DSP56KFAMUM/AD

DSP56000
Digital Signal Processor Family Manual

MOTOROLA

DSP56K FAMILY INTRODUCTION 1

DSP56K CENTRAL ARCHITECTURE OVERVIEW 2

DATA ARITHMETIC LOGIC UNIT 3

ADDRESS GENERATION UNIT 4

PROGRAM CONTROL UNIT 5

INSTRUCTION SET INTRODUCTION 6

PROCESSING STATES 7

PORT A 8

PLL CLOCK OSCILLATOR 9

ON-CHIP EMULATION (OnCE) 10

ADDITIONAL SUPPORT 11

INSTRUCTION SET DETAILS A

BENCHMARK PROGRAMS B

INDEX I

1 DSP56K FAMILY INTRODUCTION

2 DSP56K CENTRAL ARCHITECTURE OVERVIEW

3 DATA ARITHMETIC LOGIC UNIT

4 ADDRESS GENERATION UNIT

5 PROGRAM CONTROL UNIT

6 INSTRUCTION SET INTRODUCTION

7 PROCESSING STATES

8 PORT A

9 PLL CLOCK OSCILLATOR

10 ON-CHIP EMULATION (OnCE)

11 ADDITIONAL SUPPORT

A INSTRUCTION SET DETAILS

B BENCHMARK PROGRAMS

I INDEX

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

OnCE™ is a trade mark of Motorola, Inc.

© MOTOROLA INC., 1992

TABLE OF CONTENTS

Paragraph Number	Title	Page Number

SECTION 1
DSP56K FAMILY INTRODUCTION

1.1	INTRODUCTION	1-3
1.2	ORIGIN OF DIGITAL SIGNAL PROCESSING	1-3
1.3	SUMMARY OF DSP56K FAMILY FEATURES	1-9
1.4	MANUAL ORGANIZATION	1-11

SECTION 2
DSP56K CENTRAL ARCHITECTURE OVERVIEW

2.1	DSP56K CENTRAL ARCHITECTURE OVERVIEW	2-3
2.2	DATA BUSES	2-3
2.3	ADDRESS BUSES	2-4
2.4	DATA ALU	2-5
2.5	ADDRESS GENERATION UNIT	2-5
2.6	PROGRAM CONTROL UNIT	2-5
2.7	MEMORY EXPANSION PORT (PORT A)	2-6
2.8	ON-CHIP EMULATOR (OnCE)	2-6
2.9	PHASE-LOCKED LOOP (PLL) BASED CLOCKING	2-6

SECTION 3
DATA ARITHMETIC LOGIC UNIT

3.1	DATA ARITHMETIC LOGIC UNIT	3-3
3.2	OVERVIEW AND DATA ALU ARCHITECTURE	3-3
3.3	DATA REPRESENTATION AND ROUNDING	3-10
3.4	DOUBLE PRECISION MULTIPLY MODE	3-16

Table of Contents (Continued)

Paragraph Number	Title	Page Number
3.5	DATA ALU PROGRAMMING MODEL	3-19
3.6	DATA ALU SUMMARY	3-19

SECTION 4
ADDRESS GENERATION UNIT

4.1	ADDRESS GENERATION UNIT AND ADDRESSING MODES	4-3
4.2	AGU ARCHITECTURE	4-3
4.3	PROGRAMMING MODEL	4-6
4.4	ADDRESSING	4-8

SECTION 5
PROGRAM CONTROL UNIT

5.1	PROGRAM CONTROL UNIT	5-3
5.2	OVERVIEW	5-3
5.3	PROGRAM CONTROL UNIT (PCU) ARCHITECTURE	5-5
5.4	PROGRAMMING MODEL	5-8

SECTION 6
INSTRUCTION SET INTRODUCTION

6.1	INSTRUCTION SET INTRODUCTION	6-3
6.2	SYNTAX	6-3
6.3	INSTRUCTION FORMATS	6-3
6.4	INSTRUCTION GROUPS	6-20

SECTION 7
PROCESSING STATES

7.1	PROCESSING STATES	7-3
7.2	NORMAL PROCESSING STATE	7-3
7.3	EXCEPTION PROCESSING STATE (INTERRUPT PROCESSING)	7-10

Table of Contents (Continued)

Paragraph Number	Title	Page Number
7.4	RESET PROCESSING STATE	7-33
7.5	WAIT PROCESSING STATE	7-36
7.6	STOP PROCESSING STATE	7-37

SECTION 8
PORT A

8.1	PORT A OVERVIEW	8-3
8.2	PORT A INTERFACE	8-3

SECTION 9
PLL CLOCK OSCILLATOR

9.1	PLL CLOCK OSCILLATOR INTRODUCTION	9-3
9.2	PLL COMPONENTS	9-3
9.3	PLL PINS	9-9
9.4	PLL OPERATION CONSIDERATIONS	9-11

SECTION 10
ON-CHIP EMULATION (OnCE)

10.1	ON-CHIP EMULATION INTRODUCTION	10-3
10.2	ON-CHIP EMULATION (OnCE) PINS	10-3
10.3	OnCE CONTROLLER AND SERIAL INTERFACE	10-6
10.4	OnCE MEMORY BREAKPOINT LOGIC	10-11
10.5	OnCE TRACE LOGIC	10-13
10.6	METHODS OF ENTERING THE DEBUG MODE	10-14
10.7	PIPELINE INFORMATION AND GLOBAL DATA BUS REGISTER	10-16
10.8	PROGRAM ADDRESS BUS HISTORY BUFFER	10-18
10.9	SERIAL PROTOCOL DESCRIPTION	10-19
10.10	DSP56K TARGET SITE DEBUG SYSTEM REQUIREMENTS	10-19
10.11	USING THE OnCE	10-20

Paragraph Number	Title	Page Number
	SECTION 11 ADDITIONAL SUPPORT	
11.1	USER SUPPORT	11-3
11.2	MOTOROLA DSP PRODUCT SUPPORT	11-4
11.3	DSP56KADSx APPLICATION DEVELOPMENT SYSTEM	11-6
11.4	Dr. BuB ELECTRONIC BULLETIN BOARD	11-7
11.5	MOTOROLA DSP NEWS	11-16
11.6	MOTOROLA FIELD APPLICATION ENGINEERS	11-16
11.7	DESIGN HOTLINE– 1-800-521-6274	11-16
11.8	DSP HELP LINE – (512) 891-3230	11-16
11.9	MARKETING INFORMATION– (512) 891-2030	11-16
11.10	THIRD-PARTY SUPPORT INFORMATION – (512) 891-3098	11-16
11.11	UNIVERSITY SUPPORT – (512) 891-3098	11-16
11.12	TRAINING COURSES – (602) 897-3665 or (800) 521-6274	11-17
11.13	REFERENCE BOOKS AND MANUALS	11-17
	APPENDIX A INSTRUCTION SET DETAILS	
A.1	APPENDIX A INTRODUCTION	A-3
A.2	INSTRUCTION GUIDE	A-3
A.3	NOTATION	A-4
A.4	ADDRESSING MODES	A-10
A.5	CONDITION CODE COMPUTATION	A-15
A.6	PARALLEL MOVE DESCRIPTIONS	A-20
A.7	INSTRUCTION DESCRIPTIONS	A-21
A.8	INSTRUCTION TIMING	A-294
A.9	INSTRUCTION SEQUENCE RESTRICTIONS	A-305
A.10	INSTRUCTION ENCODING	A-311
	APPENDIX B BENCHMARK PROGRAMS	
B.1	INTRODUCTION	B-3
B.2	BENCHMARK PROGRAMS	B-3

LIST of FIGURES

Figure Number	Title	Page Number
1-1	Analog Signal Processing	1-4
1-2	Digital Signal Processing	1-5
1-3	DSP Hardware Origins	1-9
2-1	DSP56K Block Diagram	2-4
3-1	DSP56K Block Diagram	3-4
3-2	Data ALU	3-5
3-3	MAC Unit	3-7
3-4	DATA ALU Accumulator Registers	3-8
3-5	Saturation Arithmetic	3-10
3-6	Integer-to-Fractional Data Conversion	3-11
3-7	Bit Weighting and Alignment of Operands	3-12
3-8	Integer/Fractional Number Comparison	3-13
3-9	Integer/Fractional Multiplication Comparison	3-14
3-10	Convergent Rounding	3-15
3-11	Full Double Precision Multiply Algorithm	3-16
3-12	Single X Double Multiply Algorithm	3-17
3-13	Single X Double Multiply-Accumulate Algorithm	3-18
3-14	DSP56K Programming Model	3-19
4-1	DSP56K Block Diagram	4-4
4-2	AGU Block Diagram	4-5
4-3	AGU Programming Model	4-7
4-4	Address Register Indirect — No Update	4-10
4-5	Address Register Indirect — Postincrement	4-11
4-6	Address Register Indirect — Postdecrement	4-12
4-7	Address Register Indirect — Postincrement by Offset Nn	4-13
4-8	Address Register Indirect — Postdecrement by Offset Nn	4-14
4-9	Address Register Indirect — Indexed by Offset Nn	4-15
4-10	Address Register Indirect — Predecrement	4-16
4-11	Circular Buffer	4-19
4-12	Linear Addressing with a Modulo Modifier	4-20
4-13	Modulo Modifier Example	4-21

List of Figures (Continued)

Figure Number	Title	Page Number
4-14	Bit-Reverse Address Calculation Example	4-24
4-15	Address Modifier Summary	4-26
5-1	Program Address Generator	5-3
5-2	DSP56K Block Diagram	5-4
5-3	Three-Stage Pipeline	5-7
5-4	Program Control Unit Programming Model	5-8
5-5	Status Register Format	5-9
5-6	OMR Format	5-14
5-7	Stack Pointer Register Format	5-15
5-8	SP Register Values	5-15
5-9	DSP56K Central Processing Module Programming Model	5-18
6-1	DSP56K Central Processing Module Programming Model	6-4
6-2	General Format of an Instruction Operation Word	6-5
6-3	Operand Sizes	6-6
6-4	Reading and Writing the ALU Extension Registers	6-7
6-5	Reading and Writing the Address ALU Registers	6-7
6-6	Reading and Writing Control Registers	6-8
6-7	Special Addressing – Immediate Data	6-15
6-8	Special Addressing – Absolute Addressing	6-16
6-9	Special Addressing – Immediate Short Data	6-17
6-10	Special Addressing – Short Jump Address	6-18
6-11	Special Addressing – Absolute Short Address	6-19
6-12	Special Addressing – I/O Short Address	6-20
6-13	Hardware DO Loop	6-25
6-14	Nested DO Loops	6-26
6-15	Classifications of Parallel Data Moves	6-27
6-16	Parallel Move Examples	6-28
7-1	Fast and Long Interrupt Examples	7-13
7-2	Interrupt Priority Register (Addr X:$FFFF)	7-14
7-3	Interrupting an SWI	7-18
7-4	Illegal Instruction Interrupt Serviced by a Fast Interrupt	7-19
7-5	Illegal Instruction Interrupt Serviced by a Long Interrupt	7-20
7-6	Repeated Illegal Instruction	7-21
7-7	Trace Exception	7-23
7-8	Fast Interrupt Service Routine	7-27
7-9	Two Consecutive Fast Interrupts	7-28
7-10	Long Interrupt Service Routine	7-30
7-11	JSR First Instruction of a Fast Interrupt	7-31
7-12	JSR Second Instruction of a Fast Interrupt	7-32

List of Figures (Continued)

Figure Number	Title	Page Number
7-13	Interrupting an REP Instruction	7-34
7-14	Interrupting Sequential REP Instructions	7-35
7-15	Wait Instruction Timing	7-36
7-16	Simultaneous Wait Instruction and Interrupt	7-37
7-17	STOP Instruction Sequence	7-38
7-18	STOP Instruction Sequence Followed by IRQA	7-39
7-19	STOP Instruction Sequence Recovering with RESET	7-42
8-1	Port A Signals	8-4
9-1	PLL Block Diagram	9-3
9-2	DSP56K Block Diagram	9-4
9-3	PLL Control Register (PCTL)	9-6
10-1	OnCE Block Diagram	10-3
10-2	DSP56K Block Diagram	10-4
10-3	OnCE Controller and Serial Interface	10-6
10-4	OnCE Command Register	10-7
10-5	OnCE Status and Control Register (OSCR)	10-9
10-6	OnCE Memory Breakpoint Logic	10-12
10-7	OnCE Trace Logic Block Diagram	10-14
10-8	OnCE Pipeline Information and GDB Registers	10-16
10-9	OnCE PAB FIFO	10-17
B-1	20-Tap FIR Filter Example	B-5
B-2	Radix 2, In-Place, Decimation-In-Time FFT	B-7
B-3	8-Pole 4-Multiply Cascaded Canonic IIR Filter	B-9
B-4	LMS FIR Adaptive Filter	B-11
B-5	Real Input FFT Based on Glenn Bergland Algorithm	B-12

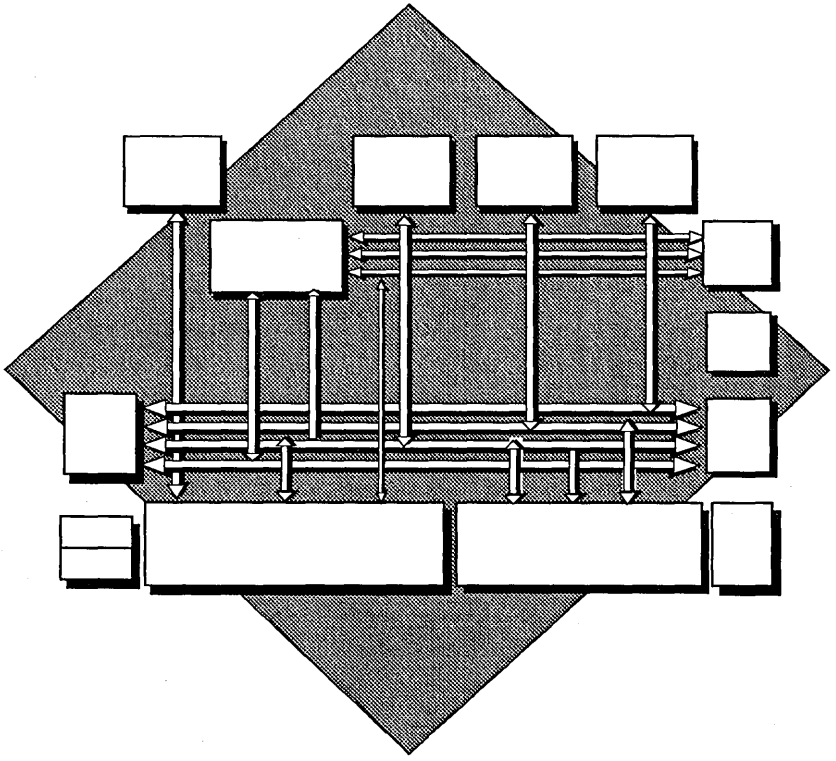
LIST of TABLES

Table Number	Title	Page Number
1-1	Benchmark Summary in Instruction Cycles	1-6
3-1	Limited Data Values	3-11
4-1	Address Register Indirect Summary	4-8
4-2	Address Modifier Summary	4-17
4-3	Bit-Reverse Addressing Sequence Example	4-23
6-1	Addressing Modes Summary	6-21
7-1	Instruction Pipelining	7-3
7-2	Status Register Interrupt Mask Bits	7-14
7-3	Interrupt Priority Level Bits	7-15
7-4	External Interrupt	7-15
7-5	Central Processor Interrupt Priorities Within an IPL	7-15
7-6	Interrupt Sources	7-16
9-1	Multiplication Factor Bits MF0-MF11	9-6
9-2	Division Factor Bits DF0-DF3	9-7
9-3	PSTP and PEN Relationship	9-8
9-4	Clock Output Disable Bits COD0-COD1	9-9
10-1	Chip Status Information	10-5
10-2	OnCE Register Addressing	10-7
10-3	Memory Breakpoint Control Table	10-10
A-1	Instruction Description Notation	A-5
A-2	DSP56K Addressing Modes	A-11
A-3	DSP56K Addressing Mode Encoding	A-12
A-4	Addressing Mode Modifier Summary	A-14
A-5	Condition Code Computations for Instructions (No Parallel Move)	A-19
A-6	Instruction Timing Summary	A-301
A-7	Parallel Data Move Timing	A-302
A-8	MOVEC Timing Summary	A-302
A-9	MOVEP Timing Summary	A-302

List of Tables (Continued)

Table Number	Title	Page Number
A-10	Bit Manipulation Timing Summary	A-303
A-11	Jump Instruction Timing Summary	A-303
A-12	RTI/RTS Timing Summary	A-304
A-13	Addressing Mode Timing Summary	A-304
A-14	Memory Access Timing Summary	A-305
A-15	Single-Bit Register Encodings	A-312
A-16	Single-Bit Special Register Encodings	A-312
A-17	Double-Bit Register Encodings	A-312
A-18	Triple-Bit Register Encodings	A-313
A-19 (a)	Four-Bit Register Encodings for 12 Registers in Data ALU	A-313
A-19 (b)	Four-Bit Register Encodings for 16 Condition Codes	A-313
A-20	Five-Bit Register Encodings for 28 Registers in Data ALU and Address ALU	A-314
A-21	Six-Bit Register Encodings for 43 Registers On-Chip	A-314
A-22	Write Control Encoding	A-314
A-23	Memory Space Bit Encoding	A-314
A-24	Program Controller Register Encoding	A-315
A-25	Condition Code and Address Encoding	A-315
A-26	Effective Addressing Mode Encoding	A-316
A-27	Operation Code K0-2 Decode	A-331
A-28	Operation Code QQQ Decode	A-332
A-29	Nonmultiply Instruction Encoding	A-333
A-30	Special Case #1	A-334
A-31	Special Case #2	A-334
B-1	27-MHz Benchmark Results for the DSP56001R27	B-4

SECTION 1
DSP56K FAMILY INTRODUCTION



1

SECTION CONTENTS

SECTION 1.1 INTRODUCTION 3
SECTION 1.2 ORIGIN OF DIGITAL SIGNAL PROCESSING 3
SECTION 1.2 SUMMARY OF DSP56K FAMILY FEATURES 9
SECTION 1.3 MANUAL ORGANIZATION 11

1.1 INTRODUCTION

The DSP56K family is Motorola's series of HCMOS, low power, 24-bit general purpose Digital Signal Processors (DSPs*). The family architecture features a modular chip layout with a standard central processing module which supports memory expansion options, different on-chip peripherals, and different package sizes. This modular implementation makes it relatively straighforward for Motorola to design and generate customer specified derivatives.

