The PDF can be any integer from 1 to 128. Note that the PD4–PD6 bits, which represent the most significant part of the PDF, are located in the PCTL1 register. The VCO oscillates at a frequency of

$$\frac{\mathrm{Fext} \cdot \mathrm{MF} \cdot 2}{\mathrm{PDF}}$$

The PDF must be chosen to ensure that the resulting VCO output frequency is in the range specified in the device’s *Technical Data* sheet. Any time a new value is written into the PD0–PD6 bits, the PLL loses the lock condition. After a time delay, the PLL relocks. The Predivider Factor bits (PD0–PD6) are set to a predetermined value during hardware reset. This value is implementation-dependent and can be found in each DSP56600 family member *User's Manual*. **Table 8-2** shows how to program the PD0–PD6 bits.

Table 8-2 Predivider Factor Bits PD0–PD6

PD[6:0]	Predivider Factor PDF
$0	1
$1	2
$2	3
• • •	• • •
$7E	127
$7F	128

8.4.4 PLL Control Register 1 (PCTL1)

The PLL Control register 1 (PCTL1) is an X I/O-mapped 16-bit read/write register used to direct the operation of the on-chip PLL. The PCTL1 control bits are described in the following sections.

PCTL1—X:$FFFD
PLL Control
Register 1
Reset = $0000
Read/Write

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
*	*	*	*	PD 6	PD 5	PD 4	*	COD	PEN	PS TP	XT LD	XT LR	DF 2	DF 1	DF 0

* Indicates reserved bits, read as 0 and should be written with 0 for future compatibility

AA0752

Figure 8-4 PLL Control Register 1(PCTL1) Format

8.4.4.1 Division Factor Bits (DF0–DF2)—Bits 0–2

The Division Factor (DF2–DF0) bits define the Division Factor (DF) of the low power divider. These bits specify any power-of-two Division Factor in the range from 2^0 to 2^7. **Table 8-3** shows the programming of the DF bits. Changing the value of the DF bits does not cause a loss of lock condition. Whenever possible, changes of the operating frequency of the chip (e.g., to enter a low power mode) should be made by changing the value of the DF2–DF0 bits rather than by changing the MF0–MF11 bits.

For MF ≤ 4, changing DF2–DF0 lengthens the instruction cycle following the PLL control register update. This is done in order to keep synchronization between EXTAL and the internal chip clock. For MF > 4, synchronization is not guaranteed and the instruction cycle is not lengthened. The DF bits are cleared by hardware reset—setting the DF to divide by 1.

Table 8-3 Division Factor Bits DF0–DF2

DF[2:0]	Division Factor DF
000	2^0
001	2^1
010	2^2
011	2^3
100	2^4
101	2^5
110	2^6
111	2^7

8.4.4.2 Crystal Range Bit (XTLR)—Bit 3

The Crystal Range (XTLR) bit controls the on-chip crystal oscillator transconductance. If the external crystal frequency is less than 200 kHz (“fork crystal”), this bit should be set in order to decrease the transconductance of the input amplifier, otherwise the internal clocks may not be stable. If the external crystal frequency is greater than 200 kHz, this bit should be cleared in order to have the full transconductance, otherwise the crystal oscillator may not function at all. Changing the XTLR bit while the PLL is active causes a loss of PLL lock and a reinitialization of the lock process. The XTLR bit is set to a predetermined value during hardware reset. This value is implementation-dependent and may vary between DSP56600 family members.

8.4.4.3 XTAL Disable Bit (XTLD)—Bit 4

The XTAL Disable (XTLD) bit controls the on-chip crystal oscillator XTAL output. When XTLD is cleared, the XTAL output pin is active, permitting normal operation of the crystal oscillator. When XTLD is set, the XTAL output pin is held high, disabling the on-chip crystal oscillator. If the on-chip crystal oscillator is not used (EXTAL is driven from an external clock source), it is recommended to set XTLD (disabling XTAL) to minimize RFI noise and power dissipation. Changing the XLTD bit while the MF value is greater than 4 causes a loss of PLL lock condition and restarts the PLL lock process. The XTLD bit is set to a predetermined value during hardware reset. This value is implementation-dependent and may vary between DSP56600 family members.

8.4.4.4 Stop Processing State Bit (PSTP)—Bit 5

The Stop Processing State (PSTP) bit controls the behavior of the PLL and of the on-chip crystal oscillator during the Stop processing state. When the PSTP bit is set, the PLL and the on-chip crystal oscillator remain operating while the chip is in the Stop state. When the PSTP bit is cleared, the PLL and the on-chip crystal oscillator are disabled when the chip enters the Stop state. For minimum power consumption during the Stop state at the cost of longer recovery time, the PSTP bit should be cleared. To enable rapid recovery when exiting the Stop state, at the cost of higher power consumption, the PSTP bit should be set. The PSTP bit is cleared by hardware reset.

8.4.4.5 PLL Enable Bit (PEN)—Bit 6

The PLL Enable (PEN) bit enables the PLL operation. When this bit is set, the PLL is enabled and the internal clocks are derived from the PLL VCO output. When this bit is cleared, the PLL is disabled and the internal clocks are derived directly from the clock connected to the EXTAL pin. When the PLL is disabled, the VCO is not operating. This helps minimize power consumption. The PEN bit can be set or cleared by software any time during the chip operation. During hardware reset, this bit receives the value of the PLL’s PINIT pin.

A relationship exists between the PSTP and PEN bits where the PEN bit adjusts PSTP’s control of the PLL operation. When the PSTP bit is set and the PEN bit is cleared, the on-chip crystal oscillator remains operating in the Stop state, but the PLL is disabled (see **Table 8-4**). This power saving feature enables rapid recovery from the Stop state when the user operates the chip with an on-chip oscillator and with the PLL disabled.

Table 8-4 PSTP and PEN Relationship

PSTP	PEN	Operation During Stop		Recovery Time From Stop	Power Consumption During Stop
		PLL	Oscillator		
0	x	Disabled	Disabled	Long	Minimal
1	0	Disabled	Enabled	Short	Lower
1	1	Enabled	Enabled	Short	Higher

8.4.4.6 Clock Output Disable Bit (COD)—Bit 7

The Clock Output Disable (COD) bit controls the output buffer of the clock at the CLKOUT pin. When the COD bit is set, the CLKOUT pin is held high. When the COD bit is cleared, the CLKOUT pin is active, providing a 50% duty cycle clock synchronized to the internal core clock. If the CLKOUT pin is not connected to external circuits, the COD bit should be set, disabling clock output and minimizing RFI noise and power dissipation. CLKOUT oscillates in all operating states except the Stop state. The COD bit is cleared by hardware reset, allowing CLKOUT to be provided.

8.4.4.7 Reserved PCTL1 Bit—Bit 8

This bit is reserved for future expansion. It is read as 0 and should be written with 0 for future compatibility.

8.4.4.8 Predivider Factor Bits (PD4–PD6)—Bits 9–11

The Predivider Factor Bits PD4–PD6 define the most significant part of the Predivider Factor (PDF) that is applied to the PLL input frequency. The PDF can be any integer from 1 to 128. **Table 8-2** on page 8-9 shows how to program the PD0–PD6 bits. Note that PD0–PD3 are located in the PCTL0 register.

8.4.4.9 Reserved PCTL1 bits—Bits 12–15

These bits are reserved for future expansion. They are read as 0 and should be written with 0 for future compatibility.

8.4.5 Voltage Controlled Oscillator (VCO)

The Voltage Controlled Oscillator (VCO) is capable of oscillating at frequencies from the minimum speed specified in a device's *Technical Data* sheet up to the maximum allowed clock input frequency.

Note: When the PLL is enabled, the chip operating frequency is half of the VCO oscillating frequency.

If the frequency of EXTAL is less than the VCO's minimum working frequency, the PINIT pin should be held low during hardware reset. Following reset, the MF value can be changed in software to the desired value, and the PEN bit set to 1.

8.4.5.1 Divide by 2

The output of the VCO is divided by 2. This results in a constant × 2 multiplication of the PLL clock output used to generate the special chip clock phases.

8.4.5.2 Frequency Divider

The Frequency Divider, which is connected to the feedback loop of the PLL, is used to multiply the incoming external clock. In the PLL close-loop, the effect of the frequency divider is to multiply the PLL input frequency by its Division Factor. The programmable Division Factor ranges from 1 to 4096, resulting in frequency multiplication in the same range. This factor is programmable using the MF0–MF11 bits in the PCTL0 register.

8.5 CLKGEN BLOCK DIAGRAM

The Clock Generator block diagram is shown in **Figure 8-5**. The components of the Clock Generator are described in the following sections.

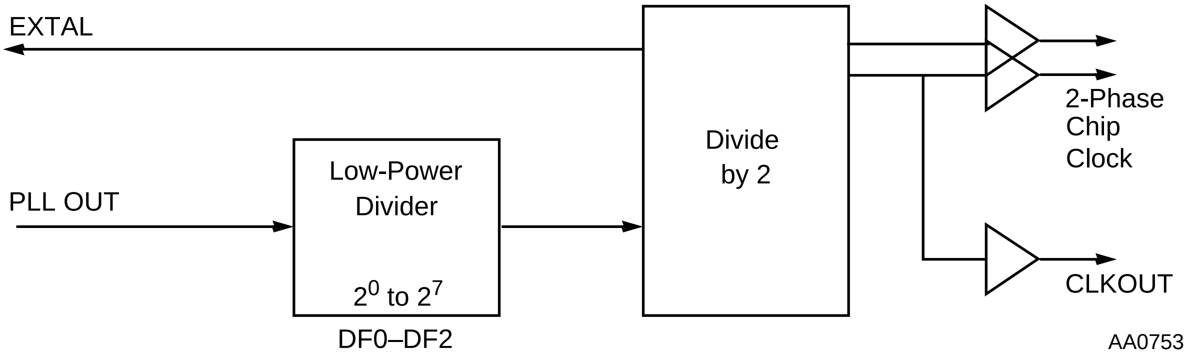


Figure 8-5 CLKGEN Block Diagram

8.5.1 Low Power Divider (LPD)

The Low Power Divider (LPD**)** divides the output frequency of the VCO by any power of 2 from 2^0 to 2^7. Since the LPD is not in the closed loop of the PLL, changes in the divide factor do not cause a loss of lock condition. This fact is particularly useful for utilizing the LPD in low power consumption modes when the chip is not involved in intensive calculations. This can result in significant power saving. When the chip is required to exit a low power mode, it can immediately do so with no time needed for clock recovery or PLL lock.

8.5.2 Divide by 2

The EXTAL clock and the output of the Low-Power Divider are selected according to the PEN bit in the PCTL1 register. The selected clock frequency is divided by two and is driven to the internal chip activity and to the CLKOUT pin.

8.5.3 Operating Frequency

The operating frequency of the chip is governed by the frequency control bits in the PLL control register as follows:

$$F_{CHIP} = \frac{F_{EXT} \times MF}{PDF \times DF} = \frac{Fvco}{DF}$$

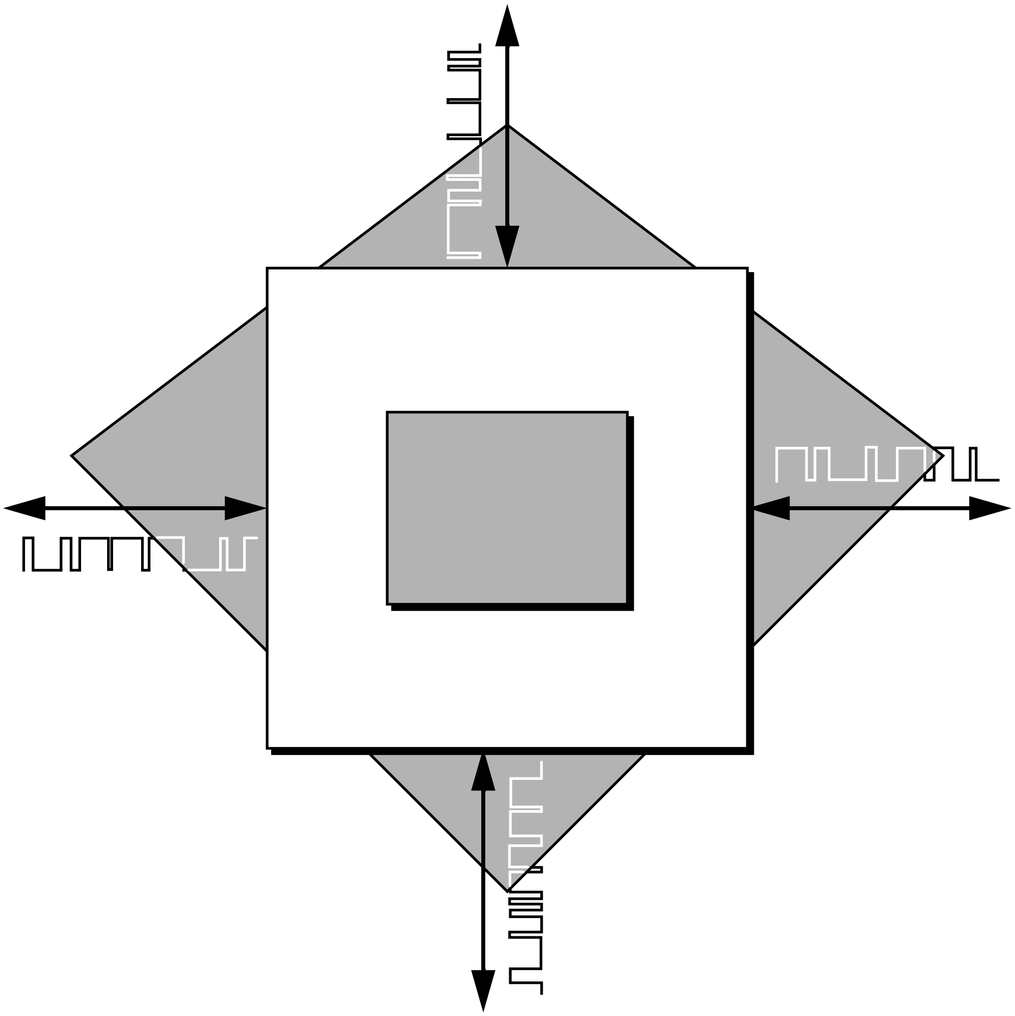
where MF is the Multiplication Factor defined by the MF0–MF11 bits, PDF is the Predivider Factor defined by the PD0–PD6 bits, and DF is the Division Factor defined by the DF0–DF2 bits. F_{CHIP} is the chip operating frequency, and F_{EXT} is the external input frequency to the chip at the EXTAL pin.

8.6 CLOCK SYNCHRONIZATION

When the PLL is enabled (the PEN bit in the PCTL1 register is set), low clock skew between EXTAL and CLKOUT is guaranteed if MF ≤ 4. CLKOUT and the internal chip clock are fully synchronized.

SECTION 9

EXTERNAL MEMORY INTERFACE (PORT A)



9.1 INTRODUCTION

Port A is the external memory expansion port that is used for program memory expansion either for program code or for data. It provides an easy to use, low part-count connection with fast or slow static memories and with I/O devices.

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 two clock cycles. External memory can be as large as 64 K × 24-bit program memory space, depending on chip configuration. An internal wait state generator can be programmed to insert as many as 31 wait states if access to slower memory or I/O device is required. A complete description of the signals provided on Port A is found in **Section 1, Signal/Connection Descriptions**, of the device's *Technical Data* sheet.

9.2 EXTERNAL MEMORY INTERFACE OPERATION

The external bus timing is defined by the operation of the Address Bus, Data Bus, and Bus Control pins described in the device's *Technical Data* sheet. The DSP56600 core external ports are designed to interface with high-speed Static RAMs and peripheral devices with Static RAM-based timing, as well as slower memory devices.

9.2.1 Static RAM Support

External bus timing is controlled by the Bus Control Register (BCR) that is described in **Bus Control Register** on page 9-5. Insertion of wait states is controlled by the BCR to provide constant bus access timing. The number of wait states for each external access is determined by the BCR.

The external memory address is defined by the Address Bus. The Memory Chip Select signal ($\overline{\text{MCS}}$) is used to generate a chip select signal for the external memory device. This signal changes the mode of the memory device from low power Standby mode to its active mode and begins the access. This allows slower memories to be used since the Chip Select signal is address-based rather than read or write enable-based. Static RAMs can be easily interfaced to the DSP56600 core bus timing. Due to the Static RAM requirement to keep the address stable during the entire bus cycle, at least one wait state must be inserted to the bus operation. **Figure 9-2** on page 9-5 shows a possible timing configuration. For detailed timing information, see the device's *Technical Data* sheet.

External Memory Interface Operation

A Static RAM access is performed in one of the following ways:

Write Access

1. A0–A15 and $\overline{\text{MCS}}$ are asserted in the middle of CLKOUT low phase.
2. $\overline{\text{WR}}$ is asserted with the rising edge of CLKOUT (for a single wait state access).
3. Data is driven in the middle of CLKOUT low phase.

Read Access

1. A0–A15 and $\overline{\text{MCS}}$ are asserted in the middle of CLKOUT low phase.
2. $\overline{\text{RD}}$ is asserted with the rising edge of CLKOUT.
3. Data is sampled in the middle of CLKOUT last high phase of the external access.

Wait states postpone the disappearance of the external address, thus increasing memory access time. In any case, Static RAM access requires at least one wait state.

Figure 9-1 shows a typical connection between a DSP56600 family member and an external SRAM memory.

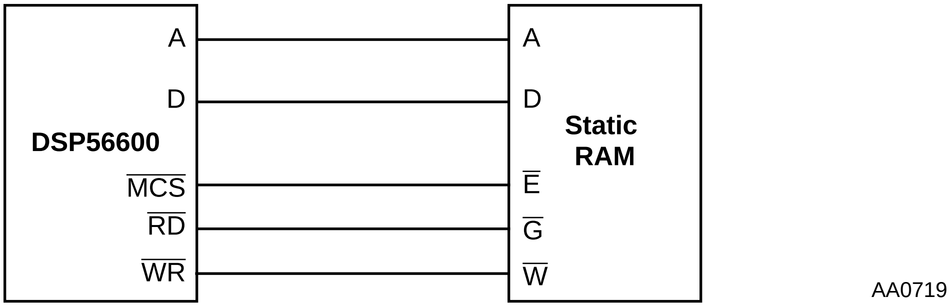
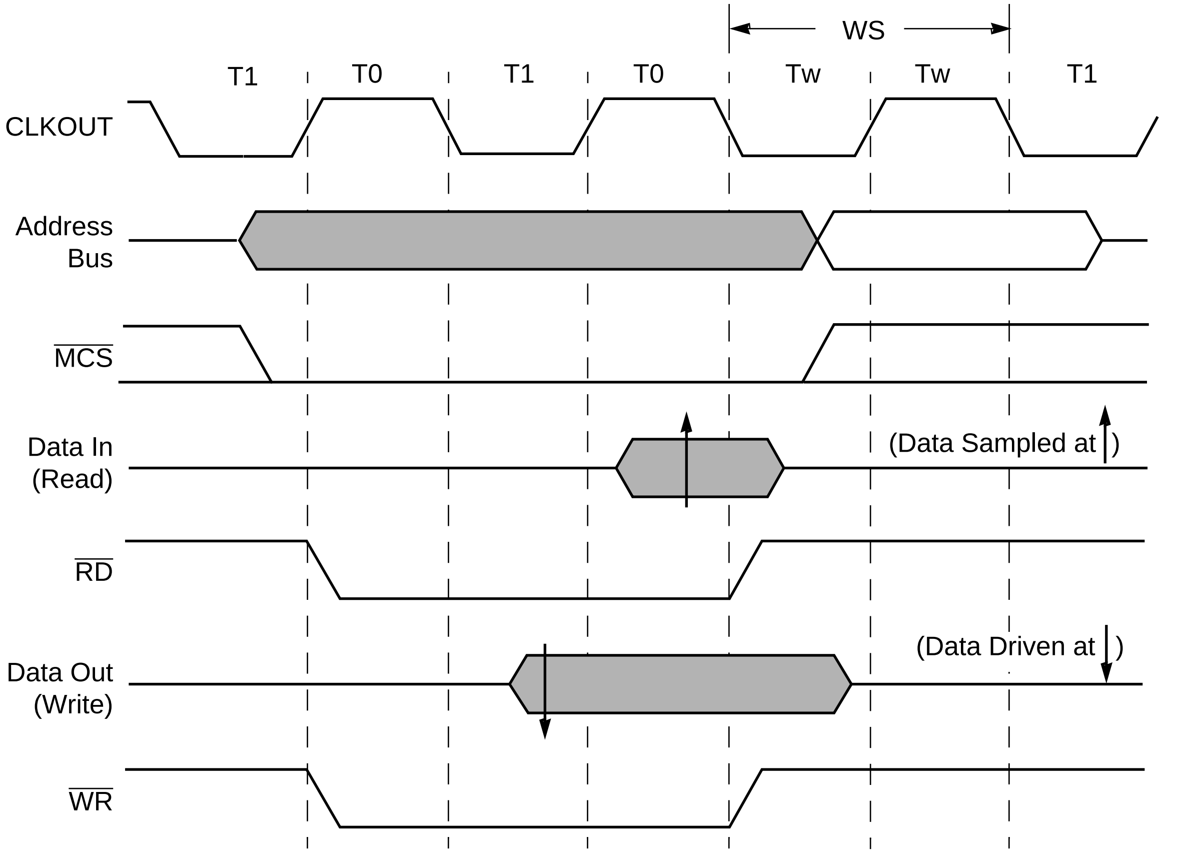


Figure 9-1 Static RAM Connection Diagram

Figure 9-2 shows a typical timing diagram for bus operation using one wait state. For detailed specifications, see the device's *Technical Data* sheet.



Note: For detailed timing specification see the device’s *Technical Data* sheet.

AA0579

Figure 9-2 Bus Operation, One Wait State—Static RAM Access

Note: The assertion of the $\overline{\text{WR}}$ signal depends on the number of wait states programmed in the BCR. If a single wait state is programmed in the BCR, the $\overline{\text{WR}}$ signal is asserted with the rising edge of CLKOUT. If the number of wait states programmed is two or three, $\overline{\text{WR}}$ assertion is delayed by half cycle of CLKOUT. If the number of wait states programmed is four or more, $\overline{\text{WR}}$ assertion is delayed by a full cycle of CLKOUT. This feature enables the connection of slow external devices that require long address setup time before write assertion in order to prevent false write.

9.2.2 Bus Control Register

The expansion port control consists of the Bus Control Register (BCR), a 16-bit read/write register used to control the external bus activity and Bus Interface Unit operation. The BCR is shown in **Figure 9-3** and described in the following paragraphs.

BCR—X:$FFFC	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bus Control Register Reset = $001F Read/Write	*	*	*	*	*	*	*	*	*	*	*	BMW 4	BMW 3	BMW 2	BMW 1	BMW 0

* Indicates reserved bits, read as 0 and should be written with 0 for future compatibility

Figure 9-3 Bus Control Register (BCR) Format

9.2.2.1 Expansion Bus Memory Wait (BMW0–BMW4)—Bits 0–4

The Expansion Bus Memory Wait control bits (BMW0–BMW4) define the number of wait states, from 0 to 31, inserted in each external SRAM access. Since SRAM memory access requires at least one wait state, the value of these bits should never be programmed as $0000.

When selecting 4 to 7 wait states, one additional wait state is inserted at the end of the access. When selecting eight or more wait states, two additional wait states are inserted at the end of the access. These trailing wait states increase the data hold time and the memory release time and do not increase the memory access time. The BMW[4:0] bits are set during hardware reset to provide the maximum of 31 wait states.

9.2.2.2 Reserved Bits—Bits 5–15

Bits 5 through 15 in the BCR are reserved. These bits should be written as 0 to ensure future compatibility.

9.2.3 Expansion Port Disable

In many applications that are sensitive to power consumption, there is no use of the expansion port because all the memory resides inside the chip itself. A special feature of the expansion port controller enables the user to reduce significantly the power consumption of the expansion port controller by setting the EBD bit in the Operating Mode Register (OMR). If this bit is set, the expansion port controller is disabled. When the EBD bit is set, the user should not attempt to access the external memory. Otherwise, improper operation will result.

9.3 PROGRAM MEMORY DATA TRANSFER

The internal and external program memories consist of 24-bit wide words. The access to a program memory word is required in any instruction fetch and in program memory move instructions. In the latter case a special 24-bit register, the Bus Switch Program Memory Register (BPMR), is used to interface between the internal and external program memory spaces and the rest of the DSP56600 core, which consists mostly of 16-bit components.

9.3.1 Bus Switch Program Memory Register (BPMR)

The Bus Switch Program Memory Register (BPMR) is a 24-bit X I/O mapped read/write register. An access to the BPMR can be either a 24-bit access or a 16-bit access. The BPMR is composed of two parts, the Bus Switch Program Memory Register Low (BPMRL) and the Bus Switch Program Memory Register High (BPMRH). Each move from a 16-bit source to a 24-bit destination is extended by the 8 lower bits of the BPMRH. Each move from a 24-bit source to a 16-bit destination truncates the 24-bit source and moves only its 16 Least Significant Bits (LSBs). Using the BPMR is the only way to access the 8 Most Significant Bits (MSBs) of any program memory address, external or internal, which is essential for many applications. For example, when using a system configuration operating mode, a hardware reset causes the DSP56600 core to jump to the mask programmed internal program memory location and execute the code fetched from this location. This code usually includes a set of program memory move instructions that load the program code to the required destination and then execute it. The access to the 8 MSBs of each 24-bit program word is available only by using BPMR.

9.3.2 BPMR Mapping

The BPMR is mapped as a 24-bit register in address BPMRG and as a pair of 16-bit and 8-bit registers in addresses x:BPMRL and x:BPMRH, respectively. The 8-bit register BPMRH is part of a 16-bit X I/O mapped register where its 8 MSBs are reserved. **Figure 9-4** shows the BMPR mapping.

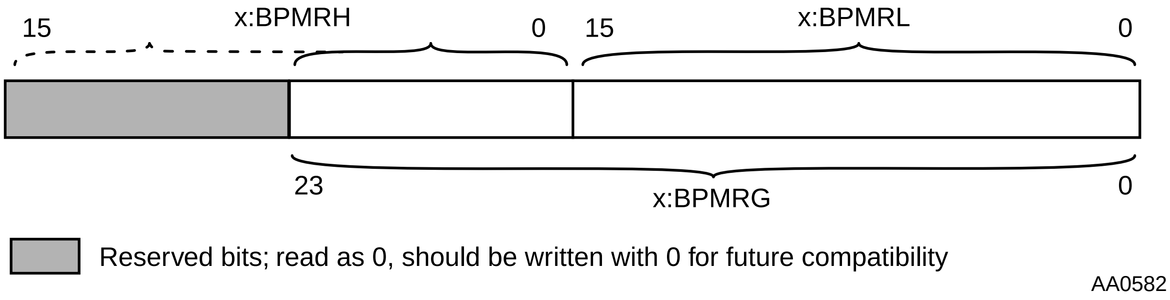

15
x:BPMRH
0
15
x:BPMRL
0
23
x:BPMRG
0
Reserved bits; read as 0, should be written with 0 for future compatibility
AA0582

Figure 9-4 BMPR Mapping

9.3.3 24-Bit Access to BPMR

A 24-bit access to BPMR is done by move instruction between BPMR and any program memory word as follows:

Example 9-1 24-Bit Access to BMPR

```
;EXAMPLE: move from P source address to P destination address

BPMRG          equ            $FFF4

               movep          p:(r0), x:BPMRG

               movep          x:BPMRG, p:$5
```

9.3.4 16-Bit Access to BPMR

A 16-bit access to the BPMR is done either to its 16 LSBs, which are mapped to x:BPMRL, or to its 8 MSBs, which are mapped to x:BPMRH. In both cases it is not treated as a special access but as a regular 16-bit X I/O access. Reading x:BPMRH will clear the 16 MSBs of the 24-bit destination or the 8 MSBs of the 16-bit destination, depending on the destination's width.

9.3.5 BPMR Usage Typical Examples

A typical usage of the BPMR is for bootstrap through external EPROM or through the Host Interface (HI08). The following code, when loaded to an external EPROM hardware reset location can load any required Program RAM segment. **Example 9-2** describes the part of it that uses the BPMR.

Example 9-2 Bootstrap Through External EPROM

```
BPMRG          equ    $FFF4
BPMRL          equ    $FFF3
BPMRH          equ    $FFF2

;==========================================================
; This is part of the routine that loads from external EPROM.
; The external EPROM is 8 bit wide.


       do #2,_LOOP1                 ; Read the 16 LSB part of the instruction.
       movem p:(r2)+,a2             ; Get the 8 LSB from ext. P mem.
       asr #8,a,a                   ; Shift 8 bit data into A1.
_

LOOP1                               ; Go get another byte.
       movep a1,x:BPMRL             ; Store the 16 LSB part in BPMRL.
       movep p:(r2)+,x:BPMRH        ; Get the 8 MSB part and store it in BPMRH.
       movep x:BPMRG,p:(r0)+        ; Store 24 Bit result in P mem.
```

A common debugging process requires the content of a segment of program memory code to be delivered to the external command controller. This information should be passed through the OGDB register. The only way to pass the 8 MSBs of each 24-bit program word to the OGDB register is by using BPMR. **Example 9-3** shows how the OGDB register is loaded by a 24-bit program memory word.

Example 9-3 Passing Program Memory Words to the OGDB Register

```
BPMRG          equ    $FFF4

BPMRL          equ    $FFF3

BPMRH          equ    $FFF2

OGDB           equ    $FFFB

;==========================================================

       movep p:(r2)+,x:BPMRG        ; Read the 24 bit data and store in BPMR.

       movep x:BPMRL,x0             ; Store the 16 LSB part in x0.

       movep x0,x:OGDB              ; Pass the 16 LSB part to OGDB.

       movep x:BPMRH,y0             ; Store the 8 MSB part in y0.

       movep y0,x:OGDB              ; Pass the 8 MSB part to OGDB.
```

9.4 PROGRAM ADDRESS TRACING MODE

The Address Tracing (AT) mode provides a means of software development in addition to the On-Chip Emulation (OnCE) circuitry. When the AT mode is enabled by setting the ATE bit in the OMR, the DSP56600 core reflects the addresses of internal fetches and program space moves (MOVEM) to the Address Bus (A0–A15), if the Address Bus is not needed by the DSP56600 core for external accesses. When an AT cycle is performed (an internal access reflected to the Address Bus), $\overline{\text{RD}}$ and $\overline{\text{WR}}$ strobes and $\overline{\text{MCS}}$ signal are not asserted. This assures that no external device is erroneously activated. The $\overline{\text{AT}}$ signal is used to indicate a new address on the Address Bus, either of an AT cycle or of an external access. The user may sample the Address Bus and the $\overline{\text{MCS}}$ signal with the falling edge of $\overline{\text{AT}}$ and sort between the AT cycles and the external accesses according to the sampled value of the $\overline{\text{MCS}}$ signal.

Note: The trace capability of the AT mechanism is not identical to the OnCE trace buffer capability. The AT mechanism provides information on fetches, not on program flow. In the AT mechanism, fetches for a jump that is not taken are sampled, although the program flow has not gone that way. The software that interprets this information must be aware of such aspects.

Figure 9-5 shows a possible configuration. For detailed timing information, see the device's *Technical Data* sheet.

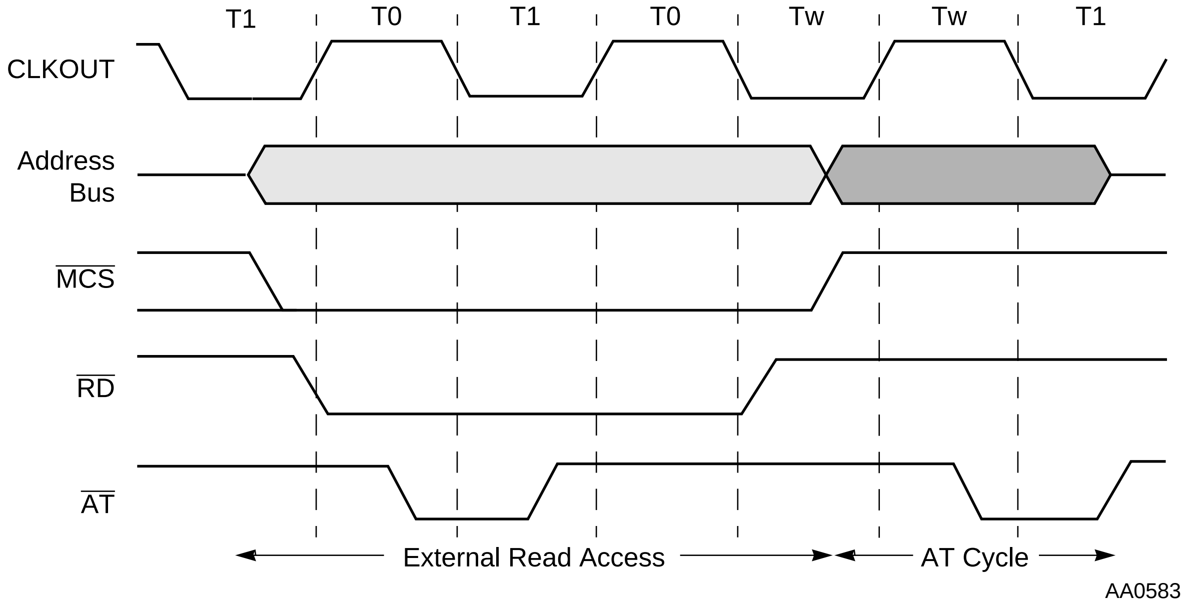
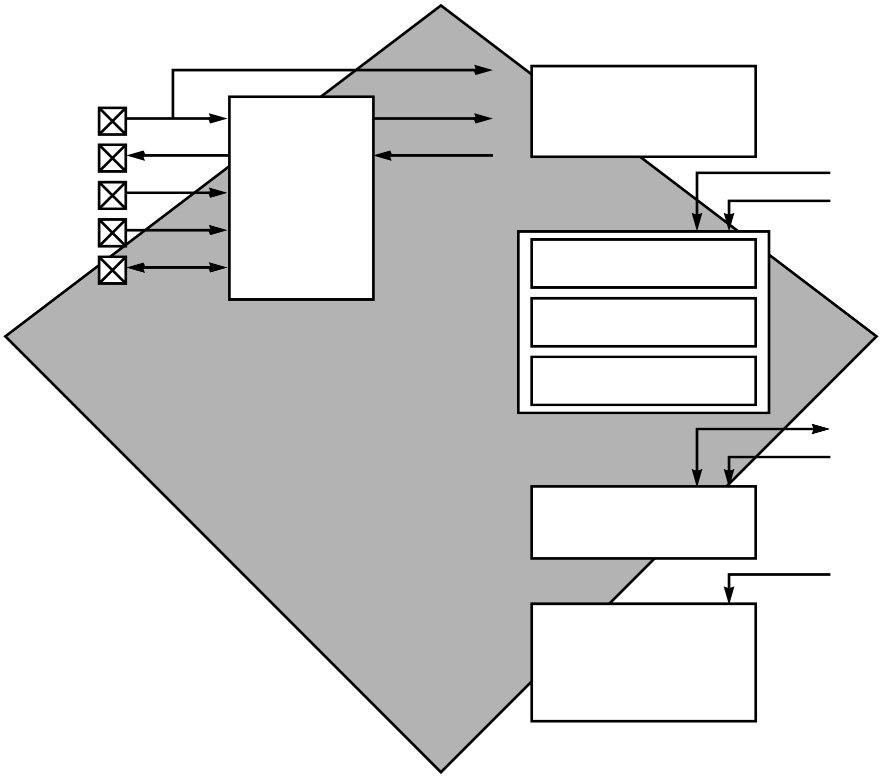


Figure 9-5 Address Tracing Possible Configuration Diagram

SECTION 10

JTAG PORT AND OnCE MODULE



10.1 INTRODUCTION

The DSP56600 core provides a dedicated user-accessible Test Access Port (TAP) that is fully compatible with the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)*. Problems associated with testing high density circuit boards have led to development of this standard under the sponsorship of the Test Technology Committee of IEEE and the Joint Test Action Group (JTAG). The DSP56600 core implementation supports circuit-board test strategies based on this standard.

The test logic includes a TAP consisting of four dedicated signal pins, a 16-state controller, and three test data registers. A Boundary Scan Register (BSR) links all device signal pins into a single shift register. The test logic, implemented utilizing static logic design, is independent of the device system logic. The DSP56600 core implementation provides the following capabilities:

- Perform boundary scan operations to test circuit-board electrical continuity (EXTEST)
- Bypass the DSP56600 core for a given circuit-board test by effectively reducing the BSR to a single cell (BYPASS)
- Sample the DSP56600 core-based device system pins during operation and transparently shift out the result in the BSR; preload values to output pins prior to invoking the EXTEST instruction (SAMPLE/PRELOAD)
- Disable the output drive to pins during circuit-board testing (HIGHZ)
- Provide a means of accessing the OnCE controller and circuits to control a target system (ENABLE_ONCE)
- Provide a means of entering the Debug mode of operation (DEBUG_REQUEST)
- Query identification information (manufacturer, part number and version) from an DSP56600 core-based device (IDCODE)
- Force test data onto the outputs of an DSP56600 core-based device while replacing its BSR in the serial data path with a single-bit register (CLAMP)

10.2 JTAG PORT

This section, which includes aspects of the JTAG implementation that are specific to the DSP56600 core, is intended to be used with the supporting IEEE 1149.1 document. The discussion includes those items required by the standard to be defined and, in certain cases, provides additional information specific to the DSP56600 core implementation. For internal details and applications of the standard, refer to the IEEE 1149.1 document. The block diagram of the DSP56600 core implementation of JTAG is shown in **Figure 10-1**.

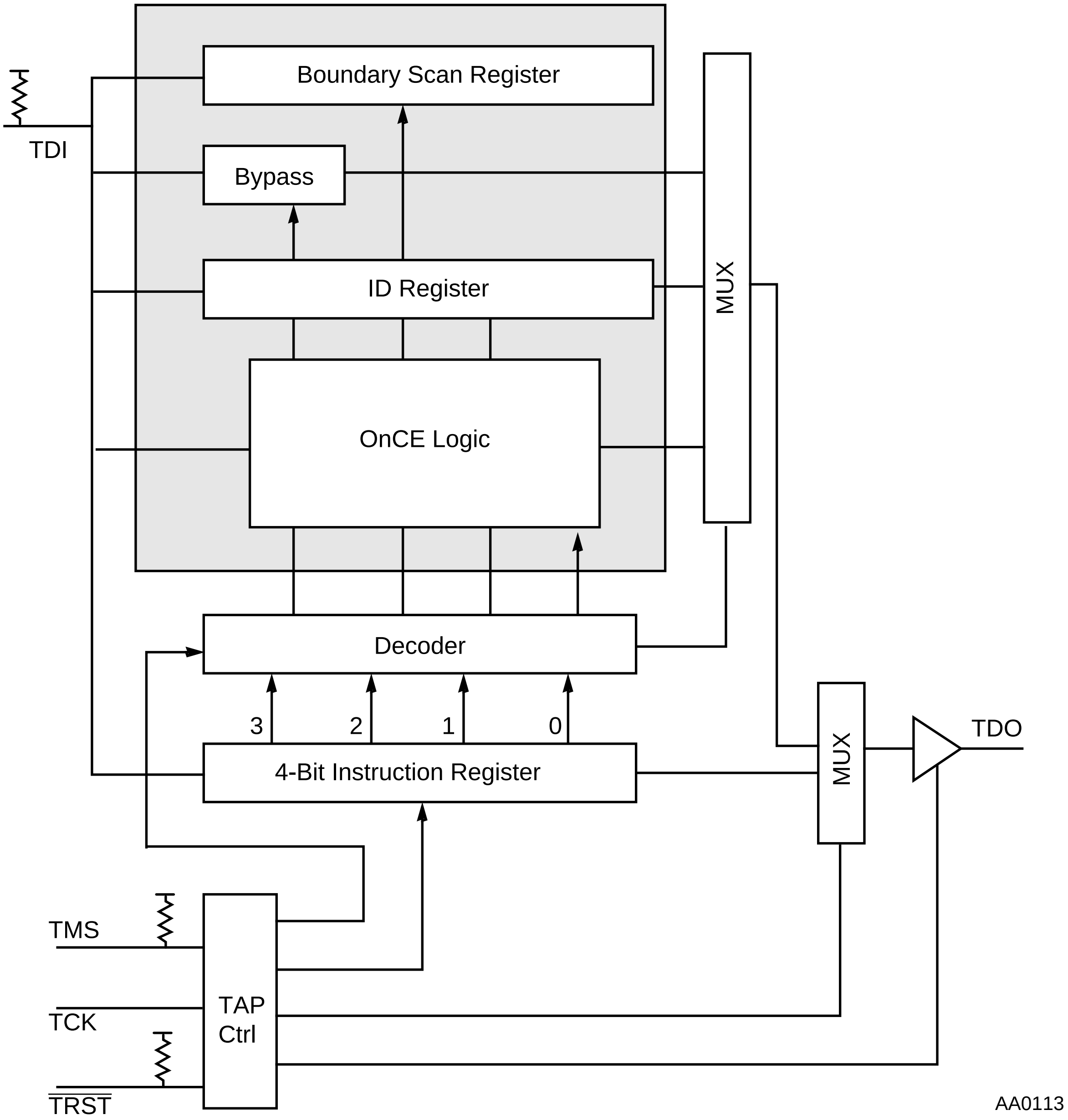


Figure 10-1 JTAG Block Diagram

The DSP56600 core implementation includes a 4-bit instruction register and three test registers: a 1-bit bypass register, a 32-bit identification register, and a BSR whose size is chip-specific. This implementation includes a dedicated TAP and five pins.

10.2.1 JTAG Pins

As described in the IEEE 1149.1 document, the JTAG port requires a minimum of four pins to support TDI, TDO, TCK, and TMS signals. The DSP56600 family also provides the optional $\overline{\text{TRST}}$ pin. The pin functions are described in the following paragraphs.

10.2.1.1 Test Clock (TCK)

The test clock input (TCK) pin is used to synchronize the test logic.

10.2.1.2 Test Mode Select (TMS)

The test mode select input (TMS) pin is used to sequence the test controller's state machine. The TMS is sampled on the rising edge of TCK and it has an internal pullup resistor.

10.2.1.3 Test Data Input (TDI)

Serial test instruction and data are received through the test data input (TDI) pin. TDI is sampled on the rising edge of TCK and it has an internal pullup resistor.

10.2.1.4 Test Data Output (TDO)

The test data output (TDO) pin is the serial output for test instructions and data. TDO is tri-stateable and is actively driven in the shift-IR and shift-DR controller states. TDO changes on the falling edge of TCK.

10.2.1.5 Test Reset ($\overline{\text{TRST}}$)

The test reset input ($\overline{\text{TRST}}$) pin is used to asynchronously initialize the test controller. The $\overline{\text{TRST}}$ pin has an internal pullup resistor.

10.2.2 TAP Controller

The TAP controller is responsible for interpreting the sequence of logical values on the TMS signal. It is a synchronous state machine that controls the operation of the JTAG logic. The state machine is shown in **Figure 10-2**. The value shown adjacent to each arc represents the value of the TMS signal sampled on the rising edge of TCK signal. For a description of the TAP controller states, please refer to the IEEE 1149.1 document.

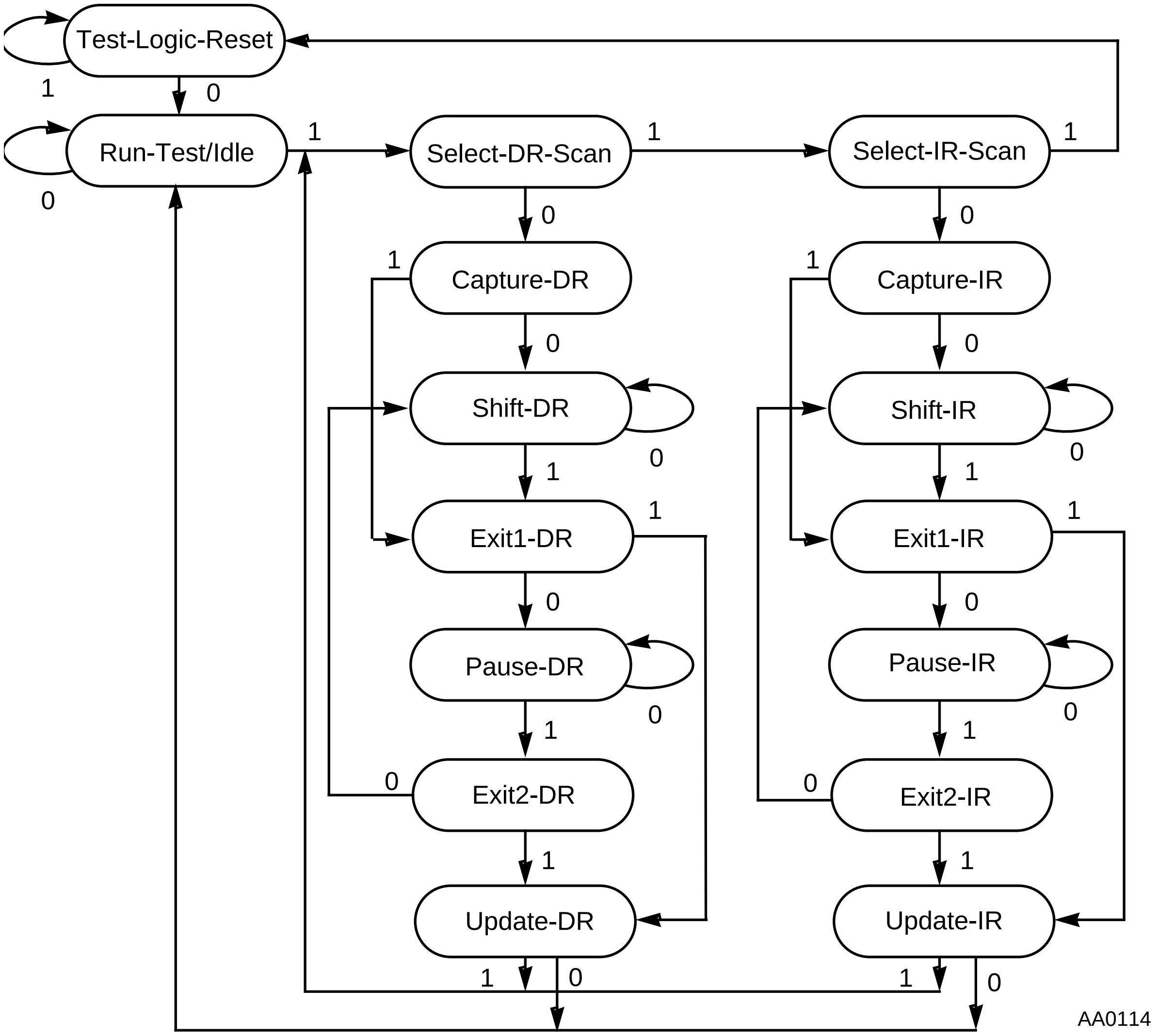


Figure 10-2 TAP Controller State Machine

10.2.3 Boundary Scan Register

The Boundary Scan Register (BSR) in the DSP56600 core JTAG implementation contains bits for all device signal and clock pins and associated control signals. All bidirectional pins have a single register bit in the BSR for pin data, and are controlled by an associated control bit in the BSR. The boundary scan bit definitions vary according to specific chip implementation. See the applicable *User's Manual* for a complete description of the BSR contents.

10.2.4 Instruction Register

The DSP56600 core JTAG implementation includes the three mandatory public instructions (EXTEST, SAMPLE/PRELOAD, and BYPASS), and also supports the optional CLAMP instruction defined by IEEE 1149.1. The HI-Z public instruction provides the capability for disabling all device output drivers. The ENABLE_ONCE public instruction enables the JTAG port to communicate with the OnCE circuitry. The DEBUG_REQUEST public instruction enables the JTAG port to force the DSP56600 core into the Debug mode of operation. The DSP56600 core includes a 4-bit instruction register without parity consisting of a shift register with four parallel outputs. Data is transferred from the shift register to the parallel outputs during the Update-IR controller state. **Figure 10-3** shows the Instruction Register configuration.

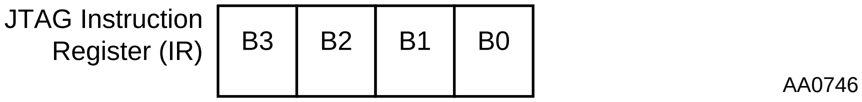


Figure 10-3 JTAG Instruction Register Format

The four bits are used to decode the eight unique instructions shown in **Table 10-1**. All other encodings are reserved for future enhancements and are decoded as BYPASS.

Table 10-1 JTAG Instructions

Code				Instruction
B3	B2	B1	B0	
0	0	0	0	EXTEST
0	0	0	1	SAMPLE/PRELOAD
0	0	1	0	IDCODE
0	0	1	1	CLAMP
0	1	0	0	HI-Z
0	1	0	1	RESERVED
0	1	1	0	ENABLE_ONCE
0	1	1	1	DEBUG_REQUEST
1	0	x	x	(Reserved)
1	1	0	x	(Reserved)
1	1	1	0	(Reserved)
1	1	1	1	BYPASS

The parallel output of the instruction register is reset to 0010 in the Test-Logic-Reset controller state, which is equivalent to the IDCODE instruction.

During the Capture-IR controller state, the parallel inputs to the instruction shift register are loaded with 01 in the Least Significant Bits (LSBs) as required by the standard. The Two Most Significant Bits (MSBs) are loaded with the values of the core status bits OS1 and OS0 from the OnCE controller. See **Core Status Bits (OS0–OS1)—Bits 6–7** on page 10-19 for a description of the status bits.

10.2.4.1 EXTEST (B[3:0] = 0000)

The external test (EXTEST) instruction selects the BSR. EXTEST also asserts internal reset for the DSP56600 core system logic to force a predictable internal state while performing external boundary scan operations.

By using the TAP, the BSR is capable of the following:

- Scanning user-defined values into the output buffers

- Capturing values presented to input pins
- Controlling the direction of bidirectional pins
- Controlling the output drive of tri-stateable output pins

For more details on the function and use of the EXTEST instruction, please refer to the IEEE 1149.1 document.

10.2.4.2 SAMPLE/PRELOAD (B[3:0] = 0001)

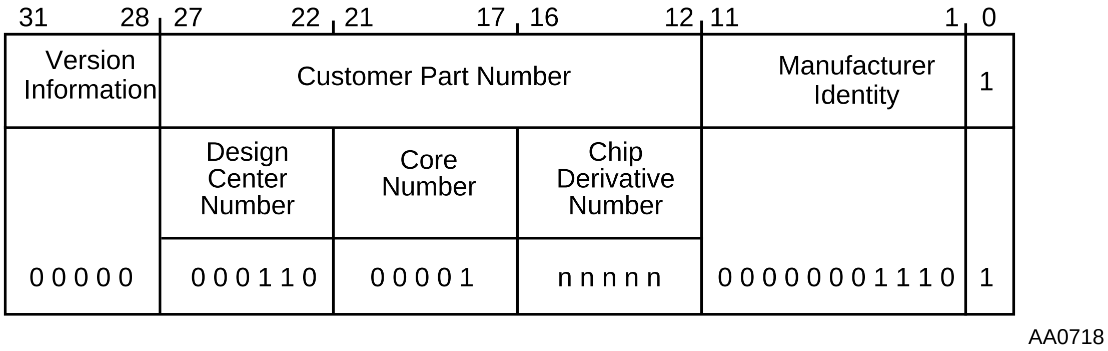
The SAMPLE/PRELOAD instruction provides two separate functions. First, it provides a means to obtain a snapshot of system data and control signals. The snapshot occurs on the rising edge of TCK in the Capture-DR controller state. The data can be observed by shifting it transparently through the BSR.

Note: Since there is no internal synchronization between the JTAG clock (TCK) and the system clock (CLK), the user must provide some form of external synchronization to achieve meaningful results.

The second function of the SAMPLE/PRELOAD instruction is to initialize the BSR output cells prior to selection of EXTEST. This initialization ensures that known data appears on the outputs when entering the EXTEST instruction.

10.2.4.3 IDCODE (B[3:0] = 0010)

The IDCODE instruction selects the ID register. This instruction is provided as a public instruction to allow the manufacturer, part number, and version of a component to be determined through the TAP. **Figure 10-4** shows the ID register configuration.



31 28	27 22	21 17	16 12	11 1	0
Version Information	Customer Part Number			Manufacturer Identity	1
	Design Center Number	Core Number	Chip Derivative Number		
00000	000110	00001	nnnnn	00000001110	1

Figure 10-4 Identification Register Configuration

One application of the ID register is to distinguish the manufacturer(s) of components on a board when multiple sourcing is used. As more components that conform to the IEEE 1149.1 standard emerge, it is desirable to allow for a system diagnostic controller unit to blindly interrogate a board design in order to determine the type of each component in each location. This information is also available for factory process monitoring and for failure mode analysis of assembled boards.

Motorola's Manufacturer Identity is 00000001110. The Customer Part Number consists of two parts: Motorola Design Center Number (bits 27:22) and a sequence number (bits 21:12). The sequence number is divided into two parts: Core Number (bits 21:17) and Chip Derivative Number (bits 16:12). Motorola Semiconductor Israel (MSIL) Design Center Number is 000110 and DSP56600 core number is 00001.

Once the IDCODE instruction is decoded, it selects the ID register , which is a 32-bit data register. Since the Bypass register loads a logic 0 at the start of a scan cycle, whereas the ID register loads a logic 1 into its LSB, examination of the first bit of data shifted out of a component during a test data scan sequence immediate following exit from Test-Logic-Reset controller state shows whether such a register is included in the design. When the IDCODE instruction is selected, the operation of the test logic has no effect on the operation of the on-chip system logic as required by the IEEE 1149.1 standard.

10.2.4.4 CLAMP (B[3:0] = 0011)

The CLAMP instruction is not included in the IEEE 1149.1 standard. It is provided as a public instruction that selects the 1-bit Bypass register as the serial path between TDI and TDO, while allowing signals driven from the component pins to be determined from the BSR. During testing of ICs on PCB, it may be necessary to place static guarding values on signals that control operation of logic not involved in the test. The EXTEST instruction could be used for this purpose, but since it selects the BSR, the required guarding signals would be loaded as part of the complete serial data stream shifted in, both at the start of the test and each time a new test pattern is entered. Since the CLAMP instruction allows guarding values to be applied using the BSR of the appropriate ICs while selecting their Bypass registers, it allows much faster testing than does the EXTEST instruction. Data in the boundary scan cell remains unchanged until a new instruction is shifted in or the JTAG state machine is set to its reset state. The CLAMP instruction also asserts internal reset for the DSP56600 core system logic to force a predictable internal state while performing external boundary scan operations.

10.2.4.5 HI-Z (B[3:0] = 0100)

The HI-Z instruction is not included in the IEEE 1149.1 standard. It is provided as a manufacturer's optional public instruction to prevent having to backdrive the output pins during circuit-board testing. When HI-Z is invoked, all output drivers, including the two-state drivers, are turned off (i.e., high impedance). The instruction selects the Bypass register. The HI-Z instruction also asserts internal reset for the DSP56600 core system logic to force a predictable internal state while performing external boundary scan operations.

10.2.4.6 ENABLE_ONCE(B[3:0] = 0110)

The ENABLE_ONCE instruction is not included in the IEEE 1149.1 standard. It is provided as a public instruction to allow the user to perform system debug functions. When the ENABLE_ONCE instruction is decoded, the TDI and TDO pins are connected directly to the OnCE registers. The particular OnCE register connected between TDI and TDO at a given time is selected by the OnCE controller depending on the OnCE instruction being currently executed. All communication with the OnCE controller is done through the Select-DR-Scan path of the JTAG TAP Controller.

10.2.4.7 DEBUG_REQUEST(B[3:0] = 0111)

The DEBUG_REQUEST instruction is not included in the IEEE 1149.1 standard. It is provided as a public instruction to allow the user to generate a debug request signal to the DSP56600 core. When the DEBUG_REQUEST instruction is decoded the TDI and TDO pins are connected to the Instruction Registers. Due to the fact that in the Capture-IR state of the TAP the OnCE status bits are captured in the Instruction shift register, the external JTAG controller must continue to shift-in the DEBUG_REQUEST instruction while polling the status bits that are shifted-out until the Debug mode of operation is entered (acknowledged by the combination 11 on OS1–OS0). After the acknowledgment of the Debug mode is received, the external JTAG controller must issue the ENABLE_ONCE instruction to allow the user to perform system debug functions.

10.2.4.8 BYPASS (B[3:0] = 1111)

The BYPASS instruction selects the single bit Bypass register, as shown in **Figure 10-5**. This creates a shift-register path from TDI to the Bypass register, and finally to TDO, circumventing the BSR. This instruction is used to enhance test efficiency when a component other than the DSP56600 core-based device becomes the device under test. When the Bypass register is selected by the current instruction, the shift-register stage is set to a logic 0 on the rising edge of TCK in the Capture-DR controller state. Therefore, the first bit shifted out after selecting the Bypass register is always a logic 0.

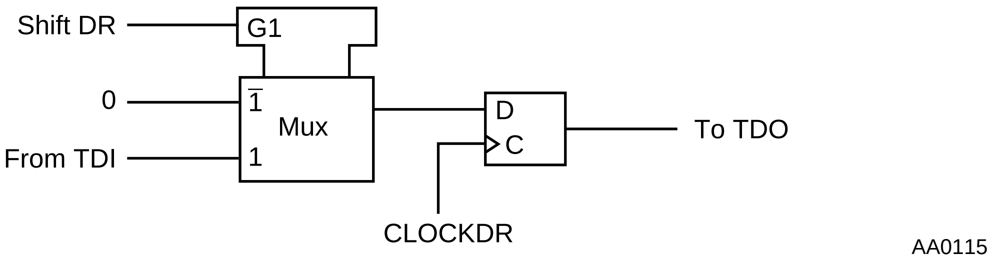


Figure 10-5 Bypass Register

10.2.5 DSP56600 Restrictions

The control afforded by the output enable signals using the BSR and the EXTEST instruction requires a compatible circuit-board test environment to avoid device-destructive configurations. The user must avoid situations in which the DSP56600 core output drivers are enabled into actively driven networks. In addition, the EXTEST instruction can be performed only after power-up or regular hardware reset while EXTAL was provided. During the execution of EXTEST, EXTAL can remain inactive.

Two constraints relate to the JTAG interface. First, the TCK input does not include an internal pullup resistor and should not be left unconnected. The second constraint is to ensure that the JTAG test logic is kept transparent to the system logic by forcing the TAP into the Test-Logic-Reset controller state, using either of two methods. During power-up, $\overline{\text{TRST}}$ must be externally asserted to force the TAP controller into this state. After power-up is concluded, TMS must be sampled as a logic 1 for five consecutive TCK rising edges. If TMS either remains unconnected or is connected to V_{CC}, then the TAP controller cannot leave the Test-Logic-Reset state, regardless of the state of TCK.

The DSP56600 core features a low-power Stop mode, which is invoked using the STOP instruction. The interaction of the JTAG interface with low-power Stop mode is as follows:

1. The TAP controller must be in the Test-Logic-Reset state to either enter or remain in the low-power Stop mode. Leaving the TAP controller Test-Logic-Reset state negates the ability to achieve low-power, but does not otherwise affect device functionality.
2. The TCK input is not blocked in low-power Stop mode. To consume minimal power, the TCK input should be externally connected to V_{CC} or GND.
3. The TMS and TDI pins include on-chip pullup resistors. In low-power Stop mode, these two pins should remain either unconnected or connected to V_{CC} to achieve minimal power consumption.

Since during Stop mode all DSP56600 core clocks are disabled, the JTAG interface provides the means of polling the device status (sampled in the Capture-IR state). For a DSP56600 derivative that does not include the $\overline{\text{DE}}$ pin, the JTAG interface provides the software means of entering the Debug mode by executing the DEBUG_REQUEST instruction.

10.3 ON-CHIP EMULATION (OnCE)

The DSP56600 core On-Chip Emulation (OnCE) module provides a means of interacting with the DSP56600 core and its peripherals non-intrusively so that a user can examine registers, memory, or on-chip peripherals, thus facilitating hardware and software development on the DSP56600 core processor. To achieve this, special circuits and dedicated pins on the DSP56600 core are defined to avoid sacrificing any user-accessible on-chip resource. The OnCE module resources can be accessed only after executing the JTAG instruction ENABLE_ONCE (these resources are accessible even when the chip is operating in Normal mode). **Figure 10-6** shows the block diagram of the OnCE module.