A standard interface between the central processing module's silicon and the on-chip expansion areas supports all memory and peripheral configurations. The architecture, on-chip peripherals, and the low power consumption of the DSP56K family minimize the complexity, cost, and time required to add the power of Digital Signal Processing (DSP) to any design.

This chapter introduces general DSP theory and discusses the features and benefits of the Motorola DSP56K family of processors. It also presents a brief description each of the sections in the manual.

1.2 ORIGIN OF DIGITAL SIGNAL PROCESSING

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

- Filtering of signals
- Convolution, which is the mixing of two signals
- Correlation, which is a comparison of two signals
- Rectification, Amplification, and/or Transformation of 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 a description of analog signal processing. The circuit in the illustration filters a signal from a sensor using an operational amplifier, and controls an actuator with the result. Since the ideal filter is impossible to design, the engineer must design the filter 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.

---

* This manual uses the acronym DSP for Digital Signal Processing or Digital Signal Processor, depending on the context

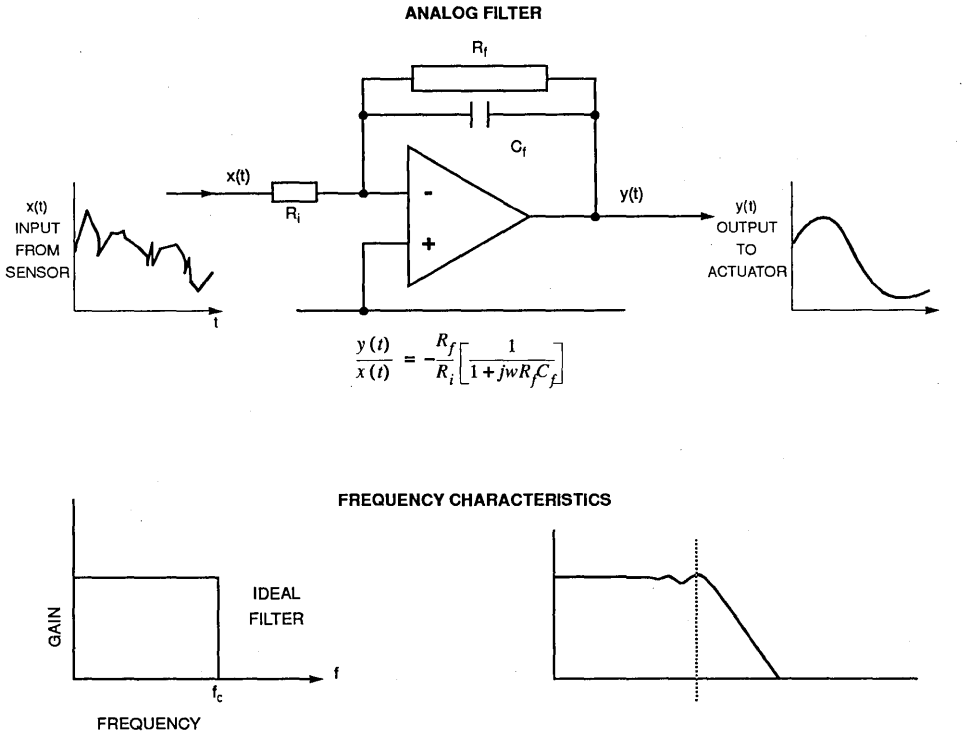


Figure 1-1 Analog Signal Processing

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

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

The filter implemented by the DSP is strictly a matter of software. The DSP can directly 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 the DSP include the following:

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

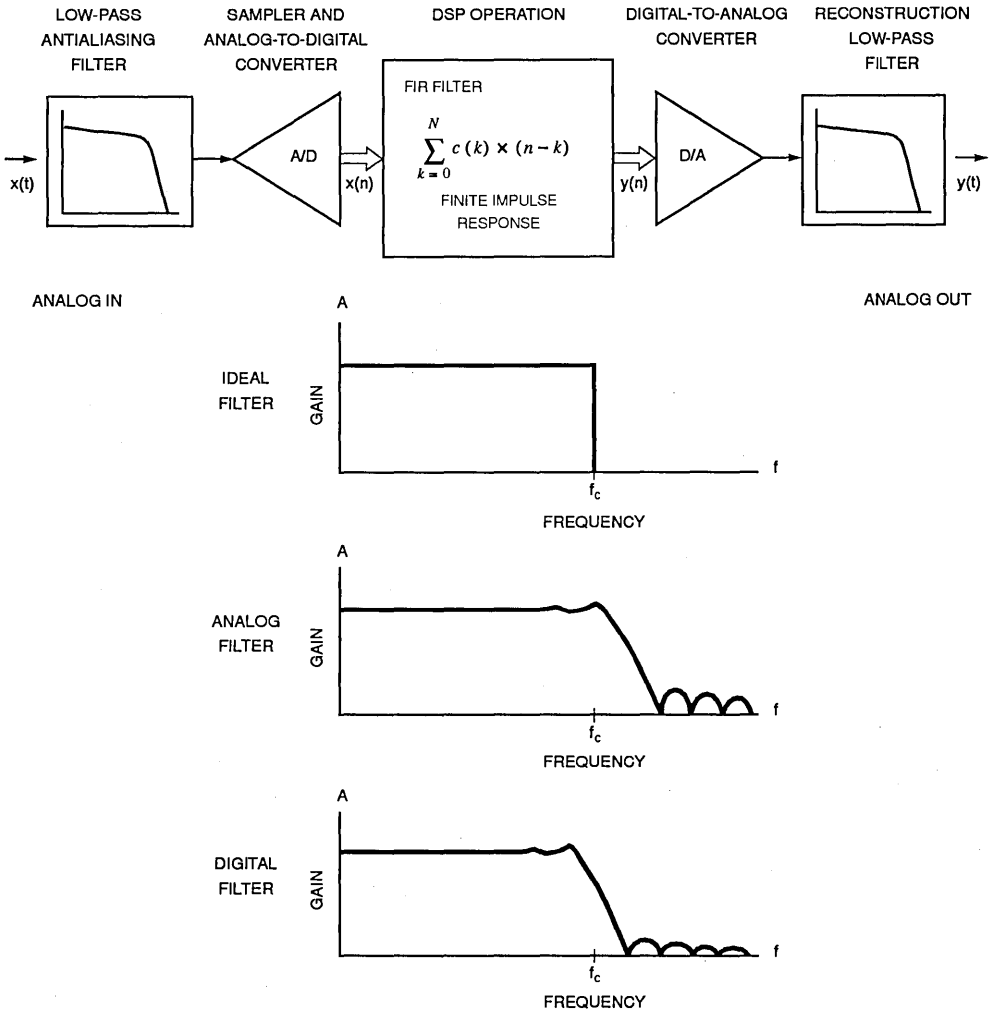


Figure 1-2 Digital Signal Processing

The DSP56K family is not designed for a particular application but is designed to execute commonly used DSP benchmarks in a minimum time for a single-multiplier architecture. For example, a cascaded, 2nd-order, four-coefficient infinite impulse response (IIR) biquad section has four multiplies for each section. For that algorithm, the theoretical minimum number of operations for a single-multiplier architecture is four per section. Table 1-1 shows a list of benchmarks with the number of instruction cycles a DSP56K chip uses compared to the number of multiplies the algorithm requires.

Table 1-1 Benchmark Summary in Instruction Cycles

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

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

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

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

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

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

Modulation
- Amplitude
- Frequency
- Phase

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

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

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

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

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

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

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

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

Instrumentation
Spectral Analysis
Waveform Generation
Transient Analysis
Data Acquisition

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

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

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

Vibration Analysis
Electric Motors
Jet Engines
Turbines

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

Digital Video
Digital Television
High-Resolution Monitors

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

Seismic Processing
Oil Exploration
Geological Exploration

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

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

MAC is the basic operation used in DSP. The DSP56K family of processors has a dual Harvard architecture optimized for MAC operations. Figure 1-3 shows how the DSP56K

architecture matches the shape of the MAC operation. The two operands, C() and X(), are directed to a multiply operation, and the result is summed. This process is built into the chip by using two separate memories (X and Y) to feed a single-cycle MAC. The entire process must occur under program control to direct the correct operands to the multiplier and save the accumulator as needed. Since the two memories and the MAC are independent, the DSP can perform two moves, a multiply and an accumulate, in a single operation. As a result, many of the benchmarks shown in Table 1-1 can be executed at or near the theoretical maximum speed for a single-multiplier architecture.

1.3 SUMMARY OF DSP56K FAMILY FEATURES

The high throughput of the DSP56K family of processors makes them well suited for communication, high-speed control, numeric processing and computer and audio applications. The main features that contribute to this high throughput include:

- **Speed** — Speeds high enough to easily address applications traditionally served by low-end floating point DSPs.

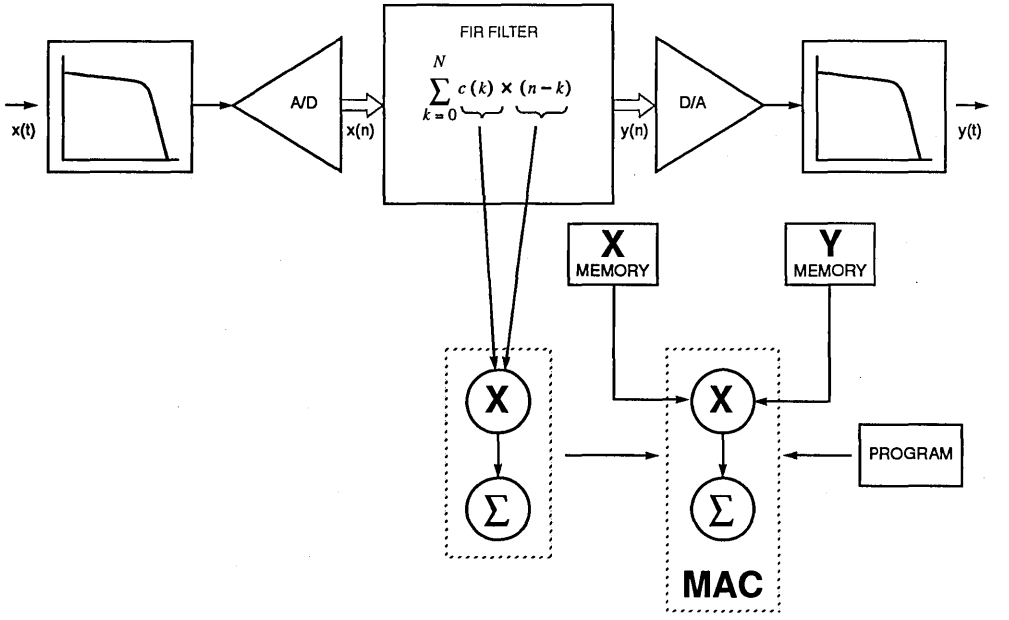


Figure 1-3 DSP Hardware Origins

- **Precision** — The data paths are 24 bits wide, providing 144 dB of dynamic range; intermediate results held in the 56-bit accumulators can range over 336 dB.
- **Parallelism** — Each on-chip execution unit (AGU, program control unit, data ALU), memory, and peripheral operates independently and in parallel with the other units through a sophisticated bus system. The data ALU, AGU, and program control unit operate in parallel so that an instruction prefetch, a 24-bit x 24-bit multiplication, a 56-bit addition, two data moves, and two address-pointer updates using one of three types of arithmetic (linear, modulo, or reverse-carry) can be executed in a single instruction cycle. This parallelism allows a four-coefficient IIR filter section to be executed in only four cycles, the theoretical minimum for single-multiplier architecture. At the same time, the two serial controllers can send and receive full-duplex data, and the host port can send/receive simplex data.
- **Flexibility** — While many other DSPs need external communications circuitry to interface with peripheral circuits (such as A/D converters, D/A converters, or host processors), the DSP56K family provides on-chip serial and parallel interfaces which can support various configurations of memory and peripheral modules
- **Sophisticated Debugging**— Motorola's on-chip emulation technology (OnCE) allows simple, inexpensive, and speed independent access to the internal registers for debugging. OnCE tells application programmers exactly what the status is within the registers, memory locations, buses, and even the last five instructions that were executed.
- **Phase-locked Loop (PLL) Based Clocking** — PLL allows the chip to use almost any available external system clock for full-speed operation while also supplying an output clock synchronized to a synthesized internal core clock. It improves the synchronous timing of the processors' external memory port, eliminating the timing skew common on other processors.
- **Invisible Pipeline** — The three-stage instruction pipeline is essentially invisible to the programmer, allowing straightforward program development in either assembly language or a high-level language such as a full Kernighan and Ritchie C.
- **Instruction Set** — The instruction mnemonics are MCU-like, making the transition from programming microprocessors to programming the chip as easy as possible. The orthogonal syntax controls the parallel execution units. The hardware DO loop instruction and the repeat (REP) instruction make writing straight-line code obsolete.

- **DSP56001 Compatibility** — All members of the DSP56K family are downward compatible with the DSP56001, and also have added flexibility, speed, and functionality.
- **Low Power** — As a CMOS part, the DSP56000/DSP56001 is inherently very low power and the STOP and WAIT instructions further reduce power requirements.

1.4 MANUAL ORGANIZATION

This manual describes the central processing module of the DSP56K family in detail and provides practical information to help the user:

- Understand the operation of the DSP56K family
- Design parallel communication links
- Design serial communication links
- Code DSP algorithms
- Code communication routines
- Code data manipulation algorithms
- Locate additional support

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

Section 2 – DSP56K Central Architecture Overview

The DSP56K central architecture consists of the data arithmetic logic unit (ALU), address generation unit (AGU), program control unit, On-Chip Emulation (OnCE) circuitry, the phase locked loop (PLL) based clock oscillator, and an external memory port (Port A). This section describes each subsystem and the buses interconnecting the major components in the DSP56K central processing module.

Section 3 – Data Arithmetic Logic Unit

This section describes in detail the data ALU and its programming model.

Section 4 – Address Generation Unit

This section specifically describes the AGU, its programming model, address indirect modes, and address modifiers.

Section 5 – Program Control Unit

This section describes in detail the program control unit and its programming model.

Section 6 – Instruction Set Introduction

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

Section 7 – Processing States

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

Section 8 – Port A

This section describes the external memory port, its control register, and control signals.

Section 9 – PLL Clock Oscillator

This section describes the PLL and its functions

Section 10 – On-Chip Emulator (OnCE)

This section describes the OnCE circuitry and its functions.

Section 11 – Additional Support

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

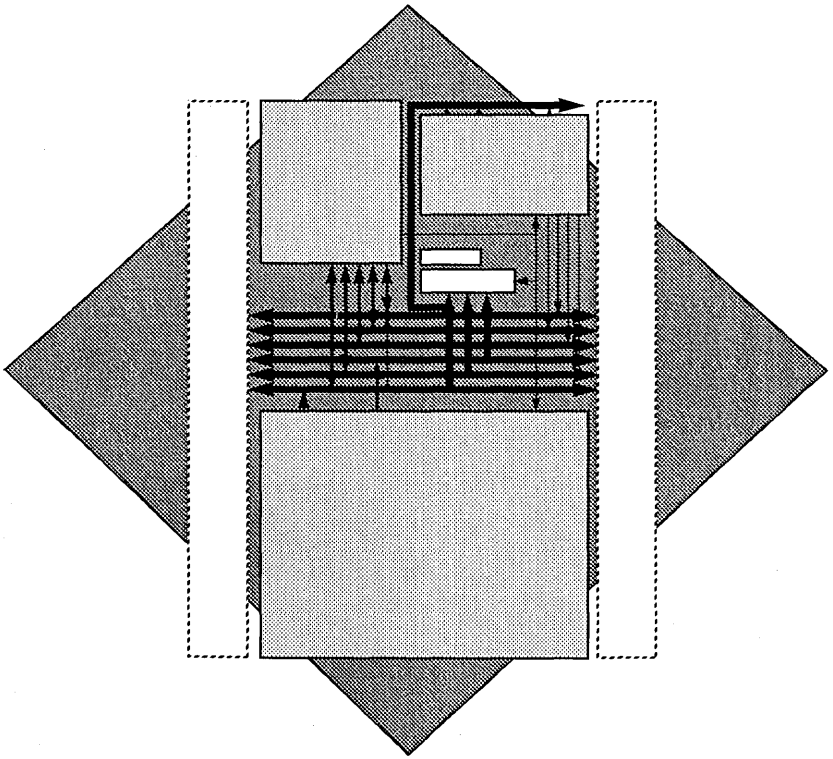
Appendix A – Instruction Set Details

A detailed description of each DSP56K family instruction, its use, and its affect on the processor are presented.

Appendix B – Benchmarks

DSP5K family benchmark results are listed in this appendix.