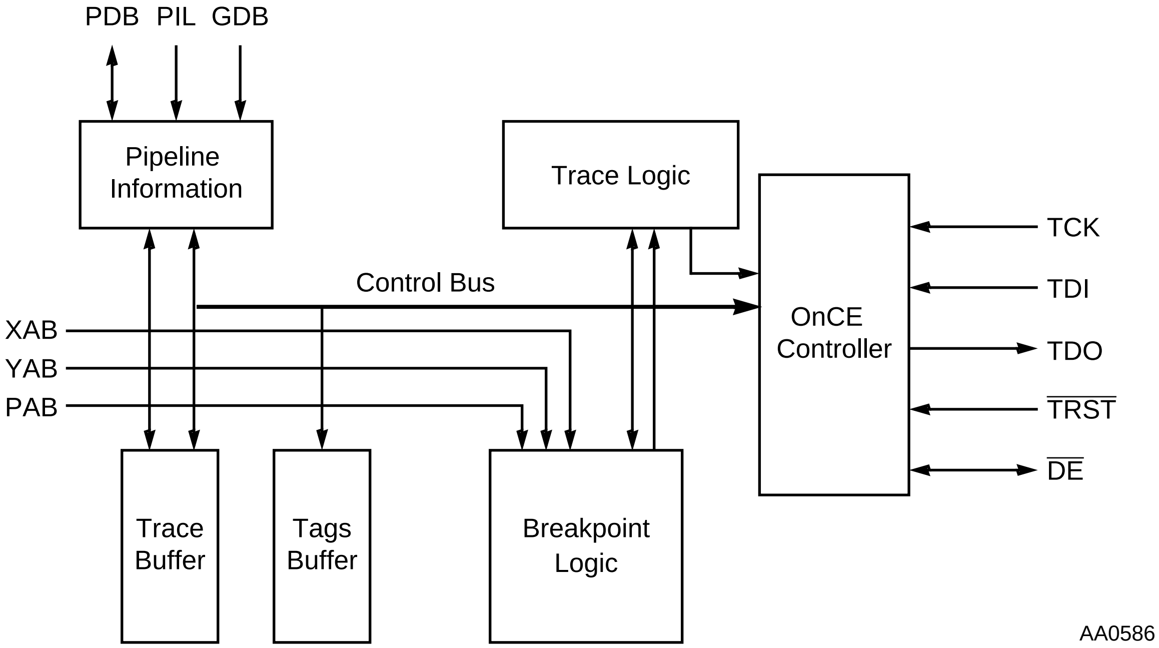


Figure 10-6 OnCE Block Diagram

The OnCE module controller functionality is accessed through the JTAG port. The JTAG pins TCK, TDI, and TDO are used to shift in and out data and instructions. See **JTAG Pins** on page 10-5 for the description of the JTAG pins. To facilitate emulation-specific functions, one additional pin, called $\overline{\text{DE}}$, is provided on certain DSP56600 family members.

The bidirectional open drain Debug Event pin ($\overline{\text{DE}}$) provides a fast means of entering the Debug mode of operation from an external command controller (when input), as well as a fast means of acknowledging the entering of the Debug mode of operation to an external command controller (when output). The assertion of this pin by a command controller causes the DSP56600 core to finish the current instruction being executed, save the instruction pipeline information, enter the Debug mode, and wait for commands to be entered from the TDI line. If the $\overline{\text{DE}}$ pin is used to enter the Debug mode, then it must be deasserted after the OnCE port responds with an acknowledge and before sending the first OnCE command. The assertion of this pin by the DSP56600 core indicates that the DSP has entered the Debug mode and is waiting for commands to be entered from the TDI line. The $\overline{\text{DE}}$ pin also facilitates multiple processor connections, as shown in **Figure 10-7**. In this way, the user can stop all the devices in the system when one of the devices enters the Debug mode. The user can also stop all the devices synchronously by asserting the $\overline{\text{DE}}$ line.

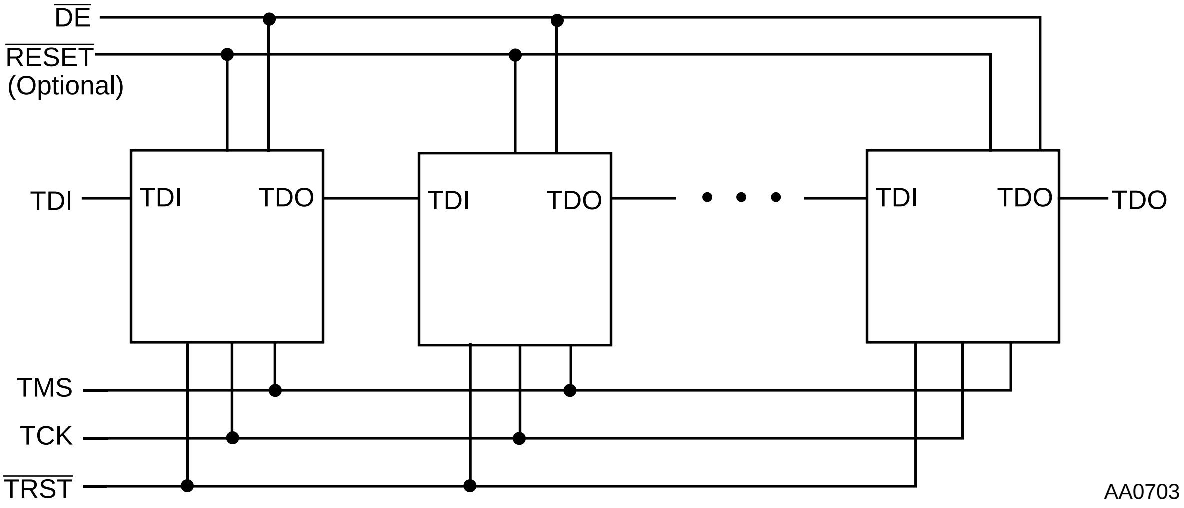


Figure 10-7 OnCE Multiprocessor Configuration

10.3.1 OnCE Controller

The OnCE Controller contains the following blocks: OnCE Command Register (OCR), OnCE Decoder, and the OnCE Status and Control Register (OSCR). **Figure 10-8** illustrates a block diagram of the OnCE controller.

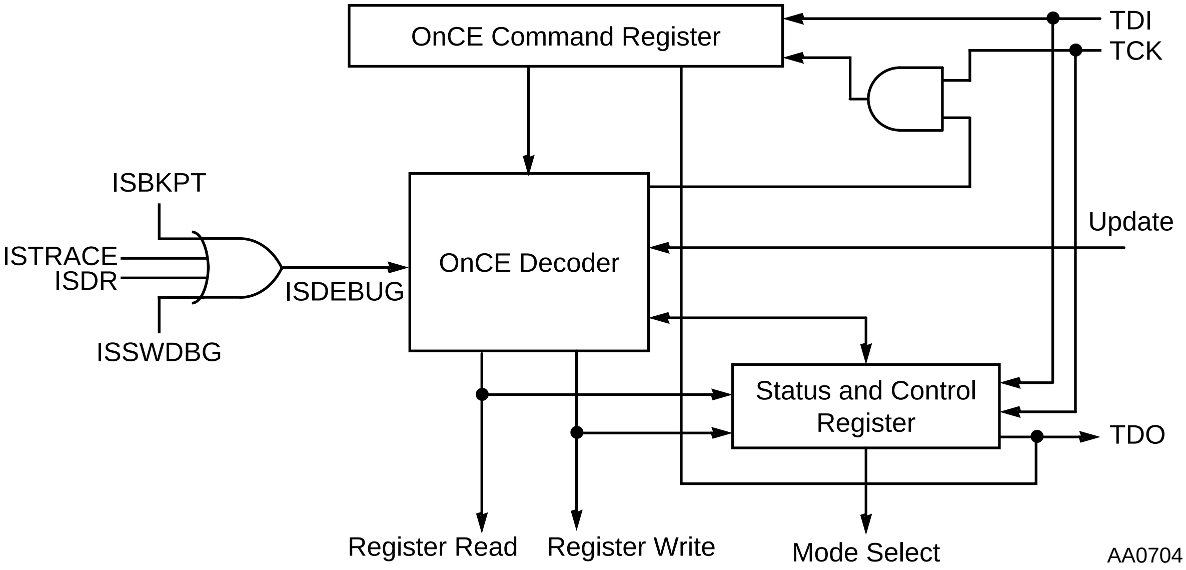


Figure 10-8 OnCE Controller

10.3.1.1 OnCE Command Register (OCR)

The OnCE Command Register (OCR) is an 8-bit shift register that receives its serial data from the TDI pin. It holds the 8-bit commands to be used as input for the OnCE Decoder. The OCR is shown in **Figure 10-9**.

OCR
OnCE Command
Register
Reset = $00
Write Only

7	6	5	4	3	2	1	0
R/W	GO	EX	RS4	RS3	RS2	RS1	RS0

AA0106

Figure 10-9 OnCE Command Register (OCR) Format

10.3.1.1.1 Register Select Bits (RS4–RS0)—Bits 0-4

The Register Select bits define which register is source/destination for the read/write operation. See **Table 10-5** for the OnCE register addresses.

10.3.1.1.2 Exit Command Bit (EX)–Bit 5

If the EX bit is set, leave Debug mode and resume normal operation. The EXIT command is executed only if the GO command is issued, and the operation is write to OPDBR or read/write to "No Register Selected". Otherwise, the EX bit is ignored. **Table 10-2** shows the definition of the EX bit.

Table 10-2 EX Bit Definition

EX	Action
0	Remain in Debug mode
1	Leave Debug mode

10.3.1.1.3 Go Command Bit (GO)—Bit 6

If the GO bit is set, execute the instruction that resides in the PIL register. To execute the instruction, the core leaves the Debug mode. The core returns to the Debug mode immediately after executing the instruction if the EX bit is cleared. The core goes on to normal operation if the EX bit is set. The GO command is executed only if the operation is write to OPDBR or read/write to "No Register Selected". Otherwise, the GO bit is ignored. **Table 10-3** shows the definition of the GO bit.

Table 10-3 GO Bit Definition

GO	Action
0	Inactive—no action taken
1	Execute instruction in PIL

10.3.1.1.4 Read/Write Command Bit ($R/\overline{W}$)—Bit 7

The $R/\overline{W}$ bit specifies the direction of data transfer.

Table 10-4 $R/\overline{W}$ Bit Definition

$R/\overline{W}$	Action
0	Write the data associated with the command into the register specified by RS4–RS0.
1	Read the data contained in the register specified by RS4–RS0.

Table 10-5 OnCE Register Select Encoding

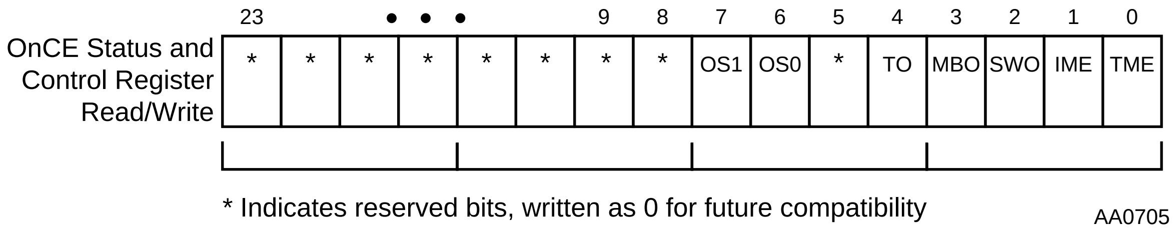
RS[4:0]	Register Selected
00000	OnCE Status and Control Register (OSCR)
00001	OnCE Memory Breakpoint Counter (OMBC)
00010	OnCE Breakpoint Control Register (OBCR)
00011	(Reserved)
00100	(Reserved)
00101	OnCE Memory Limit Register 0 (OMLR0)
00110	OnCE Memory Limit Register 1 (OMLR1)
00111	(Reserved)
01000	(Reserved)
01001	OnCE GDB Register (OGDBR)
01010	OnCE PDB Register (OPDBR)
01011	OnCE PIL Register (OPILR)
01100	PDB GO-TO Register (for GO TO command)
01101	OnCE Trace Counter (OTC)
01110	(Reserved)
01111	OnCE PAB Register for Fetch (OPABFR)
10000	OnCE PAB Register for Decode (OPABDR)
10001	OnCE PAB Register for Execute (OPABEX)
10010	Trace Buffer and Increment Pointer
10011	(Reserved)
101xx	(Reserved)
11xx0	(Reserved)
11x0x	(Reserved)
110xx	(Reserved)
11111	No Register Selected

10.3.1.2 OnCE Decoder (ODEC)

The OnCE Decoder (ODEC) supervises the entire OnCE module activity. It receives as input the 8-bit command from the OCR, a signal from JTAG Controller (indicating that 8/24 bits have been received and update of the selected data register must be performed), and a signal indicating that the core was halted. The ODEC generates all the strobes required for reading and writing the selected OnCE registers.

10.3.1.3 OnCE Status and Control Register (OSCR)

The OnCE Status and Control Register (OSCR) is a 24-bit register used to enable the Trace mode of operation and to indicate the cause of entering the Debug mode. The control bits are read/write while the status bits are read-only. The OSCR bits are cleared on hardware reset. The OSCR is shown in **Figure 10-10**.



	23		• • •				9	8	7	6	5	4	3	2	1	0
OnCE Status and Control Register Read/Write	*	*	*	*	*	*	*	*	OS1	OS0	*	TO	MBO	SWO	IME	TME

* Indicates reserved bits, written as 0 for future compatibility

Figure 10-10 OnCE Status and Control Register (OSCR) Format

10.3.1.3.1 Trace Mode Enable Bit (TME)—Bit 0

The Trace Mode Enable (TME) control bit, when set, enables the Trace mode of operation.

10.3.1.3.2 Interrupt Mode Enable Bit (IME)—Bit 1

The Interrupt Mode Enable (IME) control bit, when set, causes the chip to execute a vectored interrupt to the address VBA:$06 instead of entering the Debug mode.

10.3.1.3.3 Software Debug Occurrence Bit (SWO)—Bit 2

The Software Debug Occurrence (SWO) bit is a read-only status bit that is set when the Debug mode of operation is entered because of the execution of the DEBUG or DEBUGcc instruction with condition true. This bit is cleared when leaving the Debug mode.

10.3.1.3.4 Memory Breakpoint Occurrence Bit (MBO)—Bit 3

The Memory Breakpoint Occurrence (MBO) bit is a read-only status bit that is set when the Debug mode of operation is entered because a memory breakpoint has been encountered. This bit is cleared when leaving the Debug mode.

10.3.1.3.5 Trace Occurrence Bit (TO)—Bit 4

The Trace Occurrence (TO) bit is a read-only status bit that is set when all the following occur:

- Trace Counter = 0
- Trace mode is enabled
- Debug mode of operation is entered

This bit is cleared when leaving the Debug mode.

10.3.1.3.6 Reserved Bit 5

Bit 5 is reserved for future use. It is read as 0 and should be written with 0 for future compatibility.

10.3.1.3.7 Core Status Bits (OS0–OS1)—Bits 6–7

The Core Status (OS0,OS1) bits are read-only status bits that provide core status information. By examining the status bits, the user can determine whether the chip has entered the Debug mode. Examining SWO, MBO, and TO identifies the cause of entering the Debug mode. The user can also examine these bits and determine the cause why the chip has not entered the Debug mode after debug event assertion ($\overline{\text{DE}}$) or as a result of the execution of the JTAG Debug Request instruction (core waiting for the bus, STOP or WAIT instruction, etc.). These bits are also reflected in the JTAG instruction shift register, which allows the polling of the core status information at the JTAG level. This is useful when the DSP56600 core executes the STOP instruction (and therefore there are no clocks) to allow the reading of OSCR. See **Table 10-6** for the definition of the OS0-OS1 bits.

Table 10-6 Core Status Bits Description

OS1	OS0	DESCRIPTION
0	0	DSP56600 core is executing instructions
0	1	DSP56600 core is in Wait or STOP
1	0	DSP56600 core is waiting for bus
1	1	DSP56600 core is in Debug Mode

10.3.1.3.8 Reserved Bits 8–23

Bits 8–23 are reserved for future use. They read as 0 and should be written with 0 for future compatibility.

10.3.2 OnCE Memory Breakpoint Logic

Memory breakpoints can be set on program memory or data memory locations. In addition, the breakpoint does not have to be in a specific memory address, but within an approximate address range of where the program may be executing. This significantly increases the programmer’s ability to monitor what the program is doing in real-time.

The breakpoint logic, shown in **Figure 10-11**, contains a latch for the addresses, registers that store the upper and lower address limit, address comparators, and a breakpoint counter. Address comparators are useful in determining where a program may be getting lost or when data is being written where it should not be written. They are also useful in halting a program at a specific point to examine/change registers or memory. Using address comparators to set breakpoints enables the user to set breakpoints in RAM or ROM and while in any operating mode. Memory accesses are monitored according to the contents of the OBCR as specified in **OnCE Breakpoint Control Register (OBCR)** on page 10-21.

10.3.2.1 OnCE Memory Address Latch (OMAL)

The OnCE Memory Address Latch (OMAL) is a 16-bit register that latches the PAB, XAB or YAB on every instruction cycle according to the MBS1–MBS0 bits in OBCR.

10.3.2.2 OnCE Memory Limit Register 0 (OMLR0)

The OnCE Memory Limit Register 0 (OMLR0) is a 16-bit register that stores the memory breakpoint limit. OMLR0 can be read or written through the JTAG port. Before enabling breakpoints, OMLR0 must be loaded by the external command controller.

10.3.2.3 OnCE Memory Address Comparator 0 (OMAC0)

The OnCE Memory Address Comparator 0 (OMAC0) compares the current memory address (stored in OMAL0) with the OMLR0 contents.

10.3.2.4 OnCE Memory Limit Register 1 (OMLR1)

The OnCE Memory Limit Register 1 (OMLR1) is a 16-bit register that stores the memory breakpoint limit. OMLR1 can be read or written through JTAG port. Before enabling breakpoints, OMLR1 must be loaded by the external command controller.

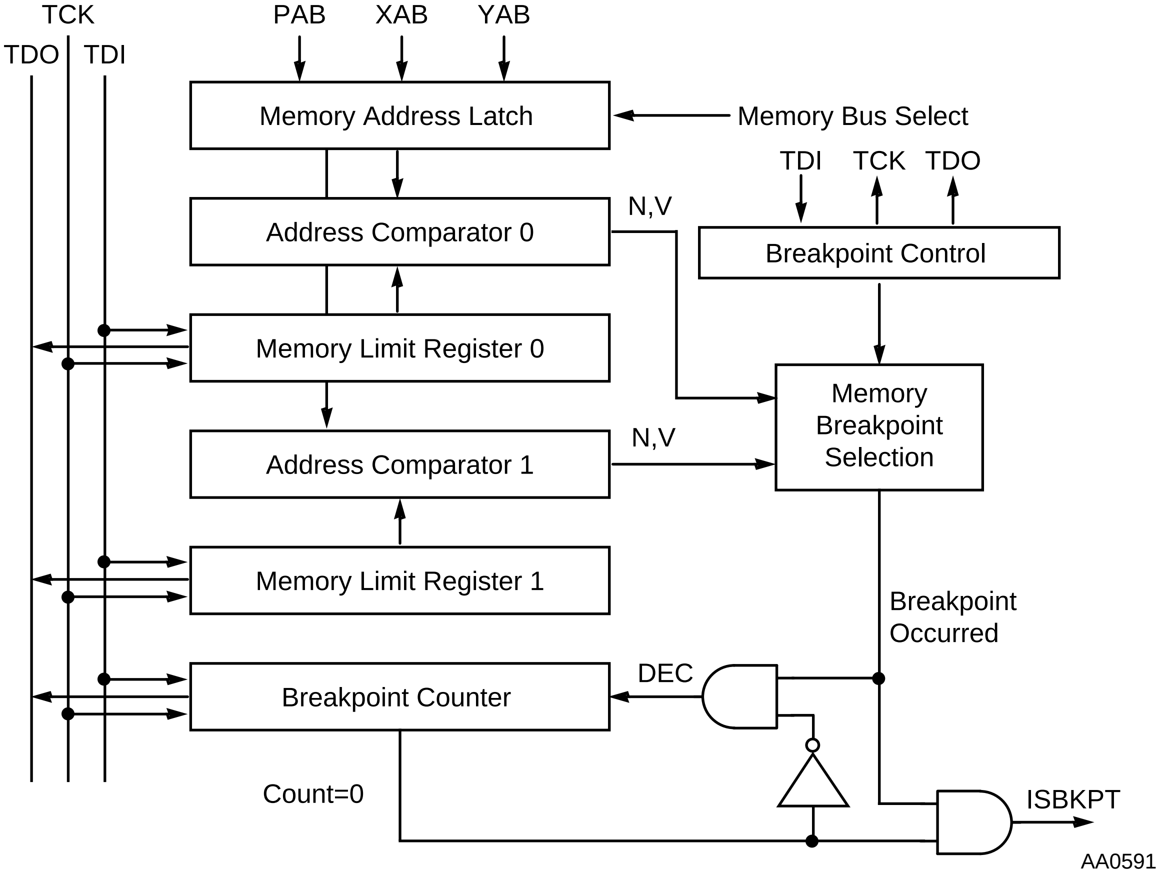


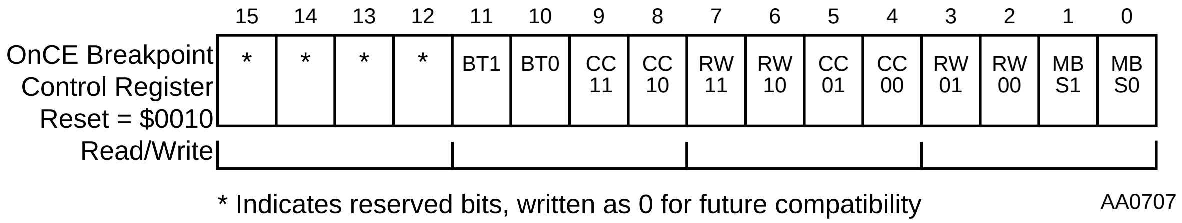
Figure 10-11 OnCE Memory Breakpoint Logic 0

10.3.2.5 OnCE Memory Address Comparator 1 (OMAC1)

The OnCE Memory Address Comparator 1 (OMAC1) compares the current memory address (stored in OMAL0) with the OMLR1 contents.

10.3.2.6 OnCE Breakpoint Control Register (OBCR)

The OnCE Breakpoint Control Register (OBCR) is a 16-bit register used to define the memory breakpoint events. OBCR can be read or written through the JTAG port. All the bits of the OBCR are cleared on hardware reset. The OBCR is described in **Figure 10-11**.



Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OnCE Breakpoint Control Register Reset = $0010	*	*	*	*	BT1	BT0	CC 11	CC 10	RW 11	RW 10	CC 01	CC 00	RW 01	RW 00	MB S1	MB S0

* Indicates reserved bits, written as 0 for future compatibility

Figure 10-12 OnCE Breakpoint Control Register (OBCR) Format

10.3.2.7 Memory Breakpoint Select Bits (MBS0–MBS1)—Bits 0–1

The Memory Breakpoint Select bits (MBS0–MBS1) enable memory breakpoints 0 and 1, allowing them to occur when a memory access is performed on P, X, or Y memory access is performed. See **Table 10-7** for the definition of the MBS0–MBS1 bits.

Table 10-7 Memory Breakpoint 0 and 1 Select Table

MBS1	MBS0	Description
0	0	Reserved
0	1	Breakpoint on P access
1	0	Breakpoint on X access
1	1	Breakpoint on Y access

10.3.2.8 Breakpoint 0 Read/Write Select Bits (RW00–RW01)—Bits 2–3

The Breakpoint 0 Read / Write Select bits (RW00–RW01) define the memory breakpoints 0 to occur when a memory address accesses is performed for read, write, or both. See **Table 10-8** for the definition of the RW00–RW01 bits.

Table 10-8 Breakpoint 0 Read / Write Select Table

RW01	RW00	Description
0	0	Breakpoint disabled
0	1	Breakpoint on write access
1	0	Breakpoint on read access
1	1	Breakpoint on read or write access

10.3.2.9 Breakpoint 0 Condition Code Select Bits (CC00–CC01)—Bits 4–5

The Breakpoint 0 Condition Code Select bits (CC00–CC01) define the condition of the comparison between the current Memory Address (OMAL0) and the Memory Limit Register 0 (OMLR0). See **Table 10-9** for the definition of the CC00–CC01 bits.

Table 10-9 Breakpoint 0 Condition Select Table

CC01	CC00	Description
0	0	Breakpoint on not equal
0	1	Breakpoint on equal
1	0	Breakpoint on less than
1	1	Breakpoint on greater than

10.3.2.10 Breakpoint1 Read/Write Select Bits (RW10–RW11)—Bits 6–7

The Breakpoint1 Read/Write Select (RW10–RW11) bits control define memory breakpoints 1 to occur when a memory address accesses is performed for read, write or both. See **Table 10-10** for the definition of the RW10–RW11 bits.

Table 10-10 Breakpoint 1 Read/Write Select Table

RW11	RW10	Description
0	0	Breakpoint disabled
0	1	Breakpoint on write access
1	0	Breakpoint on read access
1	1	Breakpoint read or write access

10.3.2.11 Breakpoint1 Condition Code Select Bits (CC10–CC11)—Bits 8–9

The Breakpoint1 Condition Code Select bits (CC10–CC11) define the condition of the comparison between the current memory address (OMAL0) and the OnCE Memory Limit Register 1 (OMLR1). See **Table 10-11** for the definition of the CC10–CC11 bits.

Table 10-11 Breakpoint 1 Condition Select Table

CC11	CC10	Description
0	0	Breakpoint on not equal
0	1	Breakpoint on equal
1	0	Breakpoint on less than
1	1	Breakpoint on greater than

10.3.2.12 Breakpoint 0 and 1 Event Select Bits (BT1–BT0)—Bits 10–11

The Breakpoint 0 and 1 Event Select bits (BT1–BT0) define the sequence between breakpoint 0 and 1. If the condition defined by BT1–BT0 is met, then the Breakpoint Counter (OMBC) is decremented. See**Table 10-12** for the definition of the BT1–BT0 bits.

Table 10-12 Breakpoint 0 and 1 Event Select Table

BT1	BT0	Description
0	0	Breakpoint 0 and Breakpoint 1
0	1	Breakpoint 0 or Breakpoint 1
1	0	Breakpoint 1 after Breakpoint 0
1	1	Breakpoint 0 after Breakpoint 1

10.3.2.13 Reserved Bits 12–15

Bits 12–15 are reserved for future use. They read as 0 and should be written with 0 for future compatibility.

10.3.3 OnCE Memory Breakpoint Counter (OMBC)

The OnCE Memory Breakpoint Counter is a 16-bit counter that is loaded with a value equal to the number of times minus one that a memory access event should occur before a memory breakpoint is declared. The memory access event is specified by the OBCR and by the memory limit registers. On each occurrence of the memory access event, the breakpoint counter is decremented. When the counter reaches 0 and a new occurrence takes place, the chip enters the Debug mode. The OMBC can be read or written through the JTAG port. Every time that the limit register is changed, or a different breakpoint event is selected in the OBCR, the breakpoint counter must be written afterwards. This ensures that the OnCE breakpoint logic is reset and that no previous events can affect the new breakpoint event selected. The breakpoint counter is cleared by hardware reset.

10.3.4 OnCE Trace Logic

Using the OnCE Trace Logic, execution of instructions in single or multiple steps is possible. The OnCE Trace Logic causes the chip to enter the Debug mode of operation after the execution of one or more instructions and wait for OnCE commands from the debug serial port. The OnCE Trace Logic block diagram is shown in **Figure 10-13**.

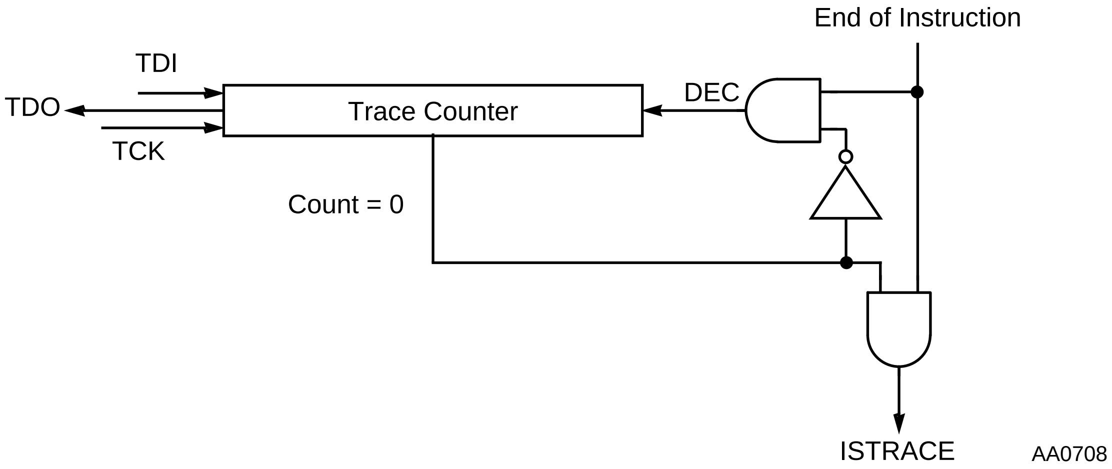


Figure 10-13 OnCE Trace Logic Block Diagram

The OnCE Trace Counter (OTC) is a 16-bit counter that can be read or written through the JTAG port. If N instructions are to be executed before entering the Debug mode, the Trace Counter should be loaded with N – 1. The Trace Counter is cleared by hardware reset.

The Trace mode has a counter associated with it so that more than one instruction can be executed before returning back to the Debug mode of operation. The objective of the counter is to allow the user to take multiple instruction steps real-time before entering the Debug mode. This feature helps the software developer debug sections of code that do not have a normal flow or are getting hung up in infinite loops. The Trace Counter also enables the user to count the number of instructions executed in a code segment.

To enable the Trace mode of operation the counter is loaded with a value, the program counter is set to the start location of the instruction(s) to be executed real-time, the TME bit is set in the OSCR and the DSP56600 core exits the Debug mode by executing the appropriate command issued by the external command controller.

Upon exiting the Debug mode, the counter is decremented after each execution of an instruction. Interrupts are serviceable and all instructions executed, including fast interrupt services and the execution of each repeated instruction, cause the Trace Counter to be decremented. Upon decrementing to 0, the DSP56600 core re-enters the Debug mode, the Trace Occurrence bit (TO) in the OSCR is set, the core Status bits OS[1:0] are set to 11, and the $\overline{\text{DE}}$ pin (if provided) is asserted to indicate that the DSP56600 core has entered Debug mode and is requesting service.

10.3.5 Methods of Entering the Debug Mode

Entering the Debug mode is acknowledged by the chip by setting the Core Status bits OS1 and OS0 and asserting the $\overline{\text{DE}}$ line. This informs the external command controller that the chip has entered the Debug mode and is waiting for commands.The DSP56600 core can disable the OnCE module if the ROM Security option is implemented. If the ROM Security is implemented, the OnCE module remains inactive until a write operation to the OGDBR is executed by the DSP56600 core.

10.3.5.1 External Debug Request During $\overline{\text{RESET}}$ Assertion

Holding the $\overline{\text{DE}}$ line asserted during the assertion of $\overline{\text{RESET}}$ causes the chip to enter the Debug mode. After receiving the acknowledge, the external command controller must negate the $\overline{\text{DE}}$ line before sending the first command.

Note: In this case, the chip does not execute any instruction before entering the Debug mode.

10.3.5.2 External Debug Request During Normal Activity

Holding the $\overline{\text{DE}}$ line asserted during normal chip activity causes the chip to finish the execution of the current instruction and then enter the Debug mode. After receiving the acknowledge, the external command controller must negate the $\overline{\text{DE}}$ line before sending the first command. This process is the same for any newly fetched instruction, including instructions fetched by the interrupt processing or instructions that will be aborted by the interrupt processing.

Note: In this case the chip completes the execution of the current instruction and stops after the newly fetched instruction enters the instruction latch.

10.3.5.3 Executing the JTAG DEBUG_REQUEST Instruction

Executing the JTAG instruction DEBUG_REQUEST asserts an internal debug request signal. Consequently, the chip finishes the execution of the current instruction and stops after the newly fetched instruction enters the instruction latch. After entering the Debug mode, the Core Status bits OS1 and OS0 are set and the $\overline{\text{DE}}$ line is asserted, thus acknowledging the external command controller that the Debug mode of operation has been entered.

10.3.5.4 External Debug Request During Stop

Executing the JTAG instruction DEBUG_REQUEST (or asserting $\overline{\text{DE}}$) while the chip is in the Stop state (i. e., has executed a STOP instruction) causes the chip to exit the Stop state and enter the Debug mode. After receiving the acknowledge, the external command controller must negate $\overline{\text{DE}}$ before sending the first command.

Note: In this case, the chip completes the execution of the STOP instruction and halts after the next instruction enters the instruction latch.

10.3.5.5 External Debug Request During Wait

Executing the JTAG instruction DEBUG_REQUEST (or asserting $\overline{\text{DE}}$) while the chip is in the Wait state (i. e., has executed a WAIT instruction) causes the chip to exit the Wait state and enter the Debug mode. After receiving the acknowledge, the external command controller must negate $\overline{\text{DE}}$ before sending the first command.

Note: In this case, the chip completes the execution of the WAIT instruction and halts after the next instruction enters the instruction latch.

10.3.5.6 Software Request During Normal Activity

Upon executing the DSP56600 core instruction DEBUG (or DEBUGcc when the specified condition is true), the chip enters the Debug mode after the instruction following the DEBUG instruction has entered the instruction latch.

10.3.5.7 Enabling Trace Mode

When the Trace mode mechanism is enabled and the Trace Counter is greater than 0, the Trace Counter is decremented after each instruction execution. Execution of an instruction when the Trace Counter = 0 causes the chip to enter the Debug mode after completing the execution of the instruction. Only instructions actually executed cause the Trace Counter to decrement. An aborted instruction does not decrement the Trace Counter and does not cause the chip to enter the Debug mode.

10.3.5.8 Enabling Memory Breakpoints

When the memory breakpoint mechanism is enabled with a Breakpoint Counter value of 0, the chip enters the Debug mode after completing the execution of the instruction that caused the memory breakpoint to occur. In case of breakpoints on executed Program memory fetches, the breakpoint is acknowledged immediately after the execution of the fetched instruction. In case of breakpoints on accesses to X, Y or P memory spaces by MOVE instructions, the breakpoint is acknowledged after the completion of the instruction following the instruction that accessed the specified address.

10.3.6 Pipeline Information and GDB Register

To restore the pipeline and to resume normal chip activity upon returning from the Debug mode, a number of on-chip registers store the chip pipeline status. **Figure 10-14** shows the block diagram of the Pipeline Information Registers with the exception of the PAB registers, which are shown in **Figure 10-15** on page 10-31.

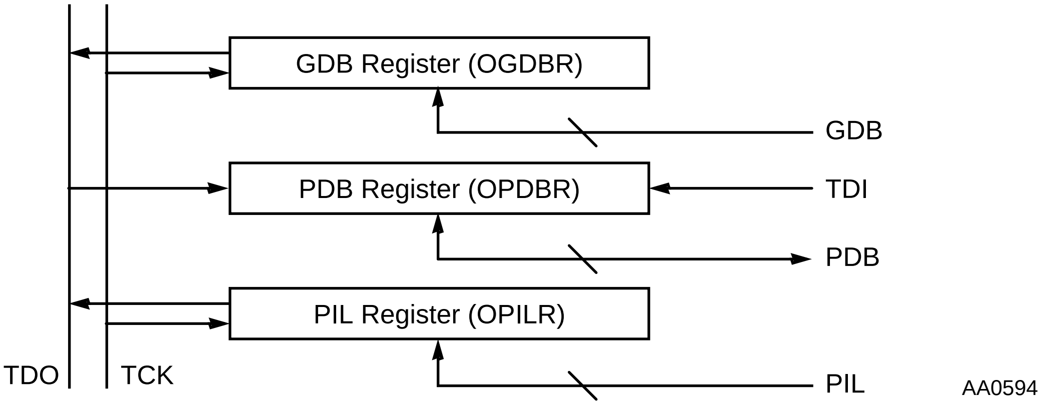


Figure 10-14 OnCE Pipeline Information and GDB Registers

10.3.6.1 OnCE PDB Register (OPDBR)

The OnCE Program Data Bus Register (OPDBR) is a 24-bit latch that stores the value of the Program Data Bus generated by the last program memory access of the core before the Debug mode is entered. The OPDBR can be read or written through the JTAG port. This register is affected by the operations performed during the Debug mode and must be restored by the external command controller when returning to Normal mode.

10.3.6.2 OnCE PIL Register (OPILR)

The OnCE PIL Register (OPILR) is a 24-bit latch that stores the value of the Instruction Latch before the Debug mode is entered. OPILR can only be read through the JTAG port.

Note: Since the Instruction Latch is affected by the operations performed during the Debug mode, it must be restored by the external command controller when returning to Normal mode. Since there is no direct write access to the Instruction Latch, the task of restoring is accomplished by writing to OPDBR with no-GO and no-EX. In this case the data written on PDB is transferred into the Instruction Latch.

10.3.6.3 OnCE GDB Register (OGDBR)

The OnCE GDB Register (OGDBR) is a 16-bit latch that can only be read through the JTAG port. The OGDBR is not actually required from a pipeline status restore point of view, but is required as a means of passing information between the chip and the external command controller. The OGDBR is mapped on the X internal I/O space at address $FFFC. Whenever the external command controller needs the contents of a register or memory location, it forces the chip to execute an instruction that brings that information to the OGDBR. Then the contents of the OGDBR are delivered serially to the external command controller by the command “READ GDB REGISTER”.

10.3.7 Trace Buffer

To ease debugging activity and keep track of program flow, the DSP56600 core provides a number of on-chip dedicated resources. There are three read-only PAB registers that give pipeline information when the Debug mode is entered, and a Trace Buffer that stores the address of the last instruction that was executed, as well as the addresses of the last eight change of flow instructions.

10.3.7.1 OnCE PAB Register for Fetch (OPABFR)

The OnCE PAB Register for Fetch Register (OPABFR) is a 16-bit register that stores the address of the last instruction whose fetch was started before the Debug mode was entered.The OPABFR can only be read through the JTAG port. This register is not affected by the operations performed during the Debug mode.

10.3.7.2 PAB Register for Decode (OPABDR)

The PAB Register for Decode Register (OPABDR) is a 16-bit register that stores the address of the instruction currently on the PDB. This is the instruction whose fetch was completed before the chip has entered the Debug mode. The OPABDR can only be read through the JTAG port. This register is not affected by the operations performed during the Debug mode.

10.3.7.3 PAB Register for Execute (OPABEX)

The PAB Register for Execute (OPABEX) register is a 16-bit register that stores the address of the instruction currently in the Instruction Latch. This is the instruction that would have been decoded and executed if the chip would not have entered the Debug mode. The OPABEX register can only be read through the JTAG port. This register is not affected by the operations performed during the Debug mode.

10.3.7.4 Trace Buffer

The Trace Buffer stores the addresses of the last eight change of flow instructions that were executed, as well as the address of the last executed instruction. The Trace Buffer is implemented as a circular buffer containing eight 17-bit registers and one 4-bit counter. All the registers have the same address, but any read access to the Trace Buffer address causes the counter to increment, thus pointing to the next Trace Buffer register. The registers are serially available to the external command controller through their common Trace Buffer address. **Figure 10-15** on page 10-31 shows the block diagram of the Trace Buffer. The Trace Buffer is not affected by the operations performed during the Debug mode except for the Trace Buffer pointer increment when reading the Trace Buffer. When entering the Debug mode, the Trace Buffer counter is pointing to the Trace Buffer register containing the address of the last executed instructions. The first Trace Buffer read obtains the oldest address and the following Trace Buffer reads get the other addresses from the oldest to the newest, in order of execution.

Notes:

1. To ensure Trace Buffer coherence, a complete set of eight reads of the Trace Buffer must be performed. This is necessary due to the fact that each read increments the Trace Buffer pointer, thus pointing to the next location. After eight reads, the pointer indicates the same location as before starting the read procedure.

2. On any change of flow instruction, the Trace Buffer stores both the address of the change of flow instruction, as well as the address of the target of the change of flow instruction. In the case of conditional change of flows, the address of the change of flow instruction is always stored (regardless of the fact that the change of flow is true or false), but if the conditional change of flow is false (i.e., not taken) the address of the target is not stored. In order to facilitate the program trace reconstruction every Trace Buffer location has an additional “invalid bit” (the 25th bit). If a conditional change of flow instruction has a “condition false”, the “invalid bit” is set, thus marking this instruction as “not taken”. Therefore, it is imperative to read seventeen bits of data

when reading the eight Trace Buffer registers. Since data is read LSB first, the "invalid bit" is the first bit to be read.

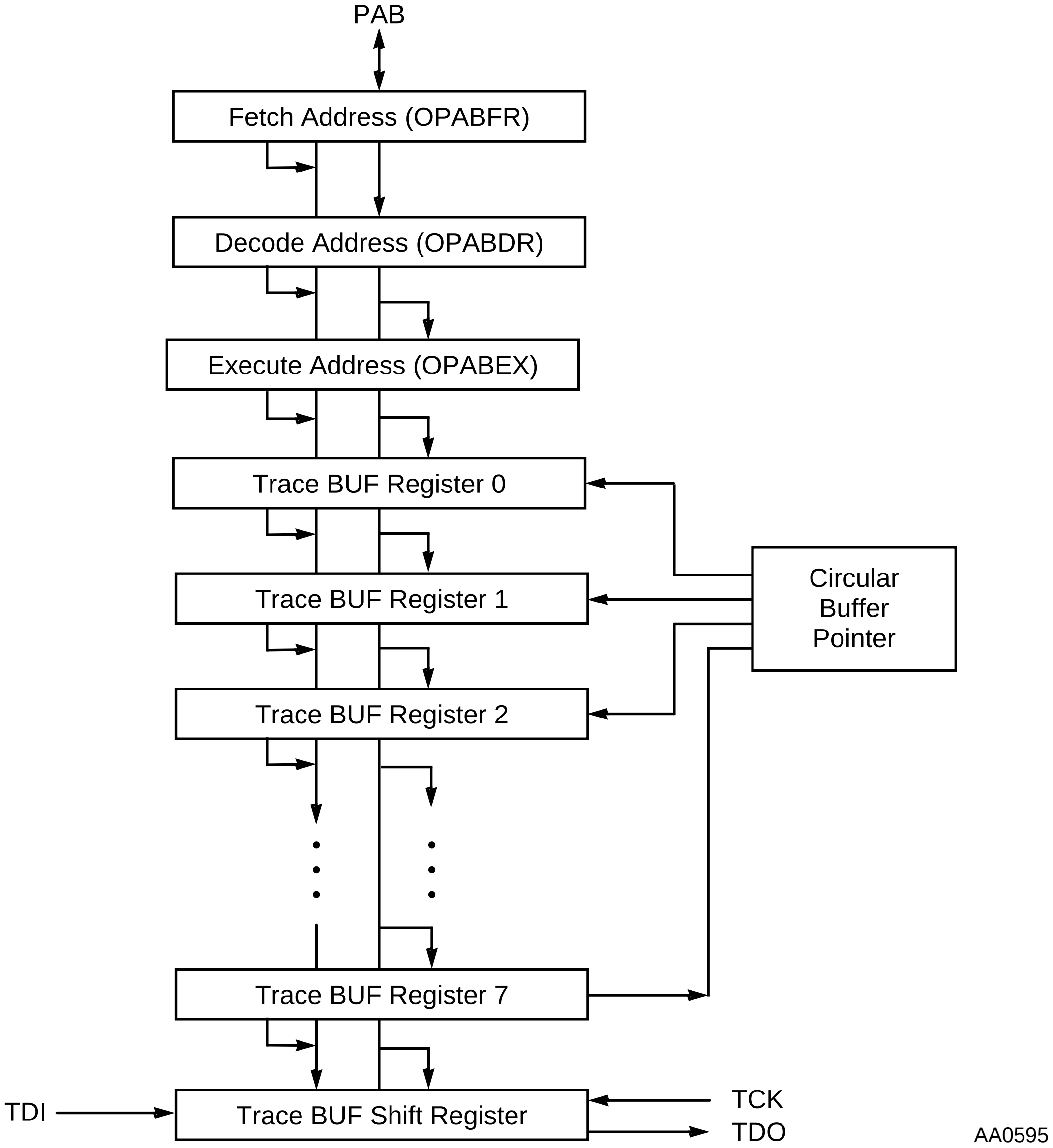


Figure 10-15 OnCE Trace Buffer Block Diagram

10.3.8 OnCE Commands and Serial Protocol

To permit an efficient means of communication between the external command controller and the DSP56600 core chip, the following protocol is adopted. Before starting any debugging activity, the external command controller has to wait for an acknowledge on the $\overline{\text{DE}}$ line indicating that the chip has entered the Debug mode (optionally the external command controller can poll the OS1 and OS0 bits in the JTAG instruction shift register). The external command controller communicates with the chip by sending 8-bit commands that can be accompanied by 24 bits of data. Both commands and data are sent or received Least Significant Bit first. After sending a command, the external command controller should wait for the DSP56600 core chip to acknowledge execution of the command. The external command controller can send a new command only after the chip has acknowledged execution of the previous command.

The OnCE commands are classified as follows:

- Read commands (when the chip delivers the required data)
- Write commands (when the chip receives data and writes the data in one of the OnCE registers)
- Commands that do not have data transfers associated with them

The commands are eight bits long and have the format shown in **Figure 10-9** on page 10-15.

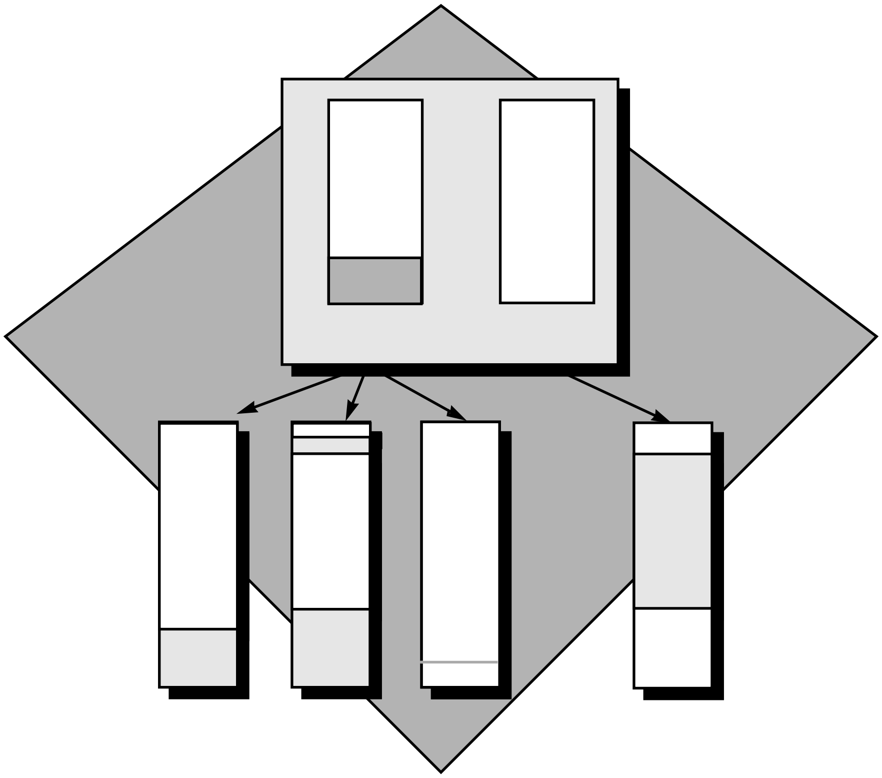
10.3.9 Target Site Debug System Requirements

A typical debug environment consists of a target system where the DSP56600 core-based device resides in the user defined hardware. The JTAG port interfaces to the external command controller through a 14-pin connector that provides connections for the five JTAG port lines, one OnCE module control line, a ground, a $\overline{\text{RESET}}$ line, and .a target power input line. The $\overline{\text{RESET}}$ line is optional and is only used to reset the DSP56600 core-based device and its associated circuitry.

The external command controller acts as the medium between the DSP56600 core target system and a host computer. The external command controller circuit acts as a JTAG port driver and host computer command interpreter. The controller issues commands based on the host computer inputs from a user interface program that communicates with the user.

SECTION 11

OPERATING MODES AND MEMORY SPACES



11.1 INTRODUCTION

This section describes the operating modes and memory spaces in the DSP56600 family.

11.2 CHIP OPERATING MODES

The DSP56600 core mode pins (MODA, MODB, MODC, and MODD) determine the reset vector address that should point to the start-up procedure when the chip leaves the Reset state. The MODA, MODB, MODC, and MODD pins are sampled as the chip leaves the Reset state. The sampled state of these pins is subject to a mask-programmed look-up table that may be used as a filter to disable the user from entering some of the operating modes. This filtered state is written to the MD, MC, MB, and MA bits in the Operating Mode Register (OMR). When the Reset state is exited, the MODA, MODB, MODC, and MODD pins become general purpose interrupt pins, $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, $\overline{\text{IRQC}}$, and $\overline{\text{IRQD}}$, respectively. When not in the Reset state, the OMR mode bits (MA, MB, MC, and MD) can be changed by software.

Table 11-1 lists the mode assignments in the DSP56600 core. The reset vector is chosen from three mask programmed addresses: RESET1, RESET2, and RESET3. Each reset vector is mask programmed to one of two different values, according to **Table 11-2**. These reset vectors are implementation-specific.

Table 11-1 DSP Core Operating Modes

MOD[D:A]	Operating Mode	Description	Reset Vector
0000	0	Expanded Mode	RESET1
0001–0111	1–7	System Configuration Mode 1–7	RESET3
1000	8	Expanded Mode 8	RESET2
1001–1111	9–15	System Configuration Mode 9–15	RESET3

Table 11-2 DSP Core Reset Vectors

RESET1 possible values	RESET2 possible values	RESET3 possible values
$C000	$4000	$0800
$8000	$0000	$0400

11.2.1 Expanded Modes (Modes 0 and 8)

In the Expanded Modes 0 and 8, a hardware reset causes the DSP56600 core to jump to the mask-programmed external program memory location RESET1 or RESET2, respectively, and execute the code fetched from this location. These locations are implementation-specific. See the appropriate *User's Manual* for more information.

11.2.2 System Configuration Modes 1–15 (Mode 1–7 and 9–15)

In the System Configuration Modes 1–7 and 9–15, a hardware reset causes the DSP56600 core to jump to the mask-programmed internal program memory location RESET3, and execute the code fetched from this location. These routines are typically implementation-specific, and can be contained in the bootstrap code.

11.3 DSP56600 CORE MEMORY MAP

The memory space of the DSP56600 core is partitioned into program memory space (P), X data memory space, and Y data memory space. The data memory space is divided into X data memory and to Y data memory in order to work with the two Address Arithmetic Logic Units (Address ALUs) and to feed two operands simultaneously to the Data ALU. Each memory space may include internal RAM, internal ROM and can be expanded off-chip under software control. The three independent memory spaces of the DSP56600 core: X data, Y data, and program, are shown in **Figure 11-1**.

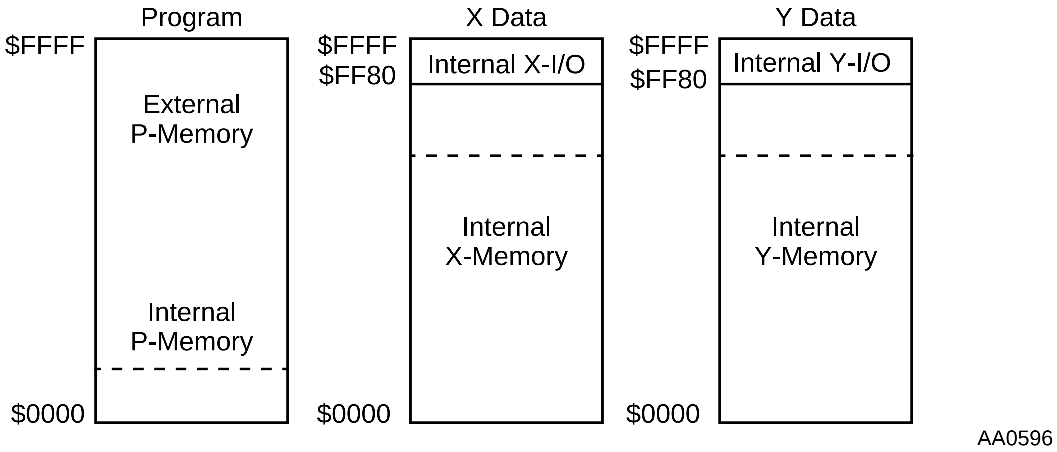


Figure 11-1 DSP Core Memory Map

Note: Individual members of the DSP 56600 family can have different amounts of X data, Y data, and program memory. Consult the appropriate *User's Manual* for more information.

11.3.1 X Data Memory Space

The X data memory space is divided into two parts:

- Internal X I/O space
- Internal X memory

11.3.1.1 Internal X I/O Space

The on-chip X I/O peripheral registers occupy the top 128 locations of the X data memory space ($FF80–$FFFF) and can be accessed by the MOVE and MOVEP instructions, as well as by bit-oriented instructions, such as the BCHG, BCLR, BSET, BTST, BRCLR, BRSET, BSCLR, BSSET, JCLR, JSET, JSCLR, and JSSET instructions. Some of the DSP56600 core registers are mapped onto the internal X I/O space as well, as listed in **Table 11-3**.

11.3.1.2 Internal X Memory

The X memory space located at locations $0000–$FF7F is used for internal X RAM or X ROM modules. The RAM modules are 256 locations each, and the ROM modules are 2048 locations each.

Table 11-3 Internal X I/O Memory Map

Register	Block	Address	Register Name and Description
IPRC	PIC	$FFFF	Interrupt Priority Register—Core
IPRP		$FFFE	Interrupt Priority Register Peripheral
PCTL0	PLL	$FFFD	PLL Control Register 0
PCTL1		$FFFC	PLL Control Register 1
OGDB	OnCE	$FFFB	ONCE GDB Register
BCR	Port A	$FFFA	Bus Control Register
IDR		$FFF9	ID Register
PAR0	PPL	$FFF8	Patch 0 Register
PAR1		$FFF7	Patch 1 Register
PAR2		$FFF6	Patch 2 Register
PAR3		$FFF5	Patch 3 Register
BPMRG	BPMR	$FFF4	BPMRG (24 bits)
BPMRL		$FFF3	BPMRL (16 bits)
BPMRH		$FFF2	BPMRH (16 bits)
Reserved	On-Chip X-I/O mapped Registers	$FFF1	Reserved for on-chip X I/O mapped register
		. . .	Reserved for on-chip X I/O mapped register
		. . .	Reserved for on-chip X I/O mapped register
		. . .	Reserved for on-chip X I/O mapped register
		$FF80	Reserved for on-chip X I/O mapped register

11.3.2 Y Data Memory Space

The Y data memory space is divided into two parts:

- Internal Y I/O space
- Internal Y memory

11.3.2.1 Internal Y I/O Space

The on-chip peripheral registers (Y I/O) occupy the top 128 locations of the Y data memory space ($FF80–$FFFF) and can be accessed by MOVE, MOVEP instructions and by bit-oriented instructions (BCHG, BCLR, BSET, BTST, BRCLR, BRSET, BSCLR, BSSET, JCLR, JSET, JSCLR and JSSET).

11.3.2.2 Internal Y Memory

The Y memory space located at locations $0000–$FF7F is used for internal Y RAM or Y ROM modules. The RAM modules are 256 locations each and the ROM modules are 2048 locations each

11.3.3 Program Memory

The Program memory space is divided into two parts:

- External program memory
- Internal program memory

11.3.3.1 External Program Memory

The program memory space located at locations $0000–$FFFF is used for expanding to external program memory. The starting address of the external program memory space is mask programmed and is dependent on the amount of on-chip program memory in the chip.

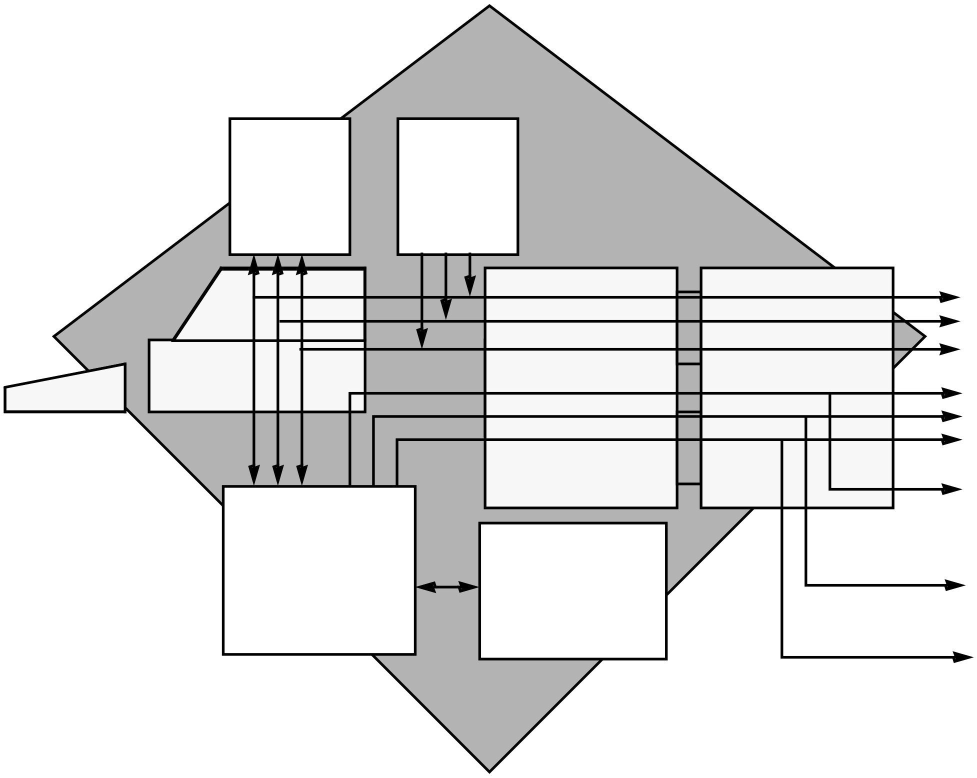
11.3.3.2 Internal Program Memory

The program memory space located at locations $0000–$FFFF is used for internal Program RAM and ROM modules, 256 locations for each RAM module and 2048 locations for each ROM module. The last address of the internal program memory is mask-programmed. The Program RAM provides a method of changing the program dynamically, allowing efficient overlaying of DSP software algorithms. The boundary between the internal and external memories is with a 256-word resolution.

DSP56600 Core Memory Map

SECTION 12

DEVELOPMENT TOOLS



12.1 INTRODUCTION

Motorola offers a full line of software and hardware Digital Signal Processor (DSP) development tools to speed development and debugging of user applications and algorithms for the DSP56600 family. The development tools include the following:

- DSP56600 Assembler
- DSP56600 Linker/Librarian
- DSP56600 Simulator
- DSP56600 C Compiler with symbolic debugger
- DSP56600 Application Development System (ADS), including hardware and interface software
- DSP56600 Graphical User Interface (GUI)

These tools can be ordered to operate on: ISA-BUS IBM PCs™, SBUS™ SUN-4 Workstations™, or Hewlett-Packard (HP) Series 7xx computers. Motorola's DSP development tools can be obtained through a local Motorola Semiconductor Sales Office or authorized distributor.

The DSP56600 Graphical User Interface (GUI) is included with the DSP56600 development tools. The GUI provides a multi-window graphical interface for the instruction simulator and application development system, giving the user source-level debug capability in assembly language and C language programs.

The DSP56600 Graphical User Interface provides display windows for the following:

- Hardware development and instruction simulator command and session windows
- Source files
- Disassembled portions of memory
- Selected set of DSP core registers
- Selected set of on-chip peripheral registers
- Selected portion of memory
- Watch window
- C language call stack
- Breakpoint window

All of the commands are accessible through the GUI’s pull-down menus, dialog boxes, tool bars, windows, and buttons. Using these tools the user selects a desired operation, such as setting a breakpoint in memory or displaying sections of memory. The interface then generates the corresponding command for the appropriate DSP56600 development tool. These commands are passed to the debugger via the GUI’s command window, and the output and other information is then displayed in the session window and other appropriate windows. The user may also enter commands directly into the command window, retaining direct control over the debugging session, if desired. There is an expression calculator and many other unique features built into the GUI.

Figure 12-1 shows the placement of the development tools in the flow of development of a user application.

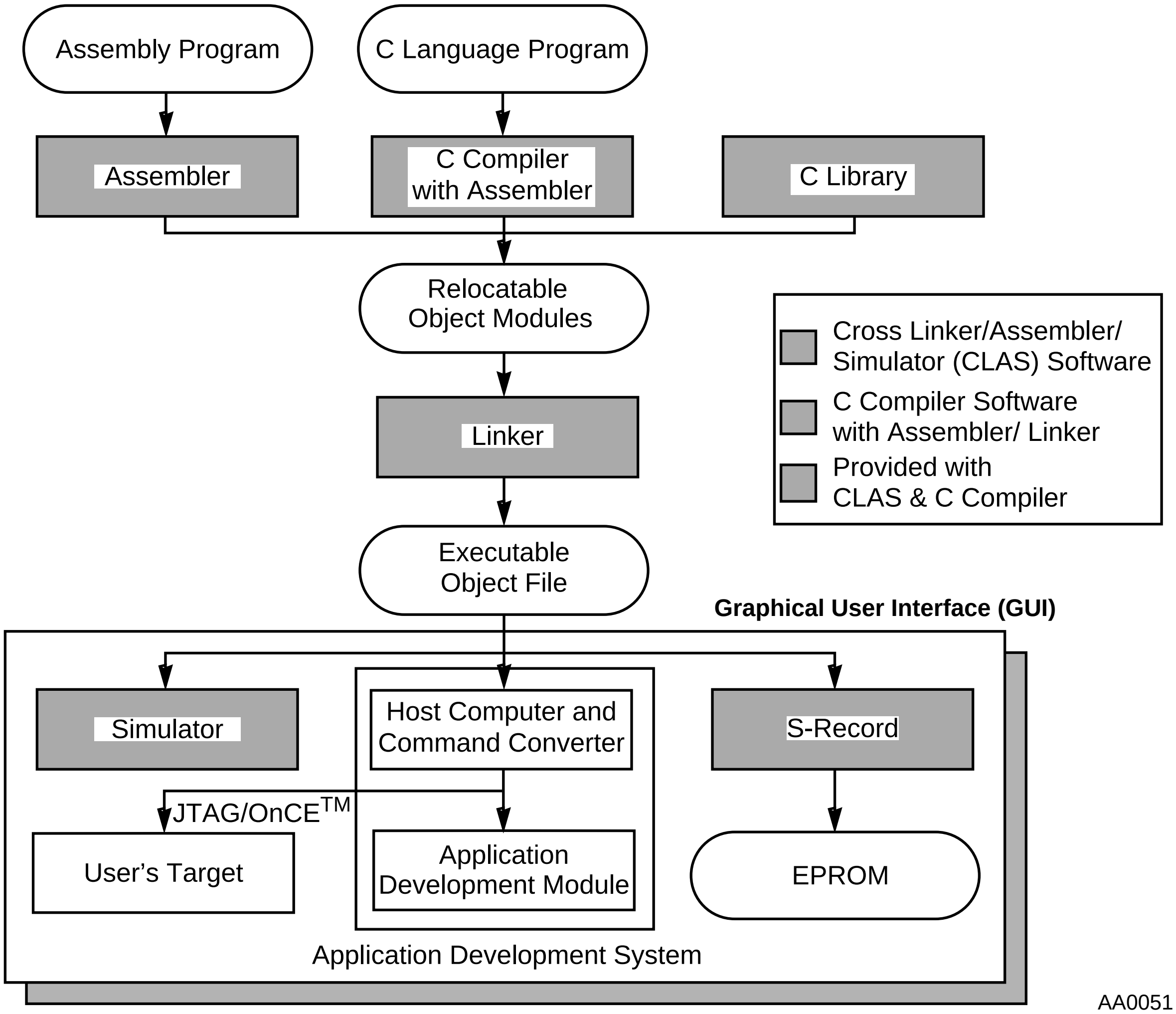

Assembly Program
C Language Program
Assembler
C Compiler with Assembler
C Library
Relocatable Object Modules
Linker
Executable Object File
Cross Linker/Assembler/ Simulator (CLAS) Software
C Compiler Software with Assembler/ Linker
Provided with CLAS & C Compiler
Graphical User Interface (GUI)
Simulator
Host Computer and Command Converter
S-Record
JTAG/OnCE™
User’s Target
Application Development Module
EPROM
Application Development System
AA0051

Figure 12-1 Development Flow

12.2 SOFTWARE DEVELOPMENT ENVIRONMENT

The software available from Motorola, the DSP56600CLASx software package, is written in the C language and consists of a relocatable macro cross assembler, linker, librarian, clock-by-clock multi-DSP-chip instruction simulator, and Graphical User Interface. These features are marketed as an integrated product. All software products run on IBM PCs, Hewlett-Packard Series 7xx computers, and SUN–4 Workstations.

The CLAS software package is designed to provide the following benefits for the programmer:

- Modular programming environment
- Full use of the DSP chip features
- A variety of data storage definitions
- Relocatability of generated code
- Symbolic debugging
- Flexible linkages of object files

A library facility is included for creating archives of the final applications code.

12.2.1 Macro Cross Assembler

The DSP56600 Assembler (ASM56600) is a full-featured macro cross assembler that translates one or more source files containing DSP instruction mnemonics, operands, and assembler directives into relocatable object modules that are relocated and linked by the Motorola DSP Linker in the Relocation mode. In the Absolute mode, the Assembler generates absolute executable files. The Assembler recognizes the full instruction set and all addressing modes of the DSP56600 family.

The features of the DSP56600 Assembler include the following:

- Produces relocatable object modules compatible with the DSP Linker program in the relocation mode.
- Produces absolute executable files compatible with the Simulator program in the absolute mode.
- Supports full instruction set, memory spaces, and parallel data transfer fields of the DSP.

- Provides modular programming features including local labels, sections, and external definition/reference directives.
- Provides nested macro libraries.
- Allows complex expression evaluation, including boolean operators.
- Gives built-in functions for data conversion, string comparison, and common transcendental math operations.
- Allows directives to define circular and bit-reversed buffers.
- Provides extensive error checking and reporting.

The unique architecture and parallel operation of the DSP demands advanced capabilities and programming aids that this Assembler readily provides. These include built-in functions for common transcendental math computations, such as sine, cosine, log, and square root functions; arbitrary expressions and modulo operations; and directives to define circular and bit-reversed data buffers. Moreover, the Assembler incorporates extensive error checking and reporting to indicate programming violations peculiar to the digital signal processing environment or stemming from the advanced features of the DSP, for instance, errors for improper nesting of hardware DO loops and improper address boundaries for circular data buffers and bit-reversed buffers.

The Assembler generates source code listings that include numbered source lines, optional titles and subtitles, optional instruction cycle counts, symbol table and cross-reference listings, and memory use reports.

12.2.2 Linker/Librarian

The DSP56600 Linker relocates and links relocatable object modules from the macro cross assembler to create an absolute executable file that can be loaded directly into the DSP56600 simulator or converted to Motorola S-record format for Programmable ROM burning.

The DSP56600 Librarian utility can merge multiple separate relocatable object modules into a single file, eliminating the need for reassembling known bug-free routines every time the mainline program is assembled.

12.2.3 Clock-by-Clock Instruction Simulator

The DSP56600 Simulator (SIM56600) is a software tool for developing programs and algorithms for the DSP. This program exactly emulates all of the functions (except for the JTAG/OnCE) of the DSP, including the DSP core, all memory and register updates associated with program code execution, the entire internal and external memory space of the DSP, all on-chip peripheral operations, and all exception processing activity. This enables the Simulator program to count clock and instruction cycles, providing an accurate measurement of code execution time that is so critical in digital signal processing applications.

The multi-DSP-chip Simulator has the same look, feel, and functions as the interface software provided with the ADS, making movement between the DSP chip's simulation and hardware environments easy.

The Simulator program executes DSP object code generated by the Linker or the Simulator's internal single-line Assembler. The object code is first loaded into the simulated DSP memory map. Then, instruction execution can proceed in single-step mode (stopping after each instruction has been executed) or until a user-defined breakpoint is encountered. During program debug, the registers or memory locations may be displayed or changed.

The Simulator package includes linkable object code libraries of simulator functions that were used to create the Simulator. The libraries allow a customized Simulator to be built and integrated with unique system simulations. Source code for some of the functions, such as the terminal I/O functions and external memory accesses, is provided to allow close simulation of the particular application.

The features of the DSP56600 Simulator include the following:

- Multiple DSP device simulation
- Source-level symbolic debug of assembly and C source programs
- Conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Instruction and cycle timing counters
- Session and/or command logging for later reference
- Input/output ASCII files for device peripherals
- Help file and help line display of simulator commands
- Macro command definition and execution

- Display enable/disable of registers and memory locations
- Hexadecimal/decimal/binary calculator

12.2.4 C Cross Compiler

The DSP56600 C Cross Compiler software package contains an optimizing C cross-compiler and symbolic debugger and can be used with Motorola's instruction Simulator and Application Development System (ADS), assembler/linker/librarian, ANSI C libraries, and COFF and Motorola S-Record utilities.

The DSP56600 C Compiler features include the following:

- Supports ISO/ANSI C, Strict ISO/ANSI C, and K&R C (pcc)
- Supports fractional data types in addition to standard C data types
- Supports pragmas for improving the compiler's processing performance
- Supports intrinsics for modulo buffer support and other useful functions
- Supports assembly code within the C code
- Provides ISO C run-time library

The C Compiler software package is supported on IBM PC and SUN-4 Workstations. (The C Compiler is not available for Hewlett-Packard computers.)

12.3 HARDWARE DEVELOPMENT ENVIRONMENT

Motorola DSP is continually developing evaluation and design tools to assist customers in evaluating new DSP products and adapting them to specific design requirements. The ADS undergoes constant evaluation and development to provide the highest level of customer support.

The basic level of customer support is the evaluation module. Customer evaluation kits containing these modules are provided to evaluate specific DSP functionality. Each kit includes the following:

- An evaluation board
- Related DSP documentation such as data sheets, user's manuals, and family manuals

- A user’s manual for the evaluation board
- A CD containing supporting software
- Documentation for recommended Motorola and third party devices that can interface with the evaluation board

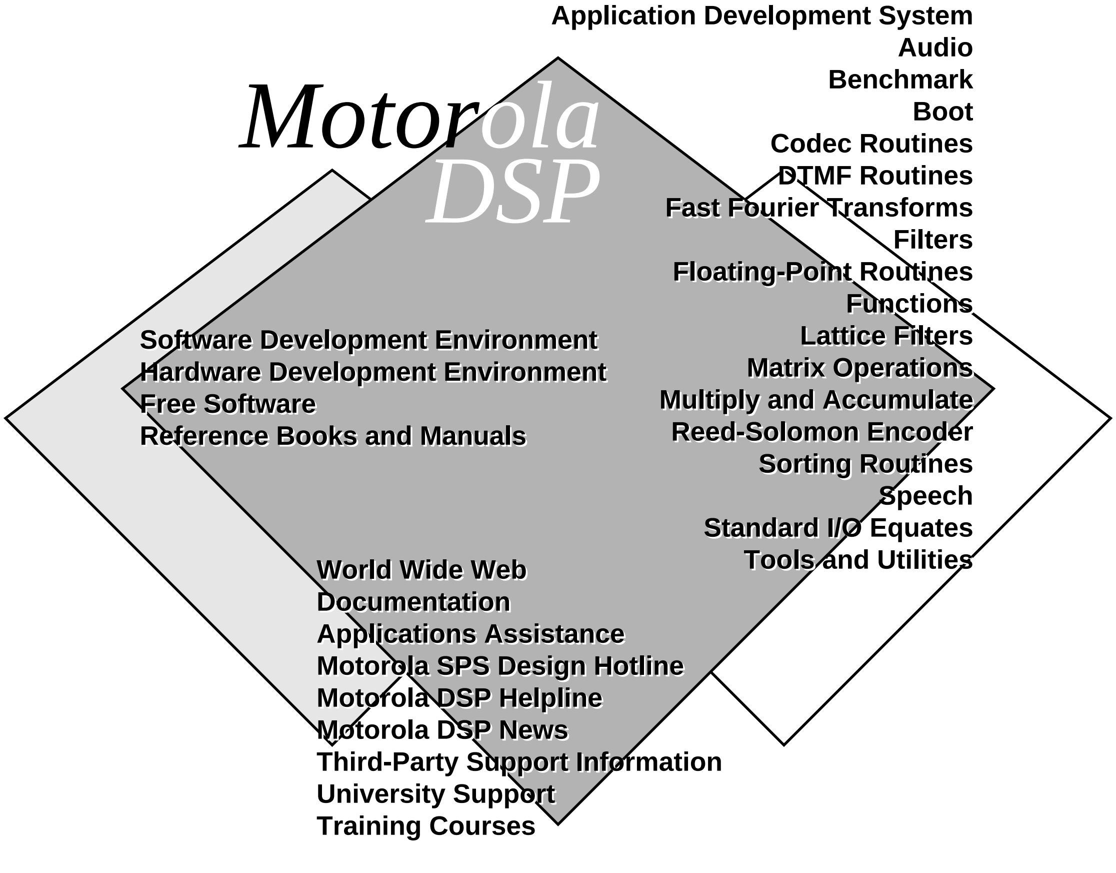
Motorola provides an upgrade path beyond chip evaluation for developing complex applications that integrate the DSP functionality. This can include the following:

- Interface hardware and software
- A command converter module
- An application development module that works with the evaluation board

For more information on Motorola’s hardware development environment, consult the Motorola DSP Home Page on the World Wide Web (see **Section 13, Additional Support**).

SECTION 13

ADDITIONAL SUPPORT



Application Development System
Audio
Benchmark
Boot
Codec Routines
DTMF Routines
Fast Fourier Transforms
Filters
Floating-Point Routines
Functions
Lattice Filters
Matrix Operations
Multiply and Accumulate
Reed-Solomon Encoder
Sorting Routines
Speech
Standard I/O Equates
Tools and Utilities

Software Development Environment
Hardware Development Environment
Free Software
Reference Books and Manuals

World Wide Web
Documentation
Applications Assistance
Motorola SPS Design Hotline
Motorola DSP Helpline
Motorola DSP News
Third-Party Support Information
University Support
Training Courses

13.1 OVERVIEW

User support from the conception of a design through completion is available from Motorola and third-party companies as shown in the following table:

Table 13-1 User Support Available

	Motorola	Third Party
Function	**Support Description**	**Support Description**
Design	• World Wide Web • Documentation • Applications Assistance • Training • Free Software	• Data Acquisition Packages • Filter Design Packages • Operating System Software • Simulator
Prototyping	• Software Development Environment • Hardware Development Environment	• In-Circuit Emulators • Data Acquisition Cards • DSP Development System Cards • Operating System Software • Debug Software
Design Verification	• Software Development Environment • Hardware Development Environment	• Data Acquisition Packages • Data Acquisition Cards • DSP Development System Cards • Application-Specific Development Tools • Debug Software

13.2 WORLD WIDE WEB

Motorola and Digital Signal Processors (DSP) Marketing maintain several World Wide Web (WWW) locations that you can access using third-party web browser software. The following web pages are good places to start:

- **Motorola DSP Home Page:**
  http://www.mot.com/SPS/DSP/
  http://www.motorola-dsp.com
- **Motorola SPS Home Page:**
  http://www.mot.com/SPS/
- **Motorola DSP ftp site:**
  http://www.mot.com/pub/SPS/DSP/LIBRARY/

13.3 MOTOROLA DSP HOME PAGE

The Motorola DSP Web site was created by the DSP Marketing Department's Web Development Group to help achieve its mission: to provide DSP information and services to Motorola customers via the World Wide Web.

Some of the more notable features of the site include:

- **What's New**
  The What's New page contains the latest information on Motorola DSP products and services.
- **Technical Documentation**
  The complete library of Motorola DSP technical documentation is available.
- **Product Overview**
  The Product Overview pages give a brief overview of each of our products.
- **Dr. BuB's Free Software Library**
  The successor to the Dr. BuB bulletin board, Dr. BuB's Free Software Library contains free software for Motorola DSPs.
- **DSP Helpline Online**
  DSP Helpline Online gives you access to the DSP Helpline, our highly trained technical support staff.

13.4 DOCUMENTATION

DSP Marketing provides the following types of documentation:

- Product Briefs
- Technical Data
- Family Manuals
- User’s Manuals
- Application Notes

These documents can be accessed in one or more of the following ways:

- Download
  (DSP Home Page → Technical Documentation)
- Order by number from Literature Distribution Center (LDC)
  - Fill out WWW on-line form
    (SPS Home Page → Literature Order)
  - Call Literature and Printing Services at (800) 441-2447
- Receive via FAX (Mfax™)
  - Fill out WWW on-line form
    (SPS Home Page → Mfax)
  - Call the Mfax automated service at (602) 244-6591

13.5 APPLICATIONS ASSISTANCE

Applications assistance is available via:

- WWW
- Motorola SPS Design Hotline
- Motorola DSP Helpline
- Motorola DSP Newsletter
- Third-Party Support
- University Support
- Training

13.5.1 WWW

The World Wide Web is described in **World Wide Web** on page 13-4.

13.5.2 Motorola SPS Design Hotline

Information and assistance for all Motorola products is available through the Motorola Customer Support Center at the following number:

(800) 521-6274

13.5.3 Motorola DSP Helpline

Information and assistance for DSP applications is available through one of the following:

- Local Motorola field office
- Email to the Helpline
  dsphelp@dsp.sps.mot.com

Note: To receive the fastest service, contact the field office first. If they are unable to help you, contact the Helpline.

13.5.4 Motorola DSP Newsletter

The Motorola DSP Newsletter is a quarterly document providing information on new products, application briefs, questions and answers, DSP product information, third-party product news, etc. This newsletter is free and is available upon request by sending a request for “DSPNEWSL/D” by one of the following methods:

- Order by FAX

(602) 994-6430

- Order online
  (DSP Home Page → Literature Order Form)

- Order from the Literature Distribution Center by phone:

(800) 441-2447

13.5.5 Third-Party Support Information

Information about third-party manufacturers who use and support Motorola DSP products is available by calling Motorola DSP Marketing at the following number:

(512) 891-3098

Third-party support includes:

- Filter design software
- Logic analyzer support
- Boards for VME, IBM-PC clones, MACII boards
- Development systems
- Data conversion cards
- Operating system software
- Debug software

Information is also available on the WWW (DSP Home Page → Development Tools) and in the DSP Newsletter.

13.5.6 University Support

The Motorola University Support program helps DSP engineers of tomorrow experience first-hand the features of Motorola’s DSP products in university DSP laboratories using Motorola-donated DSP hardware and software.

Information concerning university support programs and university discounts for all Motorola DSP products is available by calling Motorola DSP Marketing at the following number.

(512) 891-3098

13.5.7 Training Courses

Training courses conducted by Motorola are offered in your plant or at the Phoenix, Arizona training facility. Other courses conducted by Motorola's training consulting partners are available in other locations.

For more information about training, visit the Motorola WWW Training page (SPS Home Page → Other Info & Services → Training) or call **Motorola** training at:

(602) 302-8008

13.6 SOFTWARE DEVELOPMENT ENVIRONMENT

The CLASx Software Development Environment is an integrated product written in the C language that comprises the following:

- Callable modules
- Linker
- Relocatable macro cross-assembler
- Clock-by-clock instruction simulator
- Librarian
- Graphical user interface

The CLASx Software Development Environment can be used on the following platforms:

- IBM™ PCs (386 or higher) running DOS 2.x and 3.x
- Macintosh™ II running MAC OS 7.0 or later
- SUN-4™ running Sun OS 4.1.x or Solaris 2.4
- Hewlett-Packard Series 7xx running HP-UX A.09.05

For more information about the CLASx Software Development Environment, see **Software Development Environment** on page 12-5.

13.7 HARDWARE DEVELOPMENT ENVIRONMENT

The Application Development System (ADS) is a four-component system that acts as a development tool for designing and debugging real-time signal processing systems. The four components are as follows:

- User interface software
- DSP Host Interface board and cable
- Command converter board and cable
- Application Development Module (ADM) board

For more information about the Hardware Development Environment, see **Hardware Development Environment** on page 12-8.

13.8 FREE SOFTWARE

Motorola DSP Marketing provides free software associated with DSP products via the WWW. A partial list of the programs available is given in the following sections. New software will be posted on the web page as it is made available.

To download the free software that has been developed for Motorola DSPs, perform the following steps:

1. Connect to the WWW.
2. Go to the DSP Home Page.
3. Select Technical Documentation.
4. Select Dr. Bub Archives.
5. Select the Macintosh (hqx) or PC (zip) file that you want to download.

13.8.1 Application Development System (ADS)

The programs in the following table are for use with the Application Development System (ADS).

Table 13-2 ADS Software Available on the WWW

Document ID	DSP	Description	Size (K)
48compeq.asm	56000ADC16	Low-pass equiripple FIR filter source code	2.8
4compeq.lod	56000ADC16	Low-pass equiripple FIR filter hex file	0.03
48compeq.lst	56000ADC16	Low-pass equiripple FIR filter list file	5.5
aspec2.asm	56000ADC16	Real-time log-log scale spectrum analyzer program source code	24.5
aspec2.p	56000ADC16	Real-time log-log scale spectrum analyzer program s-record file	16.5
cosine.asm	56000ADC16	Cosine table source code	21.6
decode32.abl	56000ADS	32k x 24 address decoder for DSP56000	2.2
decode32.jed	56000ADS	PAL program load file	1.8
decode8k.abl	56000ADS	8k x 24 address decoder for DSP56000	2.4
decode8k.jed	56000ADS	PAL program load file	1.9

Table 13-2 ADS Software Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
evbman	56000ADC16	Evaluation board information on sample files	7.6
fftssi.lot	56000ADC16	FFT evaluation program file	8.9
intioevb.asm	56000ADC16	General interrupt I/O loop source code	3.7
intioevb.lod	56000ADC16	General interrupt I/O loop hex file	0.3
intioevb.lst	56000ADC16	General interrupt I/O loop assembled file	4.1
loopevb.asm	56000ADC16	General polled loop source code	2.4
loopevb.lod	56000ADC16	General polled loop hex file	0.2
loopevb.lst	56000ADC16	General polled loop assembled file	4.1
sxxevb.asm	56000ADC16	Sine(x)/x correction FIR filter I/O routine source code	4.5
sxxevb.lod	56000ADC16	Sine(x)/x correction FIR filter I/O routine hex file	0.7
sine.asm	56000ADC16	Sine table source code	21.6
sxxevb.lst	56000ADC16	Sine(x)/x correction FIR filter I/O routine assembled file	7.6
window1.asm	56000ADC16	Blackman window coefficients source code	43.3

13.8.2 Audio Software

The software in the following table is useful when designing audio with the DSP56000 and DSP56300 families.

Note: See Application Note APR2/D *Digital Stereo 10-Band Graphic Equalizer Using the DSP56001* for more information.

Table 13-3 Audio Software Available on the WWW

Document ID	DSP	Description	Size (K)
dge.asm	56000	Digital graphic equalizer source code	14.9
dge.lod	56000	Digital graphic equalizer hex file	2.7
dge.p	56000	Digital graphic equalizer s-record file	2.7
rvb1.asm	56000	Easy-to-read reverberation routine source code	17
rvb2.asm	56000	Same as rvb1.asm but optimized	15.4

Table 13-3 Audio Software Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
stereo.hlp	56000	stereo.asm help file	0.6

13.8.3 Benchmark Programs

The programs in the following table are useful in many applications and are typically used to benchmark against other DSP chips.

Note: See the *DSP56000 Family Manual* for more details on benchmarking.

Table 13-4 Benchmark Programs Available on the WWW

Document ID	DSP	Description	Size (K)
1-56.asm	56000 56001	20-tap FIR filter source code	4.2
2-56.asm	56000 56001	64-tap FIR filter source code	4.2
3-56.asm	56000 56001	67-tap FIR filter source code	4.2
4-56.asm	56000 56001	8-pole 4 multiply cascaded canonic IIR filter source code	3.7
5-56.asm	56000 56001	8-pole 5 multiply cascaded canonic IIR filter source code	3.9
6-56.asm	56000 56001	8-pole cascaded transposed IIR filter source code	2.8
7-56.asm	56000 56001	Dot product source code	2
a-56.asm	56000 56001	Memory to memory FFT—64 point source code	7.3
b-56.asm	56000 56001	Memory to memory FFT—256 point source code	7.3
b11.asm	96002	Real multiply source code	0.9
b110.asm	96002	N complex updates source code	0.9
b110a.asm	96002	N complex updates source code	2.7
b111.asm	96002	Complex correlation or convolution (FIR filter) source code	2.7

Table 13-4 Benchmark Programs Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
b112.asm	96002	Nth order power series (real) source code	1.7
b113.asm	96002	2nd order real biquad IIR filter source code	1.4
b114.asm	96002	N cascaded real biquad IIR filters source code	3.9
b115a.asm	96002	Fast fourier transforms source code	3.2
b115b.asm	96002	Faster radix 2 decimation in time FFT source code	8.4
b115c.asm	96002	Radix 4 decimation frequency FFT source code	8.6
b116.asm	96002	LMS adaptive filter source code	5.8
b117.asm	96002	FIR lattice filter source code	3.5
b119.asm	96002	General lattice filter source code	4.3
b12.asm	96002	N real multipliers source code	1.3
b120.asm	96002	Normalized lattice filter source code	4.6
b123.asm	96002	N point 3 x 3 2d FIR convolution source code	7.4
b124.asm	96002	Table lookup w/linear interpolation between points source code	3.5
b125.asm	96002	Argument reduction source code	3.5
b126.asm	96002	Non-IEEE floating point division source code	2.1
b127.asm	96002	Multibit rotates source code	9.6
b128.asm	96002	Bit field extraction/insertion source code	8.6
b129.asm	96002	Newton-Raphson approximation for 1.0/SQRT (x) source code	1.7
b13.asm	96002	Real update source code	1
b130.asm	96002	Newton-Raphson approximation for SQRT (x) source code	1.6
b131.asm	96002	Unsigned integer divide source code	3.3
b132.asm	96002	Signed integer divide source code	3.1
b133a.asm	96002	Graphics accept/reject, floating point version source code	2.6
b133b.asm	96002	Line accept/reject, floating point version source code	2.6
b133c.asm	96002	Line accept/reject, fixed point version source code	1.7
b133d.asm	96002	Four point polygon accept/reject source code	3.6
b133e.asm	96002	Four point polygon accept/reject (looped) source code	1.5
b134.asm	96002	Cascaded five coefficient transpose IIR filter source code	2.5

Table 13-4 Benchmark Programs Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
b135.asm	96002	3-dimensional graphics illumination source code	4.2
b136.asm	96002	Pseudorandom number generation source code	1.4
b137.asm	96002	Bezier cubic polynomial evaluation source code	3.5
b138a.asm	96002	Pack 4 bytes into a 32-bit word source code	1.1
b138b.asm	96002	Pack two 16-bit words into a single 32-bit word source code	0.9
b138c.asm	96002	Unpack a 32-bit word into four extended bytes source code	1.2
b138d.asm	96002	Unpack a 32-bit word into two 16-bit sign-extended bytes source code	1
b139.asm	96002	Nth order polynomial evaluation for two points source code	1.3
b14.asm	96002	N real updates source code	1.4
b140a.asm	96002	Graphics bit block transfer (BITBLT) source code	3.1
b140b.asm	96002	64-bit block transfer source code	3
b141.asm	96002	64 x 64-bit unsigned multiply source code	1.9
b142.asm	96002	Approximation of 1/dl source code	1
b143a.asm	96002	Line drawing source code	5.4
b143b.asm	96002	Integer incremental line drawing algorithm source code	3.6
b144.asm	96002	Wire frame graphics rendering source code	54.2
b15.asm	96002	FIR filter w/data shift source code	1.5
b16.asm	96002	Real * complex correlation or convolution (FIR filter) source code	1.5
b17.asm	96002	Complex multiply source code	1.4
b18.asm	96002	N complex multiply source code	1.8
b19.asm	96002	Complex update source code	1.5
c-56.asm	56000 56001	Memory to memory FFT—1024 point source code	10.3
d-56.asm	56000 56001	Port to memory FFT—64 point source code	16.1
d2-56.asm	56000 56001	Port to memory FFT—64 point source code	8
e-56.asm	56000 56001	Port to memory FFT—256 point source code	1.6

Table 13-4 Benchmark Programs Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
e2-56.asm	56000 56001	Port to memory FFT—256 point source code	8
f-56.asm	56000 56001	Port to memory FFT—1024 point source code	16.1
f2-56.asm	56000 56001	Port to memory FFT—1024 point source code	11.2
magsqr.asm	56000 56001	Magnitude squared macro source code	0.5
read-me	56000 56001	Motorola benchmark descriptions text file	7.6
results.100	56000 56001	**Data text files**	1.8
results.75	56000 56001	**Data text files**	1.8
sincos.asm	56000 56001	Sine-cosine table generator for FFTs source code	1.2
sincos.hlp	56000 56001	Sine-cosine table generator help file	1.1
singen.asm	56000 56001	Generates “points” samples of a sine wave source code	0.9
sqrt3.asm	56000 56001	Full precision square root by polynomial approximation source code	1.4
sqrt3.hlp	56000 56001	Full precision square root help file	1
wbh4m.asm	56000 56001	Blackman-Harris 4 term minimum sidelobe window source code	0.7

13.8.4 Boot Software

The files and software listed in the following table are for booting from an EPROM.

Table 13-5 Boot Software Available on the WWW

Document ID	DSP	Description	Size (K)
boot.art	56000	Article on booting	6.7
boot.asm	56000	Construct a boot module source code	2.3

Table 13-5 Boot Software Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
bood.lod	56000	Construct a boot module hex file	1
boot.lst	56000	Construct a boot module list file	4.1
boot.p	56000	Construct a boot module s-record file	1

13.8.5 Codec Routines

The programs in the following table perform code/decode Analog-to-Digital and Digital-to-Analog conversions.

Note: See the application note APR12/D *Twin CODEC Expansion Board for the DSP56000 Application Development System* for more information.

Table 13-6 Codec Routines Available on the WWW

Document ID	DSP	Description	Size (K)
linlog.asm	56000	Linear PCM to companded codec data conversion	4.8
linlog.hlp	56000	linlog.asm help file	1.7
loglin.asm	56000 56001	Companded codec to linear PCM data conversion	4.6
loglin.hlp	56000 56001	loglin.asm help file	1.5
loglint.asm	56000 56001	loglin.asm test program source code	2.2
loglint.hlp	56000 56001	loglint.asm help file	2

13.8.6 Demo Software

Table 13-7 Demo Software Available on the WWW

Document ID	DSP	Description	Size (K)
julia.asm	96000	Generates the Julia set	2.8

13.8.7 DTMF Routines

The programs in the following table are tone generation and detection codes for Dial Tone Multi-Frequency (DTMF) applications. The Goertzel algorithm is an optimized DTMF algorithm.

Table 13-8 DTMF Routines Available on the WWW

Document ID	DSP	Description	Size (K)
clear.cmd	56000	Clear command file	0.1
data.lod	56000	Data hex file	0.4
det.asm	56000	Subroutine used in IIR DTMF source code	5.9
dtmf.all	56000	Compilation of all DTMF source code routines	73.5
dtmf.asm	56000	Main routine used in IIR DTMF source code	10.7
dtmf.mem	56000	DTMF routine memory file	0.5
dtmfmstr.asm	56000	Main routine for multi-channel DTMF source code	7.4
dtmfmstr.mem	56000	Multi-channel DTMF routine memory file	0.04
dtmftwo.asm	56000	DTMF receiver and generator main program source code	10.3
ex56.bat	56000	56000 assembler batch file	0.1
example.lst	56000	Goertzel algorithm list file	11.6
genxd.lod	56000	X load file	0.2
genyd.lod	56000	Y load file	0.2
goertzel.asm	56000	Goertzel routine source code	4.4
goertzel.lnk	56000	Goertzel routine link file	7
goertzel.lst	56000	Goertzel routine list file	11.6
load.cmd	56000	Load command file	0.04
read.me	56000	Instructions text file	0.7
sub.asm	56000	Subroutine linked for use in IIR DTMF source code	2.5
tstgoert.mem	56000	Goertzel routine memory file	0.4

13.8.8 Encoders

Table 13-9 Reed-Solomon Encoder Available on the WWW

Document ID	DSP	Description	Size (K)
bingray.asm	56000	Binary to Gray code conversion macro source code	0.6
bingrayt.asm	56000	bingray.asm test program source code	1
newc.c	56000	Reed-Solomon coder (C source code)	4.1
readme.rs	56000	Instructions for Reed-Solomon coding text file	5.2
rscd.asm	56000	Reed-Solomon coder for DSP56000 simulator source code	5.8
table1.asm	56000	Reed-Solomon coder include file	8
table2.asm	56000	Reed-Solomon coder include file	4

13.8.9 Fast Fourier Transforms

The Fast Fourier Transforms (FFTs) in the following table include complex and real FFTs.

Note: See the application note APR4/D *Implementation of Fast Fourier Transforms on Motorola’s Digital Signal Processors.*

Table 13-10 Fast Fourier Transforms Available on the WWW

Document ID	DSP	Description	Size (K)
bergorde.asm	96000	Bergland order table generator source code	1.8
bergsinc.asm	96000	Bergland sine/cosine coefficient lookup table generator source code	0.9
bitrev.asm	56000	Converse bit reverse order to normal order in-place source code	1.1
bitrevtw.asm	56156	Sort sin and cosine coefficient look-up tables in bit reverse order for DSP56156 (source code)	1.4
cfft3n.asm	96000	Complex - Radix 2 decimation in-place FFT source code	3.2
cfft3nn.asm	96000	Complex - Radix 2 decimation in-time FFT source code	13.2
cfft56.asm	56000	512-point non-in-place FFT source code	21.7

Table 13-10 Fast Fourier Transforms Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
cfft96.asm	96000	Complex - Radix 2 Cooley-Tukey decimation-in-time FFT source code	13.2
cfft96t.asm	96000	Complex - Radix 2 Cooley-Tukey decimation-in-time FFT source code	2.3
dct1.asm	56000	Discrete cosine transform using FFT source code	5.5
dct1.hlp	56000	dct1.asm help file	1
dhit1.asm	56000	Routine to compute Hilbert transform in the frequency domain source code	1.9
dhit1.hlp	56000	dhit1.asm help file	1
fft2d256.asm	96000	256x256 complex FFT source code	
fft.asm	56000	Radix 2 decimation-in-time 512-point FFT source code	7.4
fftas.asm	56000	Radix 2 in-place decimation-in-time (smallest code size) with automatic scaling at each pass source code	3.8
fftbf.asm	56002	Radix 2 in-place decimation-in-time (smallest code size) with block floating point on the 56002 source code	4.2
fftr2a.asm	56000	Radix 2, in-place, decimation-in-time FFT (smallest) source code	3.4
fftr2a.hlp	56000	fftr2a.asm help file	2.7
fftr2aa.asm	56000	Automatic scaling FFT source code	3.2
fftr2at.asm	56000	FFT test program(fftr2a.asm) source code	1
fftr2at.hlp	56000	fftr2at.asm help file	0.6
fftr2at.bak	56000	fftr2at.asm backup file	1
fftr2at.cld	56000	Automatic scaling **c program load** file	5.7
fftr2at.lst	56000	Automatic scaling list file	51.3
fftr2b.asm	56000	Radix 2, in-place, decimation-in-time FFT (faster) source code	4.3
fftr2bf.hlp	56000	fftr2bf.asm help file	1.6
fftr2b.hlp	56000	fftr2b.asm help file	3.7
fftr2c.asm	56000	Radix 2, in-place, decimation-in-time FFT (even faster) source code	6
fftr2c.hlp	56000	fftr2c.asm help file	3.2
fftr2cc.asm	56000	Radix 2, in-place decimation-in-time complex FFT macro source code	6.5

Table 13-10 Fast Fourier Transforms Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
fftr2cc.hlp	56000	fftr2cc.asm help file	3.5
fftr2cn.asm	56000	Radix 2, decimation-in-time complex FFT macro with normally ordered input/output source code	6.6
fftr2cn.hlp	56000	fftr2cn.asm help file	2.5
fftr2cnt.asm	56000	FFT to complex number conversion source code	0.5
fftr2d.asm	56000	Radix 2, in-place, decimation-in-time FFT (using DSP56001 sine-cosine ROM tables) source code	3.7
fftr2d.hlp	56000	fftr2d.asm help file	3.5
fftr2dt.asm	56000	fftr2d.asm test program source code	1.3
fftr2dt.hlp	56000	fftr2dt.asm help file	0.6
fftr2e.asm	56000	1024 point, non-in-place, FFT (3.39 ms) source code	9
fftr2e.hlp	56000	fftr2e.asm help file	5
fftr2et.asm	56000	fftr2e.asm test program source code	1
fftr2et.hlp	56000	fftr2et.asm help file	0.4
fftr2en.asm	56000	1024 point, not-in-place, complex FFT macro with normally ordered input/output source code	9.8
fftr2en.hlp	56000	fftr2en.asm help file	4.9
fftr2bf.asm	56000	Radix-2, decimation-in-time FFT with block floating point source code	13.5
fftr2fn.asm	56000	Port to memory FFT—1024 point source code	10.4
fftr2fnt.asm	56000	Test file source code for fftr2fn.asm	0.8
gen56.asm	56001	Input signal generator for FFT on 56001 source code	0.9
norm2ber.asm	96000	Convert normal order to Berlang order source code	0.5
rfft	96002	Bergland order table generator source code	22.1
rfft56t.asm	56000	Non-in-place FFT - 1024 point source code	11.7
rfft96b.asm	96002	Real-valued FFT source code	9.3
rfft96bt.asm	96002	Real input FFT source code	2.9
rfft96t.asm	96002	Test program source code	2.5
sincos.asm	56000	Sine-cosine table generator for FFTs source code	1.2
sincosf.asm	96002	Sine-cosine table generator source code	1.3
sincosr.asm	56000	Sine-cosine table generator for rfft56.asm source code	1.5
sincos.hlp	56000	sincos.asm help file	0.9

Table 13-10 Fast Fourier Transforms Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
sinewave.asm	56000	Full-cycle sine wave table generator macro source code	1
sinewave.hlp	56000	sinewave.asm help file	1.4
split56.asm	56000	Amplifies coefficients of FFT by two source code	3.6
split96.asm	96002	Split N/2 complex FFT (hn) for N real FFT (Fn) source code	2.8

13.8.10 Filters

The programs in the following table include various Infinite Impulse Response (IIR), Finite Impulse Response (FIR), and lattice filter programs.

Note: See application note APR7/D *Implementing IIR/FIR Filters with Motorola's DSP56000/DSP56001* for more information.

Table 13-11 Filters Available on the WWW

Document ID	DSP	Description	Size (K)
fir.asm	56000	Direct form FIR filter source code	0.5
fir.hlp	56000	fir.asm help file	2.2
firt.asm	56000	fir.asm test program source code	1.2
iir1.asm	56000	Direct form second-order all pole IIR filter source code	0.7
iir1.hlp	56000	iir1.asm help file	1.8
iir1t.asm	56000	iir1.asm test program source code	1.2
iir2.asm	56000	Direct form second-order all pole IIR filter with scaling source code	0.8
iir2.hlp	56000	iir2.asm help file	2.3
iir2t.asm	56000	iir2.asm test program source code	1.3
iir3.asm	56000	Direct form arbitrary order all pole IIR filter source code	0.8
iir3.hlp	56000	iir3.asm help file	2.7
iir3t.asm	56000	iir3.asm test program source code	1.3
iir4.asm	56000	Second-order direct canonic IIR filter (biquad IIR filter) source code	0.7

Table 13-11 Filters Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
iir4.hlp	56000	iir4.asm help file	2.3
iir4t.asm	56000	iir4.asm test program source code	1.2
iir5.asm	56000	Second-order direct canonic IIR filter with scaling (biquad IIR filter) source code	0.8
iir5.hlp	56000	iir5.asm help file	2.8
iir5t.asm	56000	iir5.asm test program source code	1.3
iir6.asm	56000	Arbitrary order direct canonic IIR filter source code	0.9
iir6.hlp	56000	iir6.asm help file	3
iir6t.asm	56000	iir6.asm test program source code	1.4
iir7.asm	56000	Cascaded biquad IIR filters source code	0.9
iir7.hlp	56000	Help for iir7.asm	3.9
iir7t.asm	56000	iir7.asm test program source code	1.4
latfir1.asm	56000	Lattice FIR filter macro source code	1.2
latfir1.hlp	56000	latfir1.asm help file	6.3
latfir1t.asm	56000	latfir1.asm test program source code	1.4
latfir2.asm	56000	Lattice FIR filter macro (modified modulo count) source code	1.2
latfir2.hlp	56000	latfir2.asm help file	1.3
latfir2t.asm	56000	latfir2.asm test program source code	1.4
latgen.asm	56000	Generalized lattice FIR/IIR filter macro source code	1.3
latgen.hlp	56000	latgen.asm help file	5.5
latgent.asm	56000	latgen.asm test program source code	1.3
latiir.asm	56000	Lattice IIR filter macro source code	1.3
latiir.hlp	56000	latiir.asm help file	6.4
latiirt.asm	56000	latiir.asm test program source code	1.4
latnrm.asm	56000	Normalized lattice IIR filter macro source code	1.4
latnrm.hlp	56000	latnrm.asm help file	7.5
latnrmt.asm	56000	latnrm.asm test program source code	1.6
lms.hlp	56000	LMS Adaptive filter algorithm help file	5.8
p1	56200	Support software description	6.3
p2	56200	Adaptive filter interrupt driver flowchart	10.9
p3	56200	Adaptive filter interrupt driver program example	25.8

Table 13-11 Filters Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
p4	56200	Polled I/O flowchart	10.4
p5	56200	Polled I/O program example	24.8
p6	56200	Dual FIR filter interrupt driver flowchart	9.5
p7	56200	Dual FIR filter interrupt driver program example	28.5
p8	56200	Dual FIR filter polled I/O, flowchart	9.7
p9	56200	Dual FIR filter polled I/O program example	28.5
transiir.asm	56000	Implements the transposed IIR filter source code	2
transiir.hlp	56000	transiir.asm help file	1

13.8.11 Floating Point Routines

The programs in the following table are miscellaneous floating-point algorithms for the DSP56000 and DSP56300 families.

Table 13-12 Floating Point Routines Available on the WWW

Document ID	DSP	Description	Size (K)
durbin.asm	56000	Solution for LPC coefficients source code	5.6
durbin.hlp	56000	durbin.asm help file	2.9
float.sha	56000	Floating point routines shell archive	86.5
fpabs.asm	56000	Floating point absolute value source code	2
fpadd.asm	56000	Floating point add source code	3.9
fpcalls.hlp	56000	Subroutine calling conventions help file	11.9
fpceil.asm	56000	Floating point CEIL subroutine source code	1.8
fpcmp.asm	56000	Floating point compare source code	2.6
fpdef.hlp	56000	Storage format and arithmetic representation definition help file	10.6
fpdiv.asm	56000	Floating point divide source code	3.8
fpfix.asm	56000	Floating to fixed point conversion source code	4
fpfloat.asm	56000	Fixed to floating point conversion source code	2
fpfloor.asm	56000	Floating point FLOOR subroutine source code	2.1
fpfrac.asm	56000	Floating point FRACTION subroutine source code	1.9
fpinit.asm	56000	Library initialization subroutine source code	2.3

Table 13-12 Floating Point Routines Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
fplist.asm	56000	Test file that lists all subroutines (source code)	1.6
fpmac.asm	56000	Floating point multiply-accumulate source code	2.7
fpmpy.asm	56000	Floating point multiply source code	2.3
fpneg.asm	56000	Floating point negate source code	2
fprevs.hlp	56000	Latest revisions of floating-point library help file	1.8
fpscale.asm	56000	Floating point scaling source code	2.1
fpsqrt.asm	56000	Floating point square root source code	2.9
fpsub.asm	56000	Floating point subtract source code	3.1

13.8.12 Functions

The programs in the following table are standard mathematical functions.

Table 13-13 Functions Available on the WWW

Document ID	DSP	Description	Size (K)
exp2.asm	56000	Exponential base 2 by polynomial approximation source code	0.9
exp2.asm	56000	Exponential base 2 by polynomial approximation source code	0.9
exp2.hlp	56000	exp2.asm help file	0.8
exp2t.asm	56000	exp2.asm test program source code	1
log2.asm	56000	Log base 2 by polynomial approximation source code	1.1
log2.hlp	56000	Help for log2.asm	0.7
log2nrm.asm	56000	Normalizing base 2 logarithm macro source code	2.2
log2nrm.hlp	56000	log2nrm.asm help file	0.7
log2nrmt.asm	56000	log2nrm.asm test program source code	1.1
log2t.asm	56000	log2.asm test program source code	1
rand1.asm	56000	Pseudo random sequence generator source code	2.4
rand1.hlp	56000	rand1.asm help file	0.7
sqrt1.asm	56000	Square root by polynomial approximation, 7-bit accuracy source code	1

Table 13-13 Functions Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
sqrt1.hlp	56000	sqrt1.asm help file	0.8
sqrt1t.asm	56000	sqrt1.asm test program source code	1.1
sqrt2.asm	56000	Square root by polynomial approximation, 10-bit accuracy source code	0.9
sqrt2.hlp	56000	sqrt2.asm help file	0.8
sqrt2t.asm	56000	sqrt2.asm test program source code	1
sqrt3.asm	56000	Full precision square root macro source code	1.4
sqrt3.hlp	56000	sqrt3.asm help file	0.8
sqrt3t.asm	56000	sqrt3.asm test program source code	1
tli.asm	56000	Linear table lookup/interpolation routine for function generation source code	3.2
tli.hlp	56000	tli.asm help file	1.5

13.8.13 Matrix Operations

The programs in the following table perform matrix operations.

Table 13-14 Matrix Operations Available on the WWW

Document ID	DSP	Description	Size (K)
8-56.asm	56000 56001	[2x2] [2x2] matrix multiply source code	3.6
9-56.asm	56000 56001	[3x3] [3x3] matrix multiply source code	3.3
b121.asm	96002	[1x3] [[3x3] and [1x4] [4x4] matrix multiply source code	3.7
b122.asm	96002	[1x3] [[3x3] and [nxn] [nxn] matrix multiply source code	4.3
matmul1.asm	56000	[1x3][3x3]=[1x3] matrix multiplication source code	1.8
matmul1.hlp	56000	matmul1.asm help file	0.5
matmul2.asm	56000	General matrix multiplication, C=AB source code	2.7
matmul2.hlp	56000	matmul2.asm help file	0.8
matmul3.asm	56000	General matrix multiply-accumulate, C=AB+Q source code	2.8
matmul3.hlp	56000	matmul3.asm help file	0.9

13.8.14 Multiplier/Accumulator (MAC)

The programs in the following table perform double-precision Multiplier/Accumulator operations.

Table 13-15 Multiplier/Accumulator Operations Available on the WWW

Document ID	DSP	Description	Size (K)
dmac.asm	56000	Double precision multiplier/accumulator source code	2.9
sdm.asm	56000	Single x double multiplication using DP mode of 56000 core source code	1.1
sdmac.asm	56002	Single x double MAC using DP mode of 56002 source code	1.3

13.8.15 Sorting Routines

Table 13-16 Sorting Routines Available on the WWW

Document ID	DSP	Description	Size (K)
sort1.asm	56001	Array sort by straight selection source code	1.3
sort1.hlp	56001	sort1.asm help file	1.9
sort1t.asm	56001	sort1.asm test program source code	0.7
sort2.asm	56001	Array sort by Heapsort method source code	2.2
sort2.hlp	56001	sort2.asm help file	2
sort2t.asm	56001	sort2.asm test program source code	0.7

13.8.16 Speech

Table 13-17 Speech Available on the WWW

Document ID	DSP	Description	Size (K)
adpcm.asm	56001	32 kbps CCITT ADPCM speech coder source code	12.1
adpcm.hlp	56001	adpcm.asm help file	14.8
adpcm.uue	56001	Binary to text UNIX-to-UNIX (UU) encode file	0.4
adpcmns.asm	56001	Nonstandard ADPCM source code	54.7
adpcmns.hlp	56001	adpcmns.asm help file	10
durbin1.asm	56001	Durbin Solution for PARCOR (LPC) coefficients source code	6.4
durbin1.hlp	56001	durbin1.asm help file	3.6
lgsol1.asm	56001	Leroux-Gueguen solution for PARCOR (LPC) coefficients source code	4.9
lgsol1.hlp	56001	lgsol1.asm help file	4

13.8.17 Standard I/O Equates

The programs in the following table are useful when writing standardized assembly code.

Table 13-18 Standard I/O Equates Available on the WWW

Document ID	DSP	Description	Size (K)
intequ.asm	56000	Standard interrupt equate file source code	1.1
intequlc.asm	56000	Lower case version of intequ.asm source code	1.1
ioequ.asm	56000	Motorola standard I/O equate file source code	8.8
ioequlc.asm	56000	Lower case version of ioequ.asm source code	8.9

13.8.18 Tools and Utilities

Note: The program dos4gw.exe solves memory problems when running CLASx and ADS software.

Table 13-19 Tools and Utilities Available on the WWW

Document ID	DSP	Description	Size (K)
cldlod.hqx	56000	Convert c program load file (.cld) to load file (.lod) file, Macintosh compressed format. Includes source, executable, and makefile.	37
cldlod.nxt	56000	Convert c program load file (.cld) to load file (.lod) file, Next compressed format. Includes source, executable, and makefile.	41
cldlod.rea	56000	Read me text file.	0.5
cldlod.sn3	56000	Convert c program load file (.cld) to load file (.lod) file, Sun 3 compressed format. Includes source, executable, and makefile.	114.7
cldlod.sn4	56000	Convert c program load file (.cld) to load file (.lod) file, Sun compressed format. Includes source, executable, and makefile.	131.1
cldlod.zip	56000	Convert c program load file (.cld) to load file (.lod) file, PC compressed format. Includes source, executable, and makefile.	25.4
dos4gw.exe	56000	Convert c program load file (.cld) to load file (.lod) file, Macintosh compressed format.	231.2
dspbug	56000	Ordering information for free debug monitor for DSP56000/DSP56001	882
parity.asm	56000	Parity calculation of a 24-bit number in accumulator A source code	1641
master2.asm	96000	Multi-device simulator source code	9.8
parity.hlp	56000	parity.asm help file	936
parityt.asm	56000	parity.asm test program source code	685
parityt.hlp	56000	parityt.asm help file	259
read.me	96000	Multi-device simulation read me text file	0.3
Slave2.asm	96000	Multi-device simulation source code	
sloader.asm	56000	Serial loader from the SCI port for the DSP56001 source code	3986
sloader.hlp	56000	sloader.asm help file	2598
sloader.p	56000	Serial loader s-record file for download to EPROM source code	736
srec.c	56000	Utility to convert DSP56000 OMF format to SREC (source code)	38975
srec.doc	56000	Manual page for srec.c.	7951

Table 13-19 Tools and Utilities Available on the WWW (Continued)

Document ID	DSP	Description	Size (K)
srec.h	56000	srec.c include file	3472
srec.exe	56000	IBM PC srec executable	22065

13.8.19 Viterbi

Viterbi is a Reed-Solomon decoder of Trellis encoding.

Note: See the application note APR6/D *Convolutional Encoding and Viterbi Decoding Using the DSP56001 with a V.32 Modem Trellis Example* for more information.

Table 13-20 Viterbi Routines Available on the WWW

Document ID	DSP	Description	Size (K)
bound.d	56000	Data file	1.3
decode.asm	56000	Viterbi decoder for V.32 source code file	10.7
encode.asm	56000	Convolutional decoder for V.32 source code file	0.9
read.me	56000	Usage instructions text file	

13.9 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 DSPs. 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.

13.9.1 General DSP

Bellanger, Maurice. *Digital Processing Of Signals Theory And Practice.* New York, NY: John Wiley and Sons, 1984.

Cadzow, J. A. *Foundations Of Digital Signal Processing And Data Analysis.* New York, NY: MacMillan Publishing Company, 1987.

Candy, James V. *Signal Processing – The Modern Approach*. New York, NY: McGraw-Hill Company, Inc., 1988.

Chen, C.H. *Signal Processing Handbook*. New York, NY: Marcel Dekker, Inc., 1988.

Crochiere, R. E., and Rabiner, L. R. *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

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

Elliott, D. F. *Handbook Of Digital Signal Processing*. San Diego, CA: Academic Press, Inc., 1987.

Lim, Jae S., and Oppenheim, Alan V. *Advanced Topics In Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Oppenheim, A. V. *Applications Of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Oppenheim, A. V., and Schafer, R.W. *Discrete-time Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989.

Oppenheim, Alan V., and Schafer, Ronald W. *Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

Proakis, John G., and Manolakis, Dimitris G. *Introduction To Digital Signal Processing*. New York, NY: Macmillan Publishing Company, 1988.

Stearns, S., and Davis, R. *Signal Processing Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Rabiner, Lawrence R., Gold, and Bernard. *Theory And Application Of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

13.9.2 Digital Audio and Filters

Antoniou, Andreas. *Digital Filters: Analysis And Design*. New York, NY: McGraw-Hill Company, Inc., 1979.

Chamberlin, H. *Musical Applications Of Microprocessors (Second Edition)*. Hasbrouck Heights, NJ: Hayden Book Co., 1985.

Haykin, Simon. *Introduction To Adaptive Filters*. New York, NY: MacMillan Publishing Company, 1984.

Jackson, Leland B. *Digital Filters And Signal Processing*. Higham, MA: Kluwer Academic Publishers, 1986.

Jayant, N. S., and Noll, Peter. *Digital Coding Of Waveforms*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Kuc, Roman. *Introduction To Digital Signal Processing*. New York, NY: McGraw-Hill Company, Inc., 1988.

Mulgrew, B., and Cowan, C. *Adaptive Filter And Equalizers*. Higham, MA: Kluwer Academic Publishers, 1988.

Roberts, Richard A., and Mullis, Clifford T. *Digital Signal Processing*. New York, NY: Addison-Wesley Publishing Company, Inc., 1987.

Strawn, John. *Digital Audio Signal Processing An Anthology*. William Kaufmann, Inc., 1985.

Watkinson, John. *The Art Of Digital Audio*. Stoneham. MA: Focal Press, 1988.

Widrow, B., and Stearns, S. D. *Adaptive Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

Williams, Charles S. *Designing Digital Filters*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986.

13.9.3 C Programming Language

American National Standards Institute. *Programming Language - C*. ANSI Document X3.159-1989. American National Standards Institute, inc., 1990.

Harbison, Samuel P., and Steele, Guy L. *C: A Reference Manual*. Prentice-Hall Software Series, 1987.

Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language*. Prentice-Hall, Inc., 1978.

13.9.4 Controls

Astrom, K., and Wittenmark, B. *Adaptive Control*. New York, NY: Addison-Wesley Publishing Company, Inc., 1989.

Astrom, K., and Wittenmark, B. *Computer Controlled Systems: Theory & Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Goodwin, G., and Sin, K. *Adaptive Filtering Prediction & Control*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Kuo, B. C. *Automatic Control Systems*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

Kuo, B. C. *Digital Control Systems*. New York, NY: Holt, Reinholt, and Winston, Inc., 1980.

Moroney, P. *Issues In The Implementation Of Digital Feedback Compensators*. Cambridge, MA: The MIT Press, 1983.

Phillips, C., and Nagle, H. *Digital Control System Analysis & Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

13.9.5 Graphics

Arnold, D. B., and Bono, P. R. *CGM And CGI*. New York, NY: Springer-Verlag, 1988.

Artwick, Bruce A. *Microcomputer Displays, Graphics, And Animation*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

Bono, P. R., and Herman, I. (Eds.). *GKS Theory And Practice*. New York, NY: Springer-Verlag, 1987.

Foley, J. D., and, Van Dam, A. *Fundamentals Of Interactive Computer Graphics*. Reading MA: Addison-Wesley Publishing Company Inc., 1984.

Hall, Roy. *Illumination And Color In Computer Generated Imagery*. New York, NY: Springer-Verlag.

Hearn, D. and Baker, M. Pauline. *Computer Graphics (Second Edition)*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986.

Morteson, Michael E. *Geometric Modeling*. New York, NY: John Wiley and Sons, Inc.

Newman, William M., and Sproull, Roger F. *Principles Of Interactive Computer Graphics*. New York, NY: McGraw-Hill Company, Inc., 1979.

Pixar. *The Renderman Interface*. San Rafael, CA. 94901.

Reid, Glenn C. (Adobe Systems, Inc.). *Postscript Language Program Design*. Reading MA: Addison-Wesley Publishing Company, Inc., 1988.

Rogers, David F. *Procedural Elements For Computer Graphics*. New York, NY: McGraw-Hill Company, Inc., 1985.

13.9.6 Image Processing

Barnsley, M. F., Devaney, R. L., Mandelbrot, B. B., Peitgen, H. O., Saupe, D., and Voss, R. F. *The Science Of Fractal Images*. New York, NY: Springer-Verlag.

Ekstrom, M. P. *Digital Image Processing Techniques*. New York, NY: Academic Press, Inc., 1984.

Gonzales, Rafael C., and Wintz, Paul. *Digital Image Processing (Second Edition)*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1977.

Pratt, William K. *Digital Image Processing*. New York, NY: John Wiley and Sons, 1978.

Rosenfeld, Azriel, and Kak, Avinash C. *Digital Picture Processing*. New York, NY: Academic Press, Inc., 1982.

13.9.7 Motorola DSP Manuals

Motorola. *DSP56000 Linker/librarian Reference Manual*. Motorola, Inc., 1991.

Motorola. *DSP56000 Macro Assembler Reference Manual*. Motorola, Inc., 1991.

Motorola. *DSP56000 Simulator Reference Manual*. Motorola, Inc., 1991.

Motorola. *DSP56000/DSP56001 User's Manual*. Motorola, Inc.,1990.

13.9.8 Numerical Methods

Berliout, P., and Bizard, P. *Algorithms (The Construction, Proof, And Analysis Of Programs)*. New York, NY: John Wiley and Sons, 1986.

Golub, G. H., and Van Loan, C. F. *Matrix Computations*. John Hopkins Press, 1983.

Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. *Numerical Recipes In C - The Art Of Scientific Programming*. Cambridge University Press, 1988.

Schroeder, Manfred R. *Number Theory In Science And Communication*. New York, NY: Springer-Verlag, 1986.

13.9.9 Pattern Recognition

Bracewell, R. N. *The Fast Fourier Transform And Its Applications*. New York, NY: McGraw-Hill Company, Inc., 1986.

Brigham, E. Oran. *The Fast Fourier Transform And Its Applications*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Duda, R. O., and Hart, P. E. *Pattern Classification And Scene Analysis*. New York, NY: John Wiley and Sons, 1973.

Gardner, William A. *Statistical Spectral Analysis, A Nonprobabilistic Theory*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

James, Mike. *Classification Algorithms*. New York, NY: Wiley-Interscience, 1985. Spectral Analysis:

13.9.10 Speech

Flanagan, J. L. *Speech Analysis, Synthesis, And Perception*. New York, NY: Springer-Verlag, 1972.

Honig, Michael L., and Messerschmitt, David G. *Adaptive Filters – Structures, Algorithms, And Applications*. Higham, MA: Kluwer Academic Publishers, 1984.

Jayant, N. S., and Noll, P. *Digital Coding Of Waveforms*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Markel, J. D. and Gray, A. H., Jr. *Linear Prediction Of Speech*. New York, NY: Springer-Verlag, 1976.

O'Shaughnessy, D. *Speech Communication – Human And Machine*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1987.

Rabiner, Lawrence R., and Schafer, R. W. *Digital Processing Of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

13.9.11 Telecommunications

Lee, Edward A., and Messerschmitt, David G. *Digital Communication*. Higham, MA: Kluwer Academic Publishers, 1988.

Proakis, John G. *Digital Communications*. New York, NY: McGraw-Hill Publishing Co., 1983.

APPENDIX A

INSTRUCTION SET DETAILS

Arithmetic

ABS
ADC
ADD
ASL
ASLL
ASR
ASRR
ASRAC
CLR
CMP
DECW
DIV
IMPY
INCW
MAC
MACR
MPY
MPYR
MPYSU
MACSU
NEG
NORM
RND
SBC
SUB
Tcc
TFR
TST
TSTW

Logical

AND
ANDC
EOR
EORC
LSL
LSLL
LSR
LSRR
LSRAC
NOT
NOTC
OR
ORC
ROL
ROR

Bit-field Manipulation

BFTSTL
BFTSTH
BFCLR
BFSET
BFCHG
BRCLR
BRSET

Loop

DO
ENDDO
REP

Move

LEA
MOVE
MOVE(C)
MOVE(I)
MOVE(M)
MOVE(P)
MOVE(S)

Program Control

Bcc
BRA
DEBUG
Jcc
JMP
JSR
NOP
RTI
RTS
STOP
SWI
WAIT

A.1 INTRODUCTION

This section introduces the DSP56600 core 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 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.

A.2 INSTRUCTION FORMATS AND SYNTAX

The DSP56600 core instructions consist of one or two 24-bit words—an operation word and an optional extension word. This extension word can be either an effective address extension word or an immediate data extension word. While the extension word occupies the full 24-bit width of the program memory, only the sixteen Least Significant Bits (LSBs) are relevant for effective address extension or for immediate data. Therefore, the extension word is effectively sixteen bits wide. General formats of the instruction word are shown in **Figure A-1**. Most instructions specify data movement on the X Data Bus (XDB), Y Data Bus (YDB), and Data ALU operations in the same operation word. The DSP56600 core is designed to perform each of these operations in parallel.

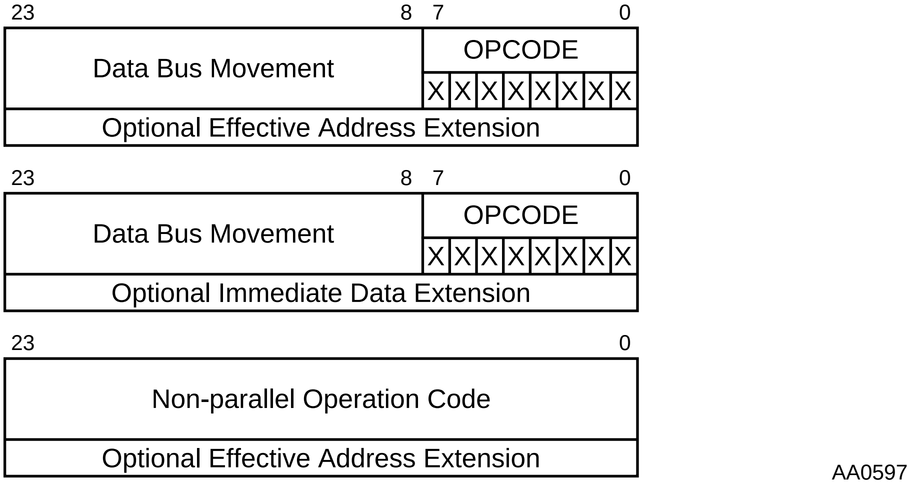

23
8
7
0
Data Bus Movement
OPCODE
X X X X X X X X
Optional Effective Address Extension
23
8
7
0
Data Bus Movement
OPCODE
X X X X X X X X
Optional Immediate Data Extension
23
0
Non-parallel Operation Code
Optional Effective Address Extension
AA0597

Figure A-1 General Formats of an Instruction 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/or YDB. This field may require additional information to fully specify the operand for certain addressing modes. An extension word following the operation word is used to provide immediate data, absolute address or address displacement, if required. Examples of operations that may include the extension word include move operation such as MOVE X:$100,X0.