SECTION 2
DSP56K CENTRAL ARCHITECTURE OVERVIEW



SECTION CONTENTS

2

SECTION 2.1 DSP56K CENTRAL ARCHITECTURE OVERVIEW 3
SECTION 2.2 DATA BUSES 3
SECTION 2.3 ADDRESS BUSES 4
SECTION 2.4 DATA ALU 5
SECTION 2.5 ADDRESS GENERATION UNIT 5
SECTION 2.6 PROGRAM CONTROL UNIT 5
SECTION 2.7 MEMORY EXPANSION PORT (PORT A) 6
SECTION 2.8 ON-CHIP EMULATOR (OnCE) 6
SECTION 2.9 PHASE-LOCKED LOOP (PLL) BASED CLOCKING 6

2.1 DSP56K CENTRAL ARCHITECTURE OVERVIEW

The DSP56K family of processors is built on a standard central processing module. In the expansion area around the central processing module, the chip can support various configurations of memory and peripheral modules which may change from family member to family member. This section introduces the architecture and the major components of the central processing module.

The central components are:

- Data Buses
- Address Buses
- Data Arithmetic Logic Unit (data ALU)
- Address Generation Unit (AGU)
- Program Control Unit (PCU)
- Memory Expansion (Port A)
- On-Chip Emulator (OnCE™) circuitry
- Phase-locked Loop (PLL) based clock circuitry

Figure 2-1 shows a block diagram of a typical DSP56K family processor, including the central processing module and a nonspecific expansion area for memory and peripherals. The following paragraphs give brief descriptions of each of the central components. Each of the components is explained in detail in subsequent chapters.

2.2 DATA BUSES

The DSP56K central processing module is organized around the registers of three independent execution units: the PCU, the AGU, and the data ALU. Data movement between the execution units occurs over four bidirectional 24-bit buses: the X data bus (XDB), the Y data bus (YDB), the program data bus (PDB), and the global data bus (GDB). (Certain instructions treat the X and Y data buses as one 48-bit data bus by concatenating them.) Data transfers between the data ALU and the X data memory or Y data memory occur over XDB and YDB, respectively. XDB and YDB are kept local on the chip to maximize speed and minimize power dissipation. All other data transfers, such as I/O transfers with peripherals, occur over the GDB. Instruction word prefetches occur in parallel over the PDB.

The bus structure supports general register-to-register, register-to-memory, and memory-to-register data movement. It can transfer up to two 24-bit words and one 56-bit word in the same instruction cycle. Transfers between buses occur in the internal bus switch.

2

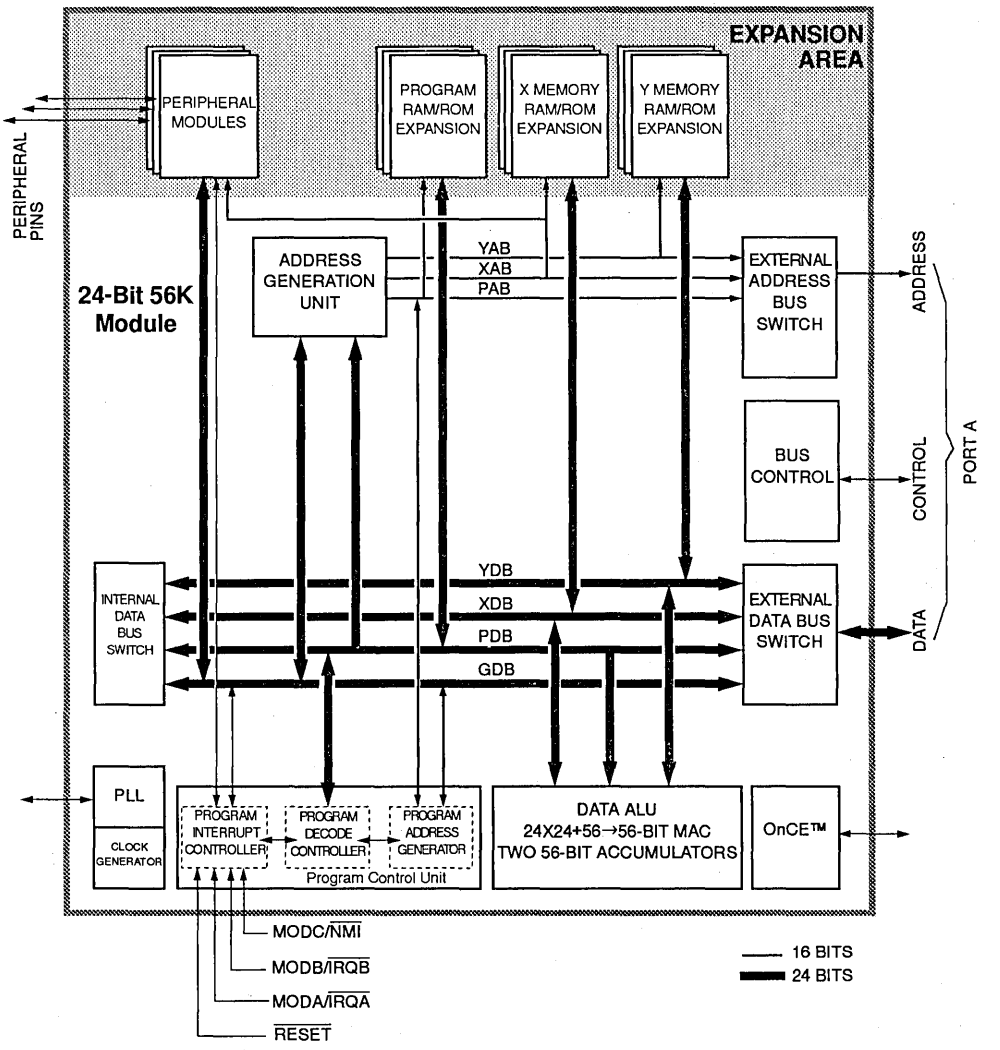


Figure 2-1 DSP56K Block Diagram

2.3 ADDRESS BUSES

Addresses are specified for internal X data memory and Y data memory on two unidirectional 16-bit buses — X address bus (XAB) and Y address bus (YAB). Program memory addresses are specified on the bidirectional program address bus (PAB). External memory spaces are addressed over a single 16-bit unidirectional address bus driven by a

three-input multiplexer that can select the XAB, the YAB, or the PAB. Only one external memory access can be made in an instruction cycle. There is no speed penalty if only one external memory space is accessed in an instruction cycle. However, if two or three external memory spaces are accessed in a single instruction, there will be a one or two instruction cycle execution delay, respectively.

A bus arbitrator controls external access.

2.3.1 Internal Bus Switch

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

2.3.2 Bit Manipulation Unit

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

2.4 DATA ALU

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

2.5 ADDRESS GENERATION UNIT

The AGU performs all of the address storage and address calculations necessary to indirectly address data operands in memory. It operates in parallel with other chip resources to minimize address generation overhead. The AGU has two identical address arithmetic units that can generate two 16-bit addresses every instruction cycle. Each of the arithmetic units can perform three types of arithmetic: linear, modulo, and reverse-carry.

2.6 PROGRAM CONTROL UNIT

The program control unit performs instruction prefetch, instruction decoding, hardware DO loop control, and interrupt (or exception) processing. It consists of three components: the program address generator, the program decode controller, and the program interrupt controller. It contains a 15-level by 32-bit system stack memory and the following six di-

rectly addressable registers: the program counter (PC), loop address (LA), loop counter (LC), status register (SR), operating mode register (OMR), and stack pointer (SP). The 16-bit PC can address 65,536 locations in program memory space.

There are four mode and interrupt control pins that provide input to the program interrupt controller. The Mode Select A/External Interrupt Request A(MODA/$\overline{\text{IRQA}}$) and Mode Select B/External Interrupt Request B (MODB/$\overline{\text{IRQB}}$) pins select the chip operating mode and receive interrupt requests from external sources.

The Mode Select C/Non-Maskable Interrupt (MODC/$\overline{\text{NMI}}$) pin provides further operating mode options and non-maskable interrupt input.

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

2.7 MEMORY EXPANSION PORT (PORT A)

Port A synchronously interfaces with a wide variety of memory and peripheral devices over a common 24-bit data bus. These devices include high-speed static RAMs, slower memory devices, and other DSPs and MPUs in master/slave configurations. This variety is possible because the expansion bus timing is programmable and can be tailored to match the speed requirements of the different memory spaces. Not all DSP56K family members feature a memory expansion port. See the individual device's User's Manual to determine if a particular chip includes this feature.

2.8 ON-CHIP EMULATOR (OnCE)

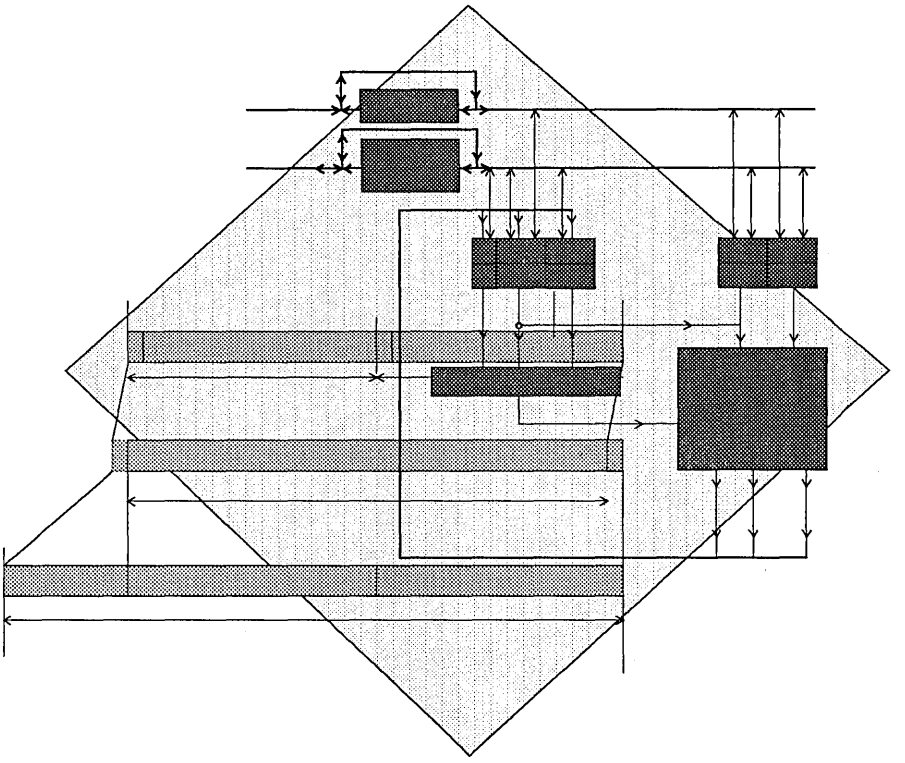
DSP56K on-chip emulation (OnCE) circuitry allows the user to interact with the DSP56K and its peripherals non-intrusively to examine registers, memory, or on-chip peripherals. It provides simple, inexpensive, and speed independent access to the internal registers for sophisticated debugging and economical system development.

Dedicated OnCE pins allow the user to insert the DSP into its target system and retain debug control without sacrificing other user accessible on-chip resources. The design eliminates the costly cabling and the access to processor pins required by traditional emulator systems.

2.9 PHASE-LOCKED LOOP (PLL) BASED CLOCKING

The PLL allows the DSP to use almost any available external system clock for full-speed operation, while also supplying an output clock synchronized to a synthesized internal clock. The PLL performs frequency multiplication, skew elimination, and low-power division.

SECTION 3
DATA ARITHMETIC LOGIC UNIT



SECTION CONTENTS

SECTION 3.1 DATA ARITHMETIC LOGIC UNIT 3

SECTION 3.2 OVERVIEW AND DATA ALU ARCHITECTURE 3

3.2.1 Data ALU Input Registers (X1, X0, Y1, Y0) 5

3.2.2 MAC and Logic Unit 6

3.2.3 Data ALU A and B Accumulators 7

3.2.4 Accumulator Shifter 9

3.2.5 Data Shifter/Limiter 9

3.2.5.1 Limiting (Saturation Arithmetic) 9

3.2.5.2 Scaling 10

SECTION 3.3 DATA REPRESENTATION AND ROUNDING 10

SECTION 3.4 DOUBLE PRECISION MULTIPLY MODE 16

SECTION 3.5 DATA ALU PROGRAMMING MODEL 19

SECTION 3.6 DATA ALU SUMMARY 19

3.1 DATA ARITHMETIC LOGIC UNIT

This section describes the operation of the Data ALU registers and hardware. It discusses data representation, rounding, and saturation arithmetic used within the Data ALU, and concludes with a discussion of the programming model.

3.2 OVERVIEW AND DATA ALU ARCHITECTURE

As described in Section 2, The DSP56K family central processing module is composed of three execution units that operate in parallel. They are the Data ALU, address generation unit (AGU), and the program control unit (PCU) (see Figure 3-1). These three units are register oriented rather than bus oriented and interface over the system buses with memory and memory-mapped I/O devices.

The Data ALU (see Figure 3-2) is the first of these execution units to be presented. It balances speed with the capability to process signals that have a wide dynamic range and performs all arithmetic and logical operations on data operands.

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

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

The Data ALU can perform any of the following operations in a single instruction cycle: multiplication, multiply-accumulate with positive or negative accumulation, convergent rounding, multiply-accumulate with positive or negative accumulation and convergent rounding, addition, subtraction, a divide iteration, a normalization iteration, shifting, and logical operations.

The components of the Data ALU are:

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

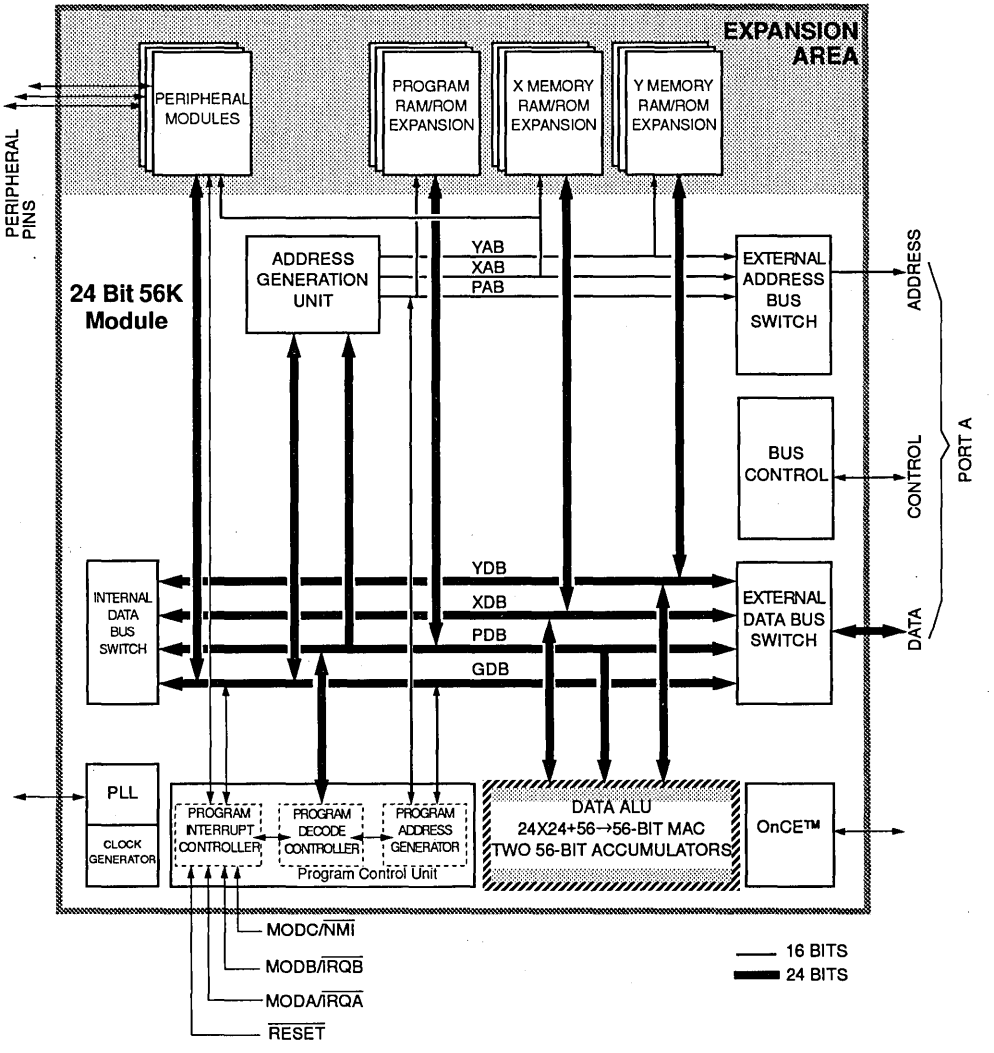


Figure 3-1 DSP56K Block Diagram

The following paragraphs describe each of these components and provide a description of data representation, rounding, and saturation arithmetic.

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

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

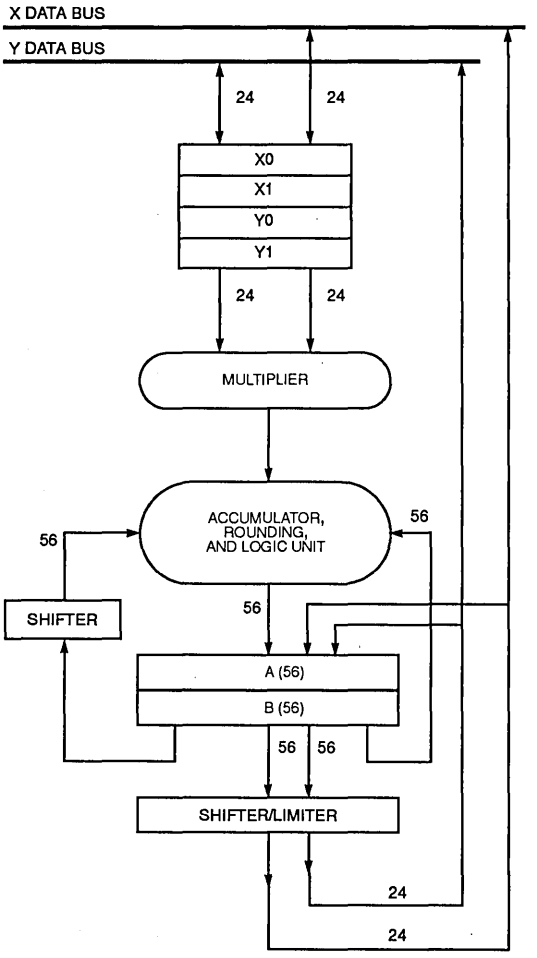


Figure 3-2 Data ALU

register contents. The registers may also be read back out to the appropriate data bus to implement memory-delay operations and save/restore operations for interrupt service routines.