The Opcode field of the operation word specifies the Data ALU operation or the Program Control Unit (PCU) operation to be performed.

The operation codes form a very versatile Microcontroller Unit (MCU) style instruction set, providing highly parallel operations in most programming situations.

The instruction syntax has two formats—parallel and non-parallel, as shown in **Table A-1** and **Table A-2**. Parallel instruction is organized into five columns: opcode, operands, two optional parallel-move fields, and an optional condition field. The condition field is used to disable the execution of the opcode if the condition is not true, and cannot be used in conjunction with the parallel move fields.

Table A-1 Parallel Instructions Format

	Opcode	Operands	XDB	YDB	Condition
Example 1	MAC	X0,Y0,A	X:(R0)+,X0	Y:(R4)+,Y0	
Example 2	MOVE		X:-(R1),X1		
Example 3	MAC	X1,Y1,B			
Example 4	MPY	X0,Y0,A			IF eq

Assembly-language source codes for some typical one-word instructions are shown in **Table A-1**. Because of the multiple bus structure and the parallelism of the DSP56600 core, as many as three data transfers can be specified in the instruction word—one on the XDB, one on the YDB, and one within the Data ALU. These transfers are explicitly specified. A fourth data transfer is implied and occurs in the PCU (instruction word prefetch, program looping control, etc.). The opcode column indicates the Data ALU operation to be performed, but may be excluded if only a MOVE operation is needed. The operands column specifies the operands to be used by the opcode. The XDB and YDB columns specify optional data transfers over the XDB and YDB and the associated addressing modes. The address space qualifiers (X:, Y:, and L:) indicate which address space is being referenced.

Non-parallel instruction is organized into two columns: opcode and operands. Assembly-language source codes for some typical one-word instructions are shown in **Table A-2**. Non-parallel instructions include all the program control, looping and peripherals read/write instructions. They also include some Data ALU instructions that are impossible to be encoded in the Opcode field of the parallel format.

Table A-2 Non-Parallel Instructions Format

	Opcode	**Operands**
Example 1:	JEQ	(R5)
Example 2:	MOVEP	#data,X:ipr
Example 3:	RTS	

A.2.1 Operand Sizes

Operand sizes are defined as follows: a byte is eight bits long, a word is sixteen bits long, a long word is 32 bits long, and an accumulator is 40 bits long, as shown in **Figure A-2**. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation.

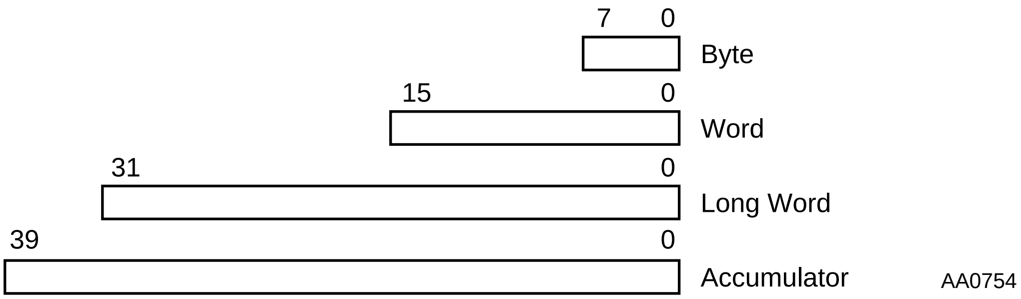


Figure A-2 Operand Sizes

A.2.2 Data Organization in Registers

The ten Data ALU registers support 8- or 16-bit data operands. Instructions support 32- or 40-bit data operands by concatenating groups of specific Data ALU registers. The eight address registers in the Address Generation Unit (AGU) support 16-bit address or data operands. The eight AGU offset registers support 16-bit offsets or may support 16-bit addresses 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. The Loop Counter (LC) and Loop Address (LA) registers support 16-bit address operands.

A.2.3 Data ALU Registers

The eight main data registers are sixteen 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) and the Most Significant Bit (MSB) is the left-most bit (Bit 15 for word operands and Bit 31 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 8–15) is sign-extended (see **Figure A-3**). 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 (e.g., 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 39).

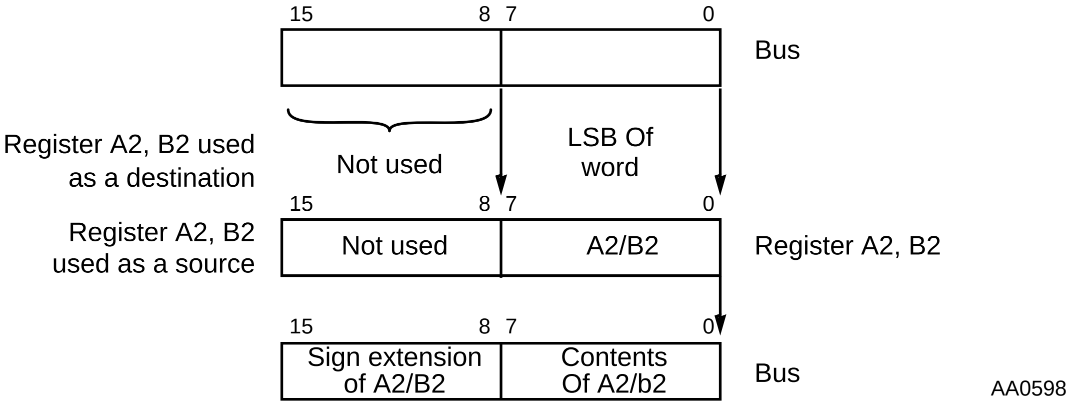


Figure A-3 Reading and Writing the ALU Extension Registers

When a 40-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 Mode Register (MR). If the data out of the shifter indicates that the accumulator extension register is in use and the data is to be moved into a 16-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 16-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 40-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 (CCR) is latched.

When a 40-bit accumulator (A or B) is specified as a *destination* operand D, any 16-bit source data to be moved into that accumulator is automatically extended to 40 bits by sign-extending the MSB of the source operand (Bit 15) and appending the source operand with sixteen 0s in the LSBs. For 16-bit source operands, both the automatic sign extension and zeroing features can be disabled by specifying the destination register to be one of the individual 16-bit accumulator registers (A1 or B1).

A.2.3.1 AGU Registers

The sixteen AGU registers, which are each sixteen bits wide, can be accessed as word operands for address, address offset, address modifier, and data storage. 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. The notation Mn is used to designate one of the eight address modifier registers, M0–M7.

A.2.3.2 Program Control Registers

The Chip Operating Mode (COM) register occupies the lower eight bits and the Extended Chip Operating Mode (EOM) register occupies the upper eight bits of the 16-bit OMR. The OMR and the Vector Base Address (VBA) register are accessed as word operands. Not all of their bits are defined. Undefined bits are read as 0 and should be written with 0 for future compatibility. The 16-bit SR has the user CCR occupying the low-order eight bits and the system MR occupying the high-order eight bits. The SR can be accessed as a word operand. The MR and CCR can be accessed individually as word operands (see **Figure A-4**).

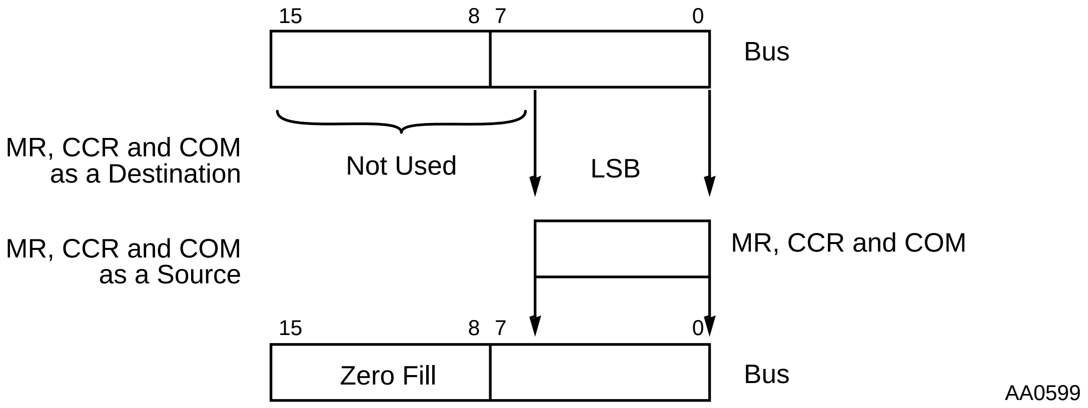


Figure A-4 Reading and Writing Control Registers

The Loop Counter (LC), Loop Last Address (LA), System Stack High (SSH), and System Stack Low (SSL) registers are sixteen bits wide and are accessed as word operands. The system Stack Pointer (SP) is a 16-bit register that is accessed as a word operand. The PC, a special 16-bit-wide program counter register, is generally referenced implicitly as a word operand, but may also be referenced explicitly (by all PC-relative operation codes) also as a word operand.

A.2.3.3 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 16-bit-wide X and Y memories can store word and byte operands. 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.

A.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.

A.3.1 Arithmetic Instructions

The arithmetic instructions perform all of the arithmetic operations within the Data ALU. 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 move operation that can be specified in parallel to the instruction marked is one of the parallel instructions listed in **Table A-7** on page A-14. Arithmetic instructions can be executed conditionally, based on the condition codes generated by the previous instructions. Conditional arithmetic instructions do not allow parallel data movement over the various data buses. **Table A-3** lists the arithmetic instructions. A "√" sign in a table cell in the "Parallel Instruction" column indicates that the corresponding instruction is a parallel instruction, while a blank table cell indicates that the instruction is not a parallel instruction.

Table A-3 Arithmetic Instructions

Mnemonic	Description	Parallel Instruction
ABS	Absolute Value	√
ADC	Add Long with Carry	√
ADD	Add	√
ADD (imm.)	Add (immediate operand)	
ADDL	Shift Left and Add	√
ADDR	Shift Right and Add	√
ASL	Arithmetic Shift Left	√
ASL (mb.)	Arithmetic Shift Left (multi-bit)	
ASL (mb., imm.)	Arithmetic Shift Left (multi-bit, immediate operand)	
ASR	Arithmetic Shift Right	√
ASR (mb.)	Arithmetic Shift Right (multi-bit)	
ASR (mb., imm.)	Arithmetic Shift Right (multi-bit, immediate operand)	
CLR	Clear an Operand	√
CMP	Compare	√
CMP (imm.)	Compare (immediate operand)	
CMPM	Compare Magnitude	√

Table A-3 Arithmetic Instructions (Continued)

Mnemonic	Description	Parallel Instruction
CMPU	Compare Unsigned	
DEC	Decrement Accumulator	
DIV	Divide Iteration	
DMAC	Double Precision Multiply-Accumulate	
INC	Increment Accumulator	
MAC	Signed Multiply-Accumulate	√
MAC (su,uu)	Mixed Multiply-Accumulate	
MACI	Signed Multiply-Accumulate (immediate operand)	
MACR	Signed Multiply-Accumulate and Round	√
MACRI	Signed Multiply-Accumulate and Round (immediate operand)	
MAX	Transfer By Signed Value	√
MAXM	Transfer By Magnitude	√
MPY	Signed Multiply	√
MPY (su,uu)	Mixed Multiply	
MPYI	Signed Multiply (immediate operand)	
MPYR	Signed Multiply and Round	√
MPYRI	Signed Multiply and Round (immediate operand)	
NEG	Negate Accumulator	√
NORMF	Fast Accumulator Normalize	
RND	Round	√
SBC	Subtract Long with Carry	√
SUB	Subtract	√

Table A-3 Arithmetic Instructions (Continued)

Mnemonic	Description	Parallel Instruction
SUB (imm.)	Subtract (immediate operand)	
SUBL	Shift Left and Subtract	√
SUBR	Shift Right and Subtract	√
Tcc	Transfer Conditionally	
TFR	Transfer Data ALU Register	√
TST	Test an Operand	√

A.3.2 Logical Instructions

The logical instructions, which execute in one instruction cycle, perform all of the logical 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 move operation that can be specified in parallel to the instruction marked is one of the parallel instructions listed in **Table A-7** on page A-14. **Table A-4** lists the logical instructions. A “√” in a table cell in the “Parallel Instruction” column indicates that the corresponding instruction is a parallel instruction, while a blank table cell indicates that the instruction is not a parallel instruction.

Table A-4 Logical Instructions

Mnemonic	Description	Parallel Instruction
AND	Logical AND	√
AND (imm.)	Logical AND (immediate operand)	
ANDI	AND Immediate to Control Register	
CLB	Count Leading Bits	

Table A-4 Logical Instructions (Continued)

Mnemonic	Description	Parallel Instruction
EOR	Logical Exclusive OR	√
EOR (imm.)	Logical Exclusive OR (immediate operand)	
EXTRACT	Extract Bit Field	
EXTRACT (imm.)	Extract Bit Field (immediate operand)	
EXTRACTU	Extract Unsigned Bit Field	
EXTRACTU (imm.)	Extract Unsigned Bit Field (immediate operand)	
INSERT	INSERT Bit Field	
INSERT (imm.)	INSERT Bit Field (immediate operand)	
LSL	Logical Shift Left	√
LSL (mb.)	Logical Shift Left (multi-bit)	
LSL (mb., imm.)	Logical Shift Left (multi-bit, immediate operand)	
LSR	Logical Shift Right	√
LSR (mb.)	Logical Shift Right (multi-bit)	
LSR (mb.,imm.)	Logical Shift Right (multi-bit, immediate operand)	
MERGE	Merge Two Half Words	
NOT	Logical Complement	√
OR	Logical Inclusive OR	√
OR (imm.)	Logical Inclusive OR (immediate operand)	
ORI	OR Immediate to Control Register	
ROL	Rotate Left	√
ROR	Rotate Right	√

A.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 contains the result of the bit test. **Table A-5** lists the bit manipulation instructions. None of the bit manipulation instructions is a parallel instruction.

Table A-5 Bit Manipulation Instructions

Mnemonic	Description	Parallel Instruction
BCHG	Bit Test and Change	
BCLR	Bit Test and Clear	
BSET	Bit Test and Set	
BTST	Bit Test	

A.3.4 Loop Instructions

The hardware DO loop executes with no overhead cycles—that is, it runs as fast as straight-line code. Replacing straight-line code with DO loops can significantly reduce program memory usage. The loop instructions control hardware looping either by initiating a program loop and establishing looping parameters, or by 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. **Table A-6** lists the loop instructions. None of the loop instructions is a parallel instruction.

Table A-6 Loop Instructions

Mnemonic	Description	Parallel Instruction
BRKcc	Conditionally Break the current Hardware Loop	
DO	Start Hardware Loop	
DO FOREVER	Start Forever Hardware Loop	
ENDDO	Abort and Exit from Hardware Loop	

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 1.

A.3.5 Move Instructions

The move instructions perform data movement over the XDB and YDB or over the GDB. Move instructions, most of which allow Data ALU opcode in parallel, do not affect the CCR, except the limit bit L, if limiting is performed when reading a Data ALU accumulator register. **Table A-7** lists the move instructions. A “√” in a table cell in the “Parallel Instruction” column indicates that the corresponding instruction is a parallel instruction, while a blank table cell indicates that the instruction is not a parallel instruction.

Table A-7 Move Instructions

Mnemonic	Description	Parallel Instruction
LUA	Load Updated Address	
LRA	Load PC-Relative Address	
MOVE	Move Data Register	√
MOVEC	Move Control Register	
MOVEM	Move Program Memory	
MOVEP	Move Peripheral Data	
U MOVE	Update Move	√
VSL	Viterbi Shift Left	

A.3.6 Program Control Instructions

The program control instructions include jumps, conditional jumps, and other instructions 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. **Table A-8** lists the program control instructions. None of the program control instructions is a parallel instruction.

Table A-8 Program Control Instructions

Mnemonic	Description	Parallel Instruction
IFcc.U	Execute Conditionally and Update CCR	
IFcc	Execute Conditionally	
Bcc	Branch Conditionally	
BRA	Branch Always	
BScc	Branch to Subroutine Conditionally	
BSR	Branch to Subroutine Always	
DEBUGcc	Enter into the Debug Mode Conditionally	
DEBUG	Enter into the Debug Mode Always	
Jcc	Jump Conditionally	
JMP	Jump Always	
JCLR	Jump if Bit Clear	
JSET	Jump if Bit Set	
JScc	Jump to Subroutine Conditionally	
JSR	Jump to Subroutine Always	
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)	
TRAPcc	Trap Conditionally	

Table A-8 Program Control Instructions (Continued)

Mnemonic	Description	Parallel Instruction
TRAP	Trap Always	
WAIT	Wait for Interrupt (Low-Power Standby)	

A.4 GUIDE TO INSTRUCTION DESCRIPTIONS

The following information is included in each instruction description:

- **Name and Mnemonic:** The mnemonic is highlighted in **bold** type for easy reference.
- **Assembler Syntax and Operation:** For each instruction syntax, the corresponding operation is symbolically described. If several operations are 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 is indicated in parentheses in both the assembler syntax and operation fields. If a letter in the mnemonic is optional, it is shown in parenthesis in the assembler syntax field.
- **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.
- **Condition Codes:** The Status Register (SR) is depicted with the condition code bits which can be affected by the instruction. Not all bits in the SR are used. Those that are reserved are indicated with a gray box covering its area.
- **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 that can be assumed by each of the variables in the various instruction fields are shown under the instruction field's heading.

A.4.1 Notation

Each instruction description contains symbols used to abbreviate certain operands and operations. **Table A-9** 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.

Table A-9 Instruction Description Notation

Symbol	Meaning
Data ALU Registers Operands	
Xn	Input Register X1 or X0 (16 bits)

Table A-9 Instruction Description Notation (Continued)

Symbol	Meaning
Yn	Input Register Y1 or Y0 (16 bits)
An	Accumulator Registers A2, A1, A0 (A2—8 bits, A1 and A0—16 bits)
Bn	Accumulator Registers B2, B1, B0 (B2—8 bits, B1 and B0—16 bits)
X	Input Register X = X1: X0 (32 bits)
Y	Input Register Y = Y1: Y0 (32 bits)
A	Accumulator A = A2: A1: A0 (40 bits)
B	Accumulator B = B2: B1: B0 (40 bits)
AB	Accumulators A and B = A1: B1 (32 bits)
BA	Accumulators B and A = B1: A1 (32 bits)
A10	Accumulator A = A1: A0 (32 bits)
B10	Accumulator B = B1:B0 (32 bits)
Program Control Unit 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)
EOM	Extended Chip Operating Mode Register (8 bits)
COM	Chip Operating Mode Register (8 bits)
OMR	Operating Mode Register = EOM:COM (16 bits)
SZ	System Stack Size Register (16 bits)
SC	System Stack Counter Register (5 bits)
VBA	Vector Base Address (16 bits, eight set to 0)
LA	Hardware Loop Address Register (16 bits)
LC	Hardware Loop Counter Register (16 bits)

Table A-9 Instruction Description Notation (Continued)

Symbol	Meaning
SP	System Stack Pointer Register (16 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 (16 locations by 32 bits)
Address Operands	
ea	Effective Address
eax	Effective Address for X Bus
eay	Effective Address for Y Bus
xxxx	Absolute or Long Displacement Address (16 bits)
xxx	Short or Short Displacement Jump Address (12 bits)
xxx	Short Displacement Jump Address (9 bits)
aaa	Short Displacement Address (7 bits, sign-extended)
aa	Absolute Short Address (6 bits, zero-extended)
pp	High I/O Short Address (6 bits, ones-extended)
qq	Low I/O Short Address (6 bits)
<. . .>	Specifies the 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 Operands	
S, Sn	Source Operand Register
D, Dn	Destination Operand Register
D [n]	Bit n of D Destination Operand Register

Table A-9 Instruction Description Notation (Continued)

Symbol	Meaning
#n	Immediate Short Data (5 bits)
#xx	Immediate Short Data (8 bits)
#xxx	Immediate Short Data (12 bits)
#xxxx	Immediate Data (16 bits)
r	Rounding Constant
#bbbb	Operand Bit Select (4 bits)
Unary Operands	
–	Negation Operator
—	Logical NOT Operator (Overbar)
PUSH	Push Specified Value onto the System Stack (SS) Operator
PULL	Pull Specified Value from the SS Operator
READ	Read the Top of the SS Operator
PURGE	Delete the Top Value on the SS Operator
| |	Absolute Value Operator
Binary Operands	
+	Addition Operator
–	Subtraction Operator
*	Multiplication Operator
÷, /	Division Operator
+	Logical Inclusive OR Operator
•	Logical AND Operator
≈	Logical Exclusive OR Operator
fi	“Is Transferred To” Operator
:	Concatenation Operator

Table A-9 Instruction Description Notation (Continued)

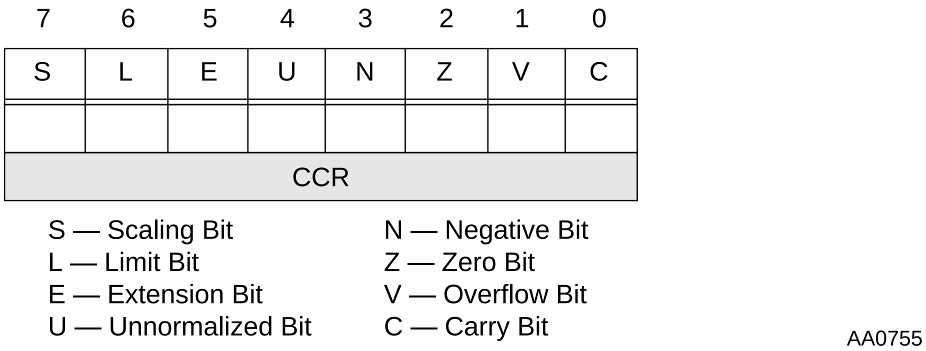
Symbol	Meaning
Addressing Mode Operators	
<<	I/O Short Addressing Mode Force Operator
<	Short Addressing Mode Force Operator
>	Long Addressing Mode Force Operator
#	Immediate Addressing Mode Operator
#>	Immediate Long Addressing Mode Force Operator
#<	Immediate Short Addressing Mode Force Operator
Mode Register Symbols	
LF	Loop Flag Bit Indicating When a DO Loop is in Progress
RM	Rounding Mode
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	
S	Block Floating Point Scaling Bit Indicating Data Growth Detection
L	Limit Bit Indicating Arithmetic Overflow and/or Data Shifting/Limiting
E	Extension Bit Indicating if the Integer Portion of Data ALU result is in Use
U	Unnormalized Bit Indicating if the Data ALU Result is Unnormalized
N	Negative Bit Indicating if Bit 39 of the Data ALU Result is Set
Z	Zero Bit Indicating if the Data ALU Result Equals Zero
V	Overflow Bit Indicating if Arithmetic Overflow has Occurred in Data ALU
C	Carry Bit Indicating if a Carry or Borrow Occurred in Data ALU Result
( )	Optional Letter, Operand, or Operation
(. . .)	Any Arithmetic or Logical Instruction Which Allows Parallel Moves
EXT	Extension Register Portion of an Accumulator (A2 or B2)

Table A-9 Instruction Description Notation (Continued)

Symbol	Meaning
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)
S/L	Shifting and/or Limiting on a Data ALU Register
Sign Ext	Sign Extension of a Data ALU Register
Zero	Zeroing of a Data ALU Register
Address ALU Registers Operands	
Rn	Address Registers R0–R7
Nn	Address Offset Registers N0–N7
Mn	Address Modifier Registers M0–M7

A.4.2 Condition Code Computation

The Condition Code Register (CCR) portion of the Status Register (SR) consists of eight defined bits, as shown in **Figure A-5**.



S — Scaling Bit
L — Limit Bit
E — Extension Bit
U — Unnormalized Bit
N — Negative Bit
Z — Zero Bit
V — Overflow Bit
C — Carry Bit

Figure A-5 Condition Codes

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 “sticky” and are not affected by Address ALU calculations or by data transfers over the XDB, YDB, or GDB. The L bit is a “sticky” overflow bit that indicates that an overflow has occurred in the Data ALU or that data limiting has occurred when moving the contents of the A and/or B accumulators. The S bit is a “sticky” bit used in block floating point operations to indicate the need to scale the number in A or B.

The full description of every instruction contains an illustration showing how the instruction affects the various condition codes. An instruction can affect a condition code according to three different rules:

Table A-10 Instruction Effect on Condition Code

Standard Mark	Effect on the Condition Code
—	This bit is unchanged by the instruction.
√	This bit is changed by the instruction, according to the standard definition of the condition code.
*	This bit is changed by the instruction, according to a special definition of the condition code, depicted as part of the instruction full description.

The standard definition of the Condition Code bits follows.

A.4.2.1 Scaling Bit (S)

The Scaling bit (S) is computed, according to the logical equations in **Table A-11**, when an instruction or a parallel move reads the contents of accumulator A or B to XDB or YDB. The S bit is a “sticky” bit, cleared only by an instruction that specifically clears it, or hardware reset.

Table A-11 S Bit Computation

S0	S1	Scaling Mode	S Bit Equation
0	0	No scaling	S = (A30 XOR A29) OR (B30 XOR B29) OR S (previous)
0	1	Scale down	S = (A31 XOR A30) OR (B31 XOR B30) OR S (previous)
1	0	Scale up	S = (A29 XOR A28) OR (B29 XOR B28) OR S (previous)
1	1	Reserved	S undefined

The S bit is used to detect data growth, which is required in Block Floating Point FFT operation. The S bit is set if the absolute value in the accumulator, before scaling, was greater or equal to 0.25 and smaller than 0.75. Typically, the bit is tested after each pass of a radix 2 decimation-in-time FFT and, if it is set, the appropriate scaling mode should be activated in the next pass. The Block Floating Point FFT algorithm is described in the Motorola application note APR4/D, *Implementation of Fast Fourier Transforms on Motorola’s DSP56000/DSP56001 and DSP96002 Digital Signal Processors.*

A.4.2.2 Limit Bit (L)

The Limit bit (L) is set if the Overflow bit (V) is set or if an instruction or a parallel move causes the data shifter/limiters to perform a limiting operation while reading the contents of accumulator A or B to the XDB or YDB bus. In Arithmetic Saturation mode, the limit bit is also set when an arithmetic saturation occurs in the Data ALU result. Not affected otherwise. The L bit is “sticky” and must be cleared only by an instruction that specifically clears it, or hardware reset.

A.4.2.3 Extension Bit (E)

The Extension bit (E) is cleared if all the bits of the signed integer portion of the Data ALU result are the same (i.e., the bit patterns are either 00. . . 00 or 11. . . 11). Otherwise, this bit is set. The signed integer portion is defined by the scaling mode, as shown in **Table A-12**.

Table A-12 Signed Integer Portion Definition

S1	S0	Scaling Mode	Integer Portion
0	0	No Scaling	Bits 39,38.............32,31
0	1	Scale Down	Bits 39,38.............33,32
1	0	Scale Up	Bits 39,38.............31,30

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 most significant eight, nine, or ten bits of that accumulator, depending on the scaling mode being used. The extension register portion of an accumulator (A2 or B2) is always the eight Most Significant 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 (S1 = S0 = 0), the signed integer portion of the A or B accumulator consists of bits 31 through 39. If the A accumulator contained the signed 40-bit value $00:8000:0000 as a result of a Data ALU operation, the E bit would be set (E = 1) since the nine Most Significant Bits of that accumulator were not all the same (i.e., neither 00...00 nor 11...11). This means that data limiting occurs if that 40-bit value is specified as a source operand in a move-type operation. This limiting operation results in either a positive or negative 16-bit or 32-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).

A.4.2.4 Unnormalized Bit (U)

The Unnormalized bit (U) is set if the two Most Significant Bits of the Most Significant Portion (MSP) of the Data ALU result are the same. This bit is cleared otherwise. The MSP is defined by the scaling mode. The U bit is computed as shown in **Table A-13**.

Table A-13 U Bit Computation

S1	S0	Scaling Mode	U Bit Computation
0	0	No Scaling	$U = \overline{(\text{Bit 31 xor Bit 30})}$
0	1	Scale Down	$U = \overline{(\text{Bit 32 xor Bit 31})}$
1	0	Scale Up	$U = \overline{(\text{Bit 30 xor Bit 29})}$

The result of calculating the U bit in this fashion is that the definition of a positive normalized number p is $0.5 \leq p < 1.0$ and the definition of negative normalized number n is $-1.0 \leq n < -0.5$.

A.4.2.5 Negative Bit (N)

The Negative bit (N) is set if the MS bit (Bit 39 in arithmetic instructions or Bit 31 in logical instructions) of the Data ALU result is set. Otherwise, this bit is cleared.

A.4.2.6 Zero Bit (Z)

The Zero bit (Z) is set if the Data ALU result equals 0. Otherwise, this bit is cleared.

A.4.2.7 Overflow Bit (V)

The Overflow bit (V) is set if an arithmetic overflow occurs in the 40-bit Data ALU result. Otherwise, this bit is cleared. This indicates that the result cannot be represented in the 40-bit accumulator; thus, the accumulator has overflowed.

In Arithmetic Saturation mode, an arithmetic overflow occurs if the Data ALU result is not representable in the accumulator without the extension part (i.e., 32-bit accumulator).

A.4.2.8 Carry Bit (C)

The Carry bit (C) is set if a carry is generated out of the MSB of the Data ALU result of an addition or if a borrow is generated out of the MSB of the Data ALU result of a subtraction. Otherwise, this bit is cleared. The carry or borrow is generated out of Bit 39 of the Data ALU result. The C bit is also affected by bit manipulation, rotate, shift, and compare instructions. The C bit is not affected by the Arithmetic Saturation mode.

A.5 INSTRUCTION DESCRIPTIONS

The following section describes each instruction in the DSP56600 core instruction set in detail. Instructions that allow parallel moves are so noted in both the **Operation** and the **Assembler Syntax** fields. 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.

Whenever an instruction uses an accumulator as both a destination operand for Data ALU operation and as a source for a parallel move operation, the parallel move operation uses the value in the accumulator prior to execution of any Data ALU operation.

ABS Absolute Value ABS

Operation:

| D | → D (parallel move)

Assembler Syntax:

ABS D (parallel move)

Description: Take the absolute value of the destination operand D and store the result in the destination accumulator.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	—
CCR							

√	This bit is changed according to the standard definition.
—	This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

ABS D

23 16 15 8	7							0
DATA BUS MOVE FIELD	0	0	1	0	d	1	1	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)

ADC Add Long with Carry ADC

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 (32 bits) can be added to the 40-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 31 of the destination accumulator (A or B).

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 16 15 8	7							0
ADC S,D	DATA BUS MOVE FIELD	0	0	1	J	d	0	0	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{S}	**J**	Source register [X,Y] (see **Table A-16** on page A-203)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)

ADD Add ADD

Operation:

S + D → D (parallel move)

#xx + D → D

#xxxx + D → D

Assembler Syntax:

ADD S,D (parallel move)

ADD #xx,D

ADD #xxxx,D

Description: Add the source operand S to the destination operand D and store the result in the destination accumulator. The source can be a register (16-bit word, 32-bit long word, or 40-bit accumulator), 6-bit short immediate, or 16-bit long immediate.

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the six bits are right-aligned and the remaining bits are zeroed to form a 16-bit source operand.

Note: The Carry bit(C) 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 31 of the destination accumulator (A or B). Thus, the C bit is always set correctly using accumulator source operands, but can be set incorrectly if A1, B1, A10, B10 or immediate operand are used as source operands and A2 and B2 are not replicas of Bit 31.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

ADD Add ADD

Instruction Formats and Opcodes:

ADD S,D

23							16	15							8	7							0
DATA BUS MOVE FIELD																0	J	J	J	d	0	0	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

ADD #xx,D

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

ADD #xxxx,D

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0	d	0	0	0
IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	**JJJ**	Source register [B/A,X,Y,X0,Y0,X1,Y1] (see **Table A-19** on page A-204)
{D}	**d**	Destination accumulator [A/B] (see **Table A-15** on page A-203)
{#xx}	**iiiiii**	6-bit Immediate Short Data
{#xxxx}		16-bit Immediate Long Data extension word

ADDL Shift Left and Add Accumulators ADDL

Operation:	**Assembler Syntax**
S + 2 ∗ D → D (parallel move)	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 0 is shifted into the LSB of D prior to the addition operation. The Carry bit (C) is set correctly if the source operand does not overflow as a result of the left shift operation. The Overflow bit (V) 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	*	√
CCR							

* V Set if overflow has occurred in A or B result or the MSB of the destination operand is changed as a result of the instruction’s left shift.

√ This bit is changed according to the standard definition.

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 … 16 15 … 8	7							0
ADDL S,D	DATA BUS MOVE FIELD	0	0	0	1	d	0	1	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

{S} The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B.

ADDR Shift Right and Add Accumulators ADDR

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 (C) is always set correctly, and the Overflow bit (V) 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 16 15 8	7							0
ADDR S,D	DATA BUS MOVE FIELD	0	0	0	0	d	0	1	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S} The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B.

AND Logical AND AND

Operation:		**Assembler Syntax:**
S • D[31:16] → D[31:16]	(parallel move)	AND S,D (parallel move)
#xx • D[31:16] → D[31:16]		AND #xx,D
#xxxx • D[31:16] → D[31:16]		AND #xxxx,D

where • denotes the logical AND operator

Description: Logically AND the source operand S with bits 31–16 of the destination operand D and store the result in bits 31–16 of the destination accumulator. The source can be a 16-bit register, 6-bit short immediate, or 16-bit long immediate. This instruction is a 16-bit operation. The remaining bits of the destination operand D are not affected.

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the six bits are right aligned and the remaining bits are zeroed to form a 16-bit source operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	—	—	—	*	*	*	—
CCR							

* N Set if bit 31 of the result is set.
* Z Set if bits 31-16 of the result are 0.
* V Always cleared.
√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

AND Logical AND AND

Instruction Formats and Opcodes:

AND S,D

23							16	15							8	7							0
DATA BUS MOVE FIELD																0	1	J	J	d	1	1	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

AND #xx,D

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

AND #xxxx,D

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0	d	1	1	0
IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	**JJ**	Source input register [X0,X1,Y0,Y1] (see **Table A-17** on page A-203)
{D}	**d**	Destination accumulator [A/B] (see **Table A-15** on page A-203)
{#xx}	**iiiiii**	6-bit Immediate Short Data
{#xxxx}		16-bit Immediate Long Data extension word

ANDI AND Immediate with Control Register ANDI

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

For CCR Operand:

* S Cleared if Bit 7 of the immediate operand is cleared.
* 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 Formats and Opcodes:

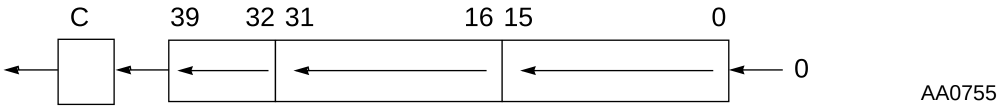
	23							16	15							8	7							0
AND(I) #xx,D	0	0	0	0	0	0	0	0	i	i	i	i	i	i	i	i	1	0	1	1	1	0	E	E

Instruction Fields:

{D}	EE	Program Controller register [MR,CCR,COM,EOM] (see **Table A-18** on page A-204)
{#xx}	iiiiiiii	Immediate Short Data

ASL Arithmetic Shift Accumulator Left ASL

Operation:



Assembler Syntax:

ASL D (parallel move)
ASL D #ii, S2,D
ASL S1,S2,D

Description:

- Single bit shift:

  Arithmetically shift the destination accumulator D one bit to the left and store the result in the destination accumulator. The MSB of D prior to instruction execution is shifted into the Carry bit (C) and a 0 is shifted into the LSB of the destination accumulator D.

- Multi-bit shift:

  The contents of the source accumulator S2 are shifted left #ii bits. Bits shifted out of position 39 are lost except for the last bit, which is latched in the C bit. The vacated positions on the right are zero-filled. The result is placed into destination accumulator D. The number of bits to shift is determined by the 6-bit immediate field in the instruction, or by the 6-bit unsigned integer located in the six LSBs of the control register S1. If a zero shift count is specified, the C bit is cleared. The difference between ASL and LSL is that ASL operates on the entire 40 bits of the accumulator, and therefore, sets the Overflow bit (V) if the number overflows.

This is a 40-bit operation.

ASL Arithmetic Shift Accumulator Left ASL

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	*	*
CCR							

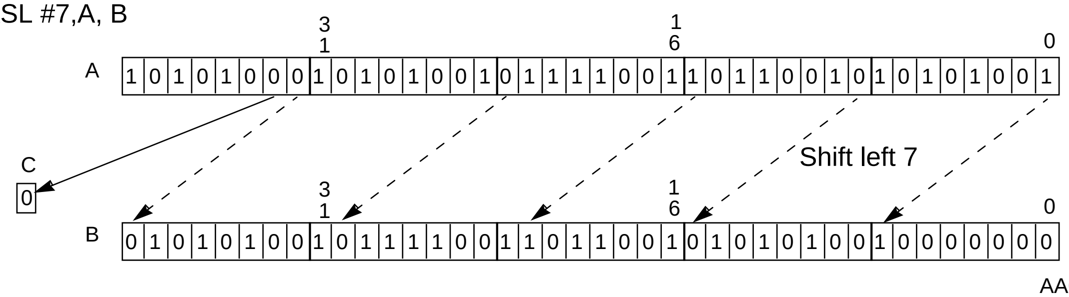
* V Set if Bit 39 is changed any time during the shift operation, cleared otherwise.

* C Set if the last bit shifted out of the operand is set, cleared for a shift count of 0, and cleared otherwise.

√ This bit is changed according to the standard definition.

Example:

ASL #7,A, B



A: 1 0 1 0 1 0 0 0 | 1 0 1 0 1 0 0 1 | 0 1 1 1 1 0 0 1 | 1 0 1 1 0 0 1 0 | 1 0 1 0 1 0 0 1

B: 0 1 0 1 0 1 0 0 | 1 0 1 1 1 1 0 0 | 1 1 0 1 1 0 0 1 | 0 1 0 1 0 1 0 0 | 1 0 0 0 0 0 0 0

AA0756

Instruction Formats and Opcodes:

ASL D

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

ASL #ii,S2,D

23							16	15							8	7							0
0	0	0	0	1	1	0	0	0	0	0	1	1	1	0	1	S	i	i	i	i	i	i	D

ASL S1,S2,D

23							16	15							8	7							0
0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	1	0	S	s	s	s	D

ASL Arithmetic Shift Accumulator Left ASL

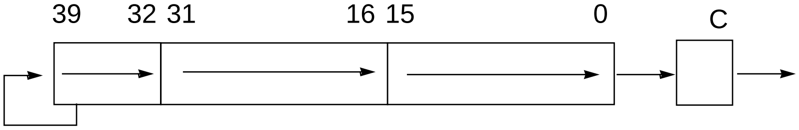
Instruction Fields:

{S2}	**S**	Source accumulator [A,B] (see **Table A-15** on page A-203)
{D}	**D**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S1}	**sss**	Control register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)
{#ii}	**iiiiii**	6-bit unsigned integer [0–40] denoting the shift amount

In the control register S1: bits 5–0 (LSB) are used as the #ii field, and the rest of the register is ignored.

ASR Arithmetic Shift Accumulator Right ASR

Operation:



AA0757

Assembler Syntax:

ASR D (parallel move)
ASR D #ii, S2,D
ASR S1,S2,D

Description:

- Single bit shift:

  Arithmetically shift the destination operand D one bit to the right and store the result in the destination accumulator. The LSB of D prior to instruction execution is shifted into the Carry bit (C), and the MSB of D is held constant.

- Multi-bit shift:
  The contents of the source accumulator S2 are shifted right #ii bits. Bits shifted out of position 0 are lost except for the last bit, which is latched in the C bit. Copies of the MSB are supplied to the vacated positions on the left. The result is placed into destination accumulator D. The number of bits to shift is determined by the 6-bit immediate field in the instruction, or by the 6-bit unsigned integer located in the six 6 LSBs of the control register S1. If a zero shift count is specified, the C bit is cleared.

This is a 40-bit operation.

Note: If the number of shifts indicated by the six LSBs of the control register or by the immediate field exceeds the value of 40, then the result is undefined.

ASR Arithmetic Shift Accumulator Right ASR

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	*	*
CCR							

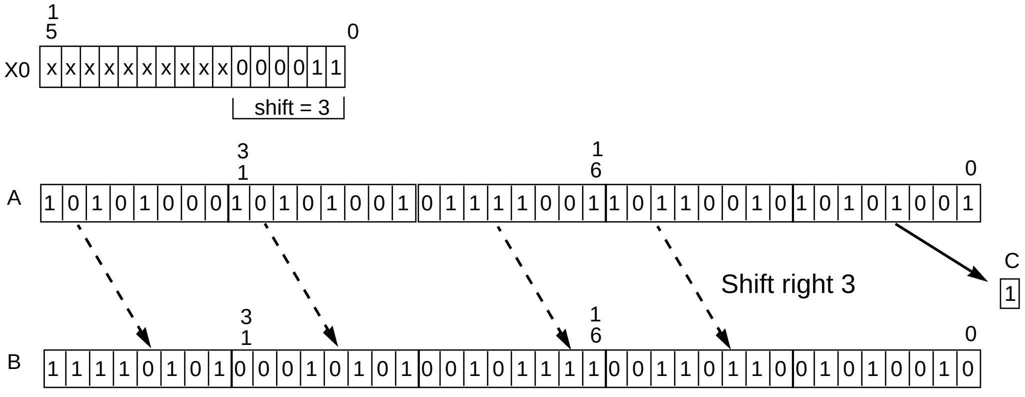
* V This bit is always cleared.

* C This bit is set if the last bit shifted out of the operand is set, cleared for a shift count of 0, and cleared otherwise.

√ This bit is changed according to the standard definition.

Example:

ASR X0,A,B



AA0758

Instruction Formats and Opcodes:

ASR D

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

ASR #ii,S2,D

23							16	15							8	7							0
0	0	0	0	1	1	0	0	0	0	0	1	1	1	0	0	S	i	i	i	i	i	i	D

ASR S1,S2,D

23							16	15							8	7							0
0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	1	1	S	s	s	s	D

ASR Arithmetic Shift Accumulator Right ASR

Instruction Fields:

{S2}	S	Source accumulator [A,B] (see **Table A-15** on page A-203)
{D}	D	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S1}	sss	Control register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)
{#ii}	iiiiii	6-bit unsigned integer [0-40] denoting the shift amount

In the control register S1: bits 5-0 (LSB) are used as the #ii field, and the rest of the register is ignored.

Bcc Branch Conditionally Bcc

Operation:

If cc, then PC + xxx → PC
else PC + 1 → PC

If cc, then PC + Rn → PC
else PC + 1 → PC

Assembler Syntax:

Bcc xxx

Bcc Rn

Description: If the specified condition is true, program execution continues at location PC + displacement. If the specified condition is false, the PC is incremented and program execution continues sequentially. The displacement is a two's-complement 16-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement and Address Register PC Relative addressing modes can be used. The Short Displacement 9-bit data is sign-extended to form the PC relative displacement. The conditions that the term "cc" can specify are listed on **Table A-47** on page A-213.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

		23							16	15							8	7							0
Bcc	xxx	0	0	0	0	0	1	0	1	C	C	C	C	0	1	a	a	a	a	0	a	a	a	a	a

		23							16	15							8	7							0
Bcc	Rn	0	0	0	0	1	1	0	1	0	0	0	1	1	R	R	R	0	1	0	0	C	C	C	C

Instruction Fields:

{cc}	**CCCC**	Condition code (see **Table A-48** on page A-214)
{xxx}	**aaaaaaaaa**	Signed PC Relative Short Displacement
{Rn}	**RRR**	Address register [R0–R7]

BCHG Bit Test and Change BCHG

Operation:		**Assembler Syntax:**	
$D[n] \rightarrow C$	$\overline{D[n]} \rightarrow D[n]$	BCHG	#n,[XorY]:ea
D[n] fi C	$\overline{D[n]} \rightarrow D[n]$	BCHG	#n,[XorY]:aa
$D[n] \rightarrow C$	$\overline{D[n]} \rightarrow D[n]$	BCHG	#n,[XorY]:pp
$D[n] \rightarrow C$	$\overline{D[n]} \rightarrow D[n]$	BCHG	#n,[XorY]:qq
$D[n] \rightarrow C$	$\overline{D[n]} \rightarrow D[n]$	BCHG	#n,D

Description: Test the nth bit of the destination operand D, complement it, and store the result in the destination location. The state of the nth bit is stored in the Carry bit (C) of the CCR register. 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

CCR Condition Codes:

For destination operand SR:

* C This bit is complemented if Bit 0 is specified, and not affected otherwise.

* V This bit is complemented if Bit 1 is specified, and not affected otherwise.

* Z This bit is complemented if Bit 2 is specified, and not affected otherwise.

* N This bit is complemented if Bit 3 is specified, and not affected otherwise.

* U This bit is complemented if Bit 4 is specified, and not affected otherwise.

* E This bit is complemented if Bit 5 is specified, and not affected otherwise.

* L This bit is complemented if Bit 6 is specified, and not affected otherwise.

* S This bit is complemented if Bit 7 is specified, and not affected otherwise.

BCHG Bit Test and Change BCHG

For other destination operands:

*	C	This bit is set if bit tested is set, and cleared otherwise.
*	V	This bit is not affected.
*	Z	This bit is not affected.
*	N	This bit is not affected.
*	U	This bit is not affected.
*	E	This bit is not affected.
*	L	This bit is set according to the standard definition.
*	S	This bit is set according to the standard definition.

MR Status Bits:

For destination operand SR:

*	I0	This bit is changed if Bit 8 is specified, and not affected otherwise.
*	I1	This bit is changed if Bit 9 is specified, and not affected otherwise.
*	S0	This bit is changed if Bit 10 is specified, and not affected otherwise.
*	S1	This bit is changed if Bit 11 is specified, and not affected otherwise.
*	FV	This bit is changed if Bit 12 is specified, and not affected otherwise.
*	SM	This bit is changed if Bit 13 is specified, and not affected otherwise.
*	RM	This bit is changed if Bit 14 is specified, and not affected otherwise.
*	LF	This bit is changed if Bit 15 is specified, and not affected otherwise.

For other destination operands: MR status bits are not affected.

Instruction Formats and Opcodes:

BCHG #n,[X or Y]:ea

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	1	M	M	M	R	R	R	O	S	0	0	b	b	b	b
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

BCHG #n,[X or Y]:aa

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	0	a	a	a	a	a	a	0	S	0	0	b	b	b	b

BCHG #n,[X or Y]:pp

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	0	p	p	p	p	p	p	0	S	0	0	b	b	b	b

BCHG Bit Test and Change BCHG

BCHG #n,[X or Y]:qq

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	1	q	q	q	q	q	q	0	S	0	0	b	b	b	b

BCHG #n,D

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	1	D	D	D	D	D	D	0	1	0	0	b	b	b	b

Instruction Fields:

{#n}	**bbbb**	Bit number [0–15]
{ea}	**MMMRRR**	Effective Address (see **Table A-23** on page A-206)
{X /Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{aa}	**aaaaaa**	Absolute Address [0-63]
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0–$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80–$FFBF]
{D}	**DDDDDD**	Destination register [all on-chip registers] (see **Table A-27** on page A-207)

BCLR Bit Test and Clear BCLR

Operation:		**Assembler Syntax:**	
D[n] → C	0 → D[n]	BCLR	#n,[XorY]:ea
D[n] → C	0 → D[n]	BCLR	#n,[XorY]:aa
D[n] → C	0 → D[n]	BCLR	#n,[XorY]:pp
D[n] → C	0 → D[n]	BCLR	#n,[XorY]:qq
D[n] → C	0 → D[n]	BCLR	#n,D

Description: Test the nth bit of the destination operand D, clear it and store the result in the destination location. The state of the nth bit is stored in the Carry bit (C) of the CCR register. 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

CCR Condition Codes:

For destination operand SR:

*	C	This bit is cleared if Bit 0 is specified, and not affected otherwise.
*	V	This bit is cleared if Bit 1 is specified, and not affected otherwise.
*	Z	This bit is cleared if Bit 2 is specified, and not affected otherwise.
*	N	This bit is cleared if Bit 3 is specified, and not affected otherwise.
*	U	This bit is cleared if Bit 4 is specified, and not affected otherwise.
*	E	This bit is cleared if Bit 5 is specified, and not affected otherwise.
*	L	This bit is cleared if Bit 6 is specified, and not affected otherwise.
*	S	This bit is cleared if Bit 7 is specified, and not affected otherwise.

BCLR Bit Test and Clear BCLR

For other destination operands:

- * C This bit is set if bit tested is set, and cleared otherwise.
- * V This bit is not affected.
- * Z This bit is not affected.
- * N This bit is not affected.
- * U This bit is not affected.
- * E This bit is not affected.
- * L This bit is set according to the standard definition.
- * S This bit is set according to the standard definition.

MR Status Bits:

For destination operand SR:

- * I0 This bit is changed if Bit 8 is specified, and not affected otherwise.
- * I1 This bit is changed if Bit 9 is specified, and not affected otherwise.
- * S0 This bit is changed if Bit 10 is specified, and not affected otherwise.
- * S1 This bit is changed if Bit 11 is specified, and not affected otherwise.
- * FV This bit is changed if Bit 12 is specified, and not affected otherwise.
- * SM This bit is changed if Bit 13 is specified, and not affected otherwise.
- * RM This bit is changed if Bit 14 is specified, and not affected otherwise.
- * LF This bit is changed if Bit 15 is specified, and not affected otherwise.

Instruction Formats and Opcodes:

BCLR #n,[X or Y]:ea

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	1	M	M	M	R	R	R	0	S	0	0	b	b	b	b
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

BCLR #n,[X or Y]:aa

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	0	a	a	a	a	a	a	0	S	0	0	b	b	b	b

BCLR #n,[X or Y]:pp

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	0	p	p	p	p	p	p	0	S	0	0	b	b	b	b

BCLR #n,[X or Y]:qq

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	0	q	q	q	q	q	q	0	S	0	0	b	b	b	b

BCLR #n,D

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	1	D	D	D	D	D	D	0	1	0	0	b	b	b	b

BCLR Bit Test and Clear BCLR

Instruction Fields:

{#n}	**bbbb**	Bit number [0-15]
{ea}	**MMMRRR**	Effective Address (see **Table A-23** on page A-206)
{X/Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{aa}	**aaaaaa**	Absolute Address [0–63]
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0–$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80–$FFBF]
{D}	**DDDDDD**	Destination register [all on-chip registers] (see **Table A-27** on page A-207)

BRA Branch Always BRA

Operation:

PC + xxx → Pc

PC + Rn → Pc

Assembler Syntax:

BRA xxx

BRA Rn

Description: Program execution continues at location PC + displacement. The displacement is a two's-complement 16-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 9-bit data is sign-extended to form the PC relative displacement.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

		23							16	15							8	7							0
BRA	xxx	0	0	0	0	0	1	0	1	0	0	0	0	1	1	a	a	a	a	0	a	a	a	a	a
BRA	Rn	0	0	0	0	1	1	0	1	0	0	0	1	1	R	R	R	1	1	0	0	0	0	0	0

Instruction Fields:

{xxx}	**aaaaaaaaa**	Signed PC Relative Short Displacement
{Rn}	**RRR**	Address register [R0–R7]

BRKcc Exit Current DO Loop Conditionally BRKcc

Operation:

If cc LA + 1→ PC; SSL(LF,FV) → SR; SP – 1 → SP
SSH → LA; SSL → LC; SP – 1 → SP
else PC + 1 → PC

Assembler Syntax:

BRKcc

Description: Exit conditionally the current hardware DO loop before the current Loop Counter (LC) equals 1. It also terminates the DO FOREVER loop. If the value of the current DO LC is needed, it must be read before the execution of the BRKcc instruction. Initially, the PC is updated from the LA, the Loop Flag (LF) and the Forever flag (FV) are restored and the remaining portion of the Status Register (SR) is purged from the system stack. The Loop Address (LA) and the LC registers are then restored from the system stack.

The conditions that the term “cc” can specify are listed on **Table A-48** on page A-214.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7							0
BRKcc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	C	C	C	C

Instruction Fields:

{cc} **CCCC** Condition code (see **Table A-48** on page A-214)

BScc Branch to Subroutine Conditionally BScc

Operation:

If cc, then PC → SSH;SR → SSL;PC + xxx → PC
else PC + 1 → PC

If cc, then PC → SSH;SR → SSL;PC + Rn → PC
else PC + 1 → PC

Assembler Syntax:

BScc xxx

BScc Rn

Description: If the specified condition is true, the address of the instruction immediately following the BScc instruction and the SR are pushed onto the stack. Program execution then continues at location PC + displacement. If the specified condition is false, the PC is incremented and program execution continues sequentially. The displacement is a 2's complement 16-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 9-bit data is sign extended to form the PC relative displacement.

The conditions that the term "cc" can specify are listed on **Table A-47** on page A-213.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

		23 … 16	15 … 8	7 … 0
BScc	xxx	0 0 0 0 0 1 0 1	C C C C 0 0 a a	a a 0 a a a a a
BScc	Rn	0 0 0 0 1 1 0 1	0 0 0 1 1 R R R	0 0 0 0 C C C C

BScc Branch to Subroutine Conditionally BScc

Instruction Fields:

{cc}	**CCCC**	Condition code (see **Table A-48** on page A-214)
{xxx}	**aaaaaaaaa**	Signed PC Relative Short Displacement
{Rn}	**RRR**	Address register [R0–R7]

BSET Bit Set and Test BSET

Operation: **Assembler Syntax:**

D[n] → C	1→ D[n]	BSET	#n,[XorY]:ea
D[n] → C	1 → D[n]	BSET	#n,[XorY]:aa
D[n] → C	1 → D[n]	BSET	#n,[XorY]:pp
D[n] → C	1 → D[n]	BSET	#n,[XorY]:qq
D[n] → C	1 → D[n]	BSET	#n,D

Description: Test the nth bit of the destination operand D, set it, and store the result in the destination location. The state of the nth bit is stored in the Carry bit (C) of the CCR register. 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.

When this instruction performs a bit manipulation/test on either the A or B 40-bit accumulator, it optionally shifts the accumulator value according to 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, the instruction acts on the limited value (limited on the maximum positive or negative saturation constant). In addition, the “L” flag in the SR is set accordingly.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

BSET Bit Set and Test BSET

CCR Condition Codes:

For destination operand SR:

* C This bit is set if Bit 0 is specified, and not affected otherwise.
* V This bit is set if Bit 1 is specified, and not affected otherwise.
* Z This bit is set if Bit 2 is specified, and not affected otherwise.
* N This bit is set if Bit 3 is specified, and not affected otherwise.
* U This bit is set if Bit 4 is specified, and not affected otherwise.
* E This bit is set if Bit 5 is specified, and not affected otherwise.
* L This bit is set if Bit 6 is specified, and not affected otherwise.
* S This bit is set if Bit 7 is specified, and not affected otherwise.

For other destination operands:

* C This bit is set if bit tested is set, and cleared otherwise.
* V This bit is not affected.
* Z This bit is not affected.
* N This bit is not affected.
* U This bit is not affected.
* E This bit is not affected.
* L This bit is set according to the standard definition.
* S This bit is set according to the standard definition.

MR Status Bits:

For destination operand SR:

* I0 This bit is changed if Bit 8 is specified, and not affected otherwise.
* I1 This bit is changed if Bit 9 is specified, and not affected otherwise.
* S0 This bit is changed if Bit 10 is specified, and not affected otherwise.
* S1 This bit is changed if Bit 11 is specified, and not affected otherwise.
* FV This bit is changed if Bit 12 is specified, and not affected otherwise.
* SM This bit is changed if Bit 13 is specified, and not affected otherwise.
* RM This bit is changed if Bit 14 is specified, and not affected otherwise.
* LF This bit is changed if Bit 15 is specified, and not affected otherwise.

For other destination operands: MR status bits are not affected.

BSET Bit Set and Test BSET

Instruction Formats and Opcodes:

BSET #n,[X or Y]:ea

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	1	M	M	M	R	R	R	0	S	1	0	b	b	b	b
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

BSET #n,[X or Y]:aa

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	0	a	a	a	a	a	a	0	S	1	0	b	b	b	b

BSET #n,[X or Y]:pp

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	0	p	p	p	p	p	p	0	S	1	0	b	b	b	b

BSET #n,[X or Y]:qq

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	0	q	q	q	q	q	q	0	S	1	0	b	b	b	b

BSET #n,D

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	1	D	D	D	D	D	D	0	1	1	0	b	b	b	b

Instruction Fields:

{#n}	**bbbb**	Bit number [0–15]
{ea}	**MMMRRR**	Effective Address (see **Table A-23** on page A-206)
{X/Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{aa}	**aaaaaa**	Absolute Address [0–63]
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0–$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80–$FFBF]
{D}	**DDDDDD**	Destination register [all on-chip registers] (see **Table A-27** on page A-207)

BSR Branch to Subroutine BSR

Operation:

PC → SSH;SR →SSL;PC + xxx → PC

PC → SSH;SR → SSL;PC + Rn → PC

Assembler Syntax:

BSR xxx

BSR Rn

Description: The address of the instruction immediately following the BSR instruction and the SR are pushed onto the stack. Program execution then continues at location PC + displacement. The displacement is a two's-complement 16-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement and Address Register PC Relative addressing modes can be used. The Short Displacement 9-bit data is sign-extended to form the PC relative displacement.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

		23							16	15							8	7							0
BSR	xxx	0	0	0	0	0	1	0	1	0	0	0	0	1	0	a	a	a	a	0	a	a	a	a	a

		23							16	15							8	7							0
BSR	Rn	0	0	0	0	1	1	0	1	0	0	0	1	1	R	R	R	1	0	0	0	0	0	0	0

Instruction Fields:

{xxx}	**aaaaaaaaa**	Signed PC Relative Short Displacement
{Rn}	**RRR**	Address register [R0–R7]

BTST Bit Test BTST

Operation:	**Assembler Syntax:**	
D[n] → C	BTST	#n,[XorY]:ea
D[n] → C	BTST	#n,[XorY]:aa
D[n] → C	BTST	#n,[XorY]:pp
D[n] → C	BTST	#n,[XorY]:qq
D[n] → C	BTST	#n,D

Description: Test the nth bit of the destination operand D. The state of the nth bit is stored in the Carry bit (C) of the CCR. 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	*
CCR							

* C This bit is set if bit tested is set, and cleared otherwise.
√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

SP—Stack Pointer:

For destination operand SSH:SP, decrement the SP by 1.
For other destination operands, the SPis not affected.

BTST Bit Test BTST

Instruction Formats and Opcodes:

BTST #n,[X or Y]:ea

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	1	M	M	M	R	R	R	O	S	1	0	b	b	b	b
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

BTST #n,[X or Y]:aa

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	0	a	a	a	a	a	a	0	S	1	0	b	b	b	b

BTST #n,[X or Y]:pp

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	0	p	p	p	p	p	p	0	S	1	0	b	b	b	b

BTST #n,[X or Y]:qq

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	1	q	q	q	q	q	q	0	S	1	0	b	b	b	b

BTST #n,D

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	1	D	D	D	D	D	D	0	1	1	0	b	b	b	b

Instruction Fields:

{#n}	**bbbb**	Bit number [0–15]
{ea}	**MMMRRR**	Effective Address (see **Table A-23** on page A-206)
{X/Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{aa}	**aaaaaa**	Absolute Address [0–63]
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0–$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80–$FFBF]
{D}	**DDDDDD**	Destination register [all on-chip registers] (see **Table A-27** on page A-207)

CLB Count Leading Bits CLB

Operation:

```
If S[39] = 0 then
9 – (Number of consecutive leading zeros in S[39:0]) → D[31:16]
else
 9 – (Number of consecutive leading ones in S[39:0]) → D[31:16]
```

Assembler Syntax:

CLB S,D

Description: Count leading 0s or 1s according to Bit 39 of the source accumulator. Scan bits 39–0 of the source accumulator starting from Bit 39. The MSP of the destination accumulator is loaded with nine minus the number of consecutive leading 1s or 0s found. The result is a signed integer in MSP whose range of possible values is from +8 to –31. This is a 40-bit operation. The LSP of the destination accumulator D is filled with 0s. The EXP of the destination accumulator D is sign-extended.

Notes: **1.** If the source accumulator is all 0s, the result is 0.

2. This instruction can be used in conjunction with NORMF instruction to specify the shift direction and amount needed for normalization.