3.2.2 MAC and Logic Unit

The MAC and logic unit shown in Figure 3-3 conduct the main arithmetic processing and perform all calculations on data operands in the DSP.

For arithmetic instructions, the unit accepts up to three input operands and outputs one 56-bit result in the following form: extension:most significant product:least significant product (EXT:MSP:LSP). The operation of the MAC unit occurs independently and in parallel with XDB and YDB activity, and its registers facilitate buffering for Data ALU inputs and outputs. Latches on the MAC unit input permit writing an input register which is the source for a Data ALU operation in the same instruction.

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

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

Convergent rounding (round-to-nearest) is performed when the instruction (for example, the signed multiply-accumulate and round (MACR) instruction) specifies adding the multiplier's product to the contents of the accumulator. The scaling mode bits in the status register specify which bit in the accumulator shall be rounded.

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

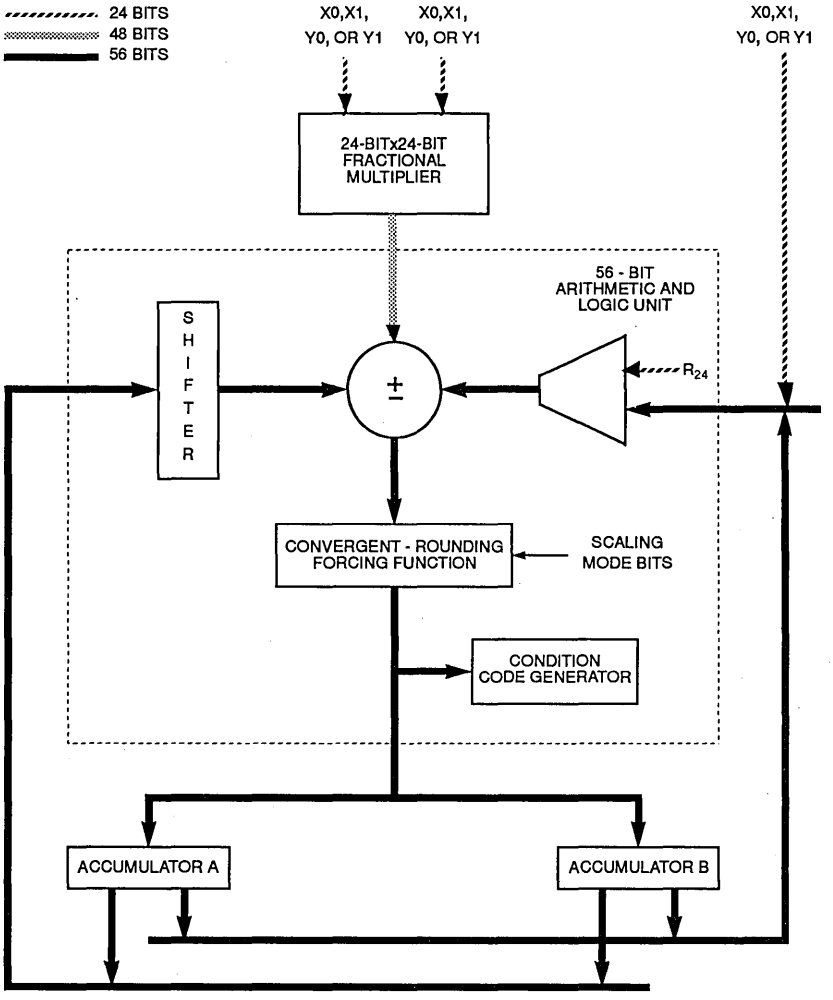
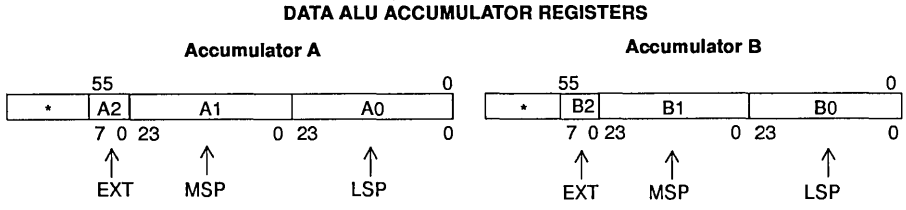


Figure 3-3 MAC Unit

3.2.3 Data ALU A and B Accumulators

The Data ALU features two general-purpose, 56-bit accumulators, A and B. Each consists of three concatenated registers (A2:A1:A0 and B2:B1:B0, respectively). The 8-bit sign extension (EXT) is stored in A2 or B2 and is used when more than 48-bit accuracy is needed; the 24-bit most significant product (MSP) is stored in A1 or B1; the 24-bit least significant product (LSP) is stored in A0 or B0 as shown in Figure 3-4.



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

Figure 3-4 DATA ALU Accumulator Registers

Overflow occurs when a source operand requires more bits for accurate representation than are available in the destination. The 8-bit extension registers offer protection against overflow. In the DSP56K chip family, the extreme values that a word operand can assume are - 1 and + 0.9999998. If the sum of two numbers is less than - 1 or greater than + 0.9999998, the result (which cannot be represented in a 24 bit word operand) has underflowed or overflowed. The 8-bit extension registers can accurately represent the result of 255 overflows or 255 underflows. Whenever the accumulator extension registers are in use, the V bit in the status register is set.

Automatic sign extension occurs when the 56-bit accumulator is written with a smaller operand of 48 or 24 bits. A 24-bit operand is written to the MSP (A1 or B1) portion of the accumulator, the LSP (A0 or B0) portion is zero filled, and the EXT (A2 or B2) portion is sign extended from MSP. A 48-bit operand is written into the MSP:LSP portion (A1:A0 or B1:B0) of the accumulator, and the EXT portion is sign extended from MSP. No sign extension occurs if an individual 24-bit register is written (A1, A0, B1, or B0).When either A or B is read, it may be optionally scaled one bit left or one bit right for block floating-point arithmetic. Sign extension can also occur when writing A or B from the XDB and/or YDB or with the results of certain Data ALU operations (such as the transfer conditionally (Tcc) or transfer Data ALU register (TFR) instructions).

Overflow protection occurs when the contents of A or B are transferred over the XDB and YDB by substituting a limiting constant for the data. Limiting does not affect the content of A or B – only the value transferred over the XDB or YDB is limited. This overflow protection occurs after the contents of the accumulator has been shifted according to the scaling mode. Shifting and limiting occur only when the *entire* 56-bit A or B accumulator is specified as the source for a parallel data move over the XDB or YDB. When individual registers A0, A1, A2, B0, B1, or B2 are specified as the source for a parallel data move, shifting and limiting are not performed.

3.2.4 Accumulator Shifter

The accumulator shifter (see Figure 3-3) is an asynchronous parallel shifter with a 56-bit input and a 56-bit output that is implemented immediately before the MAC accumulator input. The source accumulator shifting operations are as follows:

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

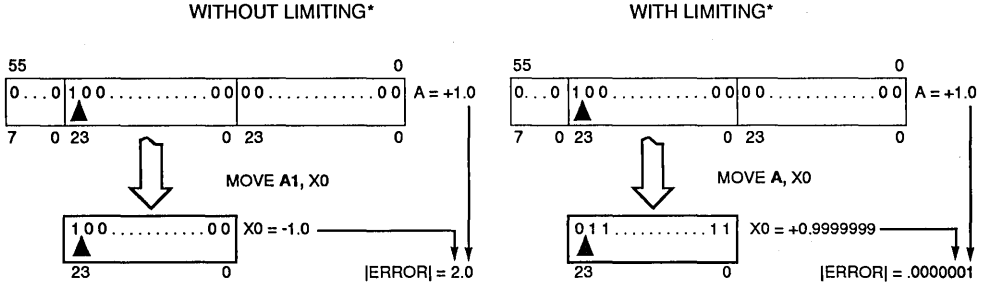
3.2.5 Data Shifter/Limiter

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

3.2.5.1 Limiting (Saturation Arithmetic)

The A and B accumulators serve as buffer registers between the MAC unit and the XDB and/or YDB. They act both as Data ALU source and destination operands.Test logic exists in each accumulator register to support the operation of the data shifter/limiter circuits. This test logic detects overflows out of the data shifter so that the limiter can substitute one of several constants to minimize errors due to the overflow. This process is called saturation arithmetic

The Data ALU A and B accumulators 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. If the contents of the selected source accumulator can be represented without overflow in the destination operand size (i.e., accumulator extension register not in use), the data limiter is disabled, and the operand is not modified. If contents of the selected source accumulator cannot be represented without overflow in the destination operand size, the data limiter will substitute a limited data value with maximum magnitude (saturated) and with the same sign as the source accumulator contents: $7FFFFF for 24-bit or $7FFFFF FFFFFF for 48-bit positive numbers, $800000 for 24-bit or $800000 000000 for 48-bit negative numbers. This process is called saturation arithmetic. The value in the accumulator register is not shifted and can be reused within the Data ALU. When limiting does occur, a flag is set and latched in the status register.Two limiters allow two-word operands to be limited independently in the same instruction cycle. The two data limiters can also be combined to form one 48-bit data limiter for long-word operands.



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

Figure 3-5 Saturation Arithmetic

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

Figure 3-5 shows the effects of saturation arithmetic on a move from register A1 to register X0. The instruction "MOVE A1,X0" causes a move without limiting, and the instruction "MOVE A,X0" causes a move of the same 24 bits with limiting. The error without limiting is 2.0; whereas, it is 0.0000001 with limiting. Table 3-1 shows a more complete set of limiting situations.

3.2.5.2 Scaling

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

3.3 DATA REPRESENTATION AND ROUNDING

The DSP56K uses a fractional data representation for all Data ALU operations. Figure 3-7 shows the bit weighting of words, long words, and accumulator operands for this representation. The decimal points are all aligned and are left justified.

Table 3-1 Limited Data Values

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

Data must be converted to a fractional number by scaling before being used by the DSP or the user will have to be very careful in how the DSP manipulates the data. Moving $3F to a 24-bit Data ALU register does not result in the contents being $00003F as might be expected. Assuming numbers are fractional, the DSP left justifies rather than right justifies. As a result, storing $3F in a 24-bit register results in the contents being $3F0000. The simplest example of scaling is to convert all integer numbers to fractional numbers by shifting the decimal 24 places to the left (see Figure 3-6). Thus, the data has not changed; only the position of the decimal has moved.

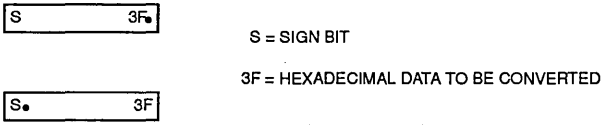


Figure 3-6 Integer-to-Fractional Data Conversion

For words and long words, the most negative number that can be represented is -1 whose internal representation is $800000 and $800000000000, respectively. The most positive word is $7FFFFF or $1 - 2^{-23}$ and the most positive long word is $7FFFFFFFFFFF or $1 - 2^{-47}$. These limitations apply to all data stored in memory and to data stored in the Data ALU input buffer registers. The extension registers associated with the accumula-

tors allow word growth so that the most positive number that can be used is approximately 256 and the most negative number is approximately -256. When the accumulator extension registers are in use, the data contained in the accumulators cannot be stored exactly in memory or other registers. In these cases, the data must be limited to the most positive or most negative number consistent with the size of the destination and the sign of the accumulator (the most significant bit (MSB) of the extension register).

To maintain alignment of the binary point when a word operand is written to accumulator A or B, the operand is written to the most significant accumulator register (A1 or B1), and its MSB is automatically sign extended through the accumulator extension register. The least significant accumulator register is automatically cleared. When a long-word operand is written to an accumulator, the least significant word of the operand is written to the least significant accumulator register A0 or B0 and the most significant word is written to A1 or B1(see Figure 3-8).

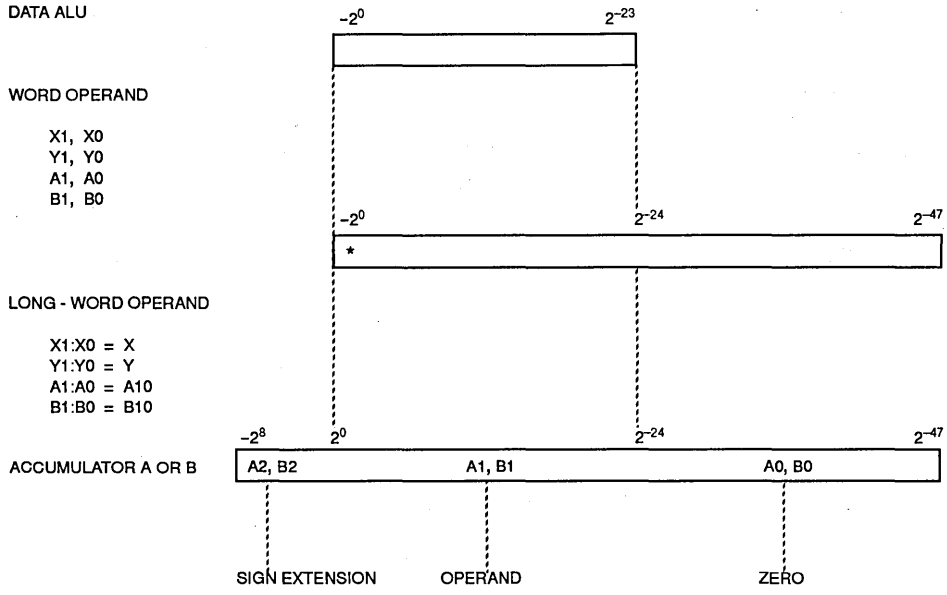


Figure 3-7 Bit Weighting and Alignment of Operands

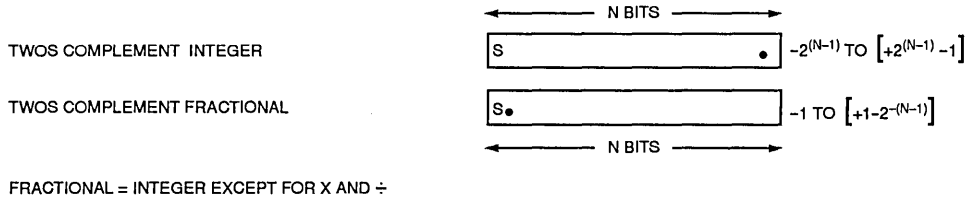


Figure 3-8 Integer/Fractional Number Comparison

A comparison between integer and fractional number representation is shown in Figure 3-8. The number representation for integers is between $\pm 2^{(N-1)}$; whereas, the fractional representation is limited to numbers between ±1. To convert from an integer to a fractional number, the integer must be multiplied by a scaling factor so the result will always be between ±1. The representation of integer and fractional numbers is the same if the numbers are added or subtracted but is different if the numbers are multiplied or divided. An example of two numbers multiplied together is given in Figure 3-9. The key difference is that the extra bit in the integer multiplication is used as a duplicate sign bit and as the least significant bit (LSB) in the fractional multiplication. The advantages of fractional data representation are as follows:

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

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

The Data ALU MAC performs rounding of the accumulator register to single precision if requested in the instruction (the A1 or B1 register is rounded according to the contents of the A0 or B0 register). The rounding method is called round-to-nearest (even) number, or convergent rounding. The usual rounding method rounds up any value above one-half and rounds down any value below one-half. The question arises as to which way one-half should be rounded. If it is always rounded one way, the results will eventually be

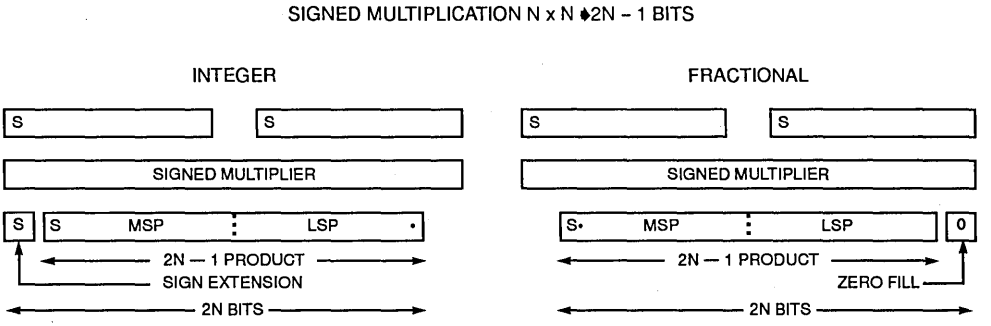
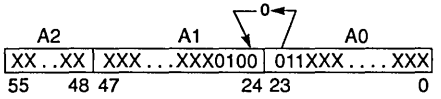


Figure 3-9 Integer/Fractional Multiplication Comparison

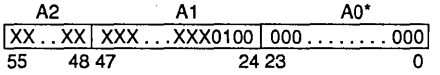
biased in that direction. Convergent rounding solves the problem by rounding down if the number is odd (LSB=0) and rounding up if the number is even (LSB=1). Figure 3-10 shows the four cases for rounding a number in the A1 (or B1) register. If scaling is set in the status register, the resulting number will be rounded as it is put on the data bus. However, the contents of the register are not scaled.

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

BEFORE ROUNDING

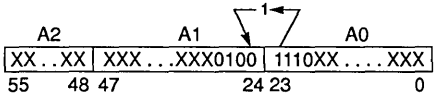


AFTER ROUNDING

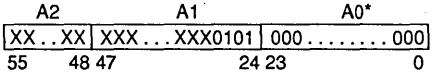


CASE II: IF A0 > $800000 (1/2), THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING

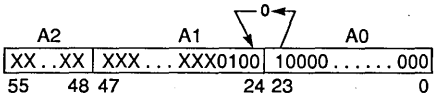


AFTER ROUNDING

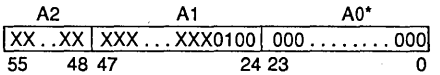


CASE III: IF A0 = $800000 (1/2), AND THE LSB OF A1 = 0,THEN ROUND DOWN (ADD NOTHING)

BEFORE ROUNDING

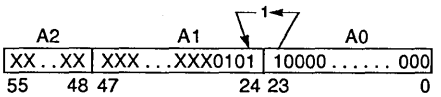


AFTER ROUNDING

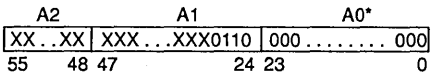


CASE IV: IF A0 = $800000 (1/2), AND THE LSB = 1, THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING



AFTER ROUNDING



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

Figure 3-10 Convergent Rounding

3.4 DOUBLE PRECISION MULTIPLY MODE

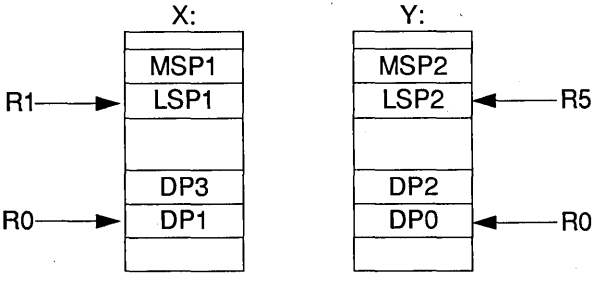
The Data ALU double precision multiply operation multiplies two 48-bit operands with a 96-bit result. The processor enters the dedicated Double Precision Multiply Mode when the user sets bit 14 (DM) of the Status Register (bit 6 of the MR register). The mode is disabled by clearing the DM bit. For information on the DM bit, see Section 5.4.2.13 - Double Precision Multiply Mode (Bit 14).

3

CAUTION:

While in the Double Precision Multiply Mode, only the double precision multiply algorithms shown in Figure 3-11, Figure 3-12, and Figure 3-13 may be executed by the Data ALU; any other Data ALU operation will give indeterminate results.

Figure 3-11 shows the full double precision multiply algorithm. To allow for pipeline delay, the ANDI instruction should not be immediately followed by a Data ALU instruction. For example, the ORI instruction sets the DM mode bit, but, due to the instruction execution pipeline, the Data ALU enters the Double Precision Multiply mode only after



DP3_DP2_DP1_DP0 = MSP1_LSP1 x MSP2_LSP2

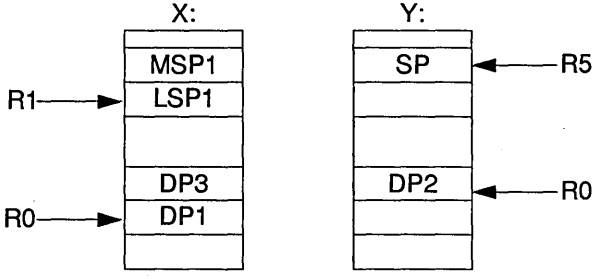
```
ori     #$40,mr                                         ;enter mode
move                x:(r1)+,x0      y:(r5)+,y0          ;load operands
mpy     y0,x0,a     x:(r1)+,x1      y:(r5)+,y1          ;LSP*LSP→a
mac     x1,y0,a                     a0,y:(r0)           ;shifted(a)+
                                                        ;  MSP*LSP→a
mac     x0,y1,a                                         ;a+LSP*MSP→a
mac     y1,x1,a     a0,x:(r0)+                          ;shifted(a)+
                                                        ;  MSP*MSP→a
move    a,l:(r0)+
andi    #$bf,mr                                         ;exit mode
non-Data ALU operation                                  ;pipeline delay
```

Figure 3-11 Full Double Precision Multiply Algorithm

one instruction cycle. The ANDI instruction clears the DM mode bit, but, due to the instruction execution pipeline, the Data ALU leaves the mode after one instruction cycle.

The double precision multiply algorithm uses the Y0 register at all stages. If the use of the Data ALU is required in an interrupt service routine, Y0 should be saved together with other Data ALU registers to be used, and should be restored before leaving the interrupt routine.

If just single precision times double precision multiply is desired, two of the multiply operations may be deleted and replaced by suitable initialization and clearing of the accumulator and Y0. Figure 3-12 shows the single precision times double precision algorithm.



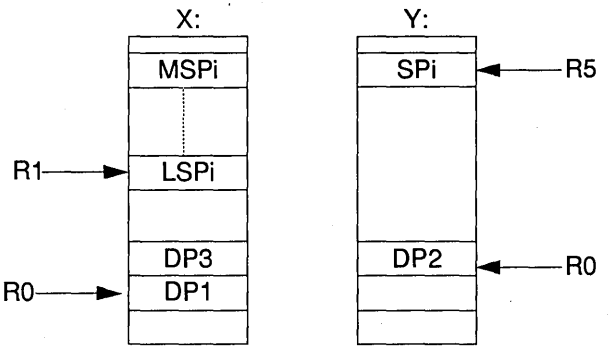
DP3_DP2_DP1 = MSP1_LSP1 x SP

```
clr a           #0,y0                          ;clear a and y0
ori  #$40,mr                                   ;enter DP mode
move            x:(r1)+,x0      y:(r5)+,y1     ;load LSP1 and SP
mac  x0,y1,a    x:(r1)+,x1                     ;LSP1*SP→a,
                                               ;load MSP1
mac  y1,x1,a    a0,x:(r0)+                     ;shifted(a)+
                                               ;  SP*MSP1→a,
                                               ;save DP1
move a,l:(r0)+                                 ;save DP3_DP2
andi #$bf,mr                                   ;exit DP mode
non-Data ALU operation                         ;pipeline delay
```

Figure 3-12 Single × Double Multiply Algorithm

Figure 3-13 shows a single precision times double precision multiply-accumulate algorithm. First, the least significant parts of the double precision values are multiplied by the single precision values and accumulated in the "Double Precision Multiply" mode. Then the DM bit is cleared and the least significant part of the result is saved to memory. The most significant parts of the double precision values are then multiplied by the single pre-

cision values and accumulated using regular MAC instructions. Note that the maximum number of single times double MAC operations in this algorithm are limited to 255 since overflow may occur (the A2 register is just eight bits long). If a longer sequence is required, it should be split into sub-sequences each with no more than 255 MAC operations.



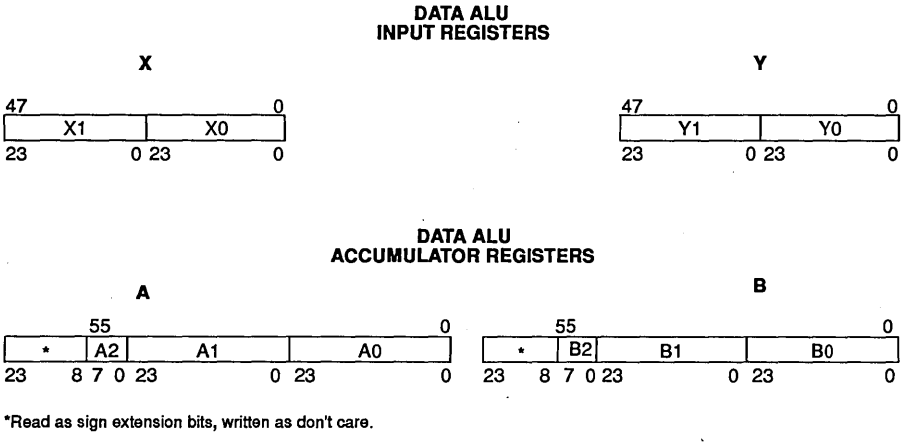
$$DP3\_DP2\_DP1 = \sum MSPi\_LSPi \times SPi$$

```
move              #N-1,m5
clr a             #0,y0                              ;clear a and y0
ori     #$40,mr                                      ;enter DP mode
move              x:(r1)+,x0      y:(r5)+,y1         ;load LSPi and SPi
rep     #N                                           ;0<N<256
mac     x0,y1,a   x:(r1)+,x0      y:(r5)+,y1         ;LSPi*SPi→a
andi    #$bf,mr                                      ;exit DP mode
move              a0,x:(r0)+                         ;save DP1
move              a1,y0
move              a2,a
move              y0,a0                              ;a2:a1→a1:a0
rep     #N
mac     x0,y1,a   x:(r1)+,x0      y:(r5)+,y1         ;load MSPi and SPi
move    a,l:(r0)+                                    ;save DP3_DP2
```

Figure 3-13 Single × Double Multiply-Accumulate Algorithm

3.5 DATA ALU PROGRAMMING MODEL

The Data ALU features 24-bit input/output data registers that can be concatenated to accommodate 48-bit data and two 56-bit accumulators, which are segmented into three 24-bit pieces that can be transferred over the buses. Figure 3-14 illustrates how the registers in the programming model are grouped.



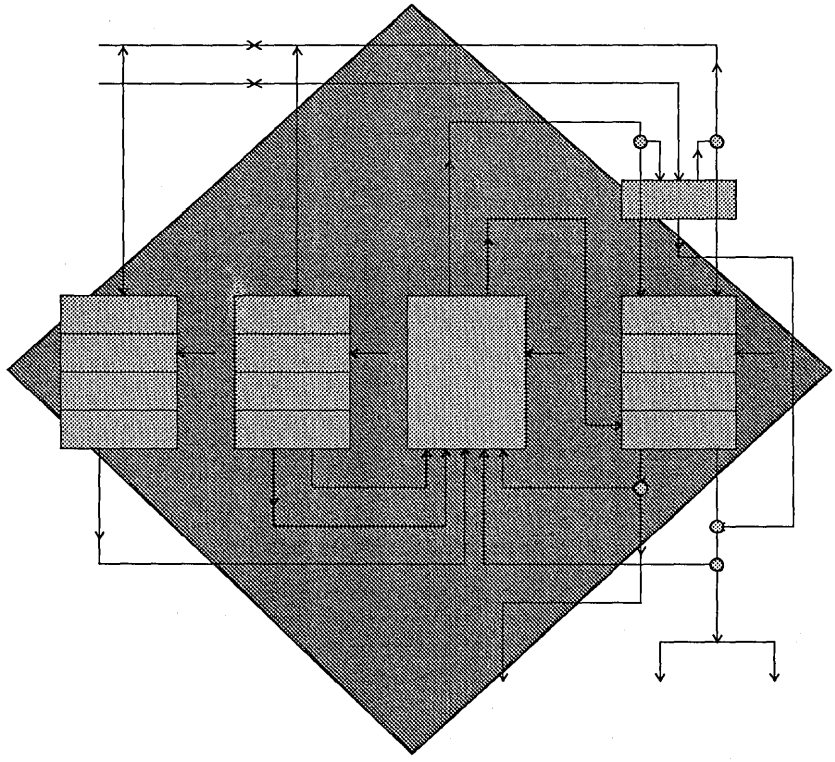
*Read as sign extension bits, written as don't care.

Figure 3-14 DSP56K Programming Model

3.6 DATA ALU SUMMARY

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

SECTION 4
ADDRESS GENERATION UNIT



SECTION CONTENTS

SECTION 4.1 ADDRESS GENERATION UNIT AND ADDRESSING MODES 3

SECTION 4.2 AGU ARCHITECTURE 3
4.2.1 Address Register Files (Rn) 3
4.2.2 Offset Register Files (Nn) 4
4.2.3 Modifier Register Files (Mn) 5
4.2.4 Address ALU 5
4.2.5 Address Output Multiplexers 6

SECTION 4.3 PROGRAMMING MODEL 6
4.3.1 Address Register Files (R0 - R3 and R4 - R7) 7
4.3.2 Offset Register Files (N0 - N3 and N4 - N7) 7
4.3.3 Modifier Register Files (M0 - M3 and M4 - M7) 8

SECTION 4.4 ADDRESSING 8
4.4.1 Address Register Indirect Modes 9
4.4.1.1 No Update 9
4.4.1.2 Postincrement By 1 9
4.4.1.3 Postdecrement By 1 9
4.4.1.4 Postincrement By Offset Nn 10
4.4.1.5 Postdecrement By Offset Nn 11
4.4.1.6 Indexed By Offset Nn 12
4.4.1.7 Predecrement By 1 13
4.4.2 Address Modifier Arithmetic Types 14
4.4.2.1 Linear Modifier (Mn=$FFFF) 16
4.4.2.2 Modulo Modifier 18
4.4.2.3 Reverse-Carry Modifier (Mn=$0000) 22
4.4.2.4 Address-Modifier-Type Encoding Summary 25

4.1 ADDRESS GENERATION UNIT AND ADDRESSING MODES

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

4.2 AGU ARCHITECTURE

The AGU is shown in the DSP56K block diagram in Figure 4-1. It uses integer arithmetic to perform the effective address calculations necessary to address data operands in memory, and contains the registers used to generate the addresses. It implements linear, modulo, and reverse-carry arithmetic, and operates in parallel with other chip resources to minimize address-generation overhead.

The AGU is divided into two identical halves, each of which has an address arithmetic logic unit (ALU) and four sets of three registers (see Figure 4-2). They are the address registers (R0 - R3 and R4 - R7), offset registers (N0 - N3 and N4 - N7), and the modifier registers (M0 - M3 and M4 - M7). The eight Rn, Nn, and Mn registers are treated as register triplets — e.g., only N2 and M2 can be used to update R2. The eight triplets are R0:N0:M0, R1:N1:M1, R2:N2:M2, R3:N3:M3, R4:N4:M4, R5:N5:M5, R6:N6:M6, and R7:N7:M7.

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

4.2.1 Address Register Files (Rn)

Each of the two address register files (see Figure 4-2) consists of four 16-bit registers. The two files contain address registers R0 - R3 and R4 - R7, which usually contain addresses used as pointers to memory. Each register may be read or written by the global data bus (GDB). When read by the GDB, 16-bit registers are written into the two least significant bytes of the GBD, and the most significant byte is set to zero. When written from the GBD, only the two least significant bytes are written, and the most significant byte is truncated. Each address register can be used as input to its associated address ALU for a register update calculation. Each register can also be written by the output of its respective address ALU. One Rn register from the low address ALU and one Rn register from the high address ALU can be accessed in a single instruction.

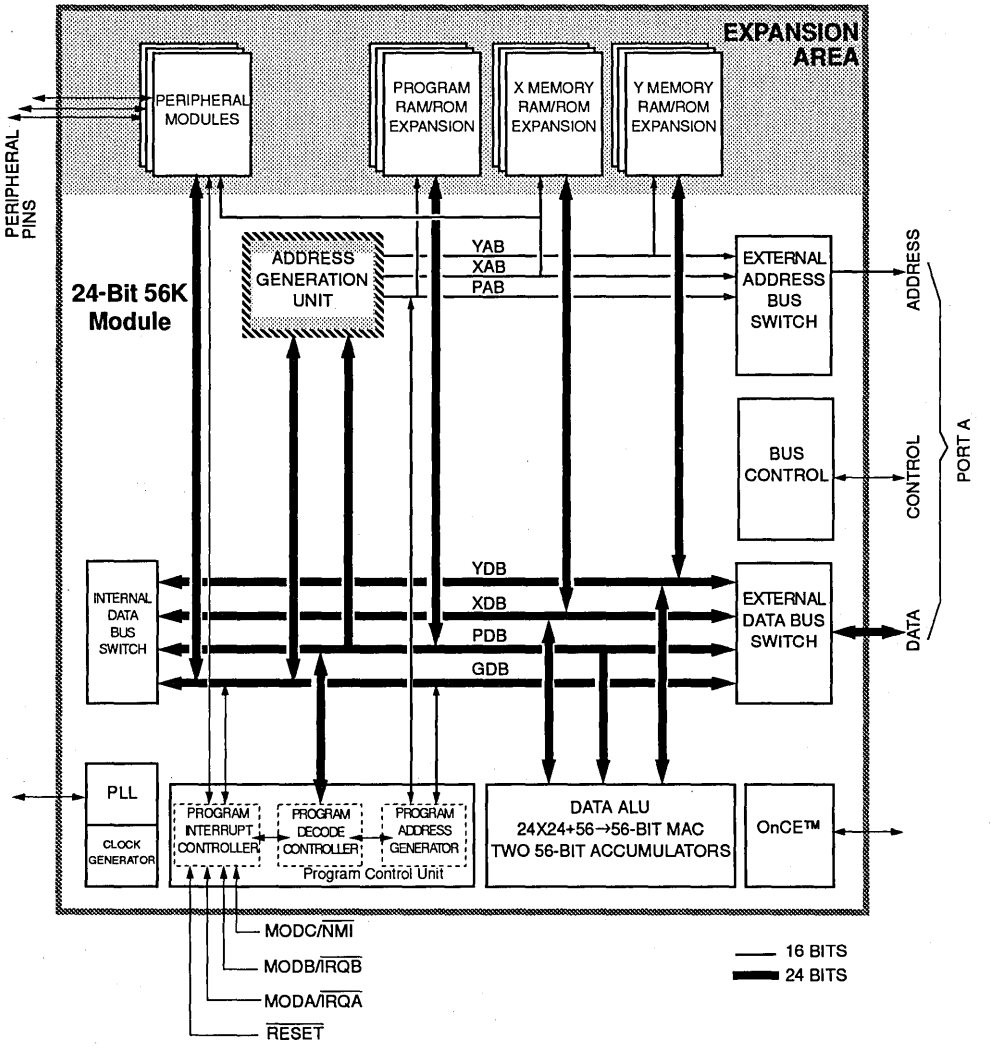


Figure 4-1 DSP56K Block Diagram

4.2.2 Offset Register Files (Nn)

Each of two offset register files shown in Figure 4-2 consists of four 16-bit registers. The two files contain offset registers N0 - N3 and N4 - N7, which contain either data or offset values used to update address pointers. Each offset register can be read or written by the GDB. When read by the GDB, the contents of a register are placed in the two least signif-

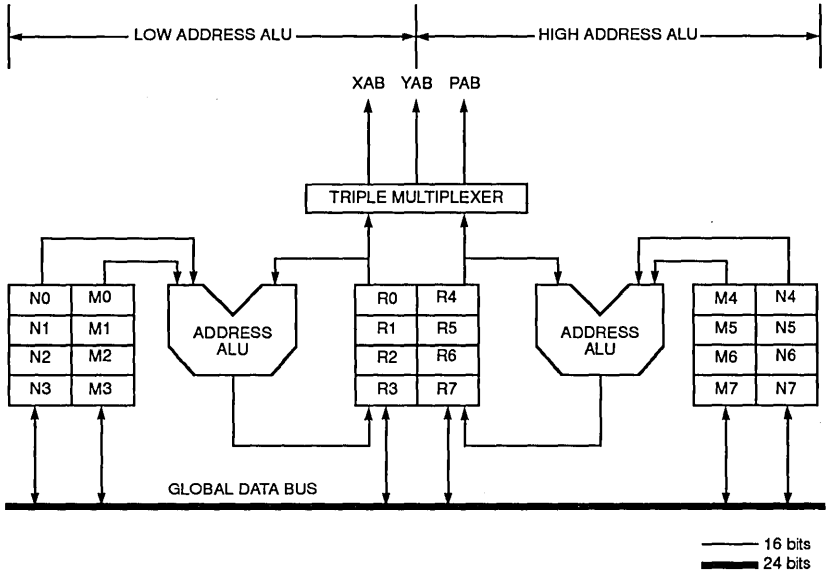


Figure 4-2 AGU Block Diagram

icant bytes, and the most significant byte on the GDB is zero extended. When a register is written, only the least significant 16 bits of the GDB are used; the upper portion is truncated.