Condition Codes:

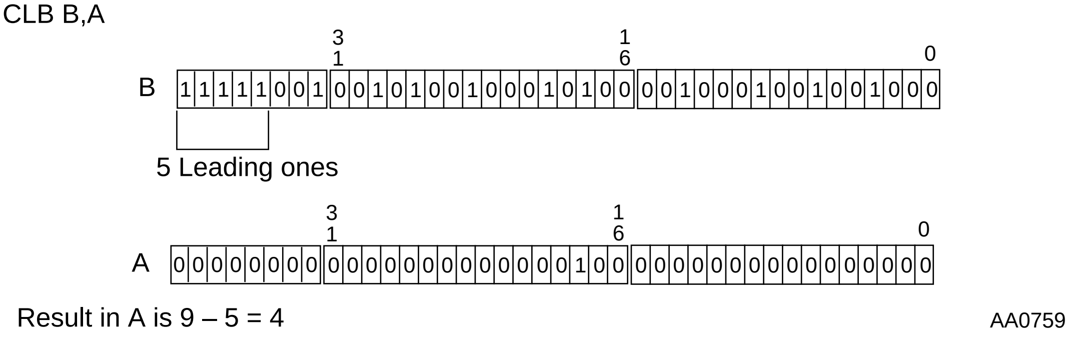
7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	*	*	*	—
CCR							

* N This bit is set if Bit 31 of the result is set, and cleared otherwise.
* Z This bit is set if bits 31–16 of the result are all 0.
* V This bit is always cleared.
— This bit is unchanged by the instruction.

CLB Count Leading Bits CLB

Example:

CLB B,A



Instruction Formats and Opcode:

	23 — 16	15 — 8	7 — 0
CLB S,D	0 0 0 0 1 1 0 0	0 0 0 1 1 1 1 0	0 0 0 0 0 0 S D

Instruction Fields:

{D}	**D**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S}	**S**	Source accumulator [A,B] (see **Table A-15** on page A-203)

CLR Clear Accumulator CLR

Operation:

0 → D (parallel move)

Assembler Syntax:

CLR D (parallel move)

Description: Clear the destination accumulator. This is a 40-bit clear instruction.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	*	*	*	*	*	—
CCR							

*	E	This bit is always cleared.
*	U	This bit is always set.
*	N	This bit is always cleared.
*	Z	This bit is always set.
*	V	This bit is always cleared.
√		This bit is changed according to the standard definition.
—		This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 16	15 8	7							0
CLR D	DATA BUS MOVE FIELD		0	0	0	1	d	0	1	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION									

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

CMP Compare CMP

Operation:

S2 – S1 (parallel move)

S2 – #xx

S2 – #xxxx

Assembler Syntax:

CMPS1, S2 (parallel move)

CMP #xx, S2

CMP #xxxx, S2

Description: Subtract the source one operand from the source two accumulator, S2, and update the CCR. The result of the subtraction operation is not stored.

The source one operand can be a register (16-bit word or 40-bit accumulator), 6-bit short immediate, or 16-bit long immediate. When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the six bits will be right-aligned and the remaining bits will be zeroed to form a 16-bit source operand.

Note: This instruction subtracts 40-bit operands. When a word is specified as the source one operand, it is sign-extended and zero-filled to form a valid 40-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 particularly applies to the case where it is extended to compare 16-bit operands, such as X0 with A1.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.

CMP Compare CMP

Instruction Formats and Opcodes:

CMP S1, S2

23 16	15 8	7 0
DATA BUS MOVE FIELD		0 J J J d 1 0 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION		

CMP #xx, S2

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

CMP #xxxx,S2

23 16	15 8	7 0
0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 0	1 1 0 0 d 1 0 1
IMMEDIATE DATA EXTENSION		

Instruction Fields:

{S1}	**JJJ**	Source one register [B/A,X0,Y0,X1,Y1] (see **Table A-29** on page A-208)
{S2}	**d**	Source two accumulator [A/B] (see **Table A-15** on page A-203)
{#xx}	**iiiiii**	6-bit Immediate Short Data
{#xxxx}		16-bit Immediate Long Data extension word

CMPM Compare Magnitude CMPM

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 CCR. The result of the subtraction operation is not stored.

Note: This instruction subtracts 40-bit operands. When a word is specified as S1, it is sign-extended and zero-filled to form a valid 40-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 particularly applies to the case where it is extended to compare 16-bit operands, such as X0 with A1.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.

Instruction Formats and Opcodes:

CMPM S1, S2

23	16 15	8	7							0
DATA BUS MOVE FIELD			0	J	J	J	d	1	1	1
OPTIONAL EFFECTIVE ADDRESS EXTENSION										

Instruction Fields:

{S1}	**JJJ**	Source one register [B/A,X0,Y0,X1,Y1] (see **Table A-29** on page A-208)
{S2}	**d**	Source two accumulator [A,B] (see **Table A-15** on page A-203)

CMPU Compare Unsigned CMPU

Operation:

S2 – S1

Assembler Syntax:

CMPU S1, S2

Description: Subtract the source one operand, S1, from the source two accumulator, S2, and update the CCR. The result of the subtraction operation is not stored.

Note: This instruction subtracts a 16- or 32-bit unsigned operand from a 32-bit unsigned operand. When a 16-bit word is specified as S1, it is aligned to the left and zero-filled to form a valid 32-bit operand. If an accumulator is specified as an operand, the value in the EXP does not affect the operation.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	√	*	*	√
CCR							

*	V	This bit is always cleared.
*	Z	This bit is set if bits 31–0 of the result are 0.
—		This bit is unchanged by the instruction.
√		This bit is changed according to the standard definition.

Instruction Formats and Opcodes:

	23							16	15							8	7							0
CMPU S1, S2	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	g	g	g	d

Instruction Fields:

{S1}	**ggg**	Source one register [A,B,X0,Y0,X1,Y1] (see **Table A-20** on page A-204)
{S2}	**d**	Source two accumulator [A,B] (see **Table A-15** on page A-203)

DEBUG Enter Debug Mode DEBUG

Operation:

Enter the Debug mode

Assembler Syntax:

DEBUG

Description: Enter the Debug mode and wait for OnCE commands.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

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

Instruction Fields: None

DEBUGcc DEBUGcc

Enter Debug Mode Conditionally

Operation:

If cc, then enter the Debug mode

Assembler Syntax:

DEBUGcc

Description: If the specified condition is true, enter the Debug mode and wait for OnCE commands. If the specified condition is false, continue with the next instruction.

The conditions that the term "cc" can specify are listed on **Table A-47** on page A-213.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7							0
DEBUGcc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	C	C	C	C

Instruction Fields:

{cc} **CCCC** Condition code (see **Table A-48** on page A-214)

DEC Decrement by One DEC

Operation:

D – 1 → D

Assembler Syntax:

DEC D

Description: Decrement by one the specified operand and store the result in the destination accumulator. One is subtracted from the LSB of D.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7							0
DEC D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	d

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

DIV Divide Iteration DIV

Operation:

```
IF    D[39]⊕S[15] = 1
      then     2 ∗ D + C + S → D
      else     2 ∗ D + C – S → D
```

Assembler Syntax:

DIV S,D

where ⊕ denotes the logical exclusive OR operator.

Description: Divide the destination operand D by the source operand S and store the result in the destination accumulator D. The 32-bit dividend must be a positive fraction that has been sign-extended to 40 bits and is stored in the full 40-bit destination accumulator D. The 16-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 LSB of the destination accumulator at the start of each DIV iteration. The formed quotient is the true quotient if the true quotient is positive. If the true quotient is negative, the formed quotient must be negated. Valid results are obtained only when $|D| < |S|$ and the operands are interpreted as fractions. This condition ensures that the magnitude of the quotient is less than 1 (i.e., a fractional quotient) and precludes division by 0.

The DIV instruction calculates one quotient bit based on the divisor and the previous partial remainder. To produce an N-bit quotient, the DIV instruction is executed N times, where N is the number of bits of precision desired in the quotient, $1 \leq N \leq 16$. Thus, for a full-precision (16-bit) quotient, sixteen DIV iterations are required. In general, executing the DIV instruction N times produces an N-bit quotient and a 32-bit remainder that has (32 – N) bits of precision and whose N MSBs are 0s. The partial remainder is not a true remainder and must be corrected due to the nonrestoring nature of the division algorithm before it can 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.

DIV Divide Iteration DIV

The DIV instruction uses a nonrestoring fractional division algorithm that consists of the following operations (see the previous **Operation** definition):

1. **Compare the source and destination operand sign bits:** An exclusive OR operation is performed on Bit 39 of the destination operand D and Bit 15 of the source operand S.
2. **Shift the partial remainder and the quotient:** The 39-bit destination accumulator D is shifted one bit to the left. The Carry bit (C) is moved into the LSB (Bit 0) of the accumulator.
3. **Calculate the next quotient bit and the new partial remainder:** The 16-bit source operand S (signed divisor) is either added to or subtracted from the Most Significant Portion (MSP) of the destination accumulator (A1 or B1), and the result is stored back into the MSP of that destination accumulator. If the result of the exclusive OR operation previously described was 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 0 (i.e., the sign bits were the same), the source operand S is subtracted from the accumulator. Because of the automatic sign extension of the 16-bit signed divisor, the addition or subtraction operation correctly sets the C bit with the next quotient bit.

For extended precision division (e.g., N-bit quotients where N > 16), the DIV instruction is no longer applicable, and a user-defined N-bit division routine is required. For more information on division algorithms, see 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.

DIV Divide Iteration DIV

Condition Codes:.Instruction Formats and Opcodes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	*	—	—	—	—	*	*
CCR							

- * L This bit is set if the Overflow bit (V) is set.
- * V This bit is set if the MSB of the destination operand is changed as a result of the instruction's left shift operation.
- * C This bit is set if Bit 39 of the result is cleared.
- — This bit is unchanged by the instruction

	23							16	15							8	7							0
DIV S,D	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	J	J	d	0	0	0

Instruction Fields:

{S}	**JJ**	Source input register [X0,X1,Y0,Y1] (see **Table A-17** on page A-203)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)

DMAC DMAC

Double-Precision Multiply-Accumulate with Right Shift

Operation:	Assembler Syntax:		
[D >> 16] ±S1 ∗ S2 → D (S1 signed, S2 signed)	DMACss	(±)S1,S2,D	(no parallel move)
[D >> 16] ±S1 ∗ S2 → D (S1 signed, S2 unsigned)	DMACsu	(±)S1,S2,D	(no parallel move)
[D >> 16] ±S1 ∗ S2 → D (S1 unsigned, S2 unsigned)	DMACuu	(±)S1,S2,D	(no parallel move)

Description: Multiply the two 16-bit source operands S1 and S2 and add / subtract the product to / from the specified 40-bit destination accumulator D, which has been previously shifted sixteen bits to the right. The multiplication can be performed on signed numbers (ss), unsigned numbers (uu), or mixed (unsigned ∗ signed, (su)). The "–" sign option is used to negate the specified product prior to accumulation. The default sign option is "+". This instruction is optimized for multiprecision multiplication support.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7							0
DMAC (±)S1,S2,D	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	s	1	s	d	k	Q	Q	Q	Q

DMAC DMAC

Double-Precision Multiply-Accumulate with Right Shift

Instruction Fields:

{S1,S2}	**QQQQ**	Source registers S1,S2 [all combinations of X0,X1,Y0, and Y1] (see **Table A-35** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
{ss,su,uu}	**ss**	[ss,su,uu] (see **Table A-45** on page A-213)

DO Start Hardware Loop DO

Operation:	Assembler Syntax:
SP + 1 → SP;LA → SSH;LC → SSL;[X or Y]:ea → LC SP + 1 → SP;PC → SSH;SR → SSL;expr – 1 → LA 1 → LF	DO [X or Y]:ea,expr
SP + 1 → SP;LA → SSH;LC → SSL;[X or Y]:aa → LC SP+1 → SP;PC → SSH;SR → SSL;expr – 1 → LA 1 → LF	DO [X or 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 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 the initial value of LC is 0, the DO loop is not executed. All address register indirect addressing modes can be used to generate the effective address of the source operand. If immediate short data is specified, the twelve LSBs of the LC register are loaded with the 12-bit immediate value, and the twelve MSBs of the LC register 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 that permits the nesting of DO loops. The DO instruction's destination operand (shown as "expr") is then loaded into the LA register. This 16-bit operand is located in the instruction's 16-bit absolute address extension word, as shown in the opcode section. The value in the PC 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 (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 LC is tested. If the LC is not equal to 1, it is decremented by one and SSH is loaded into the PC to fetch the first instruction in the loop again. When LC = 1, 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.

During the "end-of-loop" processing, the Loop Flag (LF) from the lower portion (SSL) of the Stack Pointer is written into the SR, the contents of the LA register are restored from the upper portion (SSH) of (SP – 1), the contents of LC are restored from the lower portion (SSL) of (SP – 1), and the Stack Pointer is decremented by two. Instruction fetches continue at the address of the instruction following the last instruction in the DO loop. Note that LF is the only bit in the SR that is restored after a hardware DO loop has been exited.

Notes:

1. 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 1. 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.

2. The Loop Flag (LF) is cleared by a hardware reset.

DO Start Hardware Loop DO

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	—	—	—	—	—	—
CCR							

*	S	This bit is set if the instruction sends A/B accumulator contents to XDB or YDB.
*	L	This bit is set if data limiting occurred [see Note].
—		This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

DO [X or Y]:ea, expr

23							16	15							8	7							0
0	0	0	0	0	1	1	0	0	1	M	M	M	R	R	R	0	S	0	0	0	0	0	0
ABSOLUTE ADDRESS EXTENSION WORD																							

DO [X or Y]:aa, expr

23							16	15							8	7							0
0	0	0	0	0	1	1	0	0	0	a	a	a	a	a	a	0	S	0	0	0	0	0	0
ABSOLUTE ADDRESS EXTENSION WORD																							

DO #xxx, expr

23							16	15							8	7							0
0	0	0	0	0	1	1	0	i	i	i	i	i	i	i	i	1	0	0	0	h	h	h	h
ABSOLUTE ADDRESS EXTENSION WORD																							

DO S, expr

23							16	15							8	7							0
0	0	0	0	0	1	1	0	1	1	D	D	D	D	D	D	0	0	0	0	0	0	0	0
ABSOLUTE ADDRESS EXTENSION WORD																							

DO Start Hardware Loop DO

Instruction Fields:

{ea}	**MMMRRR**	Effective Address (see **Table A-24** on page A-206)
{X/Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{expr}		16-bit Absolute Address in 16-bit extension word
{aa}	**aaaaaa**	Absolute Address [0–63]
{#xxx}	**hhhhiiiiiiii**	Immediate Short Data [0–4095]
{S}	**DDDDDD**	Source register [all on-chip registers, **except SSH**] (see **Table A-27** on page A-207)

Note: For the DO SP, expr instruction, the actual value that is 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 loads the LC with the value LC = 4. For the DO SSL, expr instruction, the LC is loaded with its previous value, which was saved on the stack by the DO instruction itself.

DO FOREVER DO FOREVER

Start Infinite Loop

Operation:

SP + 1 → SP;LA → SSH;LC → SSL
SP + 1 → SP;PC → SSH;SR → SSL;expr – 1 → LA
1 → LF; 1 →FV

Assembler Syntax:

DO FOREVER,expr

Description: Begin a hardware DO loop that is to be repeated forever and whose range of execution is terminated by the destination operand (shown above as "expr"). No overhead other than the execution of this DO FOREVER instruction is required to set up this loop. DO FOREVER loops can be nested with other types of instructions. 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 LC register is pushed onto the stack, but is not updated by this instruction.

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. Stacking the LA, LC, PC, and SR registers permits nesting DO FOREVER loops. The DO FOREVER instruction's destination operand (shown as "expr") is then loaded into the LA register. This 16-bit operand is located in the instruction's 16-bit absolute address extension word, as shown in the opcode section. The value in the PC register pushed onto the system stack is the address of the first instruction following the DO FOREVER instruction (i.e., the first actual instruction in the DO FOREVER loop). This value is read (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) and the Forever flag are set. This results in the PC being repeatedly compared with LA to determine if the last instruction in the loop has been fetched. When LA equals PC, it indicates that the last instruction in the loop has been fetched and SSH is loaded into the PC to fetch the first instruction in the loop again. The LC register is then decremented by one without being tested. This register can be used by the programer to count the number of loops already executed.

When executing a DO FOREVER loop, the instructions are actually fetched each time through the loop. Therefore, a DO FOREVER loop can be interrupted. DO FOREVER loops can also be nested. When DO FOREVER 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 FOREVER loops are improperly nested.

DO FOREVER DO FOREVER

Start Infinite Loop

Notes:

1. 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.

2. The LC register is never tested by the DO FOREVER instruction, and the only way of terminating the loop process is to use either the ENDDO or BRKcc instructions. LC is decremented every time PC = LA so that it can be used by the programmer to keep track of the number of times the DO FOREVER loop has been executed. If the programer wants to initialize LC to a particular value before the DO FOREVER, care should be taken to save it before if the DO loop is nested. If so, LC should also be restored immediately after exiting the nested DO FOREVER loop.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 … 16	15 … 8	7 … 0
DO FOREVER	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 1 1
	ABSOLUTE ADDRESS EXTENSION WORD		

Instruction Fields:

None

ENDDO End Current DO Loop ENDDO

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 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 LC registers are then restored from the system stack.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

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

Instruction Fields:

None

EOR Logical Exclusive OR EOR

Operation:

$S \oplus D[31:16] \rightarrow D[31:16]$ (parallel move)

$\#xx \oplus D[31:16] \rightarrow D[31:16]$

$\#xxxx \oplus D[31:16] \rightarrow D[31:16]$

where $\oplus$ denotes the logical XOR operator

Assembler Syntax:

EOR S,D (parallel move)

EOR #xx,D

EOR #xxxx,D

Description: Logically exclusive OR the source operand S with bits 31–16 of the destination operand D and store the result in bits 31–16 of the destination accumulator. The source can be a 16-bit register, 6-bit short immediate or 16-bit long immediate. This instruction is a 16-bit operation. The remaining bits of the destination operand D are not affected.

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the six bits are right-aligned, then the remaining bits are zeroed to form a 16-bit source operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	*	*	*	—
CCR							

* N This bit is set if Bit 31 of the result is set.
* Z This bit is set if bits 31–16 of the result are 0.
* V This bit is always cleared.
√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

EOR Logical Exclusive OR EOR

Instruction Formats and Opcodes:

EOR S,D

23							16	15							8	7							0
DATA BUS MOVE FIELD																0	1	J	J	d	0	1	1
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

EOR #xx,D

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

EOR #xxxx,D

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0	d	0	1	1
IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	**JJ**	Source register [X0,X1,Y0,Y1] (see **Table A-17** on page A-203)
{D}	**d**	Destination accumulator [A/B] (see **Table A-15** on page A-203)
{#xx}	**iiiiii**	6-bit Immediate Short Data
{#xxxx}		16-bit Immediate Long Data extension word

EXTRACT Extract Bit Field EXTRACT

Operation:

Offset = S1[5:0]
Width = S1[13:8]

S2[(offset + width – 1):offset] → D[(width – 1):0]
S2[offset + width – 1] → D[39:width] (sign extension)

Offset = #CO[5:0]
Width = #CO[13:8]

S2[(offset + width – 1):offset] → D[(width – 1):0]
S2[offset + width – 1] → D[39:width] (sign extension)

Assembler Syntax:

EXTRACT S1,S2,D

EXTRACT #CO,S2,D

Description: Extract a bit-field from source accumulator S2. The bit-field width is specified by bits 13–8 in the S1 register or in the immediate control word #CO. The offset from the Least Significant Bit is specified by bits 5–0 in the S1 register or in the immediate control word #CO. The extracted field is placed in the destination accumulator D, aligned to the right. The construction of the control register can be done by using the MERGE instruction.

This is a 40-bit operation. Bits outside the field are filled with sign extension according to the Most Significant Bit of the extracted bit field.

Note: If offset + width exceeds the value of 40, the result is undefined.

Condition Codes:

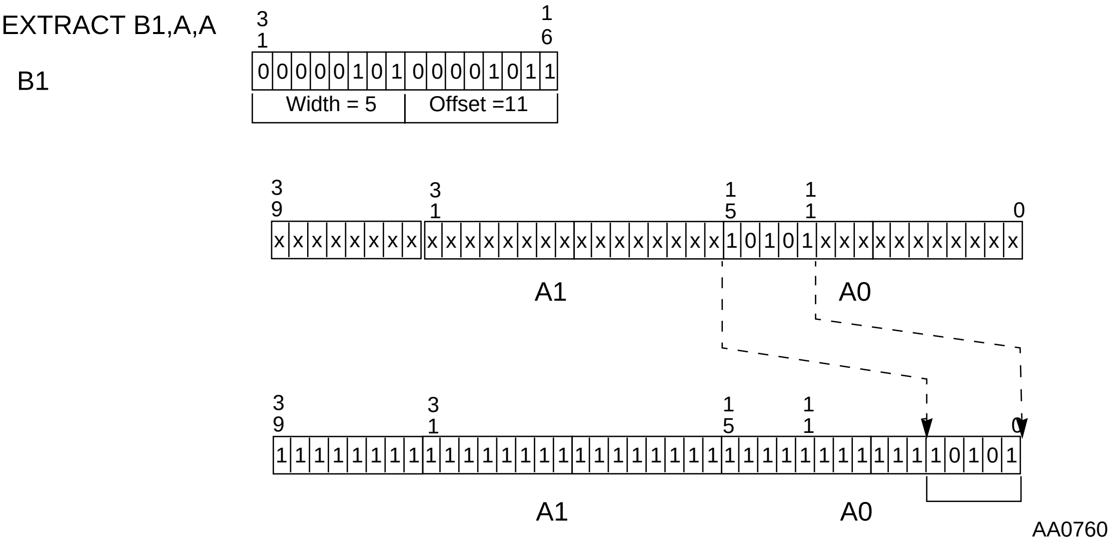
7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	√	√	√	√	*	*
CCR							

*	V	This bit is always cleared.
*	C	This bit is always cleared.
—		This bit is unchanged by the instruction.
√		This bit is changed according to the standard definition.

EXTRACT Extract Bit Field EXTRACT

Example:

EXTRACT B1,A,A



Instruction Formats and Opcodes:

	23							16	15							8	7							0
EXTRACT S1,S2,D	0	0	0	0	1	1	0	0	0	0	0	1	1	0	1	0	0	0	0	s	S	S	S	D
EXTRACT #CO,S2,D	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	s	0	0	0	D
	CONTROL WORD EXTENSION																							

Instruction Fields:

{S2}	**s**	Source accumulator [A,B] (see **Table A-15** on page A-203)
{D}	**D**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S1}	**SSS**	Control register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)
{#CO}		Control word extension.

EXTRACTU EXTRACTU

Extract Unsigned Bit Field

Operation: **Assembler Syntax:**

Offset = S1[5:0]
Width = S1[13:8]

EXTRACTU S1,S2,D

S2[(offset + width – 1):offset] → D[(width – 1):0]
zero → D[39:width]

Offset = #CO[5:0]
Width = #CO[13:8]

EXTRACTU #CO,S2,D

S2[(offset + width – 1):offset] → D[(width – 1):0]
zero fi D[39:width]

Description: Extract an unsigned bit-field from source accumulator S2. The bit-field width is specified by bits 13–8 in the S1 register or in the immediate control word #CO. The offset from the LSB is specified by bits 5–0 in the S1 register or in the immediate control word #CO. The extracted field is placed in the destination accumulator D, aligned to the right. The construction of the control register can be done by using the MERGE instruction.

This is a 40-bit operation. Bits outside the field are filled with 0s.

Note: If offset + width exceeds the value of 40, the result is undefined.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	√	√	√	√	*	*
CCR							

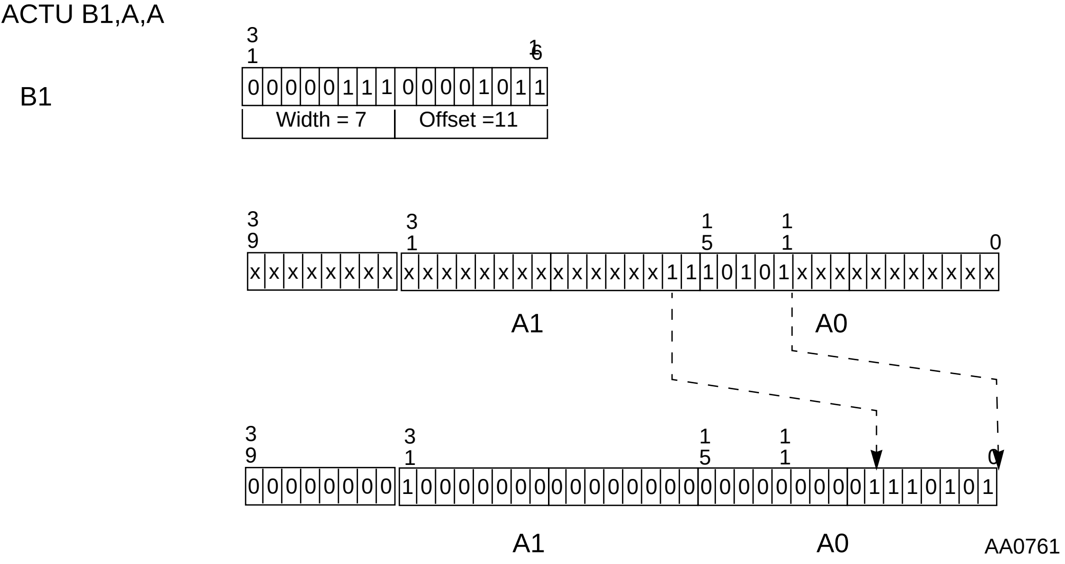
* V This bit is always cleared.
* C This bit is always cleared.
— This bit is unchanged by the instruction.
√ This bit is changed according to the standard definition.

EXTRACTU EXTRACTU

Extract Unsigned Bit Field

Example:

EXTRACTU B1,A,A



Instruction Formats and Opcodes:

	23							16	15							8	7							0
EXTRACTUS1,S2,D	0	0	0	0	1	1	0	0	0	0	0	1	1	0	1	0	1	0	0	s	S	S	S	D

	23							16	15							8	7							0
EXTRACTU#CO,S2,D	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	1	0	0	s	0	0	0	D
	CONTROL WORD EXTENSION																							

Instruction Fields:

{S2}	**s**	Source accumulator [A,B] (see **Table A-15** on page A-203)
{D}	**D**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S1}	**SSS**	Control register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)
{#CO}		Control word extension

IFcc Execute Conditionally without CCR Update IFcc

Operation:
If cc, then opcode operation

Assembler Syntax:
Opcode-Operands IFcc

Description: If the specified condition is true, execute and store result of the specified Data ALU operation. If the specified condition is false, no destination is altered. The CCR is never updated with the condition codes generated by the Data ALU operation.

The instructions that can conditionally be executed by using IFcc are the arithmetic and logical instructions that are considered as parallel instructions. See **Table A-3** on page A-9 and **Table A-4** on page A-11 for a list of those instructions.

The conditions that the term "cc" can specify are listed on **Table A-47** on page A-213.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7 0
IFcc	0	0	1	0	0	0	0	0	0	0	1	0	C	C	C	C	INSTRUCTION OPCODE

Instruction Fields:

{cc} **CCCC** Condition code (see **Table A-48** on page A-214)

IFcc.U Execute Conditionally with CCR Update IFcc.U

Operation:
If cc, then opcode operation

Assembler Syntax:
Opcode-Operands IFcc

If the specified condition is true, execute and store result of the specified Data ALU operation and update the CCR with the status information generated by the Data ALU operation. If the specified condition is false, no destination is altered and the CCR is not affected.

The instructions that can conditionally be executed by using IFcc.U are the arithmetic and logical instructions that are considered as parallel instructions. See **Table A-3** on page A-9 and **Table A-4** on page A-11 for a list of those instructions.

The conditions that the term "cc" can specify are listed on **Table A-47** on page A-213

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

* If the specified condition is true, this bit is changed according to the instruction. Otherwise, it is not changed.

Instruction Formats and Opcodes:

	23							16	15							8	7 0
IFcc.U	0	0	1	0	0	0	0	0	0	0	1	1	C	C	C	C	INSTRUCTION OPCODE

Instruction Fields:

{cc} **CCCC** Condition code (see **Table A-48** on page A-214)

ILLEGAL Illegal Instruction Interrupt ILLEGAL

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. Exiting an illegal instruction is a fatal error. A long exception routine should be used to 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 is 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, and other instructions at LA are restricted. 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 returns 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

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

Instruction Fields: None

INC Increment by One INC

Operation:

D + 1 → D

Assembler Syntax:

INC D

Description: Increment by one the specified operand and store the result in the destination accumulator. One is added from the LSB of D.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and opcodes:

	23							16	15							8	7							0
INC D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	d

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

INSERT Insert Bit Field INSERT

Operation: **Assembler Syntax:**

Offset = S1[5:0]
Width = S1[13:8]

INSERT S1,S2,D

S2[(width – 1):0] → D[(offset + width – 1):offset]

Offset = #CO[5:0]
Width = #CO[13:8]

INSERT #CO,S2,D

S2[(width-1):0] → D[(offset + width – 1):offset]

Description: Insert a bit-field into the destination accumulator D. The bit-field whose width is specified by bits 13–8 in S1 register begins at the LSB of the S2 register. This bit-field is inserted in the destination accumulator D, with an offset according to bits 5–0 in the S1 register. The S1 operand can be an immediate control word #CO. The width specified by S1 should not exceed a value of 16. The construction of the control register can be done by using the MERGE instruction.

This is a 40-bit operation. Any bits outside the field remain unchanged.

Note: If offset plus width exceeds the value of 40, the result is undefined.

Condition Codes:

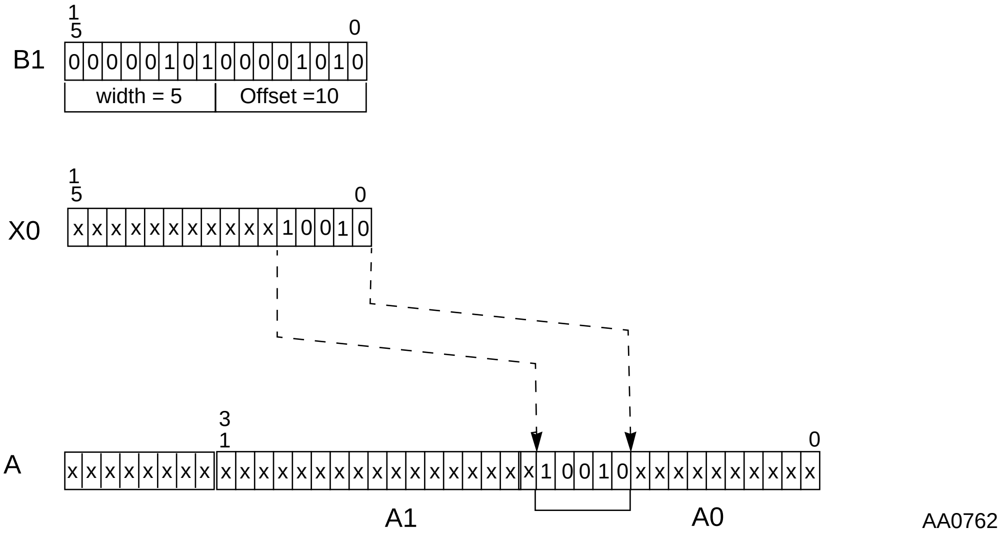
7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	√	√	√	√	*	*
CCR							

* V This bit is always cleared.
* C This bit is always cleared.
— This bit is unchanged by the instruction.
√ This bit is changed according to the standard definition.

INSERT Insert Bit Field INSERT

Example:

INSERT B1,X0,A


B1
15
0
0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0
width = 5
Offset =10
X0
15
0
x x x x x x x x x x x 1 0 0 1 0
A
31
0
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x 1 0 0 1 0 x x x x x x x x x x
A1
A0
AA0762

Instruction Formats and Opcodes:

		23							16	15							8	7							0
INSERT	S1,S2,D	0	0	0	0	1	1	0	0	0	0	0	1	1	0	1	1	0	q	q	q	S	S	S	D

		23							16	15							8	7							0
INSERT	#CO,S2,D	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	1	0	q	q	q	0	0	0	D
		CONTROL WORD EXTENSION																							

Instruction Fields:

{D}	**D**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S1}	**SSS**	Control register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)
{S2}	**qqq**	Source register [X0,X1,Y0,Y1,A0,B0] (see **Table A-20** on page A-204)
{#CO}		Control word extension

Jcc Jump Conditionally Jcc

Operation:	**Assembler Syntax:**
If cc, then 0xxx → PC else PC + 1 → PC	Jcc xxx
If cc, then ea → PC else PC + 1 → PC	Jcc ea

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 can be used for the effective address. A Fast Short Jump addressing mode can also be used. The 12-bit data is zero-extended to form the effective address.

The conditions that the term "cc" can specify are listed on **Table A-47** on page A-213.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7							0
Jcc xxx	0	0	0	0	1	1	1	0	C	C	C	C	a	a	a	a	a	a	a	a	a	a	a	a

	23							16	15							8	7							0
Jcc ea	0	0	0	0	1	0	1	0	1	1	M	M	M	R	R	R	1	0	1	0	C	C	C	C
	OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

Jcc Jump Conditionally Jcc

Instruction Fields:

{cc}	CCCC	Condition code (see **Table A-48** on page A-214)
{xxx}	aaaaaaaaaaaa	Short Jump Address
{ea}	MMMRRR	Effective Address (see **Table A-23** on page A-206)

JCLR Jump if Bit Clear JCLR

Operation:						Assembler Syntax:	
If	S{n} = 0then	xxxx	→	PC		JCLR	#n,[X or Y]:ea,xxxx
	else	PC + 1	→	PC			
If	S{n} = 0then	xxxx	→	PC		JCLR	#n,[X or Y],aa,xxxx
	else	PC + 1	→	PC			
If	S{n} = 0then	xxxx	→	PC		JCLR	#n,[X or Y]:pp,xxxx
	else	PC + 1	→	PC			
If	S{n} = 0then	xxxx	→	PC		JCLR	#n,[X or Y]:qq,xxxx
	else	PC + 1	→	PC			
If	S{n} = 0then	xxxx	→	PC		JCLR	#n,S,xxxx
	else	PC + 1	→	PC			

Description: Jump to the 16-bit absolute address in program memory specified in the instruction's 16-bit extension word if the nth 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 nth bit. All address register indirect addressing modes can be used to reference the source operand S. Absolute Short and I/O Short addressing modes can also be used.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

JCLR Jump if Bit Clear JCLR

Instruction Formats and Opcodes:

JCLR #n,[X or Y]:ea,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	1	M	M	M	R	R	R	1	S	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JCLR #n,[X or Y]:aa,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	0	a	a	a	a	a	a	1	S	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JCLR #n,[X or Y]:pp,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	0	p	p	p	p	p	p	1	S	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JCLR #n,[X or Y]:qq,xxxx

23							16	15							8	7							0
0	0	0	0	0	0	0	1	1	0	q	q	q	q	q	q	1	S	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JCLR #n,S,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	1	D	D	D	D	D	D	0	0	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

Instruction Fields:

{#n}	**bbbb**	Bit number [0–15]
{ea}	**MMMRRR**	Effective Address (see **Table A-24** on page A-206)
{X/Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{xxxx}		16-bit absolute Address extension word
{aa}	**aaaaaa**	Absolute Address [0–63]
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0–$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80–$FFBF]
{S}	**DDDDDD**	Source register [all on-chip registers] (see **Table A-27** on page A-207)

JMP Jump JMP

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 can be used for the effective address. A Fast Short Jump addressing mode can also be used. The 12-bit data is zero-extended to form the effective address.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

JMP ea

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	1	M	M	M	R	R	R	1	0	0	0	0	0	0	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

JMP xxx

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

Instruction Fields:

{xxx}	**aaaaaaaaaaaa**	Short Jump Address
{ea}	**MMMRRR**	Effective Address (see **Table A-23** on page A-206)

JScc Jump to Subroutine Conditionally JScc

Operation: **Assembler Syntax:**

If cc, then SP + 1 → SP; PC → SSH;SR → SSL;0xxx → PC JScc xxx
else PC + 1 → PC

If cc, then SP + 1 → SP; PC → SSH;SR → SSL;ea → PC JScc ea
else PC + 1 → PC

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 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 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 can be used for the effective address. A fast short jump addressing mode can also be used. The 12-bit data is zero-extended to form the effective address.

The conditions that the term “cc” can specify are listed on **Table A-47** on page A-213.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

		23							16	15							8	7							0
JScc	xxx	0	0	0	0	1	1	1	1	C	C	C	C	a	a	a	a	a	a	a	a	a	a	a	a

		23							16	15							8	7							0
JScc	ea	0	0	0	0	1	0	1	1	1	1	M	M	M	R	R	R	1	0	1	0	C	C	C	C
		OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

JScc Jump to Subroutine Conditionally JScc

Instruction Fields:

{cc}	CCCC	Condition code (see **Table A-48** on page A-214)
{xxx}	aaaaaaaaaaaa	Short Jump Address
{ea}	MMMRRR	Effective Address (see **Table A-23** on page A-206)

JSCLR Jump to Subroutine if Bit Clear JSCLR

Operation:	Assembler Syntax:

If S{n} = 0 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx → PC
else PC + 1 → PC
	JSCLR #n,[X or Y]:ea,xxxx

If S{n} = 0 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx → PC
else PC + 1 → PC
	JSCLR #n,[X or Y],aa,xxxx

If S{n} = 0 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx → PC
else PC + 1 → PC
	JSCLR #n,[X or Y]:pp,xxxx

If S{n} = 0 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx → PC
else PC + 1 → PC
	JSCLR #n,[X or Y]:qq,xxxx

If S{n} = 0 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx fiPC
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 16-bit extension word if the nth 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 nth bit of the source operand S is clear, the address of the instruction immediately following the JSCLR instruction (PC) and the SR are pushed onto the system stack. Program execution then continues at the specified absolute address in the instruction’s 16-bit extension word. If the specified memory bit is not clear, the PC is incremented and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the nth bit. All address register indirect addressing modes can be used to reference the source operand S. Absolute short and I/O short addressing modes can also be used.

JSCLR Jump to Subroutine if Bit Clear JSCLR

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

JSCLR #n,[X or Y]:ea,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	1	M	M	M	R	R	R	1	S	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSCLR #n,[X or Y]:aa,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	0	a	a	a	a	a	a	1	S	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSCLR #n,[X or Y]:pp,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	0	p	p	p	p	p	p	1	S	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSCLR #n,[X or Y]:qq,xxxx

23							16	15							8	7							0
0	0	0	0	0	0	0	1	1	1	q	q	q	q	q	q	1	S	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSCLR #n,S,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	1	D	D	D	D	D	D	0	0	0	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSCLR Jump to Subroutine if Bit Clear JSCLR

Instruction Fields:

{#n}	**bbbb**	Bit number [0–15]
{ea}	**MMMRRR**	Effective Address (see **Table A-24** on page A-206)
{X/Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{xxxx}		16-bit absolute Address extension word
{aa}	**aaaaaa**	Absolute Address [0–63]
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0–$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80–$FFBF]
{S}	**DDDDDD**	Source register [all on-chip registers] (see **Table A-27** on page A-207)

JSET Jump if Bit Set JSET

Operation:	Assembler Syntax:
If S{n} = 1 then xxxx → PC else PC + 1 → PC	JSET #n,[X or Y]:ea,xxxx
If S{n} = 1 then xxxx → PC else PC + 1 → PC	JSET #n,[X or Y],aa,xxxx
If S{n} = 1 then xxxx → PC else PC + 1 → PC	JSET #n,[X or Y]:pp,xxxx
If S{n} = 1 then xxxx → PC else PC + 1 → PC	JSET #n,[X or Y]:qq,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 16-bit extension word if the nth 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 nth bit. All address register indirect addressing modes can be used to reference the source operand S. Absolute short and I/O short addressing modes can also be used.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√	This bit is changed according to the standard definition.
—	This bit is unchanged by the instruction.

JSET Jump if Bit Set JSET

Instruction Formats and Opcodes:

JSET #n,[X or Y]:ea,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	1	M	M	M	R	R	R	1	S	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSET #n,[X or Y]:aa,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	0	a	a	a	a	a	a	1	S	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSET #n,[X or Y]:pp,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	0	p	p	p	p	p	p	1	S	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSET #n,[X or Y]:qq,xxxx

23							16	15							8	7							0
0	0	0	0	0	0	0	1	1	0	q	q	q	q	q	q	1	S	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSET #n,S,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	0	1	1	D	D	D	D	D	D	0	0	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

Instruction Fields:

{#n}	**bbbb**	Bit number [0–15]
{ea}	**MMMRRR**	Effective Address (see **Table A-24** on page A-206)
{X/Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{xxxx}		16-bit Absolute Address in extension word
{aa}	**aaaaaa**	Absolute Address [0–63]
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0–$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80–$FFBF]
{S}	**DDDDDD**	Source register [all on-chip registers] (see **Table A-27** on page A-207)

JSR Jump to Subroutine JSR

Operation:

SP + 1 → SP; PC → SSH; SR → SSL; 0xxx → PC

SP + 1 → SP; PC → SSH; SR → SSL; ea → PC

Assembler Syntax:

JSR xxx

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 can be used for the effective address. A fast short jump addressing mode can also be used. The 12-bit data is zero-extended to form the effective address.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and opcodes:

JSR ea

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	1	M	M	M	R	R	R	1	0	0	0	0	0	0	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

JSR xxx

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

Instruction Fields:

{xxx}	**aaaaaaaaaaaa**	Short Jump Address
{ea}	**MMMRRR**	Effective Address (see **Table A-23** on page A-206)

JSSET Jump to Subroutine if Bit Set JSSET

Operation:	Assembler Syntax:

If S{n} = 1 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx → PC
else PC + 1 → PC
	JSSET #n,[X or Y]:ea,xxxx

If S{n}=1 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx → PC
else PC + 1 → PC
	JSSET #n,[X or Y],aa,xxxx

If S{n}=1 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx → PC
else PC + 1 → PC
	JSSET #n,[X or Y]:pp,xxxx

If S{n}=1 then SP + 1 → SP;PC → SSH;SR → SSL;
;xxxx → PC
else PC + 1 → PC
	JSSET #n,[X or Y]:qq,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 16-bit extension word if the nth 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 nth 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 16-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 nth bit. All address register indirect addressing modes can be used to reference the source operand S. Absolute short and I/O short addressing modes can also be used.

JSSET Jump to Subroutine if Bit Set JSSET

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

JSSET #n,[X or Y]:ea,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	1	M	M	M	R	R	R	1	S	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSSET #n,[X or Y]:aa,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	0	a	a	a	a	a	a	1	S	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSSET #n,[X or Y]:pp,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	0	p	p	p	p	p	p	1	S	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSSET #n,[X or Y]:qq,xxxx

23							16	15							8	7							0
0	0	0	0	0	0	0	1	1	1	q	q	q	q	q	q	1	S	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSSET #n,S,xxxx

23							16	15							8	7							0
0	0	0	0	1	0	1	1	1	1	D	D	D	D	D	D	0	0	1	0	b	b	b	b
ABSOLUTE ADDRESS EXTENSION																							

JSSET Jump to Subroutine if Bit Set JSSET

Instruction Fields:

{#n}	**bbbb**	Bit number [0–15]
{ea}	**MMMRRR**	Effective Address (see **Table A-24** on page A-206)
{X/Y}	**S**	Memory Space [X,Y] (see **Table A-22** on page A-205)
{xxxx}		16-bit PC absolute Address extension word
{aa}	**aaaaaa**	Absolute Address [0–63]
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0–$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80–$FFBF]
{S}	**DDDDDD**	Source register [all on-chip registers] (see **Table A-27** on page A-207)

LRA Load PC Relative Address LRA

Operation:

PC + Rn → D

PC + xxxx → D

Assembler Syntax:

LRA Rn,D

LRA xxxx,D

Description: The PC is added to the specified displacement and the result is stored in destination D. The displacement is a two's-complement 16-bit integer that represents the relative distance from the current PC to the destination PC. Long Displacement and Address Register PC Relative addressing modes can be used. Note that if D is SSH, the SP is pre-incremented by one.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

LRA Rn,D

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

LRA xxxx,D

23							16	15							8	7							0
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	d	d	d	d	d
LONG DISPLACEMENT																							

Instruction Fields:

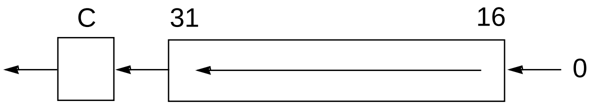
{Rn} **RRR** Address register [R0–R7]

{D} **ddddd** Destination address register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0–R7,N0–N7] (see **Table A-36** on page A-210)

{xxxx} 16-bit PC Long Displacement

LSL Logical Shift Left LSL

Operation:



AA0763

Assembler Syntax:

LSL D (parallel move)
LSL #ii,D
LSL S,D

Description:

- Single-bit shift:

  Logically shift Bits 31–16 of the destination operand D one bit to the left and store the result in the destination accumulator. Prior to instruction execution, Bit 31 of D is shifted into the carry bit C, and a 0 is shifted into Bit 16 of the destination accumulator D.

- Multi-bit shift:

  The contents of bits 31–16 of the destination accumulator D are shifted left #ii bits. Bits shifted out of position 31 are lost, except for the last bit that is latched in the Carry bit. Zeros are supplied to the vacated positions on the right. The result is placed into bits 31–16 of the destination accumulator D. The number of bits to shift is determined by the 5-bit immediate field in the instruction, or by the unsigned integer located in the control register S. If a zero shift count is specified, the carry bit is cleared.

This is a 16-bit operation. The remaining bits of the destination accumulator are not affected.

Note: The number of shifts should not exceed the value of sixteen.

LSL Logical Shift Left LSL

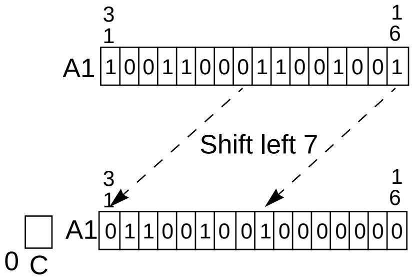
Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	*	*	*	*
CCR							

* N This bit is set if Bit 31 of the result is set.
* Z This bit is set if bits 31–16 of the result are 0.
* V This bit is always cleared.
* C This bit is set if the last bit shifted out of the operand is set, cleared for a shift count of 0, and cleared otherwise.
.
√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Example:

LSL #7, A



AA0764

Instruction Formats and Opcodes:

LSL D

23 … 8	7							0
DATA BUS MOVE FIELD	0	0	1	1	D	0	1	1
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

LSL #ii,D

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

LSL S,D

23							16	15							8	7							0
0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	s	s	s	D

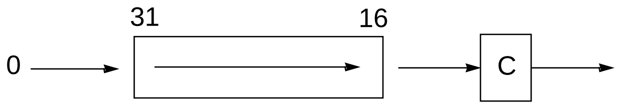
LSL Logical Shift Left LSL

Instruction Fields:

{D}	D	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S}	sss	Control register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)
{#ii}	iiiii	5-bit unsigned integer [0–16] denoting the shift amount

LSR Logical Shift Right LSR

Operation:



AA0765

Assembler Syntax:

LSR D (parallel move)
LSR #ii,D
LSR S,D

Description:

- Single-bit shift:

  Logically shift bits 31–16 of the destination operand D one bit to the right and store the result in the destination accumulator. Prior to instruction execution, Bit 16 of D is shifted into the Carry bit (C), and a 0 is shifted into Bit 31 of the destination accumulator D.

- Multi-bit shift:

  The contents of bits 31–16 of the destination accumulator D are shifted right #ii bits. Bits shifted out of position 16 are lost except for the last bit that is latched in the C bit. Zeroes are supplied to the vacated positions on the left. The result is placed into bits 31–16 of the destination accumulator D. The number of bits to shift is determined by the 5-bit immediate field in the instruction, or by the unsigned integer located in the control register S. If a zero shift count is specified, the C bit is cleared.

This is a 16-bit operation. The remaining bits of the destination register are not affected.

Note: The number of shifts should not exceed the value of sixteen.

LSR Logical Shift Right LSR

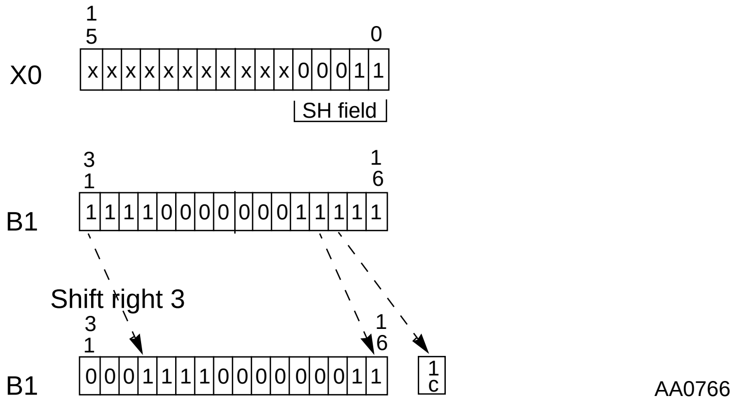
Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	*	*	*	*
CCR							

*	N	This bit is set if Bit 31 of the result is set.
*	Z	This bit is set if Bits 31–16 of the result are 0.
*	V	This bit is always cleared.
*	C	This bit is set if the last bit shifted out of the operand is set, cleared for a shift count of zero, and cleared otherwise. .
√		This bit is changed according to the standard definition.
—		This bit is unchanged by the instruction.

Example:

LSR X0,B



LSR Logical Shift Right LSR

Instruction Formats and Opcodes:

LSR D

<table>
<tr><td>23</td><td colspan="14"></td><td>8</td><td>7</td><td colspan="6"></td><td>0</td></tr>
<tr><td colspan="16">DATA BUS MOVE FIELD</td><td>0</td><td>0</td><td>1</td><td>0</td><td>D</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td colspan="24">OPTIONAL EFFECTIVE ADDRESS EXTENSION</td></tr>
</table>

LSR #ii,D

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

LSR S,D

23							16	15							8	7							0
0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0	1	1	s	s	s	D

Instruction Fields:

{D}	**D**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S}	**sss**	Control register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)
{#ii}	**iiiii**	5-bit unsigned integer [0–16] denoting the shift amount

LUA Load Updated Address LUA

Operation:	**Assembler Syntax:**
ea → D (No update performed)	LUA ea,D
Rn + aa → D	LUA (Rn + aa),D
ea → D (No update performed)	LEA ea,D
Rn + aa → D	LEA (Rn + aa),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). Only the following addressing modes can be used: Post + N, Post – N, Post + 1, Post – 1.

Note: The source address register specified in the effective address is not updated. This is the only case where an address register is not updated, although stated otherwise in the effective address mode bits.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

	23							16	15							8	7							0
LUA/L ea,D EA	0	0	0	0	0	1	0	0	0	1	0	M	M	R	R	R	0	0	0	d	d	d	d	d

	23							16	15							8	7							0
LUA/L (Rn + aa),D EA	0	0	0	0	0	1	0	0	0	0	a	a	a	R	R	R	a	a	a	a	d	d	d	d

Note: LEA is a synonym for LUA. The simulator on-line disassembly translates the opcodes into LUA.

LUA Load Updated Address LUA

Instruction Fields:

{ea}	MMRRR	Effective address (see **Table A-25** on page A-206)
{D}	ddddd	Destination address register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0–R7,N0–N7] (see **Table A-36** on page A-210)
{D}	dddd	Destination address register [R0–R7,N0–N7] (see **Table A-30** on page A-208)
{aa}	aaaaaaa	7-bit sign extended short displacement address
{Rn}	RRR	Source address register [R0–R7]

Note: RRR refers to a source address register (R0–R7), while dddd/ddddd refer to a destination address register (R0–R7 or N0–N7).

MAC Signed Multiply-Accumulate MAC

Operation:	**Assembler Syntax:**
$D \pm S1 * S2 \rightarrow D$ (parallel move)	MAC (±)S1,S2,D (parallel move)
$D \pm S1 * S2 \rightarrow D$ (parallel move)	MAC (±)S2,S1,D (parallel move)
$D \pm (S1 * 2^{-n}) \rightarrow D$ (**no** parallel move)	MAC (±)S,#n,D (**no** parallel move)

Description: Multiply the two signed 16-bit source operands S1 and S2 (**or** the signed 16-bit source operand S by the positive 16-bit immediate operand 2^{-n}) and add/subtract the product to/from the specified 40-bit destination accumulator D. The "–" sign option is used to negate the specified product prior to accumulation. The default sign option is "+".

Note: When the processor is in the Double Precision Multiply mode, the following instructions do not execute in the normal way and should only be used as part of the double precision multiply algorithm:

MAC X1, Y0, AMAC X1, Y0, B

MAC X0, Y1, AMAC X0, Y1, B

MAC Y1, X1, AMAC Y1, X1, B

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

MAC Signed Multiply-Accumulate MAC

Instruction Formats and Opcodes 1:

MAC (±)S1,S2,D
MAC (±)S2,S1,D

23	16 15	8	7							0
DATA BUS MOVE FIELD			1	Q	Q	Q	d	k	1	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION										

Instruction Fields:

{S1,S2} **QQQ** Source registers S1,S2
[X0*X0,Y0*Y0,X1*X0,Y1*Y0,X0*Y1,Y0*X0,X1*Y0,Y1*X1]
(see **Table A-31** on page A-208)

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

{±} **k** Sign [+,–] (see **Table A-34** on page A-209)

Instruction Formats and Opcode 2:

MAC (±)S,#n,D

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	0	0	0	s	s	s	s	1	1	Q	Q	d	k	1	0

Instruction Fields:

{S} **QQ** Source register [Y1,X0,Y0,X1]] (see **Table A-32** on page A-209)

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

{±} **k** Sign [+,–] (see **Table A-34** on page A-209)

{#n} **ssss** Immediate operand (see **Table A-37** on page A-210)

MACI MACI

Signed Multiply-Accumulate with Immediate Operand

Operation:

D ±#xxxx∗S → D

Assembler Syntax:

MACI (±)#xxxx,S,D

Description: Multiply the two signed 16-bit source operands #xxxx and S and add/subtract the product to/from the specified 40-bit destination accumulator D. The “–” sign option is used to negate the specified product prior to accumulation. The default sign option is “+”.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and opcode:

	23							16	15							8	7							0
MACI (±)#xxxx,S,D	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	q	q	d	k	1	0
	IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	**qq**	Source register [X0,Y0,X1,Y1] (see **Table A-33** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
#xxxx		16-bit Immediate Long Data extension word

MAC(su,uu) MAC(su,uu)

Mixed Multiply-Accumulate

Operation:	**Assembler Syntax:**
D ±S1 ∗ S2 → D (S1 unsigned, S2 unsigned)	MACuu (±)S1,S2,D (no parallel move)
D ±S1 ∗ S2 → D (S1 signed, S2 unsigned)	MACsu (±)S2,S1,D (no parallel move)

Description: Multiply the two 16-bit source operands S1 and S2 and add / subtract the product to / from the specified 40-bit destination accumulator D. One or two of the source operands can be unsigned. The “–” sign option is used to negate the specified product prior to accumulation. The default sign option is “+”.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

MACsu (±)S1,S2,D
MACuu (±)S1,S2,D

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	1	s	d	k	Q	Q	Q	Q

Instruction Fields:

{S1,S2}	**QQQQ**	Source registers S1,S2 [all combinations of X0,X1,Y0 and Y1] (see **Table A-35** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
{s}		[ss,us] (see **Table A-45** on page A-213)

MACR Signed Multiply-Accumulate and Round MACR

Operation:	**Assembler Syntax:**
D ±S1 $*$ S2 + r → D (parallel move)	MACR (±)S1,S2,D (parallel move)
D ±S1 $*$ S2 + r → D (parallel move)	MACR (±)S2,S1,D (parallel move)
D ±(S1 $*$ 2^{-n}) + r → D (**no** parallel move)	MACR (±)S,#n,D (**no** parallel move)

Description: Multiply the two signed 16-bit source operands S1 and S2 (or the signed 16-bit source operand S by the positive 16-bit immediate operand 2^{-n}), add/subtract the product to/from the specified 40-bit destination accumulator D, and then round the result using either convergent or two's-complement rounding. The rounded result is stored in the destination accumulator D.

The "–" sign option negates the specified product prior to accumulation. The default sign option is "+".

The contribution of the LSBs of the result is rounded into the upper portion of the destination accumulator. Once rounding has been completed, the LSBs of the destination accumulator D are loaded with 0s to maintain an unbiased accumulator value that can be reused by the next instruction. The upper portion of the accumulator contains the rounded result that can be read out to the data buses. Refer to the RND instruction for more complete information on the rounding process.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

MACR Signed Multiply-Accumulate and Round MACR

Instruction Formats and Opcodes 1:

	23 16	15 8	7 0
MACR (±)S1,S2,D MACR (±)S2,S1,D	DATA BUS MOVE FIELD		1 Q Q Q d k 1 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION		

Instruction Fields:

{S1,S2}	**QQQ**	Source registers S1,S2 [X0*X0,Y0*Y0,X1*X0,Y1*Y0,X0*Y1,Y0*X0,X1*Y0,Y1*X1] (see **Table A-31** on page A-208)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)

Instruction Formats and Opcode 2:

	23 16	15 8	7 0
MACR (±)S,#n,D	0 0 0 0 0 0 0 1	0 0 0 0 3 s s s	1 1 Q Q d k 1 1

Instruction Fields:

{S}	**QQ**	Source register [Y1,X0,Y0,X1] (see **Table A-32** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
{#n}	**ssss**	Immediate operand (see **Table A-37** on page A-210)

MACRI MACRI

Signed MAC and Round with Immediate Operand

Operation:

D ±#xxxx ∗ S → D

Assembler Syntax:

MACRI (±)#xxxx,S,D

Description: Multiply the two signed 16-bit source operands #xxxx and S, add/subtract the product to/from the specified 40-bit destination accumulator D, and then round the result using either convergent or two’s-complement rounding. The rounded result is stored in the destination accumulator D.

The “–” sign option negates the specified product prior to accumulation. The default sign option is “+”.

The contribution of the LSBs of the result is rounded into the upper portion of the destination accumulator. Once rounding has been completed, the LSBs of the destination accumulator D are loaded with 0s to maintain an unbiased accumulator value that can be reused by the next instruction. The upper portion of the accumulator contains the rounded result that can be read out to the data buses. Refer to the RND instruction for more complete information on the rounding process.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

	23							16	15							8	7							0
MACRI (±)#xxxx,S,D	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	q	q	d	k	1	1
	IMMEDIATE DATA EXTENSION																							

MACRI MACRI

Signed MAC and Round with Immediate Operand

Instruction Fields:

{S}	**qq**	Source register [X0,Y0,X1,Y1] (see **Table A-33** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,-] (see **Table A-34** on page A-209)
#xxxx		16-bit Immediate Long Data extension word

MAX Transfer by Signed Value MAX

Operation:

If B – A ≤ 0 then A → B

Assembler Syntax:

MAX A,B (parallel move)

Description: Subtract the signed value of the source accumulator from the signed value of the destination accumulator. If the difference is negative or 0, (A ≥ B) then transfer the source accumulator to destination accumulator. Otherwise, do not change the destination accumulator.

This is a 40-bit operation.

Note: The Carry bit signifies that a transfer has been performed.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	*
CCR							

* C This bit is cleared if the conditional transfer is performed, and set otherwise.
√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 … 16 15 … 8	7							0
MAX A, B	DATA BUS MOVE FIELD	0	0	0	1	1	1	0	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

MAXM Transfer by Magnitude MAXM

Operation:

If |B| – |A| ≤ 0 then A → B

Assembler Syntax:

MAXM A,B (parallel move)

Description: Subtract the absolute value (magnitude) of the source accumulator from the absolute value of the destination accumulator. If the difference is negative or 0 (| A | ≥ | B |), then transfer the source accumulator to the destination accumulator. Otherwise, do not change the destination accumulator.

This is a 40-bit operation.

Note: The Carry bit (C) signifies that a transfer has been performed.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	*
CCR							

* C This bit is cleared if the conditional transfer was performed, and set otherwise.
√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 16 15 8	7	0
MAXM A, B	DATA BUS MOVE FIELD	0 0 0 1	0 1 0 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION		

MERGE Merge Two Half Words MERGE

Operation:

{S[7:0],D[23:16]} → D[31:16]

Assembler Syntax:

MERGE S,D

Description: The contents of bits 7–0 of the source register are concatenated to the contents of bits 23–16 of the destination accumulator. The result is stored in the destination accumulator. This instruction is a 16-bit operation. The remaining bits of the destination accumulator D are not affected.

Note: This instruction can be used in conjunction with EXTRACT or INSERT instructions to concatenate width and offset fields into a control word.

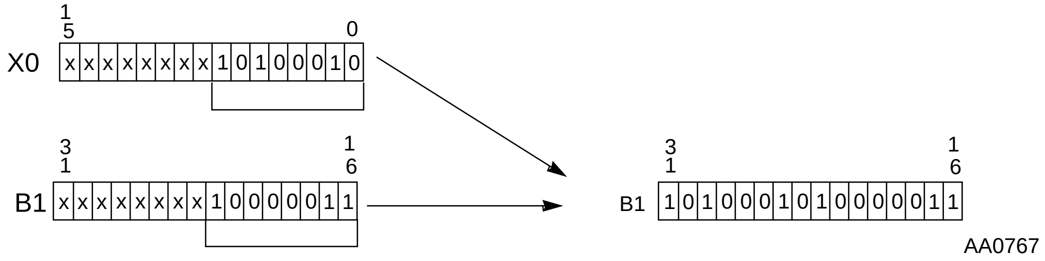
Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	*	*	*	—
CCR							

* N This bit is set if Bit 31 of the result is set.
* Z This bit is set if bits 31–16 of the result are 0.
* V This bit is always cleared.
— This bit is unchanged by the instruction.