4.2.3 Modifier Register Files (Mn)

Each of the two modifier register files shown in Figure 4-2 consists of four 16-bit registers. The two files contain modifier registers M0 - M3 and M4 - M7, which specify the type of arithmetic used during address register update calculations or contain data. Each modifier register can be read or written by the GDB. When read by the GDB, the contents of a register are placed in the two least significant bytes, and the most significant byte on the GDB is zero extended. When a register is written, only the least significant 16 bits of the GDB are used; the upper portion is truncated. Each modifier register is preset to $FFFF during a processor reset.

4.2.4 Address ALU

The two address ALUs are identical (see Figure 4-2) in that each contains a 16-bit full adder (called an offset adder), which can add 1) plus one, 2) minus one, 3) the contents of the respective offset register N, or 4) the twos complement of N to the contents of the selected address register. A second full adder (called a modulo adder) adds the summed

result of the first full adder to a modulo value, M or minus M, where M-1 is stored in the respective modifier register. A third full adder (called a reverse-carry adder) can add 1) plus one, 2) minus one, 3) the offset N (stored in the respective offset register), or 4) minus N to the selected address register with the carry propagating in the reverse direction — i.e., from the most significant bit (MSB) to the least significant bit (LSB). The offset adder and the reverse-carry adder are in parallel and share common inputs. The only difference between them is that the carry propagates in opposite directions. Test logic determines which of the three summed results of the full adders is output.

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

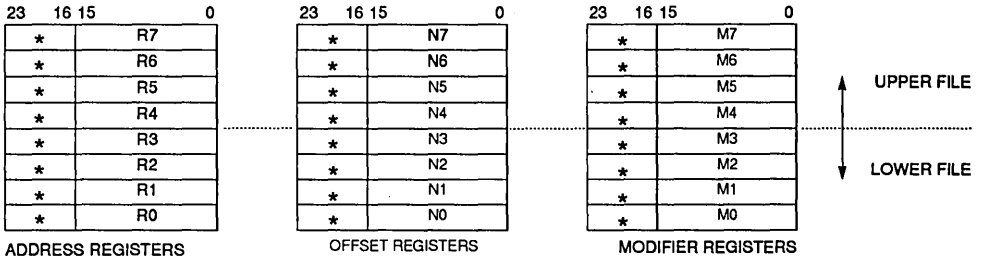
The output of the offset adder gives the result of linear arithmetic (e.g., Rn ± 1; Rn ± N) and is selected as the modulo arithmetic unit output for linear arithmetic addressing modifiers. The reverse-carry adder performs the required operation for reverse-carry arithmetic and its result is selected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

4.2.5 Address Output Multiplexers

The address output multiplexers (see Figure 4-2) select the source for the XAB, YAB, and PAB. These multiplexers allow the XAB, YAB, or PAB outputs to originate from R0 - R3 or R4 - R7.

4.3 PROGRAMMING MODEL

The programmer's view of the AGU is eight sets of three registers (see Figure 4-3). These registers can act as temporary data registers and indirect memory pointers. Automatic updating is available when using address register indirect addressing. The Mn registers can be programmed for linear addressing, modulo addressing, and bit-reverse addressing.



* Written as don't care; read as zero

Figure 4-3 AGU Programming Model

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

The eight 16-bit address registers, R0 - R7, can contain addresses or general-purpose data. The 16-bit address in a selected address register is used in the calculation of the effective address of an operand. When supporting parallel X and Y data memory moves, the address registers must be thought of as two separate files, R0 - R3 and R4 - R7. The contents of an Rn may point directly to data or may be offset. In addition, Rn can be pre-updated or post-updated according to the addressing mode selected. If an Rn is updated, modifier registers, Mn, are always used to specify the type of update arithmetic. Offset registers, Nn, are used for the update-by-offset addressing modes. The address register modification is performed by one of the two modulo arithmetic units. Most addressing modes modify the selected address register in a read-modify-write fashion; the address register is read, its contents are modified by the associated modulo arithmetic unit, and the register is written with the appropriate output of the modulo arithmetic unit. The form of address register modification performed by the modulo arithmetic unit is controlled by the contents of the offset and modifier registers discussed in the following paragraphs. Address registers are not affected by a processor reset.

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

The eight 16-bit offset registers, N0 - N7, can contain offset values used to increment/decrement address registers in address register update calculations or can be used for 16-bit general-purpose storage. For example, the contents of an offset register can be used to step through a table at some rate (e.g., five locations per step for waveform generation), or the contents can specify the offset into a table or the base of the table for indexed addressing. Each address register, Rn, has its own offset register, Nn, associated with it. Offset registers are not affected by a processor reset.

Table 4-1 Address Register Indirect Summary

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

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

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

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

4.4 ADDRESSING

The DSP56K provides three different addressing modes: register direct, address register indirect, and special. Since the register direct and special addressing modes do not necessarily use the AGU registers, they are described in SECTION 6 - INSTRUCTION SET INTRODUCTION. The address register indirect addressing modes use the registers in the AGU and are described in the following paragraphs.

4.4.1 Address Register Indirect Modes

When an address register is used to point to a memory location, the addressing mode is called "address register indirect" (see Table 4-1). The term indirect is used because the register contents are not the operand itself, but rather the address of the operand. These addressing modes specify that an operand is in memory and specify the effective address of that operand.

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

Not all possible combinations are available, such as + (Rn). The 24-bit instruction word size is not large enough to allow a completely orthogonal instruction set for all instructions used by the DSP.

An example and description of each mode is given in the following paragraphs. SECTION 6 - INSTRUCTION SET INTRODUCTION and APPENDIX A - INSTRUCTION SET DETAILS give a complete description of the instruction syntax used in these examples. In particular, XY: memory references refer to instructions in which an operand in X memory and an operand in Y memory are referenced in the same instruction.

4.4.1.1 No Update

The address of the operand is in the address register, Rn (see Table 4-1). The contents of the Rn register are unchanged by executing the instruction. Figure 4-4 shows a MOVE instruction using address register indirect addressing with no update. This mode can be used for making XY: memory references. This mode does not use Nn or Mn registers.

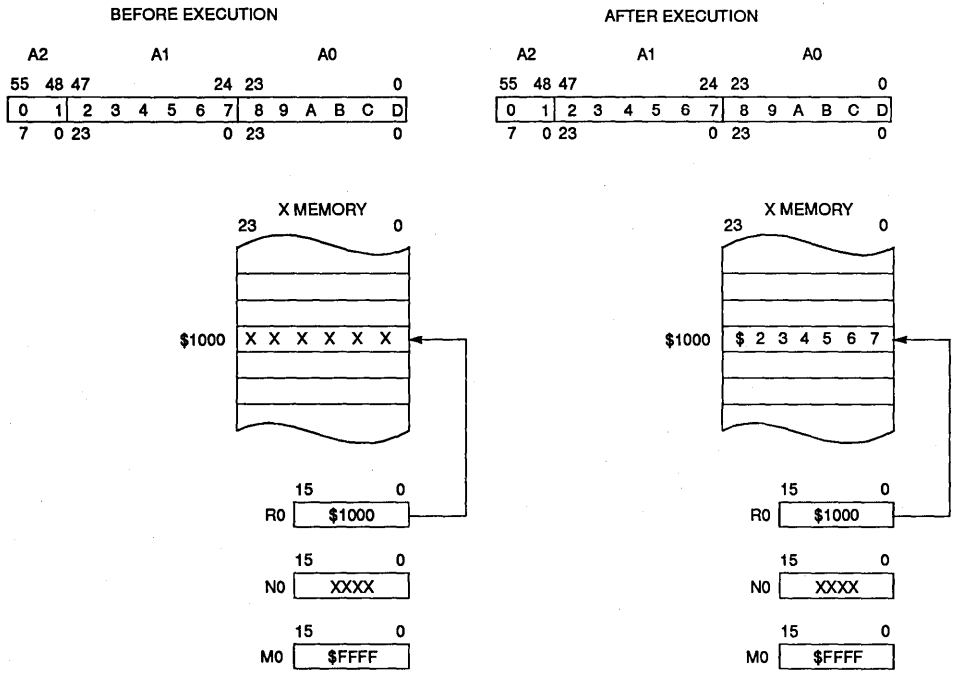
4.4.1.2 Postincrement By 1

The address of the operand is in the address register, Rn (see Table 4-1 and Figure 4-5). After the operand address is used, it is incremented by 1 and stored in the same address register. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.

4.4.1.3 Postdecrement By 1

The address of the operand is in the address register, Rn (see Table 4-1 and Figure 4-6). After the operand address is used, it is decremented by 1 and stored in the same address register. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.

EXAMPLE: MOVE A1,X: (R0)



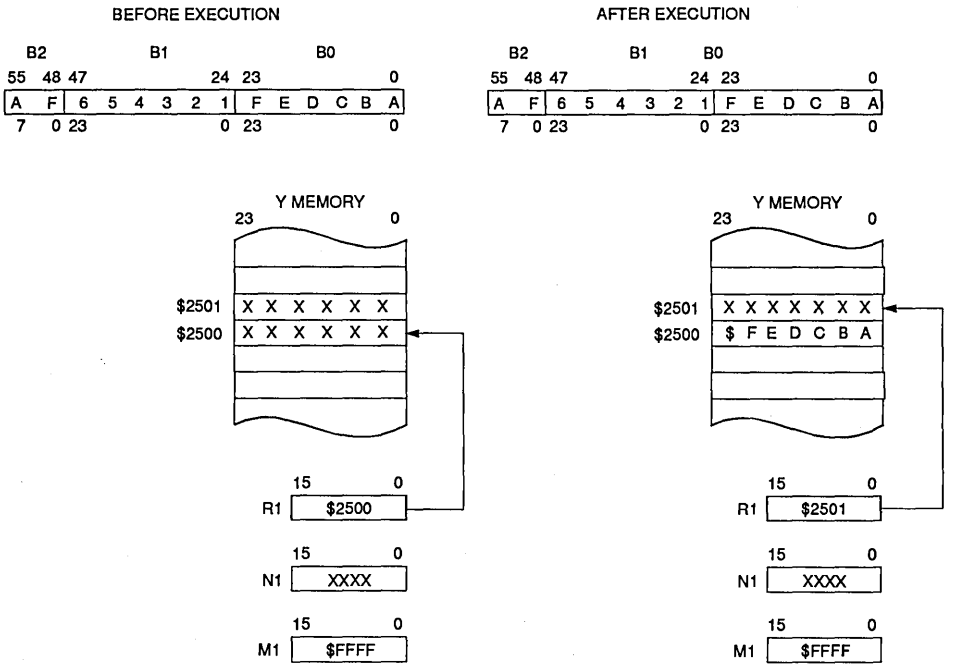
Assembler Syntax: (Rn)
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 4-4 Address Register Indirect — No Update

4.4.1.4 PostIncrement By Offset Nn

The address of the operand is in the address register, Rn (see Table 4-1 and Figure 4-7). After the operand address is used, it is incremented by the contents of the Nn register and stored in the same address register. The contents of the Nn register are unchanged. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.

EXAMPLE: MOVE B0,Y: (R1)+



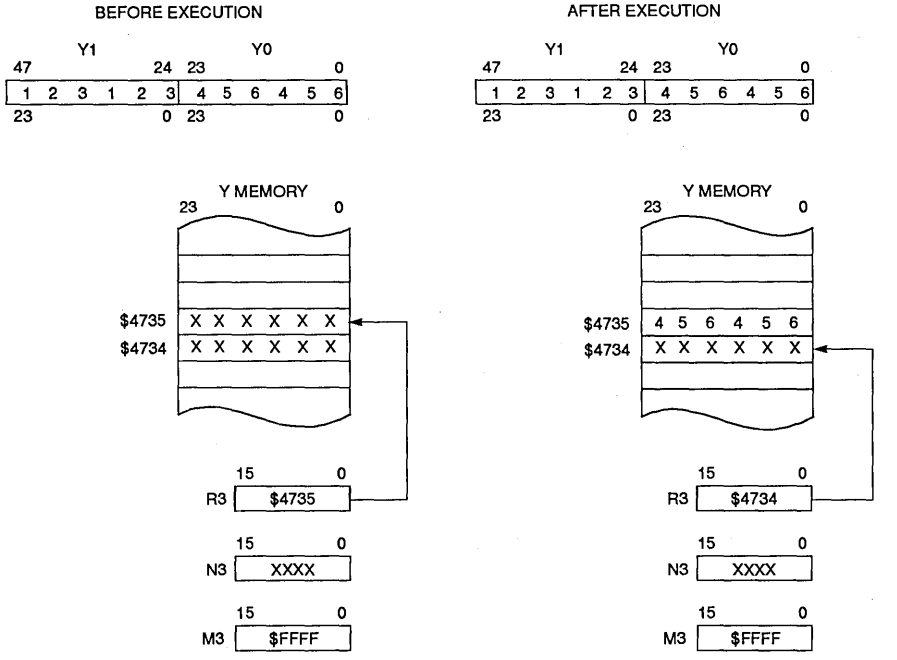
Assembler Syntax: (Rn)+
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 4-5 Address Register Indirect — Postincrement

4.4.1.5 Postdecrement By Offset Nn

The address of the operand is in the address register, Rn (see Table 4-1 and Figure 4-8). After the operand address is used, it is decremented by the contents of the Nn register and stored in the same address register. The contents of the Nn register are unchanged. This mode cannot be used for making XY: memory references, but it can be used to modify the contents of Rn without an associated data move.

EXAMPLE: MOVE Y0,Y: (R3)-



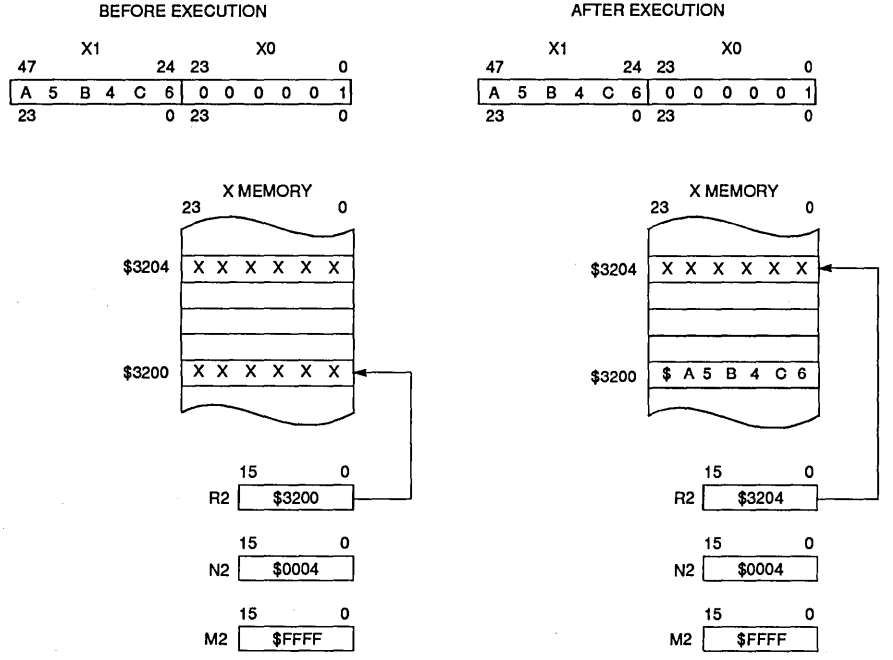
Assembler Syntax: (Rn)-
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 4-6 Address Register Indirect — Postdecrement

4.4.1.6 Indexed By Offset Nn

The address of the operand is the sum of the contents of the address register, Rn, and the contents of the address offset register, Nn (see Table 4-1 and Figure 4-9). The contents of the Rn and Nn registers are unchanged. This addressing mode, which requires an extra instruction cycle, cannot be used for making XY: memory references.

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



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

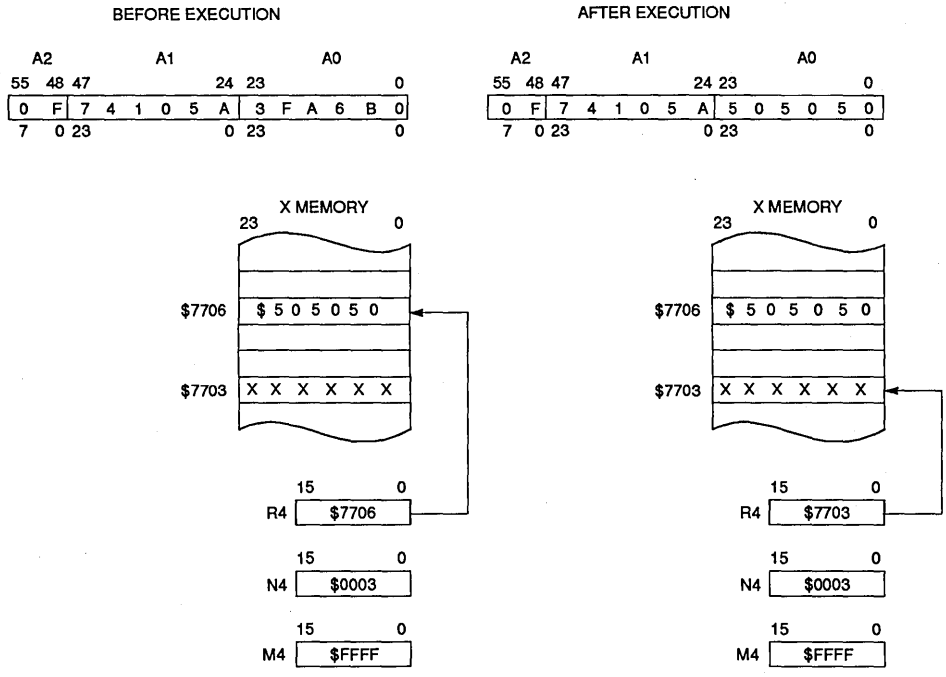
Figure 4-7 Address Register Indirect — Postincrement by Offset Nn

4.4.1.7 Predecrement By 1

The address of the operand is the contents of the address register, Rn, decremented by 1 before the operand address is used (see Table 4-1 and Figure 4-10). The contents of Rn are decremented and stored in the same address register. This addressing mode requires an extra instruction cycle. This mode cannot be used for making XY: memory references, nor can it be used for modifying the contents of Rn without an associated data move.

4

EXAMPLE: MOVE X:(R4)–N4,A0



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

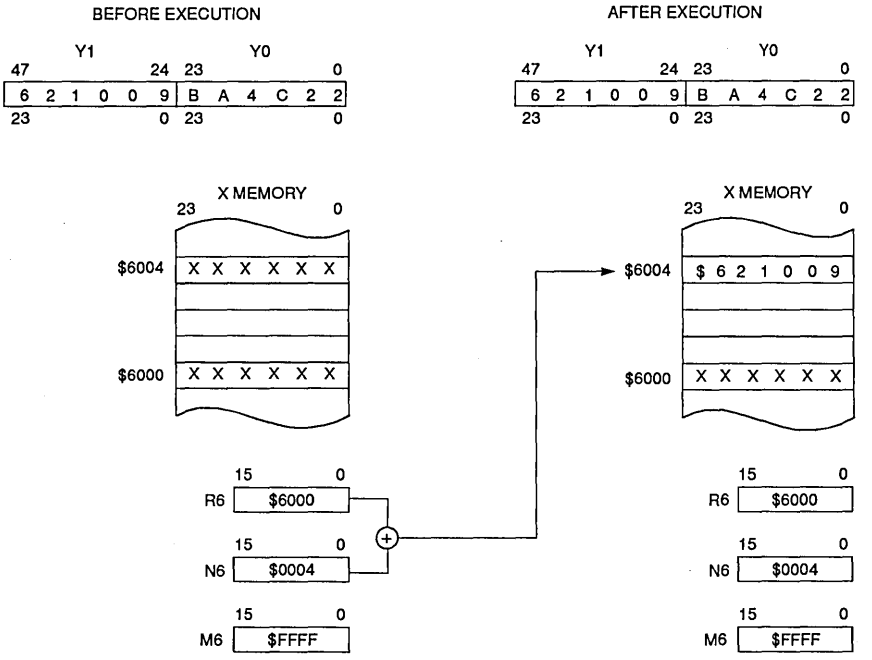
Figure 4-8 Address Register Indirect — Postdecrement by Offset Nn

4.4.2 Address Modifier Arithmetic Types

The address ALU supports linear, modulo, and reverse-carry arithmetic for all address register indirect modes. These arithmetic types easily allow the creation of data structures in memory for FIFOs (queues), delay lines, circular buffers, stacks, and bit-reversed FFT buffers.

The contents of the address modifier register, Mn, defines the type of arithmetic to be performed for addressing mode calculations. For modulo arithmetic, the contents of Mn also

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

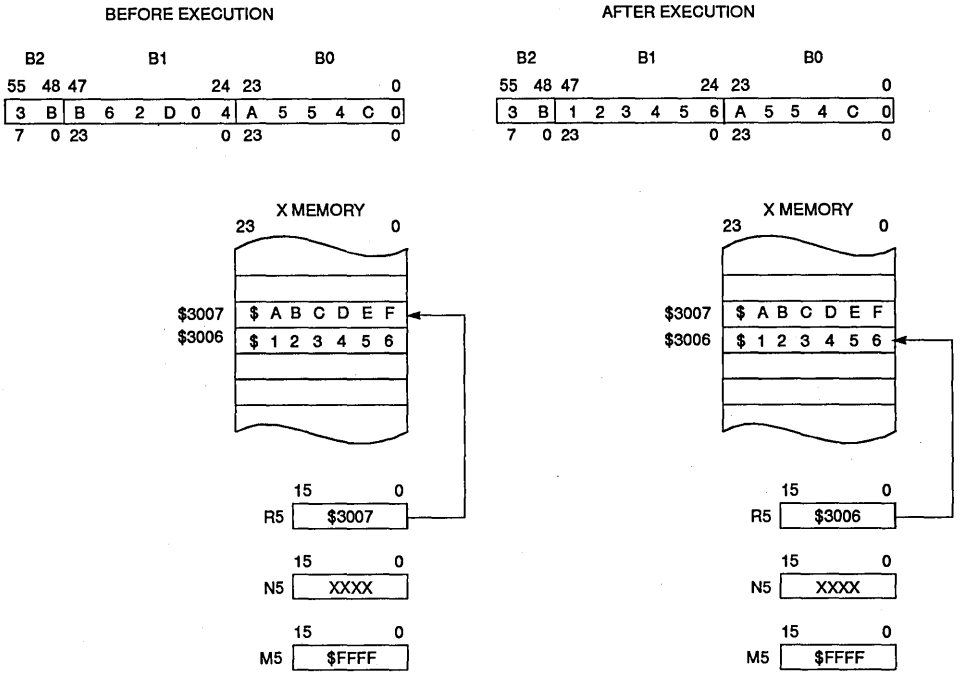


Assembler Syntax: (Rn+Nn)
Memory Spaces: P:, X:, Y:, L:
Additional Instruction Execution Time (Clocks): 2
Additional Effective Address Words: 0

Figure 4-9 Address Register Indirect — Indexed by Offset Nn

specifies the modulus, or the size of the memory buffer whose addresses will be referenced. See Table 4-2 for a summary of the address modifiers implemented on the DSP56K. The MMMM column indicates the hex value which should be stored in the Mn register.

EXAMPLE: MOVE X: -(R5),B1



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

Figure 4-10 Address Register Indirect — Predecrement

4.4.2.1 Linear Modifier (Mn=$FFFF)

When the value in the modifier register is $FFFF, address modification is performed using normal 16-bit linear arithmetic (see Table 4-2). A 16-bit offset, Nn, and + 1 or −1 can be used in the address calculations. The range of values can be considered as signed (Nn from −32,768 to + 32,767) or unsigned (Nn from 0 to + 65,535) since there is no arithmetic difference between these two data representations. Addresses are normally considered unsigned, and data is normally considered signed.

Table 4-2 Address Modifier Summary

MMMM	Addressing Mode Arithmetic
0000	Reverse Carry (Bit Reverse)
0001	Modulo 2
0002	Modulo 3
:	:
7FFE	Modulo 32767
7FFF	Modulo 32768
8000	Reserved
8001	Multiple Wrap-Around Modulo 2
8002	Reserved
8003	Multiple Wrap-Around Modulo 4
:	Reserved
8007	Multiple Wrap-Around Modulo 8
:	Reserved
800F	Multiple Wrap-Around Modulo 2^4
:	Reserved
801F	Multiple Wrap-Around Modulo 2^5
:	Reserved
803F	Multiple Wrap-Around Modulo 2^6
:	Reserved
807F	Multiple Wrap-Around Modulo 2^7
:	Reserved
80FF	Multiple Wrap-Around Modulo 2^8
:	Reserved
81FF	Multiple Wrap-Around Modulo 2^9
:	Reserved
83FF	Multiple Wrap-Around Modulo 2^{10}
:	Reserved
87FF	Multiple Wrap-Around Modulo 2^{11}
:	Reserved
8FFF	Multiple Wrap-Around Modulo 2^{12}
:	Reserved
9FFF	Multiple Wrap-Around Modulo 2^{13}
:	Reserved
BFFF	Multiple Wrap-Around Modulo 2^{14}
:	Reserved
FFFF	Linear (Modulo 2^{15})

4.4.2.2 Modulo Modifier

When the value in the modifier register falls into one of two ranges (Mn=$0001 to $7FFF or Mn= $8001 to $BFFF with the reserved gaps noted in the table), address modification is performed using modulo arithmetic (see Table 4-2).

Modulo arithmetic normally causes the address register value to remain within an address range of size M, whose lower boundary is determined by Rn. The upper boundary is determined by the modulus, or M. The modulus value, in turn, is determined by Mn, the value in the modifier register (see Figure 4-11).

There are certain cases where modulo arithmetic addressing conditions may cause the address register to jump linearly to the same relative address in a different buffer. Other cases firmly restrict the address register to the same buffer, causing the address register to wrap around within the buffer. The range in which the value contained in the modifier register falls determines how the processor will handle modulo addressing.

4.4.2.2.1 Mn=$0001 to $7FFF

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

For example, to create a circular buffer of 21 stages, M is 21, and the lower address boundary must have its five LSBs equal to zero ($2^k \geq 21$, thus $k \geq 5$). The Mn register is loaded with the value 20. The lower boundary may be chosen as 0, 32, 64, 96, 128, 160, etc. The upper boundary of the buffer is then the lower boundary plus 21. There will be an unused space of 11 memory locations between the upper address and next usable lower address. The address pointer is not required to start at the lower address boundary or to end on the upper address boundary; it can initially point anywhere within the defined modulo address range. Neither the lower nor the upper boundary of the modulo region is stored; only the size of the modulo region is stored in Mn. The boundaries are determined by the contents of Rn. Assuming the (Rn)+ indirect addressing mode, if the address register pointer increments past the upper boundary of the buffer (base address plus M–1), it will wrap around through the base address (lower boundary). Alternatively, assuming the (Rn)- indirect addressing mode, if the address decrements past the lower boundary (base address), it will wrap around through the base address plus M–1 (upper boundary).

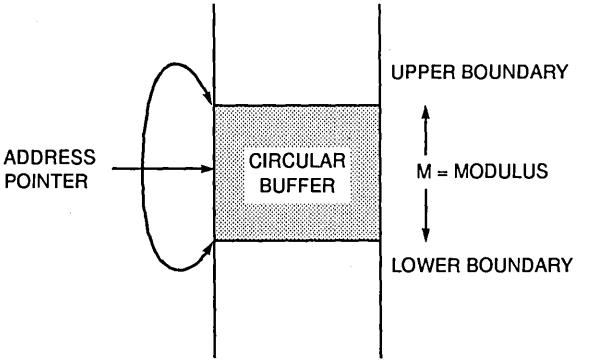


Figure 4-11 Circular Buffer

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 in this range. If Nn>M, the result is data dependent and unpredictable, except for the special case where Nn=P x 2^k, a multiple of the block size where P is a positive integer. For this special case, when using the (Rn)+ Nn addressing mode, the pointer, Rn, will jump linearly to the same relative address in a new buffer, which is P blocks forward in memory (see Figure 4-12).

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

An example of address register indirect modulo addressing is shown in Figure 4-13. Starting at location 64, a circular buffer of 21 stages is created. The addresses generated are offset by 15 locations. The lower boundary = L x (2^k) where $2^k \geq 21$; therefore, k=5 and the lower address boundary must be a multiple of 32. The lower boundary may be chosen as 0, 32, 64, 96, 128, 160, etc. For this example, L is arbitrarily chosen to be 2, making the lower boundary 64. The upper boundary of the buffer is then 84 (the lower boundary

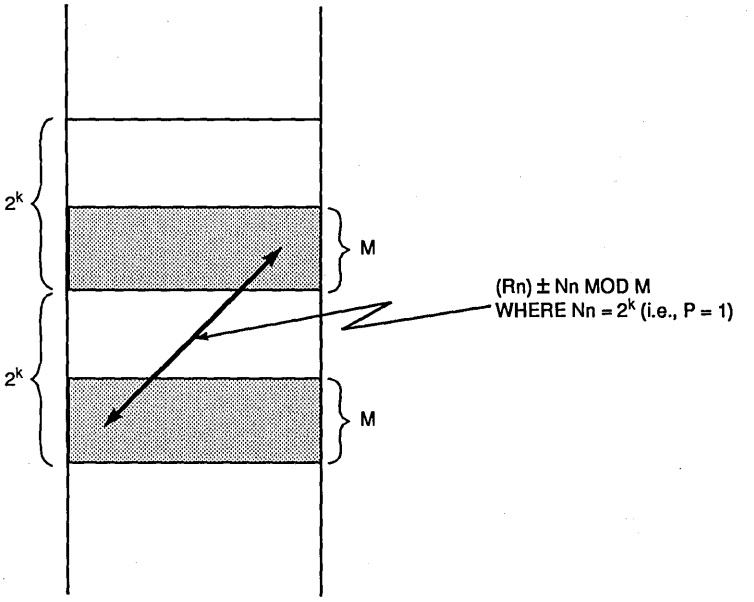


Figure 4-12 Linear Addressing with a Modulo Modifier

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

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

The MOVE instruction in Figure 4-13 takes the contents of the X0 register and moves it to a location in the X memory pointed to by (R2), and then (R2) is updated modulo 21. The

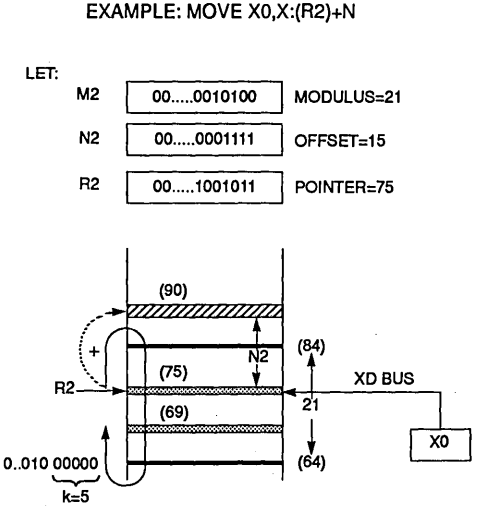


Figure 4-13 Modulo Modifier Example

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

4.4.2.2.2 Mn=$8001 to $BFFF

In this range, the modulo (M) equals (Mn+1)-$8000, where Mn is the value in the modifier register (see Table 4-2). This range firmly restricts the address register to the same buffer, causing the address register to wrap around within the buffer. This multiple wrap-around addressing feature reduces argument overhead and is useful for decimation, interpolation, and waveform generation.

The address modification is performed modulo M, where M may be **any 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 modifier register Mn least significant 14 bits while the two most significant bits are set to '10'. The lower boundary (base address) value must have zeroes in the k LSBs, where 2^k = M, and therefore must be a multiple of 2^k. The upper boundary is the lower boundary plus the modulo size minus one (base address plus M-1).

For example, to create a circular buffer of 32 stages, M is chosen as 32 and the lower address boundary must have its 5 least significant bits equal to zero (2^k = 32, thus k = 5).

The Mn register is loaded with the value $801F. The lower boundary may be chosen as 0, 32, 64, 96, 128, 160, etc. The upper boundary of the buffer is then the lower boundary plus 31.

The address pointer is not required to start at the lower address boundary and may begin anywhere within the defined modulo address range (between the lower and upper boundaries). If the address register pointer increments past the upper boundary of the buffer (base address plus M-1) it will wrap around to the base address. If the address decrements past the lower boundary (base address) it will wrap around to the base address plus M-1. If an offset Nn is used in the address calculations, 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 wrap-around cannot occur with (Rn)+, (Rn)- and -(Rn) addressing modes).

The multiple wrap-around address modifier is useful for decimation, interpolation and waveform generation since the multiple wrap-around capability may be used for argument reduction.

4.4.2.3 Reverse-Carry Modifier (Mn=$0000)

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

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

1. Set Mn=0; this selects reverse-carry arithmetic.
2. Set $Nn=2^{(k-1)}$.
3. Set Rn between the lower boundary and upper boundary in the buffer memory. The lower boundary is L x (2^k), where L is an arbitrary whole number. This

boundary gives a 16-bit binary number "xx . . . xx00 . . . 00", where xx . . . xx=L and 00 . . . 00 equals k zeros. The upper boundary is L x (2^k)+ ((2^k)−1). This boundary gives a 16-bit binary number "xx . . . xx11 . . . 11", where xx . . . xx=L and 11 . . . 11 equals k ones.