Example:

MERGE X0,B


1
5
0
X0
x x x x x x x x 1 0 1 0 0 0 1 0
3
1
1
6
B1
x x x x x x x x 1 0 0 0 0 0 1 1
3
1
1
6
B1
1 0 1 0 0 0 1 0 1 0 0 0 0 0 1 1
AA0767

MERGE Merge Two Half Words MERGE

Instruction Formats and Opcodes:

		23							16	15							8	7							0
MERGE	S,D	0	0	0	0	1	1	0	0	0	0	0	1	1	0	1	1	1	0	0	0	S	S	S	D

Instruction Fields:

{D}	**D**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{S}	**SSS**	Source register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)

MOVE Move Data MOVE

The DSP56600 core provides a set of MOVE instructions. **Table A-14** lists these instructions, which are fully described in the following pages.

Table A-14 Move Instructions

Instruction	Description	Page
MOVE	Move Data	A-132
	NO Parallel Data Move	A-133
I	Immediate Short Data Move	A-134
R	Register to Register Data Move	A-136
U	Address Register Update	A-138
X:	X Memory Data Move	A-139
X: R	X Memory and Register Data Move	A-142
Y	Y Memory Data Move	A-145
R: Y	Register and Y Memory Data Move	A-148
L:	Long Memory Data Move	A-151
X: Y	X Memory Data Move	A-153

MOVE Move Data MOVE

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 … 16	15 … 8	7 … 4	3 … 0
MOVE S,D	DATA BUS MOVE FIELD		0 0 0 0	0 0 0 0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION			

Instruction Fields: See **Parallel Move Descriptions** for data bus move field encoding.

Parallel Move Descriptions: Thirty of the sixty-two instructions allow 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. The following section contains detailed descriptions about each type of parallel move operation.

NO Parallel Data Move

Operation:

(. . . . . .)

Assembler Syntax:

(. . . . . .)

where (. . . . .) refers to any arithmetic or logical instruction that allows parallel moves

Description: Many instructions in the instruction set allow parallel moves. The parallel moves have been divided into ten opcode categories. This category is a parallel move NOP and does not involve data bus move activity.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7 0
(. . . . . .)	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	INSTRUCTION OPCODE

Instruction Format:

(defined by instruction)

I Immediate Short Data Move I

Operation:

(. . . . .), #xx → D

Assembler Syntax:

(. . . . .) #xx,D

where (. . . . .) refers to any arithmetic or logical instruction that 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 LSBs 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 MSBs 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 cannot be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 40-bit A accumulator as its destination, the parallel data bus move portion of the instruction cannot specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 40-bit B accumulator as its destination, the parallel data bus move portion of the instruction cannot specify B0, B1, B2, or B as its destination D. That is, duplicate destinations are *not* allowed within the same instruction.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Immediate Short Data Move

Instruction Formats and Opcodes:

	23							16	15							8	7 0
(. . . . .) #xx,D	0	0	1	d	d	d	d	d	i	i	i	i	i	i	i	i	INSTRUCTION OPCODE

Instruction Fields:

{#xx} **iiiiiiii** 8-bit Immediate Short Data

{D} **ddddd** Destination register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0–R7,N0–N7] (see **Table A-32** on page A-209)

R Register to Register Data Move R

Operation:

(. . . . .); S → D

Assembler Syntax:

(. . . . .) S,D

where (. . . . .) refers to any arithmetic or logical instruction that 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 cannot be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 40-bit A accumulator as its destination, the parallel data bus move portion of the instruction cannot specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 40-bit B accumulator as its destination, the parallel data bus move portion of the instruction cannot 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 can 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 MOVE A,B operation results in a 16-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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

R Register to Register Data Move R

Instruction Formats and Opcodes:

	23							16	15							8	7 0
(. . . . .) S,D	0	0	1	0	0	0	e	e	e	e	e	d	d	d	d	d	INSTRUCTION OPCODE

Instruction Fields:

{S} **eeeee** Source register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0–R7,N0–N7] (see **Table A-36** on page A-210)

{D} **ddddd** Destination register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0–R7,N0–N7] (see **Table A-36** on page A-210)

Address Register Update (U)

Operation:	**Assembler Syntax:**
(. . . . .); ea → Rn	(. . . . .) ea

where (. . . . .) refers to any arithmetic or logical instruction that allows parallel moves

Description: Update the specified address register according to the specified effective addressing mode. All update addressing modes can be used.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7 0
(. . . . .) ea	0	0	1	0	0	0	0	0	0	1	0	M	M	R	R	R	INSTRUCTION OPCODE

Instruction Fields:

{ea}	**MMRRR**	Effective Address (see **Table A-25** on page A-206)

X: X Memory Data Move X:

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
X:(Rn + xxx) → D	MOVE X:(Rn + xxx),D
X:(Rn + xxxx) → D	MOVE X:(Rn + xxxx),D
D → X:(Rn + xxx)	MOVE D,X:(Rn + xxx)
D → X:(Rn + xxxx)	MOVE D,X:(Rn + xxxx)

where (. . . . .) refers to any arithmetic or logical instruction that allows parallel moves.

Description: Move the specified word operand from/to X memory. All memory addressing modes, including absolute addressing and 16-bit immediate data, can be used. Absolute short addressing can 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 cannot be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 40-bit A accumulator as its destination, the parallel data bus move portion of the instruction cannot specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 40-bit B accumulator as its destination, the parallel data bus move portion of the instruction cannot 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 can 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.

X: X Memory Data Move X:

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.

— This bit is unchanged by the instruction.

Note: The MOVE A,X:ea operation results in a 16-bit positive or negative saturation constant being stored in the specified 16-bit X memory location if the signed integer portion of the A accumulator is in use.

Instruction Formats and Opcodes 1:

(. . . . .) X:ea,D
(. . . . .) S,X:ea
(. . . . .) #xxxx,D

23							16	15							8	7 0
0	1	d	d	0	d	d	d	W	1	M	M	M	R	R	R	INSTRUCTION OPCODE
OPTIONAL EFFECTIVE ADDRESS EXTENSION																

(. . . . .) X:aa,D
(. . . . .) S,X:aa

23							16	15							8	7 0
0	1	d	d	0	d	d	d	W	0	a	a	a	a	a	a	INSTRUCTION OPCODE

Instruction Fields:

{ea}	**MMMRRR**	Effective Address (see **Table A-21** on page A-205)
	W	Read S / Write D bit (see **Table A-38** on page A-210)
{S,D}	**ddddd**	Source/Destination registers [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0–R7,N0–N7] (see **Table A-36** on page A-210)
{aa}	**aaaaaa**	6-bit Absolute Short Address

X: X Memory Data Move X:

Instruction Formats and Opcodes 2:

MOVE X:(Rn + xxxx),D
MOVE S,X:(Rn + xxxx)

23							16	15							8	7							0
0	0	0	0	1	0	1	0	0	1	1	1	0	R	R	R	1	W	D	D	D	D	D	D
Rn RELATIVE DISPLACEMENT																							

MOVE X:(Rn + xxx),D
MOVE S,X:(Rn + xxx)

23							16	15							8	7							0
0	0	0	0	0	0	1	a	a	a	a	a	a	R	R	R	1	a	0	W	D	D	D	D

Instruction Fields:

	W	Read S / Write D bit (see **Table A-38** on page A-210)
{xxx}	**aaaaaaa**	7-bit sign extended Short Displacement Address
{Rn}	**RRR**	Address register (R0–R7)
{D}	**DDDD**	Source/Destination registers [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B] (see **Table A-39** on page A-211)
{S,D}	**DDDDDD**	Source/Destination registers [all on-chip registers] (see **Table A-27** on page A-207)

X:R X Memory and Register Data Move X:R

Operation:	**Assembler Syntax:**
Class I	
(. . . . .); X:ea → D1; S2 → D2	(. . . . .) X:ea,D1 S2,D2
(. . . . .); S1 → X:ea; S2 → D2	(. . . . .) S1,X:ea S2,D2
(. . . . .); # → D1; S2 → D2	(. . . . .) #xxxxxx,D1 S2,D2
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 that 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 16-bit immediate data, can 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 except long absolute addressing and long immediate data can 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 cannot be specified as a destination D1 in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 40-bit A accumulator as its destination, the parallel data bus move portion of the instruction cannot specify A0, A1, A2, or A as its destination D1. Similarly, if the opcode-operand portion of the instruction specifies the 40-bit B accumulator as its destination, the parallel data bus move portion of the instruction cannot specify B0, B1, B2, or B as its destination D1. That is, duplicate destinations are *not* allowed within the same instruction.

X:R X Memory and Register Data Move X:R

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register can 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—S1 and S2 can specify the same register.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Class I Instruction Formats and Opcodes:

(. . . . .) X:ea,D1 S2,D2
(. . . . .) S1,X:ea S2, D2
(. . . . .) #xxxx,D1 S2,D2

23							16	15							8	7 0
0	0	0	1	f	f	d	F	W	0	M	M	M	R	R	R	INSTRUCTION OPCODE
OPTIONAL EFFECTIVE ADDRESS EXTENSION																

Instruction Fields:

{ea}	**MMMRRR**	Effective Address (see **Table A-20** on page A-204)
	W	Read S1/Write D1 bit (see **Table A-38** on page A-210)
{S1,D1}	**ff**	S1/D1 register [X0,X1,A,B] (see **Table A-40** on page A-211)
{S2}	**d**	S2 accumulator [A,B] (see **Table A-15** on page A-203)
{D2}	**F**	D2 input register [Y0,Y1] (see **Table A-40** on page A-211)

Class II Instruction Formats and Opcodes:

(. . . . .) A → X:ea X0 → A
(. . . . .) B → X:ea X0 → B

23							16	15							8	7 0
0	0	0	0	1	0	0	d	0	0	M	M	M	R	R	R	INSTRUCTION OPCODE
OPTIONAL EFFECTIVE ADDRESS EXTENSION																

X:R X Memory and Register Data Move X:R

Instruction Fields:

{ea}	**MMMRRR**	Effective Address (see **Table A-24** on page A-206)
	d	Move opcode (see **Table A-42** on page A-211)

Y Y Memory Data Move Y

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
Y:(Rn + xxx) → D	MOVE Y:(Rn + xxx),D
Y:(Rn + xxxx) → D	MOVE Y:(Rn + xxxx),D
D → Y:(Rn + xxx)	MOVE D,Y:(Rn + xxx)
D → Y:(Rn + xxxx)	MOVE D,Y:(Rn + xxxx)

where (. . . . .) refers to any arithmetic or logical instruction that allows parallel moves

Description: Move the specified word operand from/to Y memory. All memory addressing modes, including absolute addressing and 16-bit immediate data, can be used. Absolute short addressing can 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 cannot be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 40-bit A accumulator as its destination, the parallel data bus move portion of the instruction cannot specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 40-bit B accumulator as its destination, the parallel data bus move portion of the instruction cannot 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 can 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.

Y Y Memory Data Move Y

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Note: The MOVE A,Y:ea operation results in a 16-bit positive or negative saturation constant being stored in the specified 16-bit Y memory location if the signed integer portion of the A accumulator is in use.

Instruction Formats and Opcodes 1:

(. . . . .) Y:ea,D
(. . . . .) S,Y:ea
(. . . . .) #xxxx,D

23							16	15							8	7 0
0	1	d	d	1	d	d	d	W	1	M	M	M	R	R	R	INSTRUCTION OPCODE
OPTIONAL EFFECTIVE ADDRESS EXTENSION																

(. . . . .) Y:aa,D
(. . . . .) S,Y:aa

23							16	15							8	7 0
0	1	d	d	1	d	d	d	W	0	a	a	a	a	a	a	INSTRUCTION OPCODE

Instruction Fields:

{ea}	**MMMRRR**	Effective Address (see **Table A-20** on page A-204)
	W	Read S/Write D bit (see **Table A-38** on page A-210)
{S,D}	**ddddd**	Source/Destination registers [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0–R7,N0–N7] (see **Table A-36** on page A-210)
{aa}	**aaaaaa**	Absolute Short Address

Y Y Memory Data Move Y

Instruction Formats and Opcodes 2:

MOVE Y:(Rn + xxxx),D
MOVE D,Y:(Rn + xxxx)

23							16	15							8	7							0
0	0	0	0	1	0	1	1	0	1	1	1	0	R	R	R	1	W	D	D	D	D	D	D
Rn RELATIVE DISPLACEMENT																							

MOVE Y:(Rn + xxx),D
MOVE D,Y:(Rn + xxx)

23							16	15							8	7							0
0	0	0	0	0	0	1	a	a	a	a	a	a	R	R	R	1	a	1	W	D	D	D	D

Instruction Fields:

	W	Read S/Write D bit (see **Table A-38** on page A-210)
{xxx}	**aaaaaaa**	7-bit sign extended Short Displacement Address
{Rn}	**RRR**	Address register (R0–R7)
{D}	**DDDD**	Source/Destination registers [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B] (see **Table A-39** on page A-211)
{S,D}	**DDDDDD**	Source/Destination registers [all on-chip registers] (see **Table A-27** on page A-207)

R:Y Register and Y Memory Data Move R:Y

Operation:	**Assembler Syntax:**
Class I	
(. . . . .); S1 → D1; Y:ea → D2	(. . . . .) S1,D1 Y:ea,D2
(. . . . .); S1 → D1; S2 → Y:ea	(. . . . .) S1,D1 S2,Y:ea
(. . . . .); S1 → D1; #xxxx → D2	(. . . . .) S1,D1 #xxxxxx,D2
Class II	
(. . . . .); Y0 → A; A → Y:ea	(. . . . .) Y0,A A,Y:ea
(. . . . .); Y0 → B; B → Y:ea	(. . . . .) Y0,B B,Y:ea

where (. . . . .) refers to any arithmetic or logical instruction that 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 16-bit immediate data, can 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 a one-word operand from a Data ALU accumulator to Y memory and a 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, can be used.

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 cannot be specified as a destination D2 in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 40-bit A accumulator as its destination, the parallel data bus move portion of the instruction cannot specify A0, A1, A2, or A as its destination D2. Similarly, if the opcode-operand portion of the instruction specifies the 40-bit B accumulator as its destination, the parallel data bus move portion of the instruction cannot specify B0, B1, B2, or B as its destination D2. That is, duplicate destinations are *not* allowed within the same instruction.

R:Y Register and Y Memory Data Move R:Y

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register can 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 can specify the same register.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Class I Instruction Formats and Opcodes:

(. . . . .) S1,D1 Y:ea,D2
(. . . . .) S1,D1 S2,Y:ea
(. . . . .) S1,D1 #xxxx,D2

23							16	15							8	7 0
0	0	0	1	d	e	f	f	W	1	M	M	M	R	R	R	INSTRUCTION OPCODE
OPTIONAL EFFECTIVE ADDRESS EXTENSION																

Instruction Fields :

{ea}	**MMMRRR**	Effective Address (see **Table A-20** on page A-204)
	W	Read S2/Write D2 bit (see **Table A-38** on page A-210)
{S1}	**d**	S1 accumulator [A,B] (see **Table A-15** on page A-203)
{D1}	**e**	D1 input register [X0,X1] (see **Table A-41** on page A-211)
{S2,D2}	**ff**	S2/D2 register [Y0,Y1,A,B] (see **Table A-41** on page A-211)

Class II Instruction Formats and opcodes:

(. . . . .) Y0 → A A → Y:ea
(. . . . .) Y0 → B B → Y:ea

23							16	15							8	7 0
0	0	0	0	1	0	0	d	1	0	M	M	M	R	R	R	INSTRUCTION OPCODE
OPTIONAL EFFECTIVE ADDRESS EXTENSION																

R:Y Register and Y Memory Data Move R:Y

Instruction Fields:

MMMRRR	ea = 6-bit Effective Address (see **Table A-24** on page A-206)
d	Move opcode (see **Table A-42** on page A-211)

L: Long Memory Data Move L:

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 that allows parallel moves

Description: Move one 32-bit long-word operand from/to X and Y memory. Two Data ALU registers are concatenated to form the 32-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 32-bit signed (double-precision) quantity while the X, Y, AB, and BA operands reference two separate (i.e., real and imaginary) 16-bit signed quantities. All memory alterable addressing modes can be used. Absolute short addressing can 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 cannot be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 40-bit A accumulator as its destination, the parallel data bus move portion of the instruction cannot specify A, A10, AB, or BA as destination D. Similarly, if the opcode-operand portion of the instruction specifies the 40-bit B accumulator as its destination, the parallel data bus move portion of the instruction cannot 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 can 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.

L: Long Memory Data Move L:

Note: The operands A10, B10, X, Y, AB, and BA can be used only for a 32-bit long memory move as previously described. These operands cannot be used in any other type of instruction or parallel move.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Note: The MOVE A,L:ea operation results in a 32-bit positive or negative saturation constant being stored in the specified 16-bit X and Y memory locations if the signed integer portion of the A accumulator is in use. The MOVE AB,L:ea operation results in either one or two 16-bit positive and/or negative saturation constant(s) being stored in the specified 16-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 Formats and Opcodes:

(. . . . .) L:ea,D
(. . . . .) S,L:ea

23							16	15							8	7 0
0	1	0	0	L	0	L	L	W	1	M	M	M	R	R	R	INSTRUCTION OPCODE
OPTIONAL EFFECTIVE ADDRESS EXTENSION																

(. . . . .) L:aa,D
(. . . . .) S,L:aa

23							16	15							8	7 0
0	1	0	0	L	0	L	L	W	0	a	a	a	a	a	a	INSTRUCTION OPCODE

Instruction Fields:

{ea}	**MMMRRR**	Effective Address (see **Table A-23** on page A-206)
	W	Read S/Write D bit (see **Table A-38** on page A-210)
{L}	**LLL**	Two Data ALU registers (see **Table A-28** on page A-207)
{aa}	**aaaaaa**	Absolute Short Address

X: Y: XY Memory Data Move X: Y:

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 that 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 can 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 cannot 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 40-bit A accumulator as its destination, the parallel data bus move portion of the instruction cannot specify A as its destination D1 or D2. Similarly, if the opcode-operand portion of the instruction specifies the 40-bit B accumulator as its destination, the parallel data bus move portion of the instruction cannot specify B as its destination D1 or D2. That is, duplicate destinations are *not* allowed within the same instruction. D1 and D2 cannot specify the same register.

If the instruction specifies an access to an internal X I/O and internal Y I/O modules (reflected by the address of the X memory and the Y memory), only the access to the internal X I/O module is executed. The access to the Y I/O module is discarded.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register can 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 can specify the same register.

X: Y: XY Memory Data Move X: Y:

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

(. . . . .) X:<eax>,D1 Y:<eay>,D2
(. . . . .) X:<eax>,D1 S2,Y:<eay>
(. . . . .) S1,X:<eax> Y:<eay>,D2
(. . . . .) S1,X:<eax> S2,Y:<eay>

23							16	15							8	7 0
1	w	m	m	e	e	f	f	W	r	r	M	M	R	R	R	INSTRUCTION OPCODE

Instruction Fields:

{<eax>}	**MMRRR**	5-bit X Effective Address (R0–R3 or R4–R7)
{<eay>}	**mmrr**	4-bit Y Effective Address (R4–R7 or R0–R3)
{S1,D1}	**ee**	S1/D1 register [X0,X1,A,B]
{S2,D2}	**ff**	S2/D2 register [Y0,Y1,A,B]

MMRRR,mmrr,ee,ff See **Table A-43** on page A-212
W X move Operation Control (See **Table A-38** on page A-210)
w Y move Operation Control (See **Table A-38** on page A-210)

MOVEC Move Control Register MOVEC

Operation:	Assembler Syntax:	
[X or Y]:ea → D1	MOVE(C)	[Xor Y]:ea,D1
[X or Y]:aa → D1	MOVE(C)	[Xor Y]:aa,D1
S1 → [X or Y]:ea	MOVE(C)	S1,[X or Y]:ea
S1 → [X or Y]:aa	MOVE(C)	S1,[X or 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, or move 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 the program controller registers. These registers can be moved to or from any other register or memory space. All memory addressing modes, as well as an Immediate Short Addressing mode, can be used.

If the System Stack register SSH is specified as a source operand, the Stack Pointer (SP) is post-decremented by 1 after SSH has been read. If SSH is specified as a destination operand, the SP is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

MOVEC Move Control Register MOVEC

For D1 or D2 = SR operand:

* S This bit is set according to Bit 7 of the source operand.
* L This bit is set according to Bit 6 of the source operand.
* E This bit is set according to Bit 5 of the source operand.
* U This bit is set according to Bit 4 of the source operand.
* N This bit is set according to Bit 3 of the source operand.
* Z This bit is set according to Bit 2 of the source operand.
* V This bit is set according to Bit 1 of the source operand.
* C This bit is set according to Bit 0 of the source operand.

For D1 and D2 ≠ SR operand:

* S This bit is set if data growth has been detected.
* L This bit is set if data limiting has occurred during the move.

Instruction Formats and Opcodes:

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

23							16	15							8	7							0
0	0	0	0	0	1	0	1	W	1	M	M	M	R	R	R	O	S	1	d	d	d	d	d
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

MOVE(C) [X or Y]:aa,D1
MOVE(C) S1,[X or Y]:aa

23							16	15							8	7							0
0	0	0	0	0	1	0	1	W	0	a	a	a	a	a	a	0	S	1	d	d	d	d	d

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

23							16	15							8	7							0
0	0	0	0	0	1	0	0	W	1	e	e	e	e	e	e	1	0	1	d	d	d	d	d

MOVE(C) #xx,D1

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

Instruction Fields:

{ea}	MMMRRR	Effective Address (see **Table A-20** on page A-204)
	W	Read S/Write D bit (see **Table A-38** on page A-210)
{X/Y}	S	Memory Space [X,Y] (see **Table A-22** on page A-205)
{S1,D1}	ddddd	Program Controller register [M0–M7, VBA, SR, OMR, SP, SSH,SSL,LA,LC] (see **Table A-46** on page A-213)
{aa}	aaaaaa	aa = 6-bit Absolute Short Address
{S2,D2}	eeeeee	S2/D2 register [all on-chip registers] (see **Table A-27** on page A-207)
{#xx}	iiiiiiii	#xx = 8-bit Immediate Short Data

MOVEM Move Program Memory MOVEM

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 can be any register. All memory-alterable addressing modes can be used, as well as the Absolute Short Addressing mode.

If the system stack register SSH is specified as a source operand, the system Stack Pointer (SP) is post-decremented by 1 after SSH has been read. If the system stack register SSH is specified as a destination operand, the SP is pre-incremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

MOVEM Move Program Memory MOVEM

For D1 or D2 = SR operand:

*	S	This bit is set according to Bit 7 of the source operand.
*	L	This bit is set according to Bit 6 of the source operand.
*	E	This bit is set according to Bit 5 of the source operand.
*	U	This bit is set according to Bit 4 of the source operand.
*	N	This bit is set according to Bit 3 of the source operand.
*	Z	This bit is set according to Bit 2 of the source operand.
*	V	This bit is set according to Bit 1 of the source operand.
*	C	This bit is set according to Bit 0 of the source operand.

For D1 and D2 ≠ SR operand:

*	S	This bit is set if data growth has been detected.
*	L	This bit is set if data limiting has occurred during the move.

Instruction Formats and Opcodes:

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

23							16	15							8	7							0
0	0	0	0	0	1	1	1	W	1	M	M	M	R	R	R	1	0	d	d	d	d	d	d
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

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

23							16	15							8	7							0
0	0	0	0	0	1	1	1	W	0	a	a	a	a	a	a	0	0	d	d	d	d	d	d

Instruction Fields:

{ea}	**MMMRRR**	Effective Address (see **Table A-19** on page A-204)
	W	Read S/Write D bit (see **Table A-38** on page A-210)
{ S,D}	**dddddd**	Source/Destination register [all on-chip registers] (see **Table A-27** on page A-207)
{aa}	**aaaaaa**	Absolute Short Address

MOVEP Move Peripheral Data MOVEP

Operation:	Assembler Syntax:	
[X or Y]:pp → D	MOVEP	[X or Y]:pp,D
[X or Y]:qq → D	MOVEP	[X or Y]:qq,D
[X or Y]:pp → [X or Y]:ea	MOVEP	[X or Y]:pp,[X or Y]:ea
[X or Y]:qq → [X or Y]:ea	MOVEP	[X or Y]:qq,[X or Y]:ea
[X or Y]:pp → P:ea	MOVEP	[X or Y]:pp,P:ea
[X or Y]:qq → P:ea	MOVEP	[X or Y]:qq,P:ea
S → [X or Y]:pp	MOVEP	S,[X or Y]:pp
S → [X or Y]:qq	MOVEP	S,[X or Y]:qq
[X or Y]:ea → [X or Y]:pp	MOVEP	[X or Y]:ea,[X or Y]:pp
[X or Y]:ea → [X or Y]:qq	MOVEP	[X or Y]:ea,[X or Y]:qq
P:ea → [X or Y]:pp	MOVEP	P:ea,[X or Y]:pp
P:ea → [X or Y]:qq	MOVEP	P:ea,[X or Y]:qq

Description: Move the specified operand to or from 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 can be used for the X or Y memory effective address; all memory-alterable addressing modes can be used for the P memory effective address. All the I/O space ($FF80–$FFFF) can be accessed, except for the P: reference opcode.

If the System Stack register SSH is specified as a source operand, the system Stack Pointer (SP) is post-decremented by 1 after SSH has been read. If SSH is specified as a destination operand, the SP is pre-incremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

MOVEP Move Peripheral Data MOVEP

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

For D1 or D2 = SR operand:

*	S	This bit is set according to Bit 7 of the source operand.
*	L	This bit is set according to Bit 6 of the source operand.
*	E	This bit is set according to Bit 5 of the source operand.
*	U	This bit is set according to Bit 4 of the source operand.
*	N	This bit is set according to Bit 3 of the source operand.
*	Z	This bit is set according to Bit 2 of the source operand.
*	V	This bit is set according to Bit 1 of the source operand.
*	C	This bit is set according to Bit 0 of the source operand.

For D1 and D2 ≠ SR operand:

*	S	This bit is set if data growth has been detected.
*	L	This bit is set if data limiting has occurred during the move.

Instruction Formats and Opcodes:

X: or Y: Reference (high I/O address)

MOVEP [X or Y]:pp,[X or Y]:ea
MOVEP [X or Y]:ea,[X or Y]:pp

23							16	15							8	7							0
0	0	0	0	1	0	0	s	W	1	M	M	M	R	R	R	1	S	p	p	p	p	p	p
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

X: or Y: Reference (low I/O address)

MOVEP X:qq,[X or Y]:ea
MOVEP [X or Y]:ea,X:qq

23							16	15							8	7							0
0	0	0	0	0	1	1	1	W	1	M	M	M	R	R	R	0	S	q	q	q	q	q	q
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

X: or Y: Reference (low I/O address)

MOVEP Y:qq,[X or Y]:ea
MOVEP [X or Y]:ea,Y:qq

23							16	15							8	7							0
0	0	0	0	0	1	1	1	W	0	M	M	M	R	R	R	1	S	q	q	q	q	q	q
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

MOVEP Move Peripheral Data MOVEP

P: Reference (high I/O address)

MOVEP P:ea,[X or Y]:pp
MOVEP [X or Y]:pp,P:ea

							16	15							8	7							0
0	0	0	0	1	0	0	s	W	1	M	M	M	R	R	R	0	1	p	p	p	p	p	p

P: Reference (low I/O address)

MOVEP P:ea,[X or Y]:qq
MOVEP [X or Y]:qq,P:ea

							16	15							8	7							0
0	0	0	0	0	0	0	0	1	W	M	M	M	R	R	R	0	S	q	q	q	q	q	q

Register Reference (high I/O address)

MOVEP S,[X or Y]:pp
MOVEP [X or Y]:pp,D

23							16	15							8	7							0
0	0	0	0	1	0	0	s	W	1	d	d	d	d	d	d	0	0	p	p	p	p	p	p

Register Reference: (low I/O address)

MOVEP S,X:qq
MOVEP X:qq,D

23							16	15							8	7							0
0	0	0	0	0	1	0	0	W	1	d	d	d	d	d	d	1	q	0	q	q	q	q	q

Register Reference: (low I/O address)

MOVEP S,Y:qq
MOVEP Y:qq,D

23							16	15							8	7							0
0	0	0	0	0	1	0	0	W	1	d	d	d	d	d	d	0	q	1	q	q	q	q	q

Instruction Fields:

{ea}	**MMMRRR**	Effective Address (see **Table A-21** on page A-205)
{pp}	**pppppp**	I/O Short Address [64 addresses: $FFC0-$FFFF]
{qq}	**qqqqqq**	I/O Short Address [64 addresses: $FF80-$FFBF]
{X/Y}	**S**	Memory space [X,Y] (see **Table A-22** on page A-205)
{X/Y}	**s**	Peripheral space [X,Y] (see **Table A-22** on page A-205)
	W	Read/write-peripheral (see **Table A-38** on page A-210)
{S,D}	**dddddd**	Source/Destination register [all on-chip registers] (see **Table A-27** on page A-207)

MPY Signed Multiply MPY

Operation:		Assembler Syntax:	
$\pm S1 * S2 \rightarrow D$	(parallel move)	MPY (±)S1,S2,D	(parallel move)
$\pm S1 * S2 \rightarrow D$	(parallel move)	MPY (±)S2,S1,D	(parallel move)
$\pm(S1 * 2^{-n}) \rightarrow D$	(no parallel move)	MPY (±)S,#n,D	(no parallel move)

Description: Multiply the two signed 16-bit source operands S1 and S2 and store the resulting product in the specified 40-bit destination accumulator D. Or, multiply the signed 16-bit source operand S by the positive 16-bit immediate operand 2^{-n} and store the resulting product in the specified 40-bit destination accumulator D. The "–" sign option is used to negate the specified product prior to accumulation. The default sign option is "+".

Note: When the processor is in the Double Precision Multiply mode, the following instructions do not execute in the normal way and should only be used as part of the double precision multiply algorithm:

MPY Y0,X0,A MPY Y0, X0,B

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes 1:

	23 16 15 8	7							0
MPY (±)S1,S2,D	DATA BUS MOVE FIELD	1	Q	Q	Q	d	k	0	0
MPY (±)S2,S1,D	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

MPY Signed Multiply MPY

Instruction Fields:

{S1,S2}	**QQQ**	Source registers S1,S2 [X0*X0, Y0*Y0, X1*X0, Y1*Y0, X0*Y1, Y0*X0, X1*Y0, Y1*X1] (see **Table A-31** on page A-208)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±+/-}	**k**	Sign [+,–] (see **Table A-34** on page A-209)

Instruction Formats and Opcode 2:

		23							16	15							8	7							0
MPY	(±)S,#n,D	0	0	0	0	0	0	0	1	0	0	0	0	s	s	s	s	1	1	Q	Q	d	k	0	0

Instruction Fields:

{S}	**QQ**	Source register [Y1,X0,Y0,X1] (see **Table A-32** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
{#n}	**sssss**	Immediate operand (see **Table A-37** on page A-210)

MPY(su,uu) Mixed Multiply MPY(su,uu)

Operation:	**Assembler Syntax:**
±S1 ∗ S2 → D (S1 unsigned, S2 unsigned)	MPYuu (±)S1,S2,D (no parallel move)
±S1 ∗ S2 → D (S1 signed, S2 unsigned)	MPYsu (±)S2,S1,D (no parallel move)

Description: Multiply the two 16-bit source operands S1 and S2 and store the resulting product in the specified 40-bit destination accumulator D. One or two of the source operands can be unsigned. The "–" sign option is used to negate the specified product prior to accumulation. The default sign option is "+".

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

MPY su (±)S1,S2,D
MPY uu (±)S1,S2,D

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	1	1	s	d	k	Q	Q	Q	Q

Instruction Fields:

{S1,S2}	**QQQQ**	Source registers S1,S2 [all combinations of X0,X1,Y0, and Y1] (see **Table A-35** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
{s}		[ss,us] (see **Table A-45** on page A-213)

MPYI Signed Multiply with Immediate Operand MPYI

Operation:

±#xxxx∗S → D

Assembler Syntax:

MPYI (±)#xxxx,S,D

Description: Multiply the immediate 16-bit source operand #xxxx with the 16-bit register source operand S and store the resulting product in the specified 40-bit destination accumulator D. The "–" sign option is used to negate the specified product prior to accumulation. The default sign option is "+".

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

	23							16	15							8	7							0
MPYI (±)#xxxx,S,D	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	q	q	d	k	0	0
	IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	**qq**	Source register [X0,Y0,X1,Y1] (see **Table A-33** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
#xxxx		16-bit Immediate Long Data extension word

MPYR Signed Multiply and Round MPYR

Operation:		Assembler Syntax:	
$\pm S1 * S2 + r \rightarrow D$	(parallel move)	MPYR (±)S1,S2,D	(parallel move)
$\pm S1 * S2 + r \rightarrow D$	(parallel move)	MPYR (±)S2,S1,D	(parallel move)
$\pm(S1 * 2^{-n}) + r \rightarrow D$	(no parallel move)	MPYR (±)S,#n,D	(no parallel move)

Description: Multiply the two signed 16-bit source operands S1 and S2 (**or** the signed 16-bit source operand S by the positive 16-bit immediate operand 2^{-n}), round the result using either convergent or two's-complement rounding, and store it in the specified 40-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. Once the rounding has been completed, the LSBs of the destination accumulator D are loaded with 0s to maintain an unbiased accumulator value that can be reused by the next instruction. The upper portion of the accumulator contains the rounded result that can be read out to the data buses. Refer to the RND instruction for more complete information on the rounding process.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes 1:

	23 16 15 8	7	0
MPYR (±)S1,S2,D MPYR (±)S2,S1,D	DATA BUS MOVE FIELD	1 Q Q Q	d k 0 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION		

MPYR Signed Multiply and Round MPYR

Instruction Fields 1:

{S1,S2}	**QQQ**	Source registers S1,S2 [X0*X0, Y0*Y0, X1*X0, Y1*Y0, X0*Y1, Y0*X0, X1*Y0, Y1*X1] (see **Table A-31** on page A-208)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)

Instruction Formats and Opcode 2:

	23							16	15							8	7							0
MPYR (±)S,#n,D	0	0	0	0	0	0	0	1	0	0	0	s	s	s	s	s	1	1	Q	Q	d	k	0	1

Instruction Fields 2:

{S}	**QQ**	Source register [Y1,X0,Y0,X1] (see **Table A-32** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
{#n}	**sssss**	Immediate operand (see **Table A-37** on page A-210)

MPYRI MPYRI

Signed Multiply and Round with Immediate Operand

Operation:	**Assembler Syntax:**
±#xxxx ∗ S + r → D	MPYRI (±)#xxxx,S,D

Description: Multiply the two signed 16-bit source operands #xxxx and S, round the result using either convergent or two's-complement rounding, and store it in the specified 40-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. Once the rounding has been completed, the LS bits of the destination accumulator D are loaded with 0s to maintain an unbiased accumulator value that can be reused by the next instruction. The upper portion of the accumulator contains the rounded result that can be read out to the data buses. Refer to the RND instruction for more complete information on the rounding process.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

	23							16	15							8	7							0
MPYRI (±)#xxxx,S,D	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	q	q	d	k	0	1
	IMMEDIATE DATA EXTENSION																							

MPYRI MPYRI

Signed Multiply and Round with Immediate Operand

Instruction Fields:

{S}	**qq**	Source register [X0,Y0,X1,Y1] (see **Table A-33** on page A-209)
{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)
{±}	**k**	Sign [+,–] (see **Table A-34** on page A-209)
#xxxx		16-bit Immediate Long Data extension word

NEG Negate Accumulator NEG

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 40-bit, two's-complement operation.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

NEG D

23	16 15	8	7							0
DATA BUS MOVE FIELD			0	0	1	1	d	1	1	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION										

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

NOP No Operation NOP

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

	23							16	15							8	7							0
NOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields:

None

NORMF Fast Accumulator Normalization NORMF

Operation:

If S[15] = 0 then ASR S,D
else ASL -S,D

Assembler Syntax:

NORMF S,D

Description: Arithmetically shift the destination accumulator either left or right as specified by the source operand. This instruction can be used to normalize the specified accumulator D, by arithmetically shifting it either left or right so as to bring the leading 1 or 0 to bit location 30. The number of needed shifts is specified by the source operand. This number could be calculated by a previous CLB instruction. If the source operand is negative then the accumulator is left shifted, and if the source operand is positive then it is right shifted. For normalization, the source accumulator value should be between +8 to –31.

This is a 40-bit operation.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	√	√	√	√	√	*	—
CCR							

* V This bit is set if Bit 39 is changed any time during the shift operation, and cleared otherwise.
√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Example:

```
CLB      A,B        ;Count leading bits
NORMF    B1,A       ;Normalize A.
```

If the base exponent is stored in R1 it can be updated by the following commands:

```
MOVE     B1,N1 ;    ;Update N1 with shift amount
MOVE     (R1)+N1    ;Increment or decrement exponent
```

NORMF Fast Accumulator Normalization NORMF

Prior to execution, the 40-bit A accumulator contains the value $20:0000:0000. The CLB instruction updates the B accumulator to the number of needed shifts, seven in this example. The NORMF instruction performs seven shifts to the right on A accumulator, and normalization of A is achieved. The exponent register is updated according to the number of shifts.

	Before execution	After execution
CLB A,B	A: $20:0000:0000	B: $00:0007:0000
NORMF B1,A	A: $20:0000:0000	A: $00:4000:0000

Instruction Formats and Opcode

		23							16	15							8	7							0
NORMF	S,D	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0	1	0	s	s	s	D

Instruction Fields:

{S} **sss** Source register [X0,X1,Y0,Y1,A1,B1] (see **Table A-20** on page A-204)
{D} **D** Destination accumulator [A,B] (see **Table A-15** on page A-203)

NOT Logical Complement NOT

Operation:

$\overline{D[31:16]}$ fi D[31:16] (parallel move)

Assembler Syntax:

NOT D (parallel move)

where "—" denotes the logical NOT operator

Description: Take the one's complement of bits 31–16 of the destination operand D and store the result back in bits 31–16 of the destination accumulator. This is a 16-bit operation. The remaining bits of D are not affected.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	*	*	*	—
CCR							

*	N	This bit is set if Bit 31 of the result is set.
*	Z	This bit is set if bits 31–16 of the result are 0.
*	V	This bit is always cleared.
√		This bit is changed according to the standard definition.
—		This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

NOT D

23	16 15	8	7							0
DATA BUS MOVE FIELD			0	0	0	1	d	1	1	1
OPTIONAL EFFECTIVE ADDRESS EXTENSION										

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

OR Logical Inclusive OR OR

Operation:		**Assembler Syntax:**	
S ⊕ D[31:16] → D[31:16]	(parallel move)	OR S,D	(parallel move)
#xx ⊕ D[31:16] → D[31:16]		OR #xx,D	
#xxxx ⊕ D[31:16] → D[31:16]		OR #xxxx,D	

where ⊕ denotes the logical inclusive OR operator

Description: Logically inclusive OR the source operand S with bits 31–16 of the destination operand D and store the result in bits 31–16 of the destination accumulator. The source can be a 16-bit register, 6-bit short immediate, or 16-bit long immediate. This instruction is a 16-bit operation. The remaining bits of the destination operand D are not affected.

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the six bits are right aligned, and the remaining bits are zeroed to form a 16-bit source operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	*	*	*	—
CCR							

*	N	This bit is set if Bit 31 of the result is set.
*	Z	This bit is set if bits 31–16 of the result are 0.
*	V	This bit is always cleared.
√		This bit is changed according to the standard definition.
—		This bit is unchanged by the instruction.

OR Logical Inclusive OR OR

Instruction Formats and Opcodes:

OR S,D

23							16	15							8	7							0
DATA BUS MOVE FIELD																0	1	J	J	d	0	1	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

OR #xx,D

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

OR #xxxx,D

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0	d	0	1	0
IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	**JJ**	Source input register [X0,X1,Y0,Y1] (see **Table A-17** on page A-203)
{D}	**d**	Destination accumulator [A/B] (see **Table A-15** on page A-203)
{#xx}	**iiiiii**	6-bit Immediate Short Data
{#xxxx}		16-bit Immediate Long Data extension word

ORI OR Immediate with Control Register ORI

Operation:

#xx + D → D

Assembler Syntax:

OR(I) #xx,D

where + denotes the logical inclusive OR operator

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 (CCR) is specified as the destination operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

For CCR Operand:

* S This bit is set if Bit 7 of the immediate operand is set.
* L This bit is set if Bit 6 of the immediate operand is set.
* E This bit is set if Bit 5 of the immediate operand is set.
* U This bit is set if Bit 4 of the immediate operand is set.
* N This bit is set if Bit 3 of the immediate operand is set.
* Z This bit is set if Bit 2 of the immediate operand is set.
* V This bit is set if Bit 1 of the immediate operand is set.
* C This bit is 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 Formats and Opcodes:

	23							16	15							8	7							0
OR(I) #xx,D	0	0	0	0	0	0	0	0	i	i	i	i	i	i	i	i	1	1	1	1	1	0	E	E

ORI OR Immediate with Control Register ORI

Instruction Fields:

{D}	**EE**	Program Controller register [MR,CCR,COM,EOM] (see **Table A-18** on page A-204)
{#xx}	**iiiiiiii**	Immediate Short Data

REP Repeat Next Instruction REP

Operation:	Assembler Syntax:
LC → TEMP; [X or y]:ea → LC Repeat next instruction until LC = 1 TEMP → LC	REP [X or Y]:ea
LC → TEMP; [X or Y]:aa → LC Repeat next instruction until LC = 1 TEMP → LC	REP [X or 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 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 LC value is stored in an internal temporary register. The instruction's effective address specifies the address of the value that is to be loaded into the LC. All address register indirect addressing modes can be used. The Absolute Short Addressing and the Immediate Short Addressing modes can also be used. Four 0s are inserted to the left of the 12-bit immediate value to form the 16-bit value that is to be loaded into the LC.

If the System Stack register SSH is specified as a source operand, the system Stack Pointer (SP) is post-decremented by 1 after SSH has been read.

REP Repeat Next Instruction REP

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

REP [X or Y]:ea

23							16	15							8	7							0
0	0	0	0	0	1	1	0	0	1	M	M	M	R	R	R	0	S	1	0	0	0	0	0

REP [X or Y]:aa

23							16	15							8	7							0
0	0	0	0	0	1	1	0	0	0	a	a	a	a	a	a	0	S	1	0	0	0	0	0

REP #xxx

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

REP S

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

Instruction Fields:

{ea}	MMMRRR	Effective Address (see **Table A-24** on page A-206)
{X/Y}	S	Memory Space [X,Y] (see **Table A-22** on page A-205)
{aa}	aaaaaa	Absolute Short Address
{#xxx}	hhhhiiiiiiii	Immediate Short Data
{S}	dddddd	Source register [all on-chip registers] (see **Table A-27** on page A-207)

RESET Reset On-Chip Peripheral Devices RESET

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 $\overline{\text{RESET}}$ 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 stack error, NMI, illegal instruction, Trap, Debug request, and hardware reset interrupts.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

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

Instruction Fields: None.

RND Round Accumulator RND

Operation:

D + r → D (parallel move)

Assembler Syntax:

RND D (parallel move)

Description: Round the 40-bit value in the specified destination operand D and store the result in the destination accumulator (A or B). The contribution of the LSBs of the operand is rounded into the upper portion of the operand by adding a rounding constant to the LSBs of the operand. The upper portion of the destination accumulator contains the rounded result. The boundary between the lower portion and the upper portion is determined by the scaling mode bits S0 and S1 in the Status Register (SR).

Two types of rounding can be used: convergent rounding (also called round to nearest (even)) or two's-complement rounding. The type of rounding is selected by the Rounding Mode bit (RM) in the MR portion of the SR.

In both these rounding modes a rounding constant is first added to the unrounded result. The value of the rounding constant added is determined by the scaling mode bits S0 and S1 in the SR. A 1 is positioned in the rounding constant aligned with the MSB of the current LS portion, that is, the rounding constant weight is actually equal to half the weight of the upper portion's LSB.

The following table shows the rounding position and rounding constant as determined by the scaling mode bits:

				Rounding Constant				
S1	S0	Scaling Mode	Rounding Position	39–17	16	15	14	13–0
0	0	No Scaling	15	0. . . .0	0	1	0	0. . . .0
0	1	Scale Down	16	0. . . .0	1	0	0	0. . . .0
1	0	Scale Up	14	0. . . .0	0	0	1	0. . . .0

If convergent rounding is used, the result of this addition is tested and if all the bits of the result to the right of, and including, the rounding position are cleared, then the bit to the left of the rounding position is cleared in the result. This ensures that the result is not biased.

RND Round Accumulator RND

In both rounding modes, the Least Significant Bits (LSBs) of the result are cleared. The number of LSBs cleared is determined by the Scaling Mode bits in the Status Register (SR). All bits to the right of and including the rounding position are cleared in the result.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	—
CCR							

√ This bit is changed according to the standard definition.

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

RND D

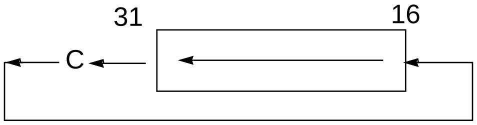
23 … 16 15 … 8	7 … 4	3 … 0
DATA BUS MOVE FIELD	0 0 0 1	d 0 0 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION		

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

ROL Rotate Left ROL

Operation:



AA0768

Assembler Syntax:

ROL D (parallel move)

Description: Rotate bits 31–16 of the destination operand D one bit to the left and store the result in the destination accumulator.The Carry bit (C) receives the previous value of Bit 31 of the operand.The previous value of the C bit is shifted into Bit 16 of the operand.This instruction is a 16-bit operation. The remaining bits of the destination operand D are not affected.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	*	*	*	*
CCR							

- * N This bit is set if Bit 31 of the result is set.
- * Z This bit is set if bits 31–16 of the result are 0.
- * V This bit is always cleared.
- * C This bit is set if Bit 31 of the destination operand is set, and cleared otherwise.
- √ This bit is changed according to the standard definition.
- — This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

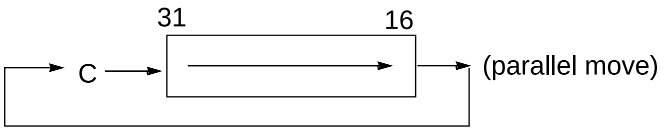
	23 16 15 8	7							0
ROL D	DATA BUS MOVE FIELD	0	0	1	1	d	1	1	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

ROR Rotate Right ROR

Operation:



AA0769

Assembler Syntax:

ROR D (parallel move)

Description: Rotate bits 31–16 of the destination operand D one bit to the right and store the result in the destination accumulator.The Carry bit (C) receives the previous value of Bit 16 of the operand.The previous value of the C bit is shifted into Bit 31 of the operand. This instruction is a 16-bit operation. The remaining bits of the destination operand D are not affected.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	*	*	*	*
CCR							

*	N	This bit is set if Bit 31 of the result is set.
*	Z	This bit is set if bits 31–16 of the result are 0.
*	V	This bit is always cleared.
*	C	This bit is set if Bit 31 of the destination operand is set, and cleared otherwise.
√		This bit is changed according to the standard definition.
—		This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

		23 16 15 8	7							0
ROR	D	DATA BUS MOVE FIELD	0	0	1	0	d	1	1	1
		OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{D}	**d**	Destination accumulator [A,B] (see **Table A-15** on page A-203)

RTI Return from Interrupt RTI

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 PC and SR values are lost.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
*	*	*	*	*	*	*	*
CCR							

*	S	This bit is set according to the value pulled from the stack.
*	L	This bit is set according to the value pulled from the stack.
*	E	This bit is set according to the value pulled from the stack.
*	U	This bit is set according to the value pulled from the stack.
*	N	This bit is set according to the value pulled from the stack.
*	Z	This bit is set according to the value pulled from the stack.
*	V	This bit is set according to the value pulled from the stack.
*	C	This bit is set according to the value pulled from the stack.

Instruction Formats and Opcode:

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

Instruction Fields: None.

RTS Return from Subroutine RTS

Operation:

SSH → PC; SP – 1 → SP

Assembler Syntax:

RTS

Description: Pull the Program Counter (PC) from the system stack. The previous PC value is lost. The Status Register (SR) is not affected.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

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

Instruction Fields: None.

SBC Subtract Long with Carry SBC

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) from the destination operand D and store the result in the destination accumulator. Long words (32-bit words) are subtracted from the 40-bit destination accumulator.

Note: The C 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 31 of the destination accumulator (A or B).

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.

Instruction Formats and Opcodes:

SBC S,D

23 16 15 8	7							0
DATA BUS MOVE FIELD	0	0	1	J	d	1	0	1
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{S} **J** Source register [X,Y] (see **Table A-16** on page A-203)
{D} **d** Destination accumulator [A,B] (see **Table A-15** on page A-203)

STOP Stop Instruction Processing STOP

Operation:	**Assembler Syntax:**
Enter the stop processing state and stop the clock oscillator	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 or the Debug Request JTAG command is detected. The clock oscillator is gated off internally. The Stop processing state is a low-power standby state.

During the Stop state, the destination port is in an idle state with the control signals held inactive, the data pins are high impedance, and the address pins are unchanged from the previous instruction.

If the exit from the Stop state is caused by a low level on the $\overline{\text{RESET}}$ pin, then the processor enters the reset processing state.

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. If no interrupt is pending, the processor will resume program execution at the instruction following the STOP instruction that caused the entry into the Stop state. Program execution (interrupt or normal flow) will resume after an internal delay counter counts:

- If the Stop Delay (SD, OMR[6]) bit is cleared—131,070 clock cycles
- If the Stop Delay (SD, OMR[6]) bit is set—24 clock cycles
- If the Stop Processing State (PSTP, PCTL1[5]) is set—8.5 clock cycles

During the clock stabilization count delay, all peripherals and external interrupts are cleared and re-enabled/arbitrated at the end of the count interval. If the $\overline{\text{IRQA}}$ pin is asserted when the STOP instruction is executed, the clock will not be gated off, and only the internal delay counter will be started.

STOP — Stop Instruction Processing — STOP

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

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

Instruction Fields:

None

SUB Subtract SUB

Operation:

D–S → D (parallel move)

D – #xx → D

D – #xxxx → D

Assembler Syntax:

SUB S, D (parallel move)

SUB #xx, D

SUB #xxxx,D

Description: Subtract the source operand from the destination operand D and store the result in the destination operand D. The source can be a register (16-bit word, 32-bit long word, or 40-bit accumulator), 6-bit short immediate, or 16-bit long immediate.

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the six bits are right-aligned and the remaining bits are zeroed to form a 16-bit source operand.

Note: The Carry bit (C) 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 31 of the destination accumulator (A or B). The C bit is always set correctly using accumulator source operands.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.

SUB Subtract SUB

Instruction Formats and Opcodes:

SUB S,D

23 16 15 8	7							0
DATA BUS MOVE FIELD	0	J	J	J	d	1	0	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

SUB #xx,D

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

SUB #xxxx,D

23							16	15							8	7							0
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0	d	1	0	0
IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	**JJJ**	Source register [B/A,X,Y,X0,Y0,X1,Y1] (see **Table A-19** on page A-204)
{D}	**d**	Destination accumulator [A/B] (see **Table A-15** on page A-203)
{#xx}	**iiiiii**	6-bit Immediate Short Data
{#xxxx}		16-bit Immediate Long Data extension word

SUBL Shift Left and Subtract Accumulators SUBL

Operation:

2 ∗ D – S → D (parallel move)

Assembler Syntax:

SUBL S,D ((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 0 is shifted into the LSB of D prior to the subtraction operation. The Carry bit (C) is set correctly if the source operand does not overflow as a result of the left shift operation. The Overflow bit (V) 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	*	√
CCR							

* V Set if overflow has occurred in the result or if the MS bit of the destination operand is changed as a result of the instruction's left shift

√ This bit is changed according to the standard definition

Instruction Formats and Opcodes:

SUBL S,D

23 16 15 8	7							0
DATA BUS MOVE FIELD	0	0	0	1	d	1	1	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{D} d Destination accumulator [A,B] (see **Table A-15** on page A-203)

{S} The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B

SUBR Shift Right and Subtract Accumulators AAA

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 (C) is always set correctly, and the Overflow bit (V) 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	√	√
CCR							

√ This bit is changed according to the standard definition.

Instruction Formats and Opcodes:

	23 16 15 8	7				0			
SUBR S,D	DATA BUS MOVE FIELD	0	0	0	0	d	1	1	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{D} d Destination accumulator [A,B] (see **Table A-15** on page A-203)

{S} The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B

Tcc Transfer Conditionally Tcc

Transfer Conditionally (Tcc)

Operation:	**Assembler Syntax:**
If cc, then S1 → D1	Tcc S1,D1
If cc, then S1 → D1 and S2 → D2	Tcc S1,D1 S2,D2
If cc, then S2 → D2	Tcc 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 conditions that the term "cc" can specify are listed on **Table A-47** on page A-213.

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Descriptions

Tcc Transfer Conditionally Tcc

Instruction Formats and Opcode:

		23							16	15							8	7							0
Tcc	S1,D1	0	0	0	0	0	0	1	0	C	C	C	C	0	0	0	0	0	J	J	J	d	0	0	0
Tcc	S1,D1 S2,D2	0	0	0	0	0	0	1	1	C	C	C	C	0	t	t	t	0	J	J	J	d	T	T	T
Tcc	S2,D2	0	0	0	0	0	0	1	0	C	C	C	C	1	t	t	t	0	0	0	0	0	T	T	T

Instruction Fields:

{cc}	**CCCC**	Condition code (see **Table A-48** on page A-214)
{S1}	**JJJ**	Source register [B/A,X0,Y0,X1,Y1] (see **Table A-29** on page A-208)
{D1}	**d**	Destination accumulator [A/B] (see **Table A-15** on page A-203)
{S2}	**ttt**	Source address register [R0–R7]
{D2}	**TTT**	Destination Address register [R0–R7]

TFR Transfer Data ALU Register TFR

Operation:

S → D (parallel move)

Assembler Syntax:

TFR S,D (parallel move)

Description: Transfer data from the specified source Data ALU register S to the specified destination Data ALU accumulator D. TFR uses the internal Data ALU data paths; thus, data does not pass through the data shifter/limiters. This allows the full 40-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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	—	—	—	—	—	—
CCR							

√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 16 15 8	7							0
TFR S,D	DATA BUS MOVE FIELD	0	J	J	J	d	0	0	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{S}	**JJJ**	Source register [B/A,X0,Y0,X1,Y1] (see **Table A-29** on page A-208)
{D}	**d**	Destination accumulator [A/B] (see **Table A-15** on page A-203)

TRAP Software Interrupt TRAP

Operation:

Begin trap exception process

Assembler Syntax:

TRAP

Description: Suspend normal instruction execution and begin TRAP 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and opcode:

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

TRAPcc Conditional Software Interrupt TRAPcc

Operation:	**Assembler Syntax:**
If cc then begin software exception processing	TRAPcc

Description: If the specified condition is true, normal instruction execution is suspended and software exception processing is initiated. The Interrupt Priority Level (I1,I0) is set to 3 in the Status Register (SR) if a long interrupt service routine is used. If the specified condition is false, instruction execution continues with the next instruction.

The conditions that the term “cc” can specify are listed on **Table A-47** on page A-213.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcode:

	23							16	15							8	7							0
TRAPcc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	C	C	C	C

Instruction Fields:

{cc} **CCCC** Condition code (see **Table A-48** on page A-214)

TST Test Accumulator TST

Operation:

S – 0 (parallel move)

Assembler Syntax:

TST S (parallel move)

Description: Compare the specified source accumulator S with 0 and set the condition codes accordingly. No result is stored although the condition codes are updated.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
√	√	√	√	√	√	*	—
CCR							

* V This bit is always cleared.
√ This bit is changed according to the standard definition.
— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23 16 15 8	7							0
TST S	DATA BUS MOVE FIELD	0	0	0	0	d	0	1	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

{S} **d** Source accumulator [A,B] (see **Table A-15** on page A-203)

VSL Viterbi Shift Left VSL

Viterbi Shift Left

Operation:

S[31:16] → X:ea; {S[14:0],i} → Y:ea

Assembler Syntax:

VSL S,i,L:ea

Description: Store the most significant part (16 bits) of the source accumulator at X memory (at effective address location), while for the least significant part (16 bits) of the source accumulator shift one bit to the left and insert 0 or 1 at the Least Significant Bit, according to operand i, and store the result at Y memory at the same address. This instruction enhances Viterbi algorithm performance (see **Viterbi Add-Compare-Select (ACS)** on page C-41).

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction.

Instruction Formats and Opcodes:

	23							16	15							8	7							0
VSL S,i,L:ea	0	0	0	0	1	0	1	S	1	1	M	M	M	R	R	R	1	1	0	i	0	0	0	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

Instruction Fields:

{S}	S	Source register A,B (see **Table A-15** on page A-203)
{i}	i	Bit value, 0 or 1 to be placed in the least significant bit of Y:<ea>
{ea}	MMMRRR	Effective address (see **Table A-21** on page A-205)

WAIT Wait for interrupt WAIT

Operation:

Disable clocks to the processor core and enter the Wait processing state

Assembler Syntax:

WAIT

Description: Enter the low-power standby 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 the WAIT instruction is executed when an interrupt is pending, the interrupt is processed. The effect is the same as if the processor never entered the Wait state. If the WAIT instruction is executed, the effect is the same as if the processor never entered the Wait state. When an unmasked interrupt or external (hardware) processor reset occurs, the processor leaves the Wait state and begins exception processing of the unmasked interrupt or reset condition. The processor also exits from the Wait state when the Debug Request ($\overline{\text{DE}}$) pin is asserted or when a Debug Request JTAG command is detected.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
—	—	—	—	—	—	—	—
CCR							

— This bit is unchanged by the instruction

Instruction Formats and Opcode:

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

Instruction Fields: None.

A.6 INSTRUCTION PARTIAL ENCODING

This section gives the encodings for the following:

- Various groupings of registers used in the instruction encodings
- Condition Code combinations
- Addressing
- Addressing modes

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.

A.6.1 Partial Encodings for Use in Instruction Encoding

Table A-15 Destination Accumulator Encoding

D/S	d/S/D
A	0
B	1

Table A-16 Data ALU Operands Encoding #1

S	J
X	0
Y	1

Table A-17 Data ALU Source Operands Encoding

S	J J
X0	0 0
Y0	0 1
X1	1 0
Y1	1 1

Table A-18 Program Control Unit Register Encoding

Register	**E E**
MR	0 0
CCR	0 1
COM	1 0
EOM	1 1

Table A-19 Data ALU Operands Encoding #2

S	**J J J**
B/A*	0 0 1
X	0 1 0
Y	0 1 1
X0	1 0 0
Y0	1 0 1
X1	1 1 0
Y1	1 1 1

* The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B.

Table A-20 Data ALU Operands Encoding #3

SSS/sss	**S,D**	**qqq**	**S,D**	**ggg**	**S,D**
000	reserved	000	reserved	000	B/A*
001	reserved	001	reserved	001	reserved
010	A1	010	A0	010	reserved
011	B1	011	B0	011	reserved
100	X0	100	X0	100	X0

Table A-20 Data ALU Operands Encoding #3 (Continued)

SSS/sss	S,D	qqq	S,D	ggg	S,D
101	Y0	101	Y0	101	Y0
110	X1	110	X1	110	X1
111	Y1	111	Y1	111	Y1

* The selected accumulator is B if the source two accumulator (selected by the **d** bit in the opcode) is A, or A if the source two accumulator is B.

Table A-21 Effective Addressing Mode Encoding #1

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
Absolute address	1 1 0 0 0 0
Immediate data	1 1 0 1 0 0

“r r r” refers to an address register R0–R7

Table A-22 Memory/Peripheral Space

Space	S
X Memory	0
Y Memory	1

Table A-23 Effective Addressing Mode Encoding #2

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
Absolute address	1 10 0 0 0

“r r r” refers to an address register R0–R7

Table A-24 Effective Addressing Mode Encoding # 3

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

“r r r” refers to an address register R0–R7

Table A-25 Effective Addressing Mode Encoding #4

Effective Addressing Mode	MMRRR
(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

“r r r” refers to an address register R0–R7

Table A-26 Triple-Bit Register Encoding

Code	1DD	DDD	TTT	NNN	FFF	EEE	VVV	GGG
000	—	A0	R0	N0	M0	—	VBA	SZ
001	—	B0	R1	N1	M1	—	SC	SR
010	—	A2	R2	N2	M2	EP	—	OMR
011	—	B2	R3	N3	M3	—	—	SP
100	X0	A1	R4	N4	M4	—	—	SSH
101	X1	B1	R5	N5	M5	—	—	SSL
110	Y0	A	R6	N6	M6	—	—	LA
111	Y1	B	R7	N7	M7	—	—	LC

Table A-27 Six-Bit Encoding For all On-Chip Registers

Destination Register	D D D D D D / d d d d d d
4 registers in Data ALU	0 0 0 1 D D
8 accumulators in Data ALU	0 0 1 D D D
8 address registers in AGU	0 1 0 T T T
8 address offset registers in AGU	0 1 1 N N N
8 address modifier registers in AGU	1 0 0 F F F
1 address register in AGU	1 0 1 E E E
2 program controller register	1 1 0 V V V
8 program controller registers	1 1 1 G G G

See **Table A-26** on page A-207 for the specific encodings.

Table A-28 Long Move Register Encoding

S	S1	S2	S S/L	D	D1	D2	D Sign Ext	D Zero	LLL
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

Instruction Partial Encoding

Table A-28 Long Move Register Encoding (Continued)

S	S1	S2	S S/L	D	D1	D2	D Sign Ext	D Zero	LLL
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

Table A-29 Data ALU Source Registers Encoding

S	J J J
B/A*	000
X0	100
Y0	101
X1	110
Y1	111

* The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B.

Table A-30 AGU Address and Offset Registers Encoding

Destination Address Register D	dddd
R0-R7	onnn
N0-N7	1nnn

Table A-31 Data ALU Multiply Operands Encoding #1

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

Note: Only the indicated S1 * S2 combinations are valid. X1 * X1 and Y1 * Y1 are not valid.

Table A-32 Data ALU Multiply Operands Encoding #2

S	Q Q
Y1	0 0
X0	0 1
Y0	1 0
X1	1 1

Table A-33 Data ALU Multiply Operands Encoding #3

S	qq
X0	0 0
Y0	0 1
X1	1 0
Y1	1 1

Table A-34 Data ALU Multiply Sign Encoding

Sign	k
+	0
–	1

Table A-35 Data ALU Multiply Operands Encoding #4

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

Table A-36 Five-Bit Register Encoding #1

D/S	ddddd / eeeee	D/S	ddddd / eeeee
X0	0 0 1 0 0	B2	0 1 0 1 1
X1	0 0 1 0 1	A1	0 1 1 0 0
Y0	0 0 1 1 0	B1	0 1 1 0 1
Y1	0 0 1 1 1	A	0 1 1 1 0
A0	0 1 0 0 0	B	0 1 1 1 1
B0	0 1 0 0 1	R0-R7	1 0 r r r
A2	0 1 0 1 0	N0-N7	1 1 n n n

“r r r” = Rn number, “n n n” = Nn number

Table A-37 Immediate Data ALU Operand Encoding

n	ssss	constant
1	0 0 0 1	0100000000000000
2	0 0 1 0	0010000000000000
3	0 0 1 1	0001000000000000
4	0 1 0 0	0000100000000000
5	0 1 0 1	0000010000000000
6	0 1 1 0	0000001000000000
7	0 1 1 1	0000000100000000
8	1 0 0 0	0000000010000000
9	1 0 0 1	0000000001000000
10	1 0 1 0	0000000000100000
11	1 0 1 1	0000000000010000
12	1 1 0 0	0000000000001000
13	1 1 0 1	0000000000000100
14	1 1 1 0	0000000000000010
15	1 1 1 1	0000000000000001

Table A-38 Write Control Encoding

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

Table A-39 ALU Registers Encoding

Destination Register	D D D D
4 registers in Data ALU	0 1 D D
8 accumulators in Data ALU	1 D D D

See **Table A-26** on page A-207 for the specific encodings.

Table A-40 X:R Operand Registers Encoding

S1,D1	f f	D2	F
X0	0 0	Y0	0
X1	0 1	Y1	1
A	1 0		
B	1 1		

Table A-41 R:Y Operand Registers Encoding

D1	e	S2,D2	f f
X0	0	Y0	0 0
X1	1	Y1	0 1
		A	1 0
		B	1 1

Table A-42 Single Bit Special Register Encoding Tables

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-43 X:Y: Move Operands Encoding Tables

X Effective Addressing Mode	MMRRR
(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
Y Effective Addressing Mode	**mmrr**
(Rn)+Nn	0 1 t t
(Rn)–	1 0 t t
(Rn)+	1 1 t t
(Rn)	0 0 t t

where the following apply:

"s s s" refers to an address register R0–R7.

"t t" refers to an address register R4–R7 or R0–R3 in the opposite address register bank from the one used in the X effective address.

S1,D1	e e	S2,D2	f f
X0	0 0	Y0	0 0
X1	0 1	Y1	0 1
A	1 0	A	1 0
B	1 1	B	1 1

Table A-44 Signed/Unsigned Partial Encoding #1

ss/su/uu	ss
ss	00
su	10
uu	11
(Reserved)	01

Table A-45 Signed/Unsigned Partial Encoding #2

su/uu	s
su	0
uu	1

Table A-46 Five-Bit Register Encoding #2

S1,D1	ddddd
M0-M7	00nnn
EP	01010
VBA	10000
SC	10001
SZ	11000
SR	11001
OMR	11010
SP	11011
SSH	11100
SSL	11101
LA	11110
LC	11111

where “n n n” = Mn number (M0–M7)

Table A-47 Condition Codes Computation Equations

	“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

Table A-47 Condition Codes Computation Equations (Continued)

	"cc" Mnemonic	Condition
ES	Extension Set	E=1
GE	Greater than or Equal	$N \oplus V=0$
GT	Greater Than	$Z+(N \oplus V)=0$
LC	Limit Clear	L=0
LE	Less than or Equal	$Z+(N \oplus V)=1$
LS	Limit Set	L=1
LT	Less Than	$N \oplus V=1$
MI	Minus	N=1
NE	Not Equal	Z=0
NR	Normalized	$Z+(\overline{U}\bullet\overline{E})=1$
PL	Plus	N=0
NN	Not Normalized	$Z+(\overline{U}\bullet\overline{E})=0$

where the following apply:

$\overline{U}$ denotes the logical complement of U.

+ denotes the logical OR operator.

• denotes the logical AND operator.

⊕ denotes the logical Exclusive OR operator.

Table A-48 Condition Codes Encoding

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

Table A-48 Condition Codes Encoding (Continued)

Mnemonic	C C C C	Mnemonic	C C C C
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

The condition code computation equations are listed in **Table A-47** on page A-213.

A.6.2 Parallel Instruction Encoding of the Operation Code

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

A.6.2.1 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 (see **Table A-31** on page A-208)
kkk = three unencoded bits k2, k1, k0
d = destination accumulator
d = 0 → A
d = 1 → B

Table A-49 Operation Code K0–2 Decode

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

A.6.2.2 Non-Multiply 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 = **0 J J J D k k k** where

J J J = 1/2 instruction number
k k k = 1/2 instruction number
D = 0 → A
D = 1 → B

Table A-50 Non-Multiply Instruction Encoding

J J J	**D = 0 Src Oper**	**D = 1 Src Oper**	**k k k**							
			0 0 0	**0 0 1**	**0 1 0**	**0 1 1**	**1 0 0**	**1 0 1**	**1 1 0**	**1 1 1**
0 0 0	B	A	MOVE[1]	TFR	ADDR	TST	*	CMP	SUBR	CMPM
0 0 1	B	A	ADD	RND	ADDL	CLR	SUB	*	SUBL	NOT
0 1 0	B	A	—	—	ASR	LSR	—	—	ABS	ROR
0 1 1	B	A	—	—	ASL	LSL	—	—	NEG	ROL
0 1 0	X1 X0	X1 X0	ADD	ADC	—	—	SUB	SBC	—	—
0 1 1	Y1 Y0	Y1 Y0	ADD	ADC	—	—	SUB	SBC	—	—
1 0 0	X0_0	X0_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
1 0 1	Y0_0	Y0_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
1 1 0	X1_0	X1_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
1 1 1	Y1_0	Y1_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM

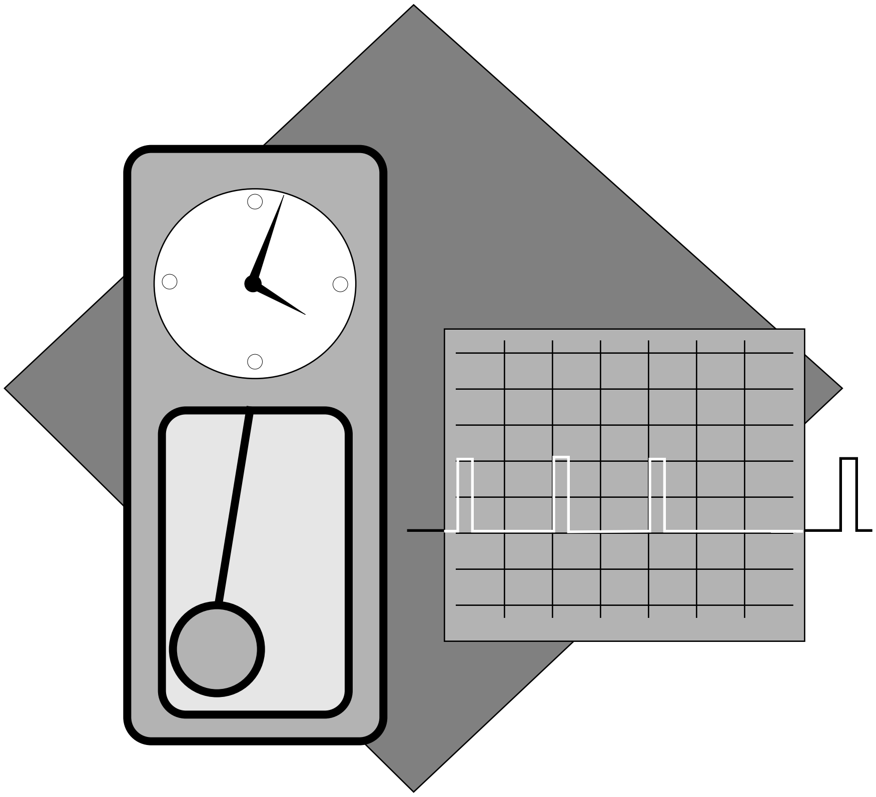
Note: 1. Special Case #1
2. * = Reserved

Table A-51 Special Case #1

OPERCODE	**Operation**
0 0 0 0 0 0 0 0	MOVE
0 0 0 0 1 0 0 0	reserved

APPENDIX B

INSTRUCTION TIMING



B.1 INTRODUCTION

This appendix describes the various aspects of execution timing analysis for each instruction mnemonic and for various instruction sequences. The section consists of the following tables and information:

- Tables showing how to calculate DSP56600 core instruction timing for each instruction mnemonic (instruction timing)
- Tables showing the number of instruction program words for each instruction mnemonic (instruction program words)
- Description of various sequences that cause timing delays and stalls in the execution (instruction sequence delays)
- Description of various instruction sequences that are forbidden and cause undefined operation (instruction sequence restrictions)

B.2 INSTRUCTION TIMING

The number of oscillator clock cycles per instruction depends on many factors, including the number of words per instruction, the addressing mode, whether the instruction fetch pipeline is full, the number of external bus accesses, cache hit/miss/burst, and the number of wait states inserted in each external access.

Table B-1 lists instruction timing, and is based on the following assumptions:

- All instruction cycles are counted in clock cycles.
- The instruction fetch pipeline is full.

The following terms are used inside the table:

- **T**—clock cycles for the normal case:
  - All instructions fetched from the internal program memory
  - No interlocks with previous instructions
  - Addressing mode is the Post-Update mode (post-increment, post-decrement and post offset by N) or the No-Update mode.
- **+ pru**—Pre-update specifies clock cycles added for using the pre-update addressing modes (pre-decrement and offset by N addressing modes).

- **+ lab**—Long absolute specifies clock cycles added for using the Long Absolute Address mode.
- **+ lim**—Long immediate specifies clock cycles added for using the long immediate data addressing mode.

Note: A dash under one or more of the columns **pru**, **lab,** or **lim** indicates that this column is not applicable to the corresponding instruction.

Table B-1 Instruction Timing, Word Count, and Encoding

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
ADD	ADD #iiiii,D	2	—	—	—
	ADD #iii,D	1	—	—	—
AND	AND #iiiiii,D	2	—	—	—
	AND #iii,D	1	—	—	—
ANDI	ANDI EE	3	—	—	—
ASL	ASL #ii,S,D	1	—	—	—
	ASL sss,S,D	1	—	—	—
ASR	ASR sss,S,D	1	—	—	—
	ASR #ii,S,D	1	—	—	—
Bcc	Bcc (PC + Rn)	4	—	—	—
	Bcc (PC + aa)	4	—	—	—
BCHG	BCHG #bbbb ,S:<aa>	2	—	—	—
	BCHG #bbbb ,S:<ea>	2	1	1	—
	BCHG #bbbb ,S:<pp>	2	—	—	—
	BCHG #bbbb ,S:<qq>	2	—	—	—
	BCHG #bbbb ,DDDDDD	2	—	—	—
BCLR	BCLR #bbbb ,S:<pp>	2	—	—	—
	BCLR #bbbb ,S:<ea>	2	1	1	—
	BCLR #bbbb ,S:<aa>	2	—	—	—

Table B-1 Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
BCLR	BCLR #bbbb ,S:<qq>	2	—	—	—
	BCLR #bbbb ,DDDDDD	2	—	—	—
BRA	BRA (PC + Rn)	4	—	—	—
	BRA (PC + aa)	4	—	—	—
BRKcc	BRKcc	5	—	—	—
BScc	BScc (PC + Rn)	4	—	—	—
	BScc (PC + aa)	4	—	—	—
BSET	BSET #bbbb ,S:<pp>	2	—	—	—
	BSET #bbbb ,S:<ea>	2	1	1	—
	BSET #bbbb ,S:<aa>	2	—	—	—
	BSET #bbbb ,DDDDDD	2	—	—	—
	BSET #bbbb ,S:<qq>	2	—	—	—
BSR	BSR (PC + Rn)	4	—	—	—
	BSR (PC + aa)	4	—	—	—
BTST	BTST #bbbb ,S:<pp>	2	—	—	—
	BTST #bbbb ,S:<ea>	2	1	1	—
	BTST #bbbb ,S:<aa>	2	—	—	—
	BTST #bbbb ,DDDDDD	2	—	—	—
	BTST #bbbb ,S:<qq>	2	—	—	—
CLB	CLB S,D	1	—	—	—
CMP	CMP #iiiiii,D	2	—	—	—
	CMP #iii,D	1	—	—	—
CMPU	CMPU ggg,D	1	—	—	—

Table B-1 Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
DEBUG/ DEBUGcc	DEBUG	1	—	—	—
	DEBUGcc	5	—	—	—
DEC	DEC	1	—	—	—
DIV	DIV	1	—	—	—
DMAC	DMAC S1,S2,D (ss,su,uu)	1	—	—	—
DO	DO #xxx,aaaa	5	—	—	—
	DO DDDDDD,aaaa	5	—	—	—
	DO S:<ea>,aaaa	5	1	—	—
	DO S:<aa>,aaaa	5	—	—	—
DO FOREVER	DO FOREVER ,(aaaa)	4	—	—	—
ENDDO	ENDDO	1	—	—	—
EOR	EOR #iiiiii,D	2	—	—	—
	EOR #iii,D	1	—	—	—
EXTRACT	EXTRACT SSS,s,D	1	—	—	—
	EXTRACT #iiii,s,D	2	—	—	—
EXTRACTU	EXTRACTU SSS,s,D	1	—	—	—
	EXTRACTU #iiii,s,D	2	—	—	—
IFcc	IFcc(.U)	1	—	—	—
ILLEGAL	ILLEGAL	5	—	—	—
INC	INC	1	—	—	—
INSERT	INSERT SSS,qqq,D	1	—	—	—
	INSERT #iiii,qqq,D	2	—	—	—
Jcc	Jcc aa	4	—	—	—
	Jcc ea	4	0	0	—

Table B-1 Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
JCLR	JCLR #bbbb ,S:<ea>,aaaa	4	1	—	—
	JCLR #bbbb ,S:<pp>,aaaa	4	—	—	—
	JCLR #bbbb ,S:<aa>,aaaa	4	—	—	—
	JCLR #bbbb ,DDDDDD,aaaa	4	—	—	—
	JCLR #bbbb ,S:<qq>,aaaa	4	—	—	—
JMP	JMP aa	3	—	—	—
	JMP ea	3	1	1	—
JScc	JScc aa	4	—	—	—
	JScc ea	4	0	0	—
JSCLR	JSCLR #bbbb ,S:<pp>,aaaa	4	—	—	—
	JSCLR #bbbb ,S:<ea>,aaaa	4	1	—	—
	JSCLR #bbbb ,S:<aa>,aaaa	4	—	—	—
	JSCLR #bbbb ,DDDDDD,aaaa	4	—	—	—
JSCLR (continued)	JSCLR #bbbb ,S:<qq>,aaaa	4	—	—	—
JSET	JSET #bbbb ,S:<pp>,aaaa	4	—	—	—
	JSET #bbbb ,S:<ea>,aaaa	4	1	—	—
	JSET #bbbb ,S:<aa>,aaaa	4	—	—	—
	JSET #bbbb ,DDDDDD,aaaa	4	—	—	—
	JSET #bbbb ,S:<qq>,aaaa	4	—	—	—
JSR	JSR aa	3	—	—	—
	JSR ea	3	1	1	—
JSSET	JSSET #bbbb ,S:<pp>,aaaa	4	—	—	—
	JSSET #bbbb ,S:<ea>,aaaa	4	1	—	—

Table B-1 Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
JSSET	JSSET #bbbb ,S:<aa>,aaaa	4	—	—	—
	JSSET #bbbb ,DDDDDD,aaaa	4	—	—	—
	JSSET #bbbb ,S:<qq>,aaaa	4	—	—	—
LSL	LSL sss,D	1	—	—	—
	LSL #ii,D	1	—	—	—
LSR	LSR #ii,D	1	—	—	—
	LSR sss,D	1	—	—	—
LRA	LRA (PC + Rn) → 0DDDDD	3	—	—	—
	LRA (PC + aaaa) → 0DDDDD	3	—	—	—
LUA, LEA	LUA ea → 0DDDDD	3	—	—	—
	LUA (Rn + aa) → 01DDDD	3	—	—	—
MACI	MACI ± #iiiiii,QQ,D	2	—	—	—
MAC	MAC ± 2**s,QQ,d	1	—	—	—
	MAC S1,S2,D (su,uu)	1	—	—	—
MACRI	MACRI ± #iiiiii,QQ,D	2	—	—	—
MACR	MACR ±2**s,QQ,d	1	—	—	—
MAX	MAX A,B	1	—	—	—
MAXM	MAXM A,B	1	—	—	—
MERGE	MERGE SSS,D	1	—	—	—
MOVE	No parallel data Move (DALU)	1	—	—	—
	MOVE #xx → DDDDD	1	—	—	—
	MOVE ddddd → DDDDD	1	—	—	—
	U move	1	—	—	—
	MOVE S:<ea>,DDDDD	1	1	1	1

Table B-1 Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
MOVE	MOVE S:<aa>,DDDDD	1	—	—	—
	MOVE S:<Rn + aa>,DDDD	2	—	—	—
	MOVE S:<Rn + aaaa>,DDDDDD	3	—	—	—
	MOVE d → X Y:<ea>,YY	1	1	1	1
	MOVE X:<ea>,XX & d → Y	1	1	1	1
	MOVE A → X:<ea> X0 A	1	1	—	—
	MOVE B → X:<ea> X0 B	1	1	—	—
	MOVE Y0 → A A Y:<ea>	1	1	—	—
	MOVE Y0 → B B Y:<ea>	1	1	—	—
	MOVE L:<ea>,LLL	1	1	1	—
	MOVE L:<aa>,LLL	1	—	—	—
	MOVE X:<ea>,XX & Y:<ea>,YY	1	—	—	—
MOVEC	MOVEC #xx → 1DDDDD	1	—	—	—
	MOVEC S:<ea>,1DDDDD	1	1	1	1
	MOVEC S:<aa>,1DDDDD	1	—	—	—
	MOVEC DDDDDD,1ddddd	1	—	—	—
MOVEM	MOVEM P:<ea>,DDDDDD	6	1	1	—
	MOVEM P:<aa>,DDDDDD	6	—	—	—
MOVEP	MOVEP S:<pp>,s:<ea>	2	1	1	0
	MOVEP S:<pp>,P:<ea>	6	1	1	—
	MOVEP S:<pp>,DDDDDD	1	—	—	—
	MOVEP X:<qq>,s:<ea>	2	1	1	0
	MOVEP Y:<qq>,s:<ea>	2	1	1	0
	MOVEP X:<qq>,DDDDDD	1	—	—	—

Table B-1 Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
MOVEP	MOVEP Y:<qq>,DDDDDD	1	—	—	—
	MOVEP S:<qq>,P:<ea>	6	1	1	-
MPY	MPY S1,S2,D (su,uu)	1	—	—	—
	MPY ± 2**s,QQ,d	1	—	—	—
MPYI	MPYI ± #iiiiii,QQ,D	2	—	—	—
MPYR	MPYR ± 2**s,QQ,d	1	—	—	—
MPYRI	MPYRI ± #iiiiii,QQ,D	2	—	—	—
NOP	NOP	1	—	—	—
NORMF	NORMF SSS,D	1	—	—	—
OR	OR #iiiiii,D	2	—	—	—
	OR #iii,D	1	—	—	—
ORI	ORI EE	3	—	—	—
REP	REP #xxx	5	—	—	—
	REP DDDDDD	5	—	—	—
	REP S:<ea>	5	1	—	—
	REP S:<aa>	5	—	—	—
RESET	RESET	7	—	—	—
RTI/RTS	RTI	3	—	—	—
	RTS	3	—	—	—
STOP	STOP	10	—	—	—
SUB	SUB #iiiiii,D	2	—	—	—
	SUB #iii,D	1	—	—	—

Table B-1 Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
Tcc	Tcc JJJ → D ttt TTT	1	—	—	—
	Tcc JJJ → D	1	—	—	—
	Tcc ttt → TTT	1	—	—	—
TRAP/ TRAPcc	TRAP	9	—	—	—
	TRAPcc	9	—	—	—
VSL	VSL S,i,L:ea	1	1	1	—
WAIT	WAIT	10	—	—	—

B.3 INSTRUCTION SEQUENCE DELAYS

Because of the pipelined nature of the DSP56600 core, certain instruction sequences can cause a delay in the execution of instructions involved in that sequences. Most of these sequences are caused by a source-destination conflict or by the need to access the external bus. These are the six types of sequence delays:

- External bus wait states
- Instruction fetch delays
- Data ALU interlocks
- Address generation interlocks
- Stack extension delays
- Pipeline interlocks

B.3.1 External Bus Wait States

An external bus wait state is caused by an instruction accessing the external bus for data read or write. In this case, the execution time of the instruction is increased by the number of clock cycles equal to the number of wait states that is programmed for that external data access. The exact number of wait states depends on the type of memory accessed, as described in **Bus Control Register** on page 9-5.

B.3.2 Instruction Fetch Delays

At an external instruction fetch, the effective number of stall states in the pipeline is the number specified in the Bus Control Register (BCR).

B.3.3 Data ALU Interlock

A Data ALU interlock can be caused by one of the following sequences:

- Arithmetic stall
- Transfer stall
- Status stall

B.3.3.1 Arithmetic Stall

An arithmetic stall interlock is caused by an instruction that uses one of the Data ALU registers (A0, A1, A2, B0, B1, or B2) or accumulators (A or B) as a source register for the move portion of that instruction when the preceding instruction was an arithmetic instruction (an instruction that uses the internal Data ALU data paths) that used the same accumulator as its destination. The execution of the initiating instruction is delayed by one clock cycle.

B.3.3.2 Transfer Stall

A transfer stall interlock is caused by an instruction that uses one of the Data ALU registers (A0, A1, A2, B0, B1, or B2) or accumulators (A or B) as a source register for the move portion of that instruction when the preceding instruction used the corresponding accumulator or one of the Data ALU registers that comprise the accumulator as its destination register in the move portion of that instruction. The execution of the initiating instruction is delayed by one instruction cycle.

B.3.3.3 Status Stall

A status stall interlock is caused by an instruction that reads the contents of the Status Register (SR) for either move operation or bit testing when the preceding or the second preceding instruction was an arithmetic instruction (an instruction that uses the internal Data ALU data paths). The execution of the initiating instruction is delayed by 2 instruction cycles for a move operation or 1 instruction cycle for bit testing.

B.3.4 Address Registers Interlocks

An address register interlock can be caused by one of the following sequences.

B.3.4.1 Conditional Transfer Interlock

A conditional transfer interlock is caused by a Transfer On-Condition (Tcc) instruction followed by an instruction that explicitly specifies one of the address generation registers (R0–R7) as its source operand. The execution of the second instruction is delayed by one instruction cycle.

B.3.4.2 Address Generation Interlock

An address generation interlock is caused by a move portion of an instruction that uses one of the AGU registers R0–R7 for address generation or for address calculation, while one of the three preceding instruction cycles uses one of the register set (Ri, Ni or Mi) members as a destination register in its move portion. Consider **Example B-1**:

Example B-1 Address Generation Interlock

```
I1 MOVE #$addr,R0

I2 NOP

I3 NOP

I4 NOP

I5 MOVE #$offset,N0

I6 MOVE X:(R0)+,Y1
```

In this example, the instruction I6 causes an address generation interlock because it used R0 as the source for address generation on the X Address Bus while the preceding instruction, I5, used N0 as its destination.

Three types of address generation interlock exist, as follows:

- Type0
- Type1
- Type2

These types depend on the distance, in term of clock cycles, between the instruction causing the interlock and the preceding instruction that used the AGU register as a destination. **Figure B-1** describes an example to each of the types:

Instruction Sequence Delays

Type0 Interlock	Type1 Interlock	Type2 Interlock
`I1 MOVE #$addr,R0`	`I1 MOVE #$addr,R0`	`I1 MOVE #$addr,R0`
`I2 MOVE X:(R0)+,Y1`	`I2 CLR A`	`I2 CLR A`
	`I3 MOVE X:(R0)+,Y1`	`I3 INC B`
		`I4 MOVE X:(R0)+,Y1`
Three NOP instructions are inserted	Two NOP instructions are inserted	One NOP instruction is inserted

AA0770

Figure B-1 Types of Address Generation Interlock

When a Type0 Address Generation Interlock is detected (during the decoding of I2 in the example), three NOP clock cycles are automatically inserted before the execution of the instruction starts. When a Type1 Address Generation Interlock is detected (during the decoding of I3 in the example), two NOP clock cycles are automatically inserted before the execution of the instruction starts. When a Type2 Address Generation Interlock is detected (during the decoding of I4 in the example), one NOP clock cycle is inserted before the execution of the instruction starts.

Note: Only clock cycles are counted to determine when interlock cycles should be inserted.

Whenever an instruction using one of the AGU registers as an address generation enters the decoding stage of the DSP56600 core, the distance from that instruction to the preceding instruction that used the register as destination is measured in term of clock cycles to determine the existence and type of address generation interlock. Once an address generation interlock is detected, the appropriate number of NOP clock cycles is inserted. The following instructions take these additional cycles into account for the detection of a possible new address generation interlock. **Example B-2** demonstrates this feature.

Example B-2 Detection of Address Generation Interlock

```
I1 MOVE #$addr,R0

I2 CLR A

I3 MOVE X:(R0)+,Y1

I4 MOVE X:(R0)+,Y0
```

In this example, a Type1 address generation interlock is detected during the decoding phase of I, 3 and two NOP cycles are inserted before the execution of that instruction. During the decoding of I4, no address generation interlock is detected, so no NOP cycles are inserted.

However, if I3 had been an instruction that did not use R0, a Type2 address generation interlock would have been detected during the decoding phase of I4, and one NOP cycle would have been inserted before the execution of that instruction.

B.3.5 Stack Extension Delays

Some instructions access the System Stack (SS) as part of their normal activity. Whenever the SS is either completely full or completely empty, the special stack extension mechanism is engaged and the access is completed only after an access to data memory is automatically performed. This delays the decoding and the execution phases of that instruction. A stack-full or a stack-empty state is defined by the contents of the Stack Counter (SC) register. When the stack counter equals 14, it means that the on-chip hardware stack has fourteen words (a stack word is a 48-bit long word combined from the low and the high portions of the stack) inside. The stack is declared as stack-full, and any additional push operation activates the stack extension mechanism. When the stack counter equals 2, it means that the on-chip hardware stack has only two words inside. The stack is declared as stack-empty, and any additional pop operations activate the stack extension mechanism.

The instructions/cases listed in **Table B-2** cause an access to the system stack and may engage the stack extension mechanism:

Table B-2 Instructions that Access the System Stack

Instruction	Description
SUBcc	This denotes all the conditional and unconditional Jump to Subroutine instructions (e.g., JSR, JSSET, and so forth). These instructions perform a stack PUSH operation that stores the PC and the SR on top of the stack, for the use of the 'Return from Subroutine' instruction that will terminate the subroutine execution.
RET	This denotes the two Return from Subroutine instructions, RTS and RTI. These instructions perform a stack POP operations that pulls the PC and (optionally) the SR out from the top of stack in order to return back to the calling procedure and to restore the status bits and loop flag state.

Table B-2 Instructions that Access the System Stack (Continued)

Instruction	Description
END-OF-DO	This condition is achieved by the internal hardware inside the Program Control Unit. This hardware detects the case where a fetch from the last address of a loop is initiated when the Loop Counter equals 1. This condition defines the end of the loop, thus performing a stack POP operation. This POP operation restores the loop flag, purges the top of stack (PC:SR) and pulls LA and LC from the new top of stack.
LOOP	This denotes all the hardware-loop initiating instructions (e.g., DO) with all their options. These instructions perform a stack double-PUSH operation that first stores the previous values of LA and LC on top of the stack. Then the DO instruction stores the contents of SR and PC on the new top of stack. This PC value is used every loop iteration in order to go back to the top of loop location and start fetch from there. DO performs two accesses to the stack instead of the normal single access done by most stack operations.
ENDDO	This special instruction forces an end-of-do condition during a hardware loop. Like END-OF-DO, ENDDO performs two accesses to the stack instead of the normal single access done by most stack operations.
SSHWR	This denotes all the explicit stack PUSH instructions that use SSH as their destination (e.g., the MOVE R0,SSH instruction).
SSHRD	This denotes all the explicit stack POP instructions that use SSH as their source (e.g., the MOVE SSH,Y1 instruction).

Table B-3 shows how many clock cycles are added in the various instructions/cases described on the previous pages.

Table B-3 Stack Extension Delays

CASE	Stack Full Condition (+ clock cycles)	Stack Empty Condition (+ clock cycles)
SUBcc	2	—
RET	—	3
END-OF-DO	—	5
DO	4	—

Table B-3 Stack Extension Delays (Continued)

CASE	Stack Full Condition (+ clock cycles)	Stack Empty Condition (+ clock cycles)
ENDDO	—	5
SSHWR	2	—
SSHRD	—	3

B.3.6 Program Flow-Control Delays

During the execution of flow-control instructions, some boundary cases exist and introduce interlocks to the program flow. These interlocks lengthen the decoding phase of the instructions, thus delaying the execution of them. The following sequences represent unusual operations that probably would never be used. The detection of these cases and hence the generation of interlocks is done in order to maintain an object code compatibility between the DSP56600 core and the 56000 family of DSPs.

The following terms are used in this subsection:

- I1—An address of an instruction, where I2, I3, and I4 are used to indicate the next instructions in the program flow
- MOVE—any type of MOVE, MOVEM, MOVEP, MOVEC, BSET, BCHG, BCLR, and BTST
- LA—the last address of a DO LOOP
- LA – 1—the address of an instruction word located at LA – 1
- CR—Control Register, every one of the registers LA, LC, SR, SP, SSH, SSL, and OMR

B.3.6.1 JMP to LA or to LA – 1

When I1 is any type of JMP with its target address equal to LA, the decoding phase of the instruction following the instruction at LA is delayed by 2 clock cycles.

When I1 is any type of JMP with its target address equal to LA – 1, the decoding phase of the instruction following the instruction at LA is delayed by 1 clock cycle.

B.3.6.2 RTI to LA or to LA – 1

When I1 is an RTI instruction whose return address is LA, the decoding phase of the instruction following the instruction at LA is delayed by 2 clock cycles.

When I1 is an RTI instruction whose return address is LA – 1, the decoding phase of the instruction following the instruction at LA is delayed by 1 clock cycle.

B.3.6.3 Conditional Instructions

When I1 is a conditional change of flow instruction (such as Jcc) and the condition is false, the decoding phase of I2 is delayed by 1 clock cycle.

B.3.6.4 Interrupt Abort

When I1 is an instruction with a decoding phase that is longer than one cycle, it may be aborted by the Interrupt Control Unit. In this case, a 1 clock cycle “hole” is inserted into the pipeline, after which the instruction at the interrupt vector is decoded.

B.3.6.5 Degenerated DO loop

When I1 is a DO loop but the loop contains only one instruction, the decoding phase of I1 is lengthened by 1 clock cycle.

B.3.6.6 Annulled REP and DO

If the repeat count of a REP instruction is zero, then the decoding phase of the REP instruction is lengthened by 1 clock cycle. If the repeat count of a DO instruction is zero, then the decoding phase of the DO instruction is lengthened by 3 clock cycles.

B.4 INSTRUCTION SEQUENCE RESTRICTIONS

Because of the pipelined nature of the DSP56600 core central processor, certain instruction sequences are forbidden. Use of these sequences causes undefined operation. Most of these restricted sequences cause contention for an internal resource, such as the Stack Register. The DSP Assembler flags these as assembly errors.

The following terms are used in this subsection:

- MOVE—any type of MOVE, MOVEM, MOVEP, MOVEC
- MOVEM—any type of MOVE to/from the Program space
- LA—the last address of a DO LOOP

- Two-words <inst>—a double-word instruction in which the second word is used as an immediate data or absolute address
- Single-word <inst>—an instruction with an addressing mode that does not need a second word extension

B.4.1 Restrictions Near the End of DO Loops

Proper DO loop operation is not guaranteed if an instruction sequence similar to one of the sequences described below is used.

B.4.1.1 At LA – 5

The following instructions should not start at address LA – 5:

- Single-word or Two-words MOVE to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}

B.4.1.2 At LA – 4

The following instructions should not start at address LA – 4:

- Single-word or Two-words MOVE to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}

B.4.1.3 At LA – 3

The following instructions should not start at address LA – 3:

- BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE from SSH, SSL
- Two-words JMP, Jcc, JSR, JScc
- JSET, JCLR, JSSET, JSCLR
- Two-words MOVEM

B.4.1.4 At LA – 2

The following instructions should not start at address LA – 2:

- DO, DO FOREVER

- MOVE to/from {LA, LC, SP,SC, SSH, SSL,SZ, VBA, OMR}
- BCHG, BSET, BCLR, BTST on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- JMP, Jcc, JSR, JScc, JSET, JCLR, JSSET, JSCLR, BRA, Bcc, BSR, BScc
- MOVEM
- ANDI, ORI on MR
- BRKcc, ENDDO, REP
- STOP, WAIT, DEBUG, DEBUGcc, TRAP, TRAPcc, ILLEGAL

B.4.1.5 At LA – 1

The following instructions should not start at address LA – 1:

- DO, DO FOREVER
- MOVE to/from {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BCHG, BSET, BCLR, BTST on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- JMP, Jcc, JSR, JScc, JSET, JCLR, JSSET, JSCLR, BRA, Bcc, BSR, BScc
- MOVEM
- ANDI, ORI on MR
- BRKcc, ENDDO, REP
- STOP, WAIT, DEBUG, DEBUGcc, TRAP, TRAPcc, ILLEGAL

B.4.1.6 At LA

The following instructions should not start at address LA:

- Any Two-word instruction
- MOVE to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE from SSH, SSL
- BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BTST on SSH
- JMP, JSR, BRA, BSR, Jcc, JScc, Bcc, BScc
- MOVEM
- RESET
- RTI, RTS

- ANDI, ORI on MR
- BRKcc, ENDDO, REP
- STOP, WAIT, DEBUG, DEBUGcc, TRAP, TRAPcc, ILLEGAL

B.4.2 General DO Restrictions

The following describes general restrictions on DO instructions:

- A DO loop should be initialized and aborted by using only the following instructions: DO, ENDDO, and BRKcc.
- The LF and the FV bits in the Status Register (SR) should not be explicitly changed by using the MOVE, BCHG, BSET, BCLR, ANDI, or ORI instructions.
- Proper DO loop operation is not guaranteed if an instruction sequence similar to one of the sequences described below is used.
  - SSH cannot be used as the source for the Loop-Count for a DO instruction
  - The following instructions should not appear within four words before a DO or DO FOREVER:
    - BCHG, BCLR, BSET, MOVE on/to SSH,SSL
    - BCHG, BCLR, BSET, MOVE on/to SP, SC
  - The following instructions should not appear immediately before a DO or DO FOREVER:
    - MOVE from SSH
    - BTST on SSH
    - BCHG, BCLR, BSET, MOVE to/on {LA, LC, SP, SC, SSH, SSL}
    - JSR, JScc, JSSET, JSCLR to LA whenever LF is set
    - BSR, BScc, to LA whenever LF is set
  - The following instructions should not appear in a DO or DO FOREVER loop:
    - {JMP, Jcc, JSR, JScc, JSET, JCLR, JSSET, JSCLR, BRA, Bcc, BSR, BScc} to LA

B.4.2.1 ENDDO Restrictions

The instructions in the following list should not appear within four words before an ENDDO instruction:

- BCHG, BCLR, BSET, MOVE on/to SSH,SSL
- BCHG, BCLR, BSET, MOVE on/to SP, SC

The instructions in the following list should not appear immediately before an ENDDO instruction:

- ANDI, ORI on MR
- MOVE from SSH
- BTST on SSH
- BCHG, BCLR, BSET, MOVE on/to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}

B.4.2.2 BRKcc Restrictions

The instructions in the following list should not appear immediately before a BRKcc instruction:

- Every arithmetic instruction
- IFcc, Tcc
- BCHG, BCLR, BSET, MOVE on/to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}

B.4.2.3 RTI and RTS Restrictions

The instructions in the following list should not appear within four words before an RTI or RTS instruction:

- BCHG, BCLR, BSET, MOVE on/to SSH,SSL
- BCHG, BCLR, BSET, MOVE on/to SP, SC

The instructions in the following list should not appear immediately before an RTI instruction:

- MOVE, BCHG, BCLR, BSET on {SSH, SSL, SP, SC}
- MOVE, BTST from/on SSH
- ANDI, ORI on {MR, CCR}
- ENDDO

The instructions in the following list should not appear immediately before an RTS instruction:

- MOVE, BCHG, BCLR, BSET on {SSH, SSL, SP, SC}
- MOVE, BTST from/on SSH
- ENDDO

B.4.3 SR Manipulation Restrictions

Changing values of bits in the Status Register (SR) should not be done by explicitly using one of the MOVE, BCHG, BSET, BCLR instruction, but only by using the ANDI or ORI instructions.

B.4.4 SP/SC and SSH/SSL Manipulation Restrictions

The instructions in List A should not be executed within four instructions before executing any of the instructions in List B.

List A

- MOVE to (SP, SC)
- BCHG, BSET, BCLR on (SP, SC)

List B

- MOVE to/from {SSH,SSL}
- BTST, BCHG, BSET, BCLR on {SSH,SSL}
- JSET, JCLR, JSSET, JSCLR on {SSH,SSL}

B.4.5 Fast Interrupt Routines

The following instructions cannot be used in a fast interrupt routine:

- DO, DO FOREVER, REP
- ENDDO, BRKcc

- RTI, RTS
- STOP, WAIT
- TRAP, TRAPcc
- ANDI, ORI on {MR, CCR}
- MOVE from SSH
- BTST on SSH
- MOVE to {LA, LC, SP, SC, SSH, SSL}
- BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL}

B.4.6 REP Restrictions

The REP instruction can repeat any single-word instruction except the REP instruction itself and any instruction that changes program flow. The following instructions are not allowed to follow a REP instruction (cannot be repeated):

- REP, DO, DO FOREVER
- ENDDO, BRKcc
- JMP, Jcc, JCLR, JSET
- JSR, JScc, JSCLR, JSSET
- BRA, Bcc
- BSR, BScc
- RTS, RTI
- TRAP, TRAPcc
- WAIT, STOP

B.4.7 Stack Extension Restrictions

The following instructions, related to the operation of the on-chip hardware stack extension, may not be used whenever the stack extension is enabled:

- MOVE to EP
- BCHG, BSET, BCLR on EP

- MOVE to SC with a value greater than 15

The following instructions, related to the operation of the on-chip hardware stack extension, may not be placed in the stack error vector locations whenever the stack extension is enabled:

- JSR, JScc, JSCLR, JSSET
- BSR, BScc

B.5 PERIPHERAL PIPELINE RESTRICTIONS

The DSP56600 core is based on a highly optimized pipeline engine. Despite the relatively deep pipeline (seven stages), the latency effects normally associated with long pipelines have been kept to a minimum because most of these effects are transparent to the user. Design techniques, such as forwarding and interlocking, alleviate the need for the user to have a thorough knowledge of the machine's pipeline in order to avoid data dependencies. This knowledge becomes relevant only when further optimization of the code is pursued. The cases where transparency does not exist (e.g., the Pointer Restrictions) are detected by the assembler that generates an appropriate warning message.

There is, however, an aspect of the machine's pipeline that is exposed to the user and this is the area of peripheral activity. This section describes the cases in which the user must take precautions in order to achieve the desired functionality.

B.5.1 Polling a Peripheral Device for Write

When writing data to a peripheral device, there is a two-cycle pipeline delay until any status bits affected by this operation are updated. For example, the user operates a peripheral port using the polling technique. The user looks for the Data Empty flag to be set. After this status bit is set, the user writes new data to the Transmit Data register. If the user attempts to read the status bit within the next two cycles, due to the pipeline delays associated with the peripheral operations, the flag is mistakenly read as set. Therefore, the user assumes that the Transmit Data register is empty and writes a new data word that in fact overwrites the previously written data. In order to achieve the correct functionality, the user must wait at least two cycles before attempting to read the Status Register following a write to the Transmit Data register.

Example B-3 shows the correct sequence for transmit operations.

Example B-3 Providing a Wait for Proper Data Writes

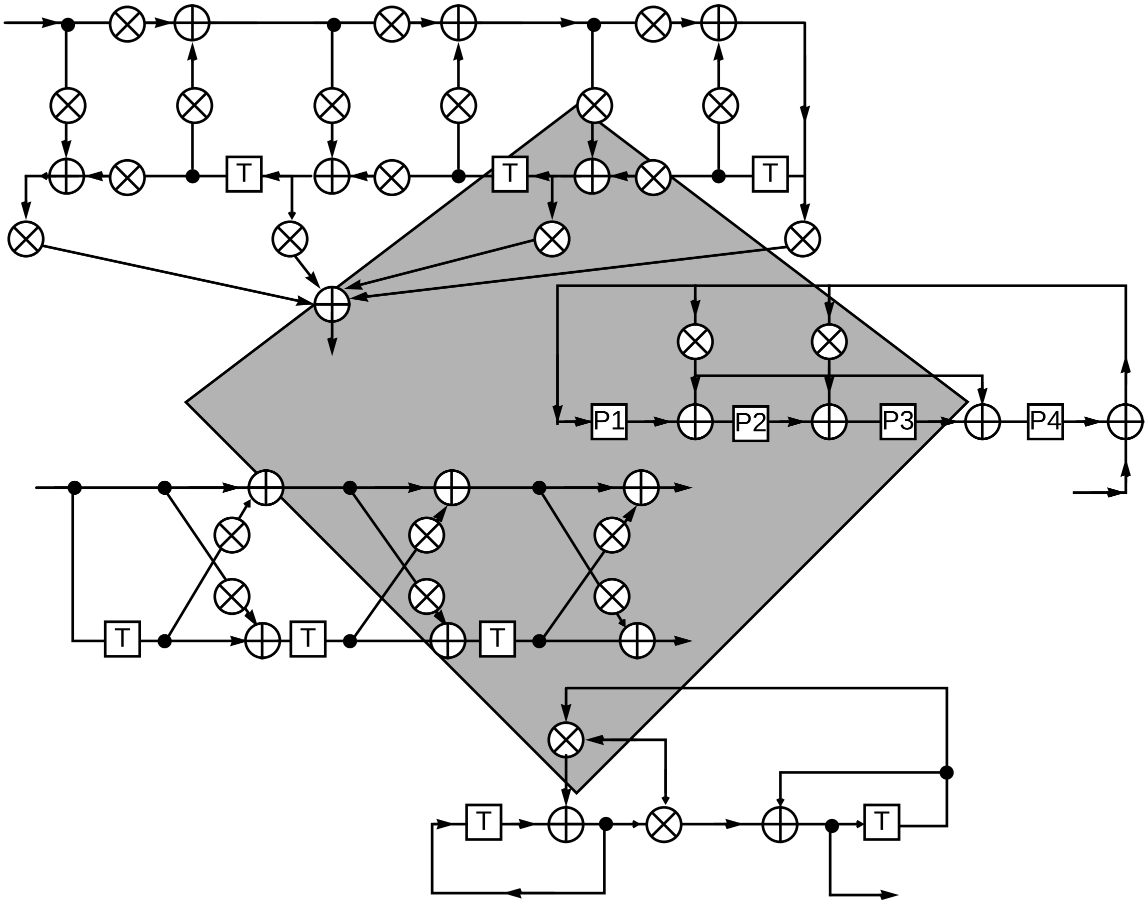
```
send
       movep  x:(r0)+,x:STX         ; send new data
       nop                          ; pipeline delay
       nop                          ; pipeline delay
poll
       jclr   #TDE,x:SCSR,poll      ; wait for data empty
       jmp    send                  ; go to send data
```

B.5.2 Writing to a Read-Only Register

Writing to a read-only register is an operation that normally has no effect, but if a read operation from the same register is attempted within the following two cycles, the value of the read data is the value of the data that was written, instead of the unchanged data of the read-only register. In order to ensure that the correct data is read after the write operation, the user must wait at least two cycles before performing the read.

APPENDIX C

BENCHMARK PROGRAMS



C.1 BENCHMARK OVERVIEW

The following benchmarks illustrate the source code syntax and programming techniques for the DSP56600 core. **Table C-1** lists the DSP benchmark programs provided in this appendix.

Table C-1 List of Benchmark Programs

Benchmark	Page	Number of Words	Clock Cycles	Sample Rate or Execution Time for 60 MHz Clock Cycle
Real Multiply	C-5	3	4	67 ns
Parsing a Hoffman Code Data Stream	C-50	7	2N + 8	33.3N + 133 ns
Real Update	C-6	4	5	83 ns
N Real Updates	C-7	9	2N + 8	33.3N + 133.6 ns
Real Correlation or Convolution (FIR Filter)	C-8	6	N + 14	60/(N + 14) MHz
Real * Complex Correlation or Convolution (FIR Filter)	C-10	9	2N + 10	30/(N + 5) MHz
Complex Multiply	C-11	6	7	117 ns
N Complex Multiplies	C-12	9	5N + 9	66.7N + 150.3 ns
Complex Update	C-13	7	8	133 ns
N Complex Updates	C-14	9	4N + 9	66.7N + 150.3 ns
Complex Correlation or Convolution (FIR Filter)	C-17	16	4N + 13	30/(2N + 5.5) MHz
Nth Order Power Series (Real)	C-19	10	2N + 11	33.3N + 183.7ns
2nd Order Real Biquad IIR Filter	C-20	7	9	150.3 ns
N Cascaded Real Biquad IIR Filter	C-21	10	5N + 10	12/(N + 2) MHz
N Radix-2 FFT Butterflies (DIT, In-Place Algorithm)	C-22	12	8N + 9	133.6N + 150.3 ns
True (Exact) LMS Adaptive Filter	C-24	15	3N + 16	60/(3N + 17) MHz

Table C-1 List of Benchmark Programs (Continued)

Benchmark	Page	Number of Words	Clock Cycles	Sample Rate or Execution Time for 60 MHz Clock Cycle
Delayed LMS Adaptive Filter	C-27	13	3N + 12	60/(3N + 12) MHz
FIR Lattice Filter	C-29	10	3N + 10	60/(3N + 10) MHz
All Pole IIR Lattice Filter	C-31	12	4N + 8	30/(2N + 4) MHz
General Lattice Filter	C-33	14	5N + 19	60/(5N + 19) MHz
Normalized Lattice Filter	C-35	15	5N + 19	60/(5N + 19) MHz
[1 ¥ 3][3 ¥ 3] Matrix Multiplication	C-37	13	14	233.8 ns
N Point 3 ¥ 3 2-D FIR Convolution	C-38	19	$11N^2$ + 8N + 7	60/($11N^2$ + 8N + 7) MHz

C.2 SET OF BENCHMARKS

The following benchmarks illustrate the source code syntax and programming techniques for the DSP56600 core. The assembly language source is organized into six columns, as shown in **Table C-2**.

Table C-2 Example of Assembly Language Source

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
`FIR`	`MAC`	`X0,Y0,A`	`X:(R0)+,X0`	`Y:(R4)+,Y0`	`;Do each tap`	1	1

The Label column is used for program entry points and end of loop indication. The Opcode column indicates the Data ALU, Address ALU, or Program Controller operation to be performed. The Opcode column must always be included in the source code. The Operands column specifies the operands to be used by the opcode. The X Bus Data specifies an optional data transfer over the X Bus and the addressing mode to be used. The Y Bus Data specifies an optional data transfer over the Y Bus and the addressing mode to be used. The Comment column is used for documentation purposes and does not affect the assembled code. The P column provides the number of Program words used by the operation, and should not be included in the source code. The T column provides the number of clock cycles used by the operation, and should not be included in the source code.

C.2.1 Real Multiply

Equation C-1:

$$c = a \times b$$

Example C-1 Real Multiply

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move		x:(r0),x0	y:(r4),y0	;	1	1
	mpyr	x0,y0,a			;	1	1
	move		a,x:(r1)		;	1	2 i'lock
					Totals	3	4

C.2.2 N Real Multiplies

Equation C-2:

$$c(i) = a(i) \times b(i) \qquad i = 1, 2, \ldots, N$$

Table C-3 N Real Multiplies Memory Map

Pointer	X memory	Y memory
r0	a(i)	
r4		b(i)
r1	c(i)	

Example C-2 N Real Multiplies

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0			;		
	move	#BADDR,r4			;		
	move	#CADDR,r1			;		
	move		x:(r0)+,x0	y:(r4)+,y0	;	1	1
	mpyr	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	;	1	1

Example C-2 N Real Multiplies (Continued)

	do	#N-1,end			;	2	5
	mpyr	x0,y0,a	a,x:(r1)+	y:(r4)+,y0	;	1	1
	move		x:(r0)+,x0		;	1	1
end					;		
	move		a,x:(r1)+		;	1	1
					Totals	7	2N + 8

C.2.3 Real Update

Equation C-3:

$$d = c + a \times b$$

Example C-3 Real Update

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0					
	move	#BADDR,r4					
	move	#CADDR,r1					
	move	#DADDR,r2					
	move		x:(r0),x0	y:(r4),y0	;	1	1
	move		x:(r1),a		;	1	1
	macr	x0,y0,a			;	1	1
	move		a,x:(r2)		;	1	2 i’lock
					Totals	4	5

C.2.4 N Real Updates

Equation C-4:

$$d(i) = c(i) + a(i) \times b(i) \qquad i = 1, 2, \ldots, N$$

Table C-4 N Real Updates Memory Map

Pointer	X memory	Y memory
r0	a(i)	
r4		b(i)
r1	c(i)	
r5		d(i)

Example C-4 N Real Updates

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0			;		
	move	#BADDR,r4			;		
	move	#CADDR,r1			;		
	move	#DADDR,r5			;		
	move		x:(r0)+,x0	y:(r4)+,y0	;	1	1
	move		x:(r1)+,a		;	1	1
	move		x:(r1)+,b		;	1	1
	do	#N/2,end			;	2	5
	macr	x0,y0,a	x:(r0)+,x1	y:(r4)+,y1	;	1	1
	macr	x1,y1,b	x:(r0)+,x0	y:(r4)+,y0	;	1	1
	move	x:(r1)+,a	a,y:(r5)+		;	1	1
	move	x:(r1)+,b	b,y:(r5)+		;	1	1
end							
					Totals	9	2N + 8

C.2.5 Real Correlation or Convolution (FIR Filter)

Equation C-5:

$$c(n) = \sum_{i=0}^{N-1} [a(i) \times b(n-i)]$$

Table C-5 Real Correlation or Convolution (FIR Filter) Memory Map

Pointer	X memory	Y memory
r0	a(i)	
r4		b(i)

Example C-5 Real Correlation or Convolution (FIR Filter)

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`move`	`#AADDR,r0`					
	`move`	`#BADDR,r4`			;		
	`move`	`#N – 1,m4`			;		
	`move`	`m4,m0`			;		
	`movep`	`y:input,y:(r4)`			;	1	2
	`clr`	`a`	`x:(r0)+,x0`	`y:(r4)–,y0`	;	1	1
	`rep`	`#N – 1`			;	1	5
	`mac`	`x0,y0,a`	`x:(r0)+,x0`	`y:(r4)–,y0`	;	1	1
	`macr`	`x0,y0,a`		`(r4)+`	;	1	1
	`movep`	`a,y:output`			;	1	2 i’lock
					Totals	6	N + 14

Table C-6 Real Correlation or Convolution (FIR Filter) Memory Map

Pointer	X memory	Y memory
r0	a(i)	
r1	b(i)	

Example C-6 Real Correlation or Convolution (FIR Filter)

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`move`	`#AADDR,r0`					
	`move`	`#BADDR,r1`			;		
	`move`	`#N-1,m1`			;		
	`move`	`m1,m0`			;		
	`movep`	`y:input,x:(r1)`			;	1	2
	`clr`	`a`	`x:(r0)+,x1`		;	1	1
	`do`	`#N-1,end`			;	2	5
	`move`		`x:(r1)-,x0`		;	1	1
	`mac`	`x0,x1,a`	`x:(r0)+,x1`		;	1	1
`end`					;		
	`move`		`x:(r1)-,x0`		;	1	1
	`macr`	`x0,x1,a`	`(r1)+`		;	1	1
	`movep`	`a,y:output`			;	1	2 i'lock
					Totals	9	2N + 10

C.2.6 Real * Complex Correlation or Convolution (FIR Filter)

Equation C-6:

$$cr(n) = jci(n) = \sum_{i=0}^{N-1} [(ar(i) + jai(i)) \times b(n-i)]$$

$$cr(n) = \sum_{i=0}^{N-1} ar(i) \times b(n-i) \qquad ci(n) = \sum_{i=0}^{N-1} ai(i) \times b(n-i)$$

Table C-7 Real * Complex Correlation or Convolution (FIR Filter) Memory Map

Pointer	X memory	Y memory
r0	ar(i)	ai(i)
r4	b(i)	
r1	cr(n)	ci(n)

Example C-7 Real * Complex Correlation or Convolution (FIR Filter)

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`move`	`#AADDR,r0`			;		
	`move`	`#BADDR,r4`			;		
	`move`	`#CADDR,r1`			;		
	`move`	`#N-1,m4`			;		
	`move`	`m4,m0`			;		
	`movep`	`y:input,x:(r4)`			;	1	2
	`clr`	`a`	`x:(r0),x0`		;	1	1
	`clr`	`b`	`x:(r4)-,x1`	`y:(r0)+,y0`	;	1	1
	`do`	`#N-1,end`			;	2	5
	`mac`	`x0,x1,a`	`x:(r0),x0`		;	1	1
	`mac`	`y0,x1,b`	`x:(r4)-,x1`	`y:(r0)+,y0`	;	1	1

Example C-7 Real * Complex Correlation or Convolution (FIR Filter) (Continued)

end							
	macr	x0,x1,a			;	1	1
	macr	y0,x1,b	(r4)+		;	1	1
	move		a,x:(r1)		;	1	1
	move			b,y:(r1)	;	1	1
					Totals	11	2N + 11

C.2.7 Complex Multiply

Equation C-7:

$$cr + jci = (ar + jai) \times (br + jbi)$$

$$cr = ar \times br - ai \times bi \qquad ci = ar \times bi + ai \times br$$

Table C-8 Complex Multiply Memory Map

Pointer	X memory	Y memory
r0	ar	ai
r4	br	bi
r1	cr	ci

Example C-8 Complex Multiply

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0					
	move	#BADDR,r4					
	move	#CADDR,r1					

Example C-8 Complex Multiply (Continued)

	move		x:(r0),x1	y:(r4),y0	;	1	1
	mpy	y0,x1,b	x:(r4),x0	y:(r0),y1	;	1	1
	macr	x0,y1,b			;	1	1
	mpy	x0,x1,a			;	1	1
	macr	-y0,y1,a		b,y:(r1)	;	1	1
	move		a,x:(r1)		;	1	2 i'lock
					Totals	6	7

C.2.8 N Complex Multiplies

Equation C-8:

$$cr(i) + jci(i) = (ar(i) + jai(i)) \times (br(i) + jbi(i)) \qquad i = 1, 2, \ldots, N$$

$$cr(i) = ar(i) \times br(i) - ai(i) \times bi(i)$$

$$ci(i) = ar(i) \times bi(i) + ai(i) \times br(i)$$

Table C-9 N Complex Multiplies Memory Map

Pointer	X memory	Y memory
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r5	cr(i)	ci(i)

Example C-9 N Complex Multiplies

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0			;		
	move	#BADDR,r4			;		
	move	#CADDR-1,r5			;		

Example C-9 N Complex Multiplies (Continued)

	move		x:(r0),x1	y:(r4),y0	;	1	1
	move		x:(r5),a		;	1	1
	do	#N,end			;	2	5
	mpy	y0,x1,b	x:(r4)+,x0	y:(r0)+,y1	;	1	1
	macr	x0,y1,b	a,x:(r5)+		;	1	1
	mpy	-y0,y1,a		y:(r4),y0	;	1	1
	macr	x0,x1,a	x:(r0),x1	b,y:(r5)	;	1	1
end							
	move	a,x:(r5)			;	1	2 i'lock
					Totals	9	4N + 9

C.2.9 Complex Update

Equation C-9:

$$dr + jdi = (cr + jci) + (ar + jai) \times (br + jbi)$$

$$dr = cr + ar \times br - ai \times bi \qquad di = ci + ar \times bi + ai \times br$$

Table C-10 Complex Update Memory Map

Pointer	X memory	Y memory
r0	ar	ai
r4	br	bi
r1	cr	ci
r2	dr	di

Example C-10 Complex Update

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0					

Example C-10 Complex Update (Continued)

```
move      #BADDR,r4
move      #CADDR,r1
move      #DADDR,r2
move                                  y:(r1),b     ;        1    1
move                   x:(r0),x1      y:(r4),y0    ;        1    1
mac       y0,x1,b      x:(r4),x0      y:(r0),y1    ;        1    1
macr      x0,y1,b      x:(r1),a                    ;        1    1
mac       x0,x1,a                                  ;        1    1
macr      -y0,y1,a                    b,y:(r2)     ;        1    1
move                   a,x:(r2)                    ;        1    2 i'lock
                                                   Totals   7    8
```

C.2.10 N Complex Updates

Equation C-10:

$$dr(i) + jdi(i) = (cr(i) + jci(i)) + (ar(i) + jai(i)) \times (br(i) + jbi(i))$$

$$dr(i) = cr(i) + ar(i) \times br(i) - ai(i) \times bi(i)$$

$$di(i) = ci(i) + ar(i) \times bi(i) + ai(i) \times br(i)$$

$$i = 1, 2, \ldots, N$$

Table C-11 N Complex Updates Memory Map

Pointer	X memory	Y memory
r0	ar(i) ; ai(i)	
r4		br(i) ; bi(i)
r1	cr(i) ; ci(i)	
r5		dr(i) ; di(i)

Example C-11 N Complex Updates

Lab el	Opcod e	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0			;		
	move	#BADDR,r4			;		
	move	#CADDR,r1			;		
	move	#DADDR-1,r5			;		
	move		x:(r0)+,x1	y:(r4)+,y0	;	1	1
	move		x:(r1)+,b	y:(r5),a	;	1	1
	do	#N,end	;2 5		;	2	5
	mac	y0,x1,b	x:(r0)+,x0	y:(r4)+,y1	;	1	1
	macr	-x0,y1,b	x:(r1)+,a	a,y:(r5)+	;	1	1
	mac	x0,y0,a	x:(r1)+,b	b,y:(r5)+	;	1	2 i'lock
	macr	x1,y1,a	x:(r0)+,x1	y:(r4)+,y0	;	1	1
end							
	move			a,y:(r5)+	;	1	2 i'lock
					Totals	9	5N + 9

Table C-12 N Complex Updates Memory Map

Pointer	X memory	Y memory
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r1	cr(i)	ci(i)
r5	dr(i)	di(i)

Example C-12 N Complex Updates

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0			;		
	move	#BADDR,r4			;		
	move	#CADDR,r1			;		
	move	#DADDR-1,r5			;		
	move		x:(r5),a		;	1	1
	move		x:(r0),x1	y:(r4),y0	;	1	1
	move		x:(r4)+,x0	y:(r1),b	;	1	1
	do	#N,end			;	2	5
	mac	y0,x1,b	a,x:(r5)+	y:(r0)+,y1	;	1	1
	macr	x0,y1,b	x:(r1)+,a		;	1	1
	mac	-y0,y1,a	y:(r4),y0		;	1	1
	macr	x0,x1,a	x:(r0),x1	b,y:(r5)	;	1	1
	move		x:(r4)+,x0	y:(r1),b	;	1	1
end							
	move		a,x:(r5)		;	1	1
					Totals	11	5N + 9

C.2.11 Complex Correlation or Convolution (FIR Filter)

Equation C-11:

$$cr(n) + jci(n) = \sum_{i=0}^{N-1} [(ar(i) + jai(i)) \times (br(n-i) + jbi(n-i))]$$

$$cr(n) = \sum_{i=0}^{N-1} [ar(i) \times br(n-i) - ai(i) \times bi(n-i)]$$

$$ci(n) = \sum_{i=0}^{N-1} [ar(i) \times bi(n-i) + ai(i) \times br(n-i)]$$

Table C-13 Complex Correlation or Convolution (FIR Filter) Memory Map

Pointer	X memory	Y memory
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r1	cr(i)	ci(i)

Example C-13 Complex Correlation or Convolution (FIR Filter)

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`move`	`#AADDR,r0`			;		
	`move`	`#BADDR,r4`			;		
	`move`	`#CADDR,r1`					
	`move`	`#N-1,m4`					
	`move`	`#m4,m0`					
	`movep`	`y:input,x:(r4)`				1	2
	`movep`	`y:input,y:(r4)`				1	2
	`clr`	`a`			;	1	1

Example C-13 Complex Correlation or Convolution (FIR Filter) (Continued)

`clr`	`b`	`x:(r0),x1`	`y:(r4),y0`	`;`	1	1
`do`	`#N-1,end`			`;`	2	5
`mac`	`y0,x1,b`	`x:(r4)-,x0`	`y:(r0)+,y1`	`;`	1	1
`mac`	`x0,y1,b`			`;`	1	1
`mac`	`x0,x1,a`			`;`	1	1
`mac`	`-y0,y1,a`	`x:(r0),x1`	`y:(r4),y0`	`;`	1	1
`end`						
`mac`	`y0,x1,b`	`x:(r4),x0`	`y:(r0)+,y1`	`;`	1	1
`macr`	`x0,y1,b`			`;`	1	1
`mac`	`x0,x1,a`			`;`	1	1
`macr`	`-y0,y1,a`			`;`	1	1
`move`			`b,y:(r1)`	`;`	1	1
`move`		`a,x:(r1)`		`;`	1	1
				Totals	16	4N + 13

C.2.12 Nth Order Power Series (Real)

Equation C-12:

$$c = \sum_{i=0}^{N-1} [a(i) \times b^{i}]$$

Table C-14 Nth Order Power Series (Real) Memory Map

Pointer	X memory	Y memory
r0	a(i)	
r4		b
r1	c	

Example C-14 Nth Order Power Series (Real)

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`move`	`#AADDR,r0`			;		
	`move`	`#BADDR,r4`					
	`move`	`#CADDR,r1`					
	`move`		`x:(r0)+,a`		;	1	1
	`move`			`y:(r4),x0`		1	1
	`mpyr`	`x0,x0,b`	`x:(r0)+,y0`		;	1	1
	`move`			`b,y1`	;	1	2 i'lock
	`do`	`#N-1,end`			;	2	5
	`mac`	`y0,x0,a`	`x:(r0)+,y0`		;	1	1
	`mpyr`	`x0,y1,b`	`b,x0`		;	1	1
`end`							
	`macr`	`y0,x0,a`			;	1	1
	`move`		`a,x:(r1)`		;	1	2 i'lock
					Totals	10	2N + 11

C.2.13 2nd Order Real Biquad IIR Filter

Equation C-13:

$$w(n)/2 = x(n)/2 - (a1)/2 \times w(n-1) - (a2)/2 \times w(n-2)$$

$$y(n)/2 = w(n)/2 + (b1)/2 \times w(n-1) + (b2)/2 \times w(n-2)$$

Table C-15 2nd Order Real Biquad IIR Filter Memory Map

Pointer	X memory	Y memory
r0	w(n-2), w(n-1)	
r4		a2/2, a1/2, b2/2, b1/2

Example C-15 2nd Order Real Biquad IIR Filter

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`ori`	`#$08,mr`			;		
	`move`	`#AADDR,r0`			;		
	`move`	`#BADDR,r4`			;		
	`move`	`#1,m0`					
	`move`	`#3,m4`					
	`movep`	`y:input,a`			;	1	1
	`rnd`	`a`	`x:(r0)+,x0`	`y:(r4)+,y0`	;	1	1
	`mac`	`-y0,x0,a`	`x:(r0)-,x1`	`y:(r4)+,y0`	;	1	1
	`mac`	`-y0,x1,a`	`x1,x:(r0)+`	`y:(r4)+,y0`	;	1	1
	`mac`	`y0,x0,a`	`a,x:(r0)`	`y:(r4),y0`	;	1	2 i'lock
	`macr`	`y0,x1,a`			;	1	1
	`movep`	`a,y:output`			;	1	2 i'lock
					Totals	7	9

C.2.14 N Cascaded Real Biquad IIR Filter

Equation C-14:

$$w(n)/2 = x(n)/2 - (a1)/2 \times w(n-1) - (a2)/2 \times w(n-2)$$
$$y(n)/2 = w(n)/2 + (b1)/2 \times w(n-1) + (b2)/2 \times w(n-2)$$

Table C-16 N Cascaded Real Biquad IIR Filter Memory Map

Pointer	X memory	Y memory
r0	w(n-2)1, w(n-1)1, w(n-2)2, ...	
r4		(a2/2)1, (a1/2)1, (b2/2)1, (b1/2)1, (a2/2)2, ...

Example C-16 N Cascaded Real Biquad IIR Filter

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`ori`	`#$08,mr`			;		
	`move`	`#AADDR,r0`			;		
	`move`	`#BADDR,r4`			;		
	`move`	`#(2N-1),m0`			;		
	`move`	`#(4N-1),m4`			;		
	`move`		`x:(r0)+,x0`	`y:(r4)+,y0`	;	1	1
	`movep`	`y:input,a`			;	1	1
	`do`	`#N,end`			;	2	5
	`mac`	`-y0,x0,a`	`x:(r0)-,x1`	`y:(r4)+,y0`	;	1	1
	`mac`	`-y0,x1,a`	`x1,x:(r0)+`	`y:(r4)+,y0`	;	1	1
	`mac`	`y0,x0,a`	`a,x:(r0)+`	`y:(r4)+,y0`	;	1	2 i'lock
	`mac`	`y0,x1,a`	`x:(r0)+,x0`	`y:(r4)+,y0`	;	1	1
`end`							
	`rnd`	`a`			;	1	1
	`movep`	`a,y:output`			;	1	2 i'lock
					Totals	10	5N + 10

C.2.15 N Radix-2 FFT Butterflies (DIT, In-Place Algorithm)

Equation C-15:

$$ar' = ar + cr \times br - ci \times bi \qquad br' = ar - cr \times br + ci \times bi = 2 \times ar - ar'$$

$$ai' = ai + ci \times br + cr \times bi \qquad bi' = ai - ci \times br - cr \times bi = 2 \times ai - ai'$$

Table C-17 N Radix-2 FFT Butterflies (DIT, In-Place Algorithm) Memory Map

Pointer	X memory	Y memory
r0	ar(i)	ai(i)
r1	br(i)	bi(i)
r6	cr(i)	ci(i)
r4	ar′(i)	ai′(i)
r5	br′(i)	bi′(i)

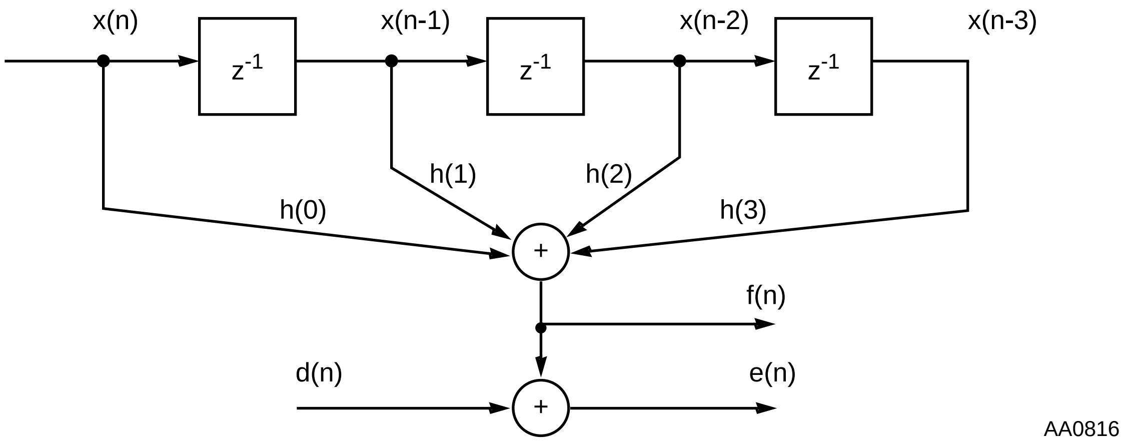
Example C-17 N Radix-2 FFT Butterflies (DIT, In-Place Algorithm)

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#AADDR,r0			;		
	move	#BADDR,r1			;		
	move	#CADDR,r6			;		
	move	#ATADDR,r4			;		
	move	#BTADDR-1,r5			;		
	move		x:(r1),x1	y:(r6),y0	;	1	1
	move		x:(r5),a	y:(r0),b		1	1
	do	#N,end			;	2	5
	mac	y0,x1,b	x:(r6)+n,x0	y:(r1)+,y1	;	1	1
	macr	x0,y1,b	a,x:(r5)+	y:(r0),a	;	1	1
	subl	b,a			;	1	1
	move		x:(r0),b	b,y:(r4)	;	1	1

Example C-17 N Radix-2 FFT Butterflies (DIT, In-Place Algorithm) (Continued)

```
        mac     x0,x1,b         x:(r0)+,a     a,y:(r5)      ;          1    1
        macr    -y0,y1,b        x:(r1),x1     y:(r6),y0     ;          1    1
        subl    b,a             b,x:(r4)+     y:(r0),b      ;          1    2 i'lock
end
        move                    a,x:(r5)+                   ;          1    2 i'lock
                                                           Totals     12    8N + 9
```

C.2.16 True (Exact) LMS Adaptive Filter



x(n)—Input sample at time n
d(n)—Desired signal at time n
f(n)—FIR filter output at time n
H(n)—Filter coefficient vector at time n. H = {h0,h1,h2,h3}
X(n)—Filter state variable vector at time N, X = {x(n),x(n – 1),x(n – 2),x(n – 3)}
u—Adaptation Gain
NTAPS—Number of coefficient taps in the filter. For this example, NTAPS = 4

Figure C-1 True (Exact) LMS Adaptive Filter

Table C-18 System Equations

True LMS Algorithm	Delayed LMS Algorithm
$e(n) = d(n) - H(n) \times (n)$	$e(n) = d(n) - H(n) \times (n)$
$H(n + 1) = H(n) + uX(n)e(n)$	$H(n + 1) = H(n) + uX(n - 1)e(n - 1)$

Table C-19 LMS Algorithms

True LMS Algorithm	Delayed LMS Algorithm
Get input sample	Get input sample
Save input sample	Save input sample

Table C-19 LMS Algorithms (Continued)

True LMS Algorithm	Delayed LMS Algorithm
Do FIR	Do FIR
Get d(n), find e(n)	Update coefficients
Update coefficients	Get d(n), find e(n)
Output f(n)	Output f(n)
Shift vector X	Shift vector X

Table C-20 True (Exact) LMS Adaptive Filter Memory Map

Pointer	X memory	Y memory
r0	x(n), x(n – 1), x(n – 2), x(n – 3)	
r4, r5		h(0), h(1), h(2), h(3)

Example C-18 True (Exact) LMS Adaptive Filter

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`move`	`#-2,n0`			`;`		
	`move`	`n0,n4`					
	`move`	`#NTAPS-1,m0`			`;`		
	`move`	`m0,m4`			`;`		
	`move`	`m0,m5`			`;`		
	`move`	`#AADDR+NTAPS-1,r0`			`;`		
	`move`	`#BADDR,r4`			`;`		
	`move`	`r4,r5`			`;`		
`_getsmp`							
	`movep`	`y:input,x0`		`input sample`		1	1
	`clr`	`a`	`x0,x:(r0)+`	`y:(r4)+,y0`	`; save`	1	1
					`;X(n), get h0`		

Example C-18 True (Exact) LMS Adaptive Filter (Continued)

```
         rep      #NTAPS-1                                        ; do fir      1    5
                                                                  ; do taps
         mac      x0,y0,b       x:(r0)+,x0      y:(r4)+,y0        ;             1    1
                                                                  ; last tap
         macr     x0,y0,b                                         ;             1    1
; Get d(n), subtract fir output, multiply by "u",
; put the result in y1.
; This section is application dependent.
         move     x:(r0)+,x0    y:(r4)+,a                                       1    1
         movep    b,y:output    ; output fir if desired                         1    1
         move                   y:(r4)+,b                                       1    1
         do       #NTAPS/2,cup                                    ;             2    5
         macr     x0,x1,a       x:(r0)+,x0      y:(r4)+,y0        ;             1    1
         macr     x0,x1,b       x:(r0)+,x0      y:(r4)+,y1        ;             1    1
         tfr      y0,a                          a,y:(r5)+                       1    1
         tfr      y0,b                          b,y:(r5)+                       1    1
cup
         move                   x:(r0)+n0,x0    y:(r4)+n4,y0      ;             1    1
; continue looping (jmp _getsmp)
                                                                  Total        15    3N + 16
```

C.2.17 Delayed LMS Adaptive Filter

- 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)

Table C-21 Delayed LMS Adaptive Filter Memory Map

Pointer	X memory	Y memory
r0	x(n), x(n – 1), x(n – 2), x(n – 3), x(n – 4)	
r5, r4		dummy, h(0), h(1), h(2), h(3)

Example C-19 Delayed LMS Adaptive Filter

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`move`	`#STATE,r0`			`; start of X`		
	`move`	`#2,n0`			`; used for pointer update`		
	`move`	`#NTAPS,m0`			`; number of filter taps`		
	`move`	`#COEF+1,r4`			`; start of H`		
	`move`	`m0,m4`			`; number of filter taps`		
	`move`	`#COEF,r5`			`; start of H-1`		
	`move`	`m4,m5`			`; number of filter taps`		
	`movep`	`y:input,a`			`; get input sample`	1	1
	`move`	`a,x:(r0)`			`; save input sample`	1	1
	`clr`	`a`	`x:(r0)+,x0`		`; x0<-x(n)`	1	1
	`move`		`x:(r0)+,x1`	`y:(r4)+,y0`		1	1
					`; x1<-x(n-1); y0<-h(0)`		
	`do`	`#TAPS/2,lms`			`;`	2	5
	`;a<-h(0)*x(n) b<-h(0) Y<-dummy`						

Example C-19 Delayed LMS Adaptive Filter (Continued)

`mac      x0,y0,a      y0,b       b,y:(r5)+`	1	2 i'lock
`;b<-H(0)=h(0)+e*x(n-1), x0<-x(n-2), y0<-h(1)`		
`macr     x1,y1,b      x:(r0)+,x0   y:(r4)+,y0   ;`	1	1
`;a<-a+h(1)*x(n-1); b<-h(1); Y(0)<-H(0)`		
`mac      x1,y0,a      y0,b       b,y:(r5)+    ;`	1	2 i'lock
`;b<-H(1)=h(1)+e*x(n-2); x1<-x(n-3); y0<-h(2)`		
`macr     x0,y1,b      x:(r0)+,x1   y:(r4)+,y0   ;`	1	1
`lms`		
`movep    a,y:output`	1	1
`move     b,y:(r5)+                ; Y<-last coef`	1	1
`move     (r0)-n0                  ; update pointer`	1	1
Totals	13	3N + 12

C.2.18 FIR Lattice Filter

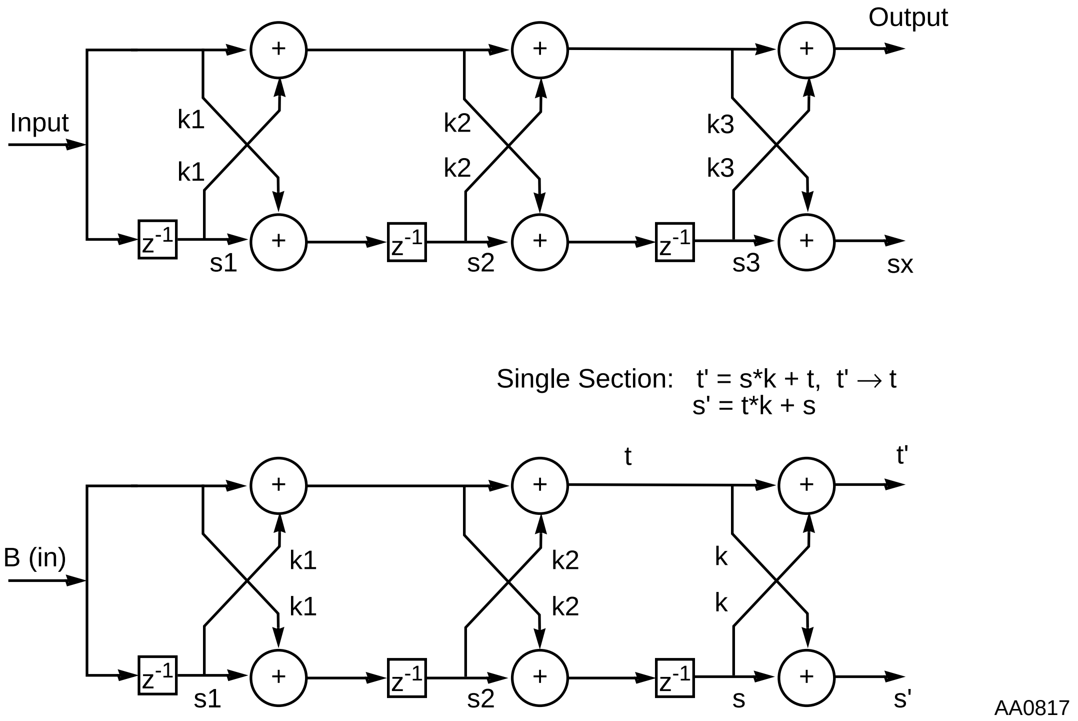


Figure C-2 FIR Lattice Filter

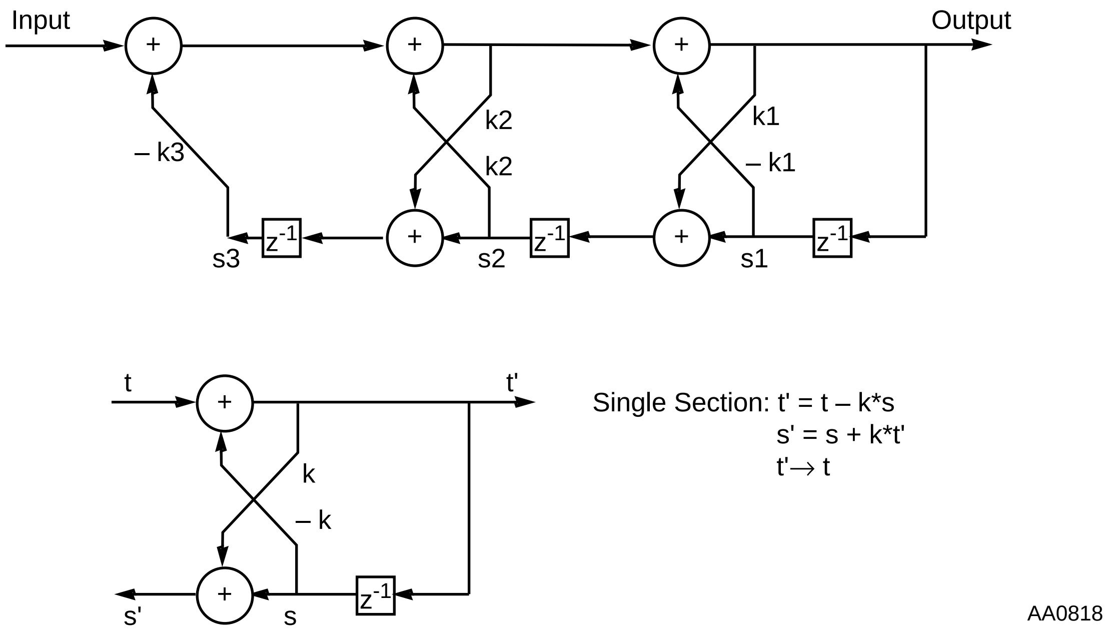
Table C-22 FIR Lattice Filter Memory Map

Pointer	X memory	Y memory
r0	s1, s2, s3, sx	
r4		k1, k2, k3

Example C-20 FIR Lattice Filter

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#S,r0			; point to s		
	move	#N,m0			; N = number of k coefficients		
	move	#K,r4			; point to k coefficients		
	move	#N-1,m4			; mod for k's		
	movep	y:datin,b			; get input	1	1
	move	b,a			; save first state	1	1
	move		x:(r0),x0	y:(r4)+,y0	; get s, get k	1	1
	do	#N,_elat			;	2	5
	macr	x0,y0,b		b,y1	; s*k+t,copy t ; for mul	1	1
	tfr	x0,a	a,x:(r0)+		; save s', ; copy next s	1	1
	macr	y1,y0,a	x:(r0),x0	y:(r4)+,y0	; t*k+s, get s, ; get k	1	1
_elat							
	move		a,x:(r0)+	y:(r4)-,y0	; adj r4, ; dummy load	1	1
	movep	b,y:datout			; output sample	1	1
					Totals	10	3N + 10

C.2.19 All Pole IIR Lattice Filter



Single Section: t' = t – k*s
s' = s + k*t'
t'→ t

Figure C-3 All Pole IIR Lattice Filter

Table C-23 All Pole IIR Lattice Filter Memory Map

Pointer	X memory	Y memory
r0	k3, k2, k1	
r4		s3, s2, s1

Example C-21 All Pole IIR Lattice Filter

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	`move`	`#k+N-1,r0`			`;point to k`		
	`move`	`#N-1,m0`			`;number of k's-1`		
	`move`	`#STATE,r4`			`;point to filter states`		
	`move`	`m0,m4`			`;mod for states`		
	`move`	`#1,n4`			`;`		

Example C-21 All Pole IIR Lattice Filter (Continued)

```
         movep     y:datin,a                    y:(r4)+,b     ;get input        1    1
         move                   x:(r0)-,x0      y:(r4)+,y0    ;get s, get k     1    1
         macr      -x0,y0,a     x:(r0)-,x0      y:(r4),y0     ;s*k+t            1    1
         do        #N-1,_endl                                 do sections       2    5
                   at
         macr      -x0,y0,a                     y:(r4)+,y1    ;                 1    1
         tfr       y1,b         a,x1            b,y:(r4)      ;                 1    2 i'lock
         macr      x1,x0,b      x:(r0)-,x0      y:(r4),y0                       1    1
_endlat
         movep     a,y:datout                                                   1    1
         move                   x:(r0)+,x0      y:(r4)+,r0    ;output sample    1    1
         move      b,y:(r4)+                                  ;save s'          1    1
;save last s', update r4
         move                   a,y:(r4)                                        1    1
                                                              Totals           12    4N + 8
```

C.2.20 General Lattice Filter

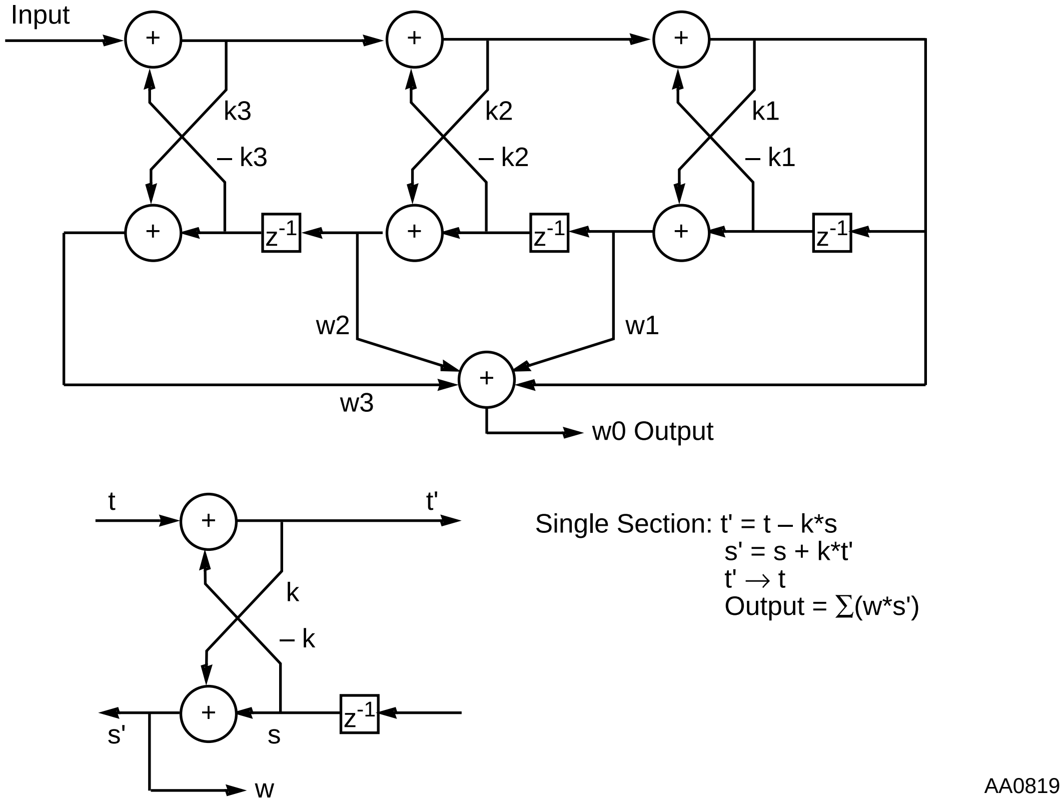


Figure C-4 General Lattice Filter

Table C-24 General Lattice Filter Memory Map

Pointer	X memory	Y memory
r0	k3, k2, k1, w3, w2, w1, w0	
r4		s4, s3, s2, s1

Example C-22 General Lattice Filter

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#K,r0		;point to coefficients			
	move	#2*N,m0		;mod 2*(# of k's)+1			
	move	#STATE,r4		;point to filter states			
	move	#-2,n4					
	move	#N,m4		;mod on filter states			
	movep	y:datin,a		;get input		1	1
	move		x:(r0)+,x0	y:(r4)-,y0		1	1
	do	#N,_endlat				2	5
	macr	-x0,y0,a			;	1	1
	tfr	y0,b	a,x1	b,y:(r4)+n4	;	1	2 i'lock
	macr	x1,x0,b	x:(r0)+,x0	y:(r4)-,y0	;	1	1
_endlat							
	move			b,y:(r4)+	;save s'	1	2 i'lock
	clr	a		a,y:(r4)+	;save last s', ; update r4	1	1
	move			y:(r4)+,y0		1	1
	rep	#N			;	1	5
	mac	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	;s*w+out, ; get s, get w	1	1
	macr	x0,y0,a			;last mac	1	1
	movep	a,y:datout			;output sample	1	2 i’lock
					Totals	14	5N + 19

C.2.21 Normalized Lattice Filter

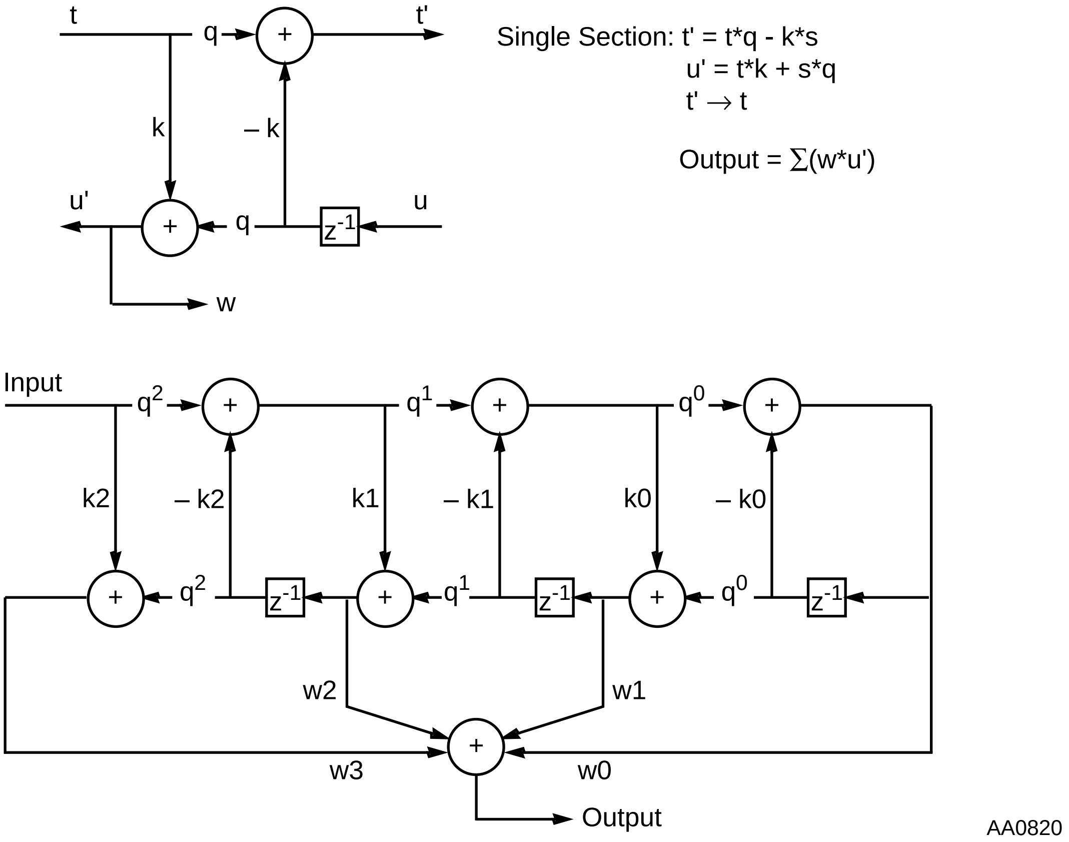


Figure C-5 Normalized Lattice Filter

Table C-25 Normalized Lattice Filter Memory Map

Pointer	X memory	Y memory
r0	q2, k2, q1, k1, q0, k0, w3, w2, w1, w0	
r4		sx, s2, s1, s0

Example C-23 Normalized Lattice Filter

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#COEF,r0			; point to ; coefficients		
	move	#3*N,m0			; mod on ; coefficients		
	move	#STATE+1,r4			; point to ; state variables		
	move	#N,m4			; mod on filter ; states		
	movep	y:datin,y0			; get input sample	1	1
	move		x:(r0)+,x1		; get q in the ; table	1	1
	do	#N,_elat				2	5
	mpy	x1,y0,a	x:(r0)+,x0	y:(r4),y1	; q * t,get k,get s	1	1
	macr	-x0,y1,a		b,y:(r4)+	; q * t - k * s, ; save new s	1	1
	mpy	x0,y0,b			; k * t	1	1
	macr	x1,y1,b	x:(r0)+,x1	a,y0	; k * t + q * s ; get next q,set t'	1	1
_elat							
	move	b,y:(r4)+			; save second ; last state	1	2 i'lock
	move	a,y:(r4)+			; save last state	1	1
	clr	a		y:(r4)+,y0	; clear a, get ; first state	1	1
	rep	#N				1	5
	mac	x1,y0,a	x:(r0)+,x1	y:(r4)+,y0	; fir taps	1	1
	macr	x1,y0,a	(r4)+		; round, ; adj pointer	1	1
	movep	a,y:datout			; output sample	1	2 i'lock
					Total	15	5N + 19

C.2.22 [1 × 3][3 × 3] Matrix Multiplication

Example C-24 [1 × 3][3 × 3] Matrix Multiplication

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
`_init`							
	`move`	`#MAT_A,r0`			`;point to A matrix`		
	`move`	`#MAT_B,r4`			`;point to B matrix`		
	`move`	`#MAT_X,r1`			`;output X matrix`		
	`move`	`#2,m0`			`mod 3`		
	`move`	`#8,m4`			`;mod 9`		
	`move`	`m0,m1`			`;mod 3`		
`_start`							
	`move`	`x:(r0)+,x0`	`y:(r4)+,y0`			1	1
	`mpy`	`x0,y0,a`	`x:(r0)+,x0`	`y:(r4)+,y0`		1	1
	`mac`	`x0,y0,a`	`x:(r0)+,x0`	`y:(r4)+,y0`		1	1
	`macr`	`x0,y0,a`	`x:(r0)+,x0`	`y:(r4)+,y0`		1	1
	`mpy`	`x0,y0,b`	`x:(r0)+,x0`	`y:(r4)+,y0`		1	1
	`move`			`a,y:(r1)+`		1	1
	`mac`	`x0,y0,b`	`x:(r0)+,x0`	`y:(r4)+,y0`		1	1
	`macr`	`x0,y0,b`	`x:(r0)+,x0`	`y:(r4)+,y0`		1	1
	`mpy`	`x0,y0,a`	`x:(r0)+,x0`	`y:(r4)+,y0`		1	1
	`move`			`b,y:(r1)+`		1	1
	`mac`	`x0,y0,a`	`x:(r0)+,x0`	`y:(r4)+,y0`		1	1
	`macr`	`x0,y0,a`				1	1
	`move`			`a,y:(r1)+`		1	2 i’lock
`_end`							
					Totals	13	14

C.2.23 N Point 3 × 3 2-D FIR Convolution

The two dimensional FIR uses a [3 × 3] coefficient mask:

c(1,1) c(1,2) c(1,3)

c(2,1) c(2,2) c(2,3)

c(3,1) c(3,2) c(3,3)

stored in Y memory in the order:

c(1,1), c(1,2), c(1,3), c(2,1), c(2,2), c(2,3), c(3,1), c(3,2), c(3,3).

The image is an array of 512 × 512 pixels. To provide boundary conditions for the FIR filtering, the image is surrounded by a set of 0s such that the image is actually stored as a 514 × 514 array.

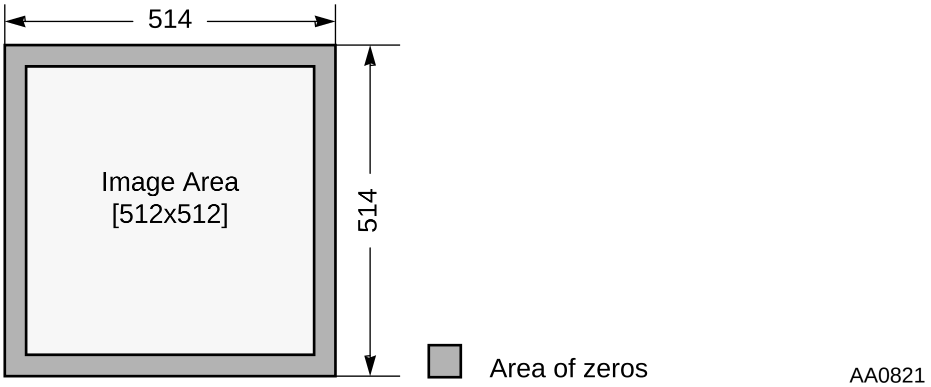


Figure C-6 FIR Filtering

The image (with boundary) is stored in row major storage. The first element of the array image(,) is image(1,1) followed by image(1,2). The last element of the first row is image(1,514) followed by the beginning of the next column image(2,1). These are stored sequentially in the array "im" in X memory:

- Image(1,1) maps to index 0, image(1,514) maps to index 513;
- Image(2,1) maps to index 514 (row major storage).

Although many other implementations are possible, this is a realistic type of image environment where the actual size of the image may not be an exact power of 2. Other possibilities include storing a 512 × 512 image but computing only a 511 × 511 result, computing a 512 × 512 result without boundary conditions but throwing away the pixels on the border, etc.

Table C-26 N Point 3 × 3 2-D FIR Convolution Memory Map

Pointer		

r0	image(n,m)
	image(n,m+1)
	image(n,m+2)	

r1	image(n+514,m)
	image(n+514,m+1)
	image(n+514,m+2)	

r2	image(n+2*514,m)
	image(n+2*514,m+2)
	image(n+2*514,m+3)	

r4	FIR coefficients	
r5	output image	

Example C-25 N Point 3 × 3 2-D FIR Convolution

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	move	#MASK,r4		;point to coefficients			
	move	#8,m4			;mod 9		
	move	#IMAGE,r0			;top boundary		
	move	#IMAGE+514,r1		;left of first pixel			
;left of first pixel 2nd row							
	move	#IMAGE+2*514,r2			;		
;adjust. for end of row							
	move	#2,n1			;		
	move	n1,n2			;		
	move	#IMAGEOUT,r5			;output image		

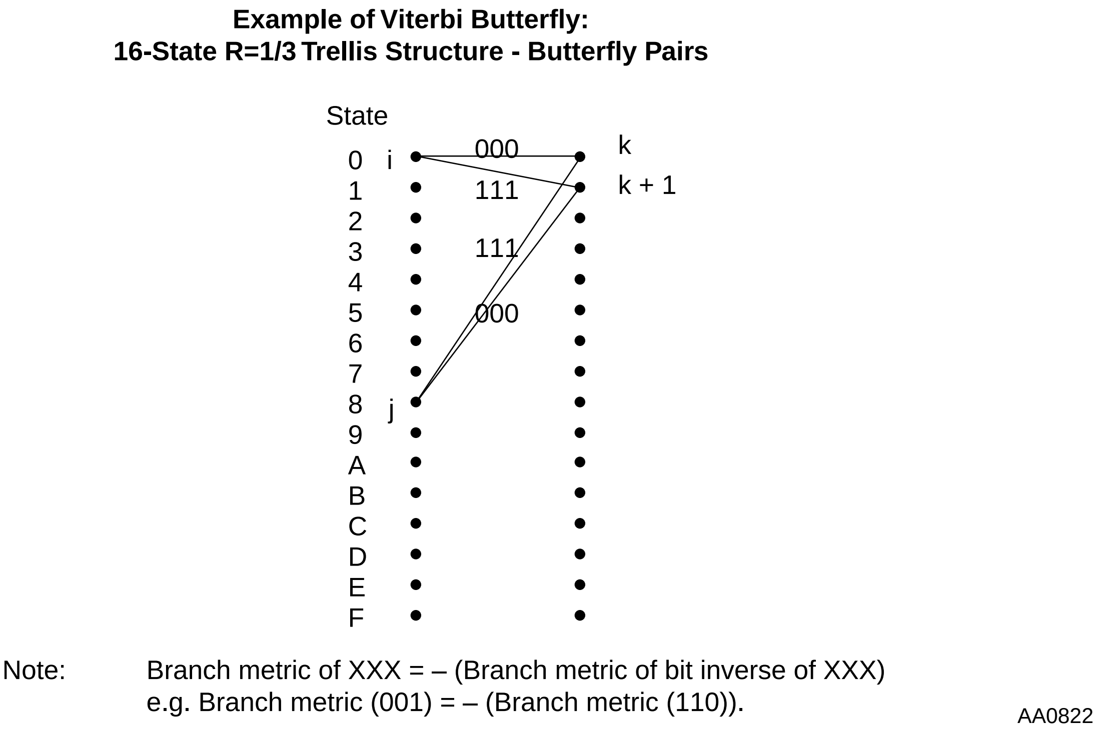
Example C-25 N Point 3 × 3 2-D FIR Convolution (Continued)

```
;first element, c(1,1)
        move                x:(r0)+,x0    y:(r4)+,y0    ;              1  1
        do        #512,row                              ;              2  5
        do        #512,col                              ;              2  5
        mpy       x0,y0,a   x:(r0)+,x0    y:(r4)+,y0    ;c(1,2)        1  1
        mac       x0,y0,a   x:(r0)-,x0    y:(r4)+,y0    ;c(1,3)        1  1
        mac       x0,y0,a   x:(r1)+,x0    y:(r4)+,y0    ;c(2,1)        1  1
        mac       x0,y0,a   x:(r1)+,x0    y:(r4)+,y0    ;c(2,2)        1  1
        mac       x0,y0,a   x:(r1)-,x0    y:(r4)+,y0    ;c(2,3)        1  1
        mac       x0,y0,a   x:(r2)+,x0    y:(r4)+,y0    ;c(3,1)        1  1
        mac       x0,y0,a   x:(r2)+,x0    y:(r4)+,y0    ;c(3,2)        1  1
        mac       x0,y0,a   x:(r2)-,x0    y:(r4)+,y0    ;c(3,3)        1  1
; preload, get c(1,1)
        macr      x0,y0,a   x:(r0)+,x0    y:(r4)+,y0    ;              1  1
;output image sample
        move                              a,y:(r5)+     ;              1  2 i'lock
col
; adjust pointers for frame boundary, adj r0,r5 w/dummy loads
        move                x:(r0)+,x0    y:(r5)+,y1    ;              1  1
; adj r1,r5 w/dummy loads
        move                x:(r1)+n1,x0  y:(r5)+,y1    ;              1  1
; adj r2 (dummy load y1), preload x0 for next pass
        move                x:(r0)+,x0                  ;              1  1
        move                              y:(r2)+n2,y1                 1  1
row
```

Total	P = 19
	T = $11N^2 + 8N + 7$

C.2.24 Viterbi Add-Compare-Select (ACS)

This routine implements Viterbi algorithm kernel. The algorithm is parametric and fits any valid values of Trellis states number and any branch metrics.



Example of Viterbi Butterfly:
16-State R=1/3 Trellis Structure - Butterfly Pairs

Note: Branch metric of XXX = – (Branch metric of bit inverse of XXX)
e.g. Branch metric (001) = – (Branch metric (110)).

Figure C-7 Viterbi Butterfly

Given Branch Metric value (BrM), ACS should perform:

- Fetch path metric of state(i) – S_i.
- Fetch path metric of state(j) – S_j.
- Add BrM to S_i.
- Subtract BrM from S_j.
- Compare and select the greater of the two:
  Next $S_k = \text{Max}(S_i + \text{BrM}, S - \text{BrM})$.
- Store the result in next-state path-metric memory location.
- Update the state’s Trellis history with the selection bit.
- Perform the similar task for:
  Next $S_{k+1} = \text{Max}(S_i - \text{BrM}, S_j + \text{BrM})$.

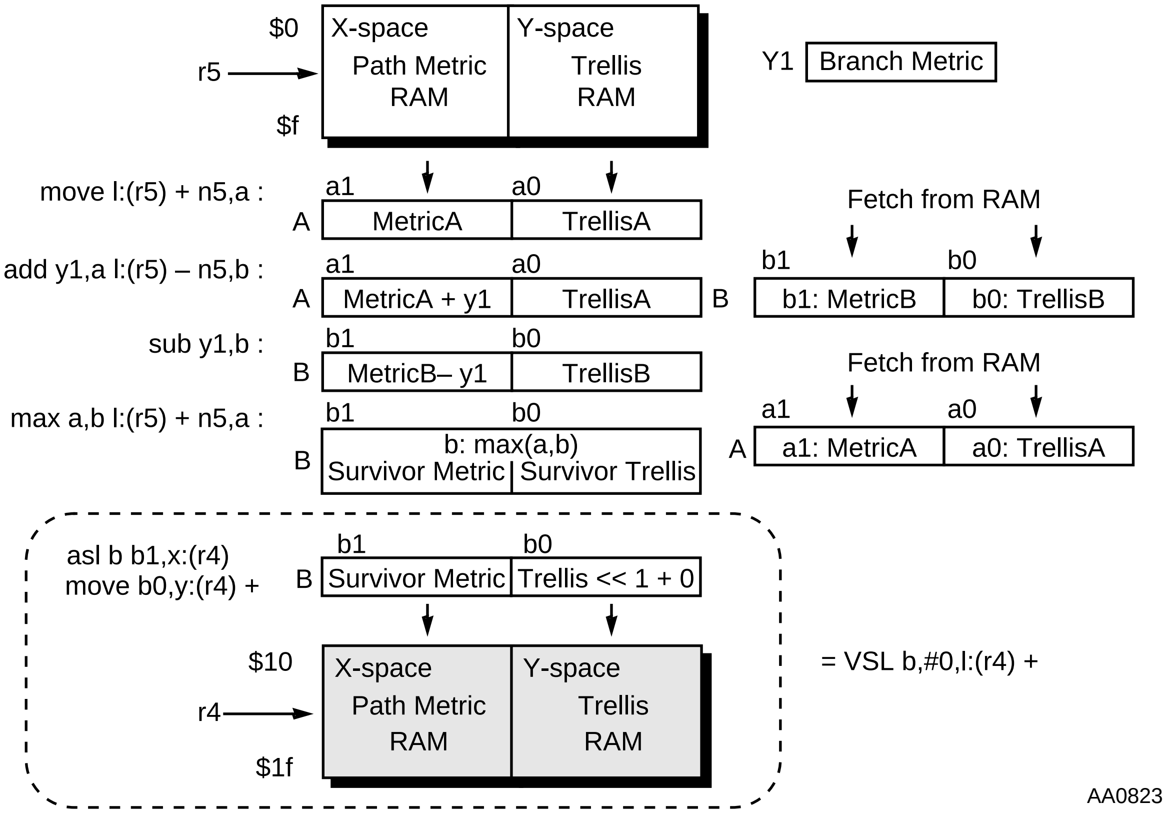


Figure C-8 ACS Butterfly—First Half

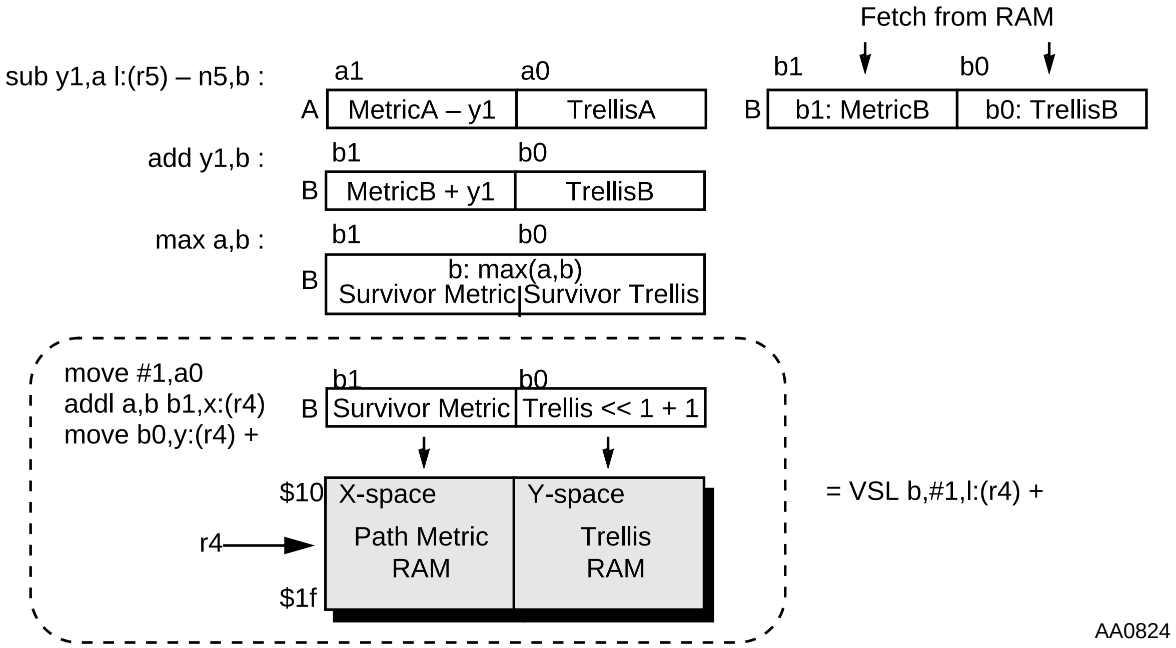


Figure C-9 ACS Butterfly—Second Half

Example C-26 Viterbi Add-Compare-Select (ACS)

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
	; **r0** - R/W pointer to branch-metric table. ; **r4** - write pointer - path metric Present State tables. ; **r5** - read pointer - path metric tables Previous State. ; **n5** - bit-count value, used for decode loop. ; **y1** - given Brm for ACS loop ; **x0** - tmp register						
ComputeBrMtrc:					;		
	; for the general case, assuming that the branch metrics are ; calculated and prepared as table at y:(r0) location						
	move			y:(r0)+,y1		1	
; load first branch metric.							
	move		l:(r5)+n5,a			1	
; a0 <- trellis, a1 <- PathMetr							
	; main ACS loop						

Example C-26 Viterbi Add-Compare-Select (ACS) (Continued)

```
        do      #NoOfAcsButt,NextStage                       ;                      2
        add     y1,a           l:(r5)-n5,b                                           1
; a=a+y1, b0 <- trellis, b1 <- PthMt
        sub     y1,b                                         ; b=b-y1               1
        max     a,b            l:(r5)+n5,a                                           1
; b=max(a,b) | refetch a
        vsl                    b,#0,l:(r4)+                                          1
; store survivor path metric & trellis
        sub     y1,a           l:(r5)-n5,b                                           1
; a=a-y1 | refetch b
        add     y1,b           x:(r5)+,x0    y:(r0)+,y1                              1
; b=b+y1 | increment r5 | load next brm.
        max     a,b            l:(r5)+n5,a                                           1
; b=max(a,b) | fetch next a
        vsl                    b,#1,l:(r4)+                                          1
; store survivor path metric & trellis
NextStage:
        move                   #branch_tbl,r0                                        2
; set r0 to start of br. metric table.
                                                             Total                 14
```

C.2.25 Parsing a Data Stream

This routine implements parsing of a data stream for MPEG audio. The data stream, composed by concatenated words of variable length, is allocated in consecutive memory words. The word lengths reside in another memory buffer. The routine extracts words from the data stream according to their length. Two consecutive words are read from the stream buffer and are concatenated in the accumulator. Using bit offset and the specified length, a field of variable length can be extracted. The decision whether to load a new memory word into the accumulator from the stream is determined when bit offset overflow to the LSP of the accumulator.

The following describes the pointers and registers used by the routine:

- r0—pointer to the buffer in X memory containing the variable length stream
- r5—pointer to buffer in Y memory where the length of each field is stored
- r4—pointer to a location that stores the “bits offset”, number of bits left to be consumed, 48 initially
- r3—pointer to a location storing the constant 24
- r1—used as temporary storage (no need to initialize)
- y1—stores the length of the field to be extracted
- x0—stores 24

Table C-27 Parsing Data Stream Memory Map

Pointer	X memory	Y mem
r0	stream buffer	
r5		length buffer
r4		“bits offset”
r3	‘24‘	

Example C-27 Parsing Data Stream

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
init_		; this is the initialization code					
	move	#stream_buffer,r0					
	move	#length_buffer,r5					
	move	#bits_offset,r4					
	move	#boundary,r3					
	move	#>48,b					
	move	#>24,x0					
	move		x0,x:(r3)	b,y:(r4)			
Get_bits							
		; bring length of next field and ‘24‘					
	move		x:(r3),x0	y:(r5)+,y1		1	1
		; bring word for parsing and “bits offset”					
	move		x:(r0)+,a	y:(r4),b		1	1
		; bring next word for parsing, point back to first word					
	move		x:(r0)-,a0			1	1
		;calculate new “bits offset”, r1 points to current word					
	sub	y1,b	r0,r1			1	1
		; save “bits offset” in x1					
	move	b,x1				1	2
		; merge width and offset					
	merge	y1,b				1	1
		; extract the field according to b, place it in a					
	extract	b1,a,a				1	1
		; restore “bits offset”, r0 points to next word					

Example C-27 Parsing Data Stream (Continued)

tfr	x1,b	(r0)+	1	1
	; compare “bits offset” to 24, extracted word to a1			
cmp	x0,b	a0,a	1	1
	; if “bits offset” is less or equal 24 another word is needed ; update “bits offset” and point to next word			
add	x0,b	ifle	1	1
tgt		r1,r0	1	1
	;save “bits field” in memory			
move		b1,y:(r4)	1	1
		Totals	12	13

C.2.26 Creating a Data Stream

This routine creates a data stream for MPEG audio. Words of variable length are concatenated and stored in consecutive memory words. The words for generating the stream are allocated in a memory buffer, and are aligned to the right. The word lengths reside in another memory buffer. The word and its length are loaded for insertion. A word is read from the stream buffer into the accumulator. Using a bit offset and the specified length, a field of variable length is inserted into the accumulator. The accumulator is stored back containing the new concatenated field. The decision whether to read a new word from the stream is determined when bit offset overflow to the LSP of the accumulator.

The following describes the pointers and registers used by the routine:

- r0—pointer to a buffer in X memory, containing the variable length codes—the code is right-aligned at each location
- r2—pointer to a buffer in X memory containing the stream generated
- r4—pointer to a buffer in Y memory where the actual length of each field is stored
- r3—pointer to a location that stores the “bits offset,” the number of bits left to be consumed, 48 initially
- r5—pointer to a location storing the constant 24
- r1—used as temporary storage (no need to initialize)
- x0—stores the current word to be inserted
- y1—stores the length of the code brought in x0
- y0—stores 24

Table C-28 Creating Data Stream Memory Map

Pointer	X memory	Y memory
r0	data buffer	
r2	stream buffer	
r4		length buffer
r3		“bits offset”
r5		24

Example C-28 Creating Data Stream

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
init_		;this is the initialization code					
	move	#data_buffer,r0					
	move	#stream_buffer,r2					
	move	#length_buffer,r4					
	move	#bits_offset,r3					
	move	#boundary,r5					
	move	#>48,b					
	move	#>24,y0					
	move		b,x:(r3)	y0,y:(r5)			
Put_bits							
		;bring code and its length					
	move		x:(r0)+,x0	y:(r4)+,y1		1	1
		;bring “bits offset” and ‘24‘					
	move		x:(r3),b	y:(r5),y0		1	1
		; calculate new “bits offset”, bring current word ; from stream buffer					
	sub	y1,b	x:(r2),a			1	1
		; save “bits offset” in x1					
	move	b,x1				1	2
		; merge width and offset					
	merge	y1,b				1	1
		; insert the field according to b, place it in a					
	insert	b1,x0,a				1	1
		; restore “bits offset”, r1 points to current word					
	tfr	x1,b	r2,r1			1	1

Example C-28 Creating Data Stream (Continued)

```
                     ;compare "bits offset" to 24, send new word to stream buffer

          cmp        y0,b         a1,x:(r2)+                                        1   1

                     ; send a0 to next location in stream buffer in case of
                     ; crossing boundary

          move                    a0,x:(r2)                                         1   2

                     ; if "bits offset" is less or equal 24 then update
                     ; "bits offset" and point to the next word in stream buffer

          add        y0,b         ifle                                              1   1

          tgt                     r1,r2                                             1   1

                     ;save "bits offset" in memory

          move                    b1,y:(r4)                                         1   1

                                                                        Totals     12  14
```

C.2.27 Parsing a Hoffman Code Data Stream

This routine implements the parsing of a Hoffman code data stream. The routine extracts a bit field from the stream. Two consecutive words are brought to the accumulator from the stream buffer. An address word is extracted using a bit offset and a field length. The field length is determined by the number of bits needed by the address of the two Hoffman code lookup tables. A word is loaded from the first lookup table. If the hit bit in the word is not set, then a field of variable length is extracted. The length of the extracted field is specified in the length field in the word. The bit offset is updated according to the length of the extracted word.

If the hit bit in the word is set, a new address word is read from the stream. A word is brought from the second lookup table. The bit field is extracted according to the same guidelines.

The flow chart in **Figure C-10** demonstrates the parsing process:

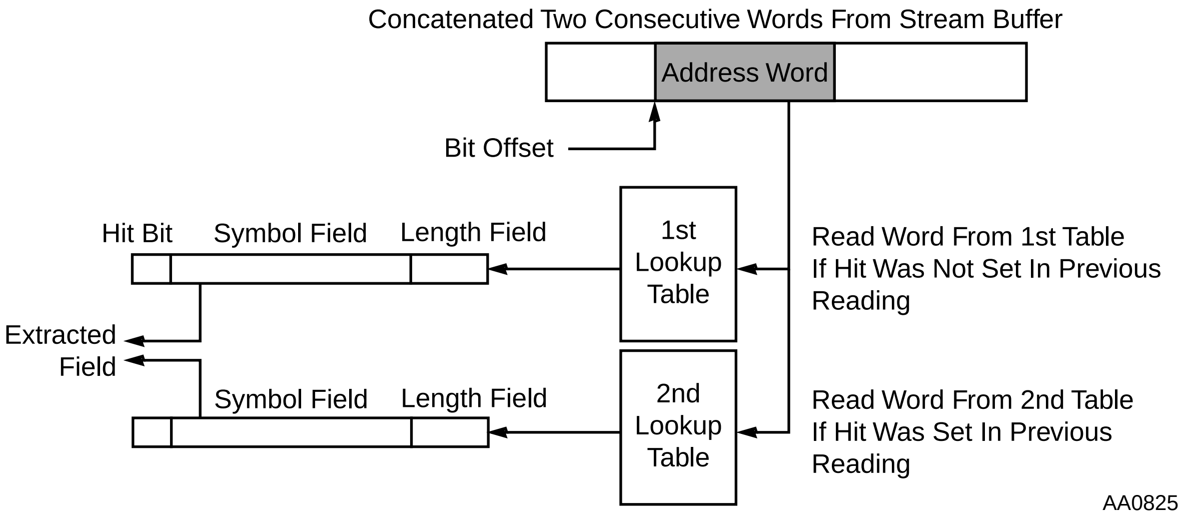

Concatenated Two Consecutive Words From Stream Buffer
Address Word
Bit Offset
Hit Bit
Symbol Field
Length Field
1st Lookup Table
Read Word From 1st Table If Hit Was Not Set In Previous Reading
Extracted Field
Symbol Field
Length Field
2nd Lookup Table
Read Word From 2nd Table If Hit Was Set In Previous Reading
AA0825

Figure C-10 Parsing Process

The following describes the pointers and registers used by the routine:

- r0—pointer to the buffer in X memory containing the stream
- r1—used as temporary storage (no need to initialize)
- r3—pointer to buffer in Y memory where the extracted fields are stored
- r5—pointer to a location that stores the “bits offset”, number of bits left to be consumed, 48 initially
- r2—pointer to the right table
- r6—pointer to the first lookup table
- r7—pointer to the second lookup table
- r4—pointer to constants

Table C-29 Parsing Hoffman Code Data Stream Memory Map

pointer	X memory	Y memory
r0	stream buffer	
r3	extracted data buffer	
r5		“bits offset”

Table C-29 Parsing Hoffman Code Data Stream Memory Map

pointer	X memory	Y memory
r4		#no.1 address bus length
		#no.2 mask word for length field
		#no.3 merged width and offset
		‘24‘
r6	first lookup table	
r7	second lookup table	

Example C-29 Parsing Hoffman Code Data Stream

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment	P	T
init_		;this is the initialization code					
	move	#stream_buffer,r0					
	move	#data_buffer,r3					
	move	#bits_offset,r5					
	move	#constants,r4					
	move	#first_table,r2					
	move	#first_table,r6					
	move	#second_table,r7					
		;move constants to memory					
	move	#>48,b					
	move	b,y:(r5)					
	move	#>3,n4					
	move	#n0_1,y1					
	move	y1,y:(r4)+					
	move	#n0_2,y1					

Example C-29 Parsing Hoffman Code Data Stream (Continued)

```
          move       y1,y:(r4)+
          move       #n0_3,y1
          move       y1,y:(r4)+
          move       #>24,y1
          move       y1,y:(r4)-n4
Get_bits
                     ;bring word from stream, and "bits-offset"
          move                   x:(r0)+,a         y:(r5)+,b                    1   1
                     ;bring next word from stream, and address length
          move                                     y:(r4)+,y0                   1   1
          move                   x:(r0)-,a0                                     1   1
                     ;calculate new "bits offset", and save old one in x1
          sub        y0,b        b,x1                                           1   1
                     ;merge width and offset
          merge      y0,b                                                       1   1
                     ;extract the field according to b, place it in a
          extract    b1,a,a                                                     1   1
                     ;move address to n2
          move                   a0,n2                                          1   1
                     ;bring mask for length field in lookup table words
          move                   y:(r4)+,y1                                     1   1
                     ;bring the merged offset and length for extactionf
          move                   y:(r4)+,x0                                     1   1
                     ;r1 points to current address for extracted field
          move                   r3,r1                                          1   1
```

Example C-29 Parsing Hoffman Code Data Stream (Continued)

```
          ;bring word from lookup table
move                  x:(r2+n2),a                               1    1
          ;extract the field according to x0, place it in b
extract   x0,a,b                                                1    1
          ;test if hit bit is set, r2 points s first lookup table
tst       a           r6,r2                                     1    1
          ; if hit bit is set, r2 points second lookup table, a holds
          address length
tmi       y0,a        r7,r2                                     1    1
          ;restore “bit offset” , send extracted field to memory
tfr       x1,b        b0,x:(r3)+                                1    1
          ; if hit bit is set, restore r3
tmi                   r1,r3                                     1    1
          ;mask length field , save pointer to current stream word
and       y1,a        r0,r1                                     1    1
          ;calculate new “bits offset”, y1 holds ’24’
sub       a,b         y:(r4)-n4,y1                              1    1
          ;compare “bits offset” to 24, update steam pointer
cmp       y1,b        (r0)+                                     1    1
          ; if “bits offset” is less or equal 24 another word is needed -
          ; update “bits offset” and point to next word
add       y1,b        ifle                                      1    1
tgt                   r1,r0                                     1    1
          ;save “bits field” in memory
move                  b1,y:(r5)                                 1    1
                                                    Totals     22   22
```

INDEX