4. Use the (Rn)+ Nn addressing mode.

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

Postincrementing by + N generates the address sequence (0, 512, 256, 768, 128, 640,...), which is added to the lower boundary. This sequence (0, 512, etc.) is the scrambled FFT data order for sequential frequency points from 0 to 2π. Table 4-3 shows the successive contents of Rn when using (Rn)+ Nn updates.

Table 4-3 Bit-Reverse Addressing Sequence Example

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

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

The term bit reverse with respect to reverse-carry arithmetic is descriptive. The lower boundary that must be used for the bit-reverse address scheme to work is L x (2^k). In the previous example shown in Table 4-3, L=3 and k=10. The first address used is the lower boundary (3072); the calculation of the next address is shown in Figure 4-14. The k LSBs of the current contents of Rn (3,072) are swapped:

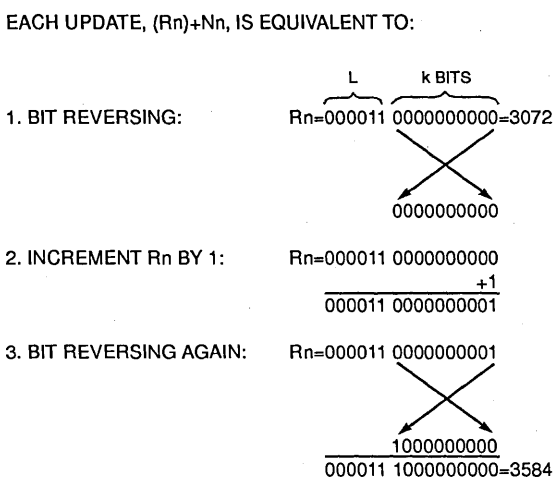


Figure 4-14 Bit-Reverse Address Calculation Example

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

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

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

The result is Rn equals 3,584.

4.4.2.4 Address-Modifier-Type Encoding Summary

There are three address modifier types:

- Linear Addressing
- Reverse-Carry Addressing
- Modulo Addressing

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

Figure 4-15 gives examples of the three addressing modifiers using 8-bit registers for simplification (all AGU registers are 16 bit). The addressing mode used in the example, postincrement by offset Nn, adds the contents of the offset register to the contents of the address register after the address register is accessed. The results of the three examples are as follows:

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

LINEAR ADDRESS MODIFIER

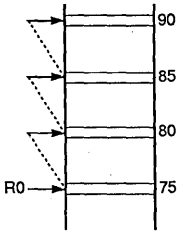
M0 = 255 = 1111 1111 FOR LINEAR ADDRESSING WITH R0

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

POSTINCREMENT BY OFFSET N0: R0 = 80 = 0101 0000

POSTINCREMENT BY OFFSET N0: R0 = 85 = 0101 0101

POSTINCREMENT BY OFFSET N0: R0 = 90 = 0101 1010



4

MODULO ADDRESS MODIFIER

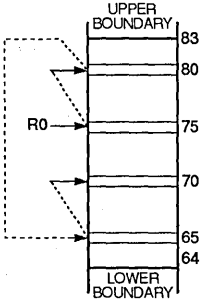
M0 = 19 = 0001 0011 FOR MODULO 20 ADDRESSING WITH R0

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

POSTINCREMENT BY OFFSET N0: R0 = 80 = 0101 0000

POSTINCREMENT BY OFFSET N0: R0 = 65 = 0100 0001

POSTINCREMENT BY OFFSET N0: R0 = 70 = 0100 0110



REVERSE-CARRY ADDRESS MODIFIER

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

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

POSTINCREMENT BY OFFSET N0: R0 = 72 = 0100 1000

POSTINCREMENT BY OFFSET N0: R0 = 68 = 0100 0100

POSTINCREMENT BY OFFSET N0: R0 = 76 = 0100 1100

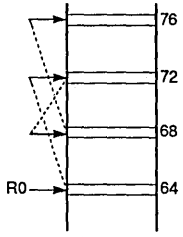
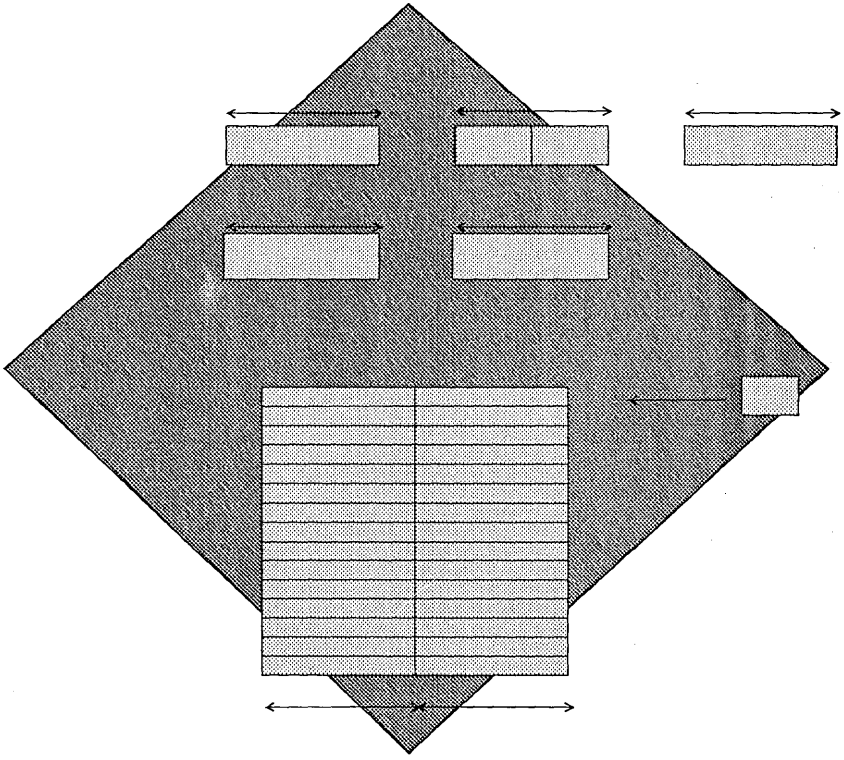


Figure 4-15 Address Modifier Summary

SECTION 5
PROGRAM CONTROL UNIT



SECTION CONTENTS

SECTION 5.1 PROGRAM CONTROL UNIT 3

SECTION 5.2 OVERVIEW 3

SECTION 5.3 PROGRAM CONTROL UNIT (PCU) ARCHITECTURE 5
- 5.3.1 Program Decode Controller 5
- 5.3.2 Program Address Generator (PAG) 5
- 5.3.3 Program Interrupt Controller 6
- 5.3.4 Instruction Pipeline Format 6

SECTION 5.4 PROGRAMMING MODEL 8
- 5.4.1 Program Counter 8
- 5.4.2 Status Register 9
  - 5.4.2.1 Carry (Bit 0) 10
  - 5.4.2.2 Overflow (Bit 1) 10
  - 5.4.2.3 Zero (Bit 2) 10
  - 5.4.2.4 Negative (Bit 3) 10
  - 5.4.2.5 Unnormalized (Bit 4) 10
  - 5.4.2.6 Extension (Bit 5) 11
  - 5.4.2.7 Limit (Bit 6) 11
  - 5.4.2.8 Scaling Bit (Bit 7) 11
  - 5.4.2.9 Interrupt Masks (Bits 8 and 9) 12
  - 5.4.2.10 Scaling Mode (Bits 10 and 11) 12
  - 5.4.2.11 Reserved Status (Bit 12) 13
  - 5.4.2.12 Trace Mode (Bit 13) 13
  - 5.4.2.13 Double Precision Multiply Mode (Bit 14) 13
  - 5.4.2.14 Loop Flag (Bit 15) 13
- 5.4.3 Operating Mode Register 14
- 5.4.4 System Stack 14
- 5.4.5 Stack Pointer Register 15
  - 5.4.5.1 Stack Pointer (Bits 0–3) 16
  - 5.4.5.2 Stack Error Flag (Bit 4) 16
  - 5.4.5.3 Underflow Flag (Bit 5) 16
  - 5.4.5.4 Reserved Stack Pointer Registration (Bits 6–23) 17
- 5.4.6 Loop Address Register 17
- 5.4.7 Loop Counter Register 17
- 5.4.8 Programming Model Summary 17

5.1 PROGRAM CONTROL UNIT

This section describes the hardware of the program control unit (PCU) and concludes with a description of the programming model. The instruction pipeline description is also included since understanding the pipeline is particularly important in understanding the DSP56K family of processors.

5.2 OVERVIEW

The program control unit is one of the three execution units in the central processing module (see Figure 5-2). It performs program address generation (instruction prefetch), instruction decoding, hardware DO loop control, and exception (interrupt) processing. The programmer sees the program control unit as six registers and a hardware system stack (SS) as shown in Figure 5-1. In addition to the standard program flow-control resources, such as a program counter (PC), complete status register (SR), and SS, the program control unit features registers (loop address (LA) and loop counter (LC)) dedicated to supporting the hardware DO loop instruction.

The SS is a 15-level by 32-bit separate internal memory which stores the PC and SR for subroutine calls, long interrupts, and program looping. The SS also stores the LC and LA registers. Each location in the SS is addressable as a 16-bit register, system stack high (SSH) and system stack low (SSL). The stack pointer (SP) points to the SS locations.

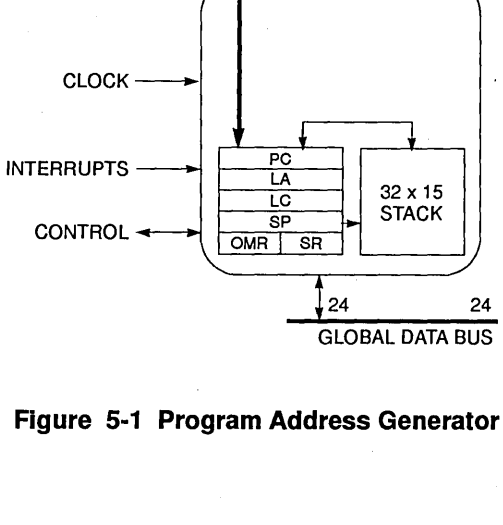


Figure 5-1 Program Address Generator

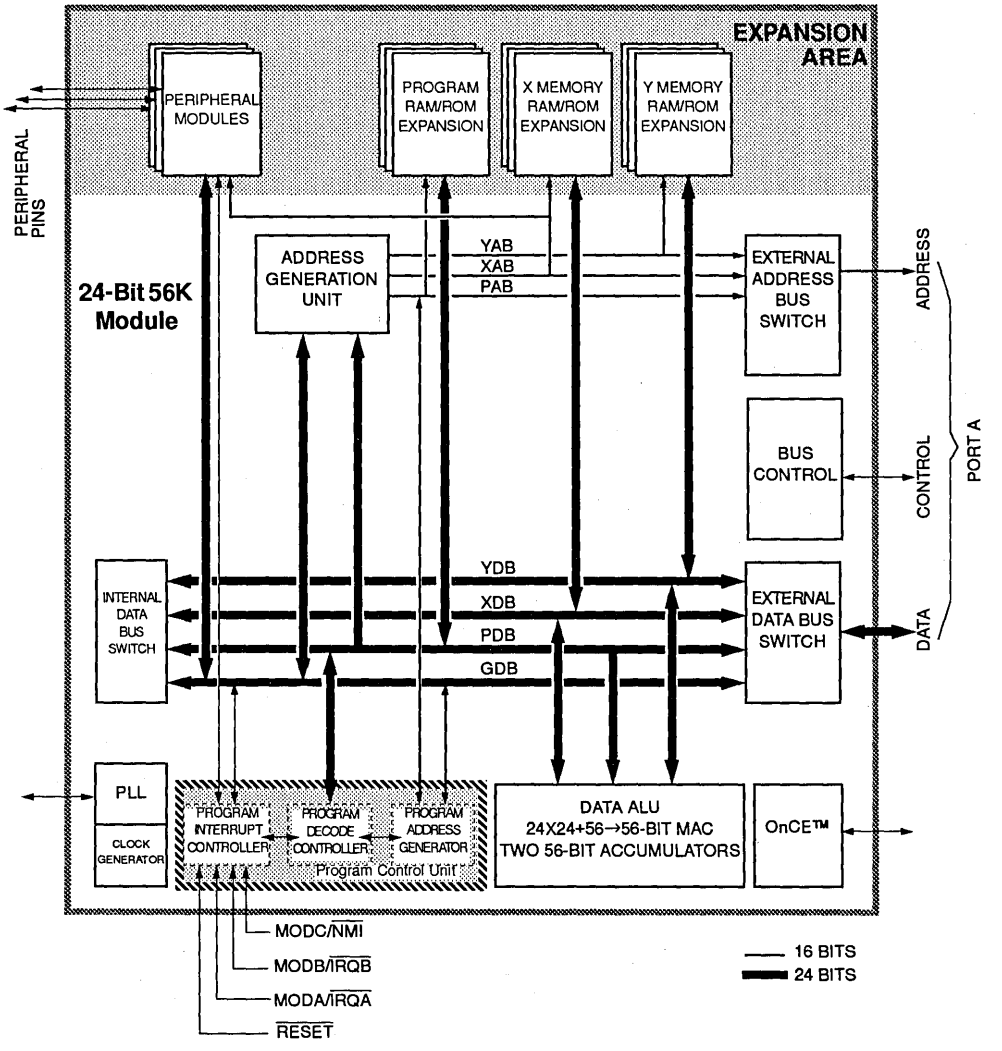


Figure 5-2 DSP56K Block Diagram

All of the PCU registers are read/write to facilitate system debugging. Although none of the registers are 24 bits, they are read or written over 24-bit buses. When they are read, the least significant bits (LSBs) are significant, and the most significant bits (MSBs) are zeroed as appropriate. When they are written, only the appropriate LSBs are significant, and the MSBs are written as don't care.

The program control unit implements a three-stage (prefetch, decode, execute) pipeline and controls the five processing states of the DSP: normal, exception, reset, wait, and stop.

5.3 PROGRAM CONTROL UNIT (PCU) ARCHITECTURE

The PCU consists of three hardware blocks: the program decode controller (PDC), the program address generator (PAG), and the program interrupt controller (PIC).

5.3.1 Program Decode Controller

The PDC contains the program logic array decoders, the register address bus generator, the loop state machine, the repeat state machine, the condition code generator, the interrupt state machine, the instruction latch, and the backup instruction latch. The PDC decodes the 24-bit instruction loaded into the instruction latch and generates all signals necessary for pipeline control. The backup instruction latch stores a duplicate of the prefetched instruction to optimize execution of the repeat (REP) and jump (JMP) instructions.

5.3.2 Program Address Generator (PAG)

The PAG contains the PC, the SP, the SS, the operating mode register (OMR), the SR, the LC register, and the LA register (see Figure 5-1).

The PAG provides hardware dedicated to support loops, which are frequent constructs in DSP algorithms. A DO instruction loads the LC register with the number of times the loop should be executed, loads the LA register with the address of the last instruction word in the loop (fetched during one loop pass), and asserts the loop flag in the SR. The DO instruction also supports nested loops by stacking the contents of the LA, LC, and SR prior to the execution of the instruction. Under control of the PAG, the address of the first instruction in the loop is also stacked so the loop can be repeated with no overhead. While the loop flag in the SR is asserted, the loop state machine (in the PDC) will compare the PC contents to the contents of the LA to determine if the last instruction word in the loop was fetched. If the last word was fetched, the LC contents are tested for one. If LC is not equal to one, then it is decremented, and the SS is read to update the PC with the address of the first instruction in the loop, effectively executing an automatic branch. If the LC is equal to one, then the LC, LA, and the loop flag in the SR are restored with the stack contents, while instruction fetches continue at the incremented PC value (LA + 1). More information about the LA and LC appears in Section 5.3.4 Instruction Pipeline Format.

The repeat (REP) instruction loads the LC with the number of times the next instruction is to be repeated. The instruction to be repeated is only fetched once, so throughput is increased by reducing external bus contention. However, REP instructions are not

interruptible since they are fetched only once. A single-instruction DO loop can be used in place of a REP instruction if interrupts must be allowed.

5.3.3 Program Interrupt Controller

The PIC receives all interrupt requests, arbitrates among them, and generates the interrupt vector address.

Interrupts have a flexible priority structure with levels that can range from zero to three. Levels 0 (lowest level), 1, and 2 are maskable. Level 3 is the highest interrupt priority level (IPL) and is not maskable. Two interrupt mask bits in the SR reflect the current IPL and indicate the level needed for an interrupt source to interrupt the processor. Interrupts cause the DSP to enter the exception processing state which is discussed fully in SECTION 7 – PROCESSING STATES.

The four external interrupt sources include three external interrupt request inputs ($\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, and $\overline{\text{NMI}}$) and the $\overline{\text{RESET}}$ pin. $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ can be either level sensitive or negative edge triggered. The nonmaskable interrupt ($\overline{\text{NMI}}$) is edge sensitive and is a level 3 interrupt. MODA/$\overline{\text{IRQA}}$, MODB/$\overline{\text{IRQB}}$, and MODC/$\overline{\text{NMI}}$ pins are sampled when $\overline{\text{RESET}}$ is deasserted. The sampled values are stored in the operating mode register (OMR) bits MA, MB, and MC, respectively (see Section 5.4.3 for information on the OMR). Only the fourth external interrupt, $\overline{\text{RESET}}$, and Illegal Instruction have higher priority than $\overline{\text{NMI}}$.

The PIC also arbitrates between the different I/O peripherals. The currently selected peripheral supplies the correct vector address to the PIC.

5.3.4 Instruction Pipeline Format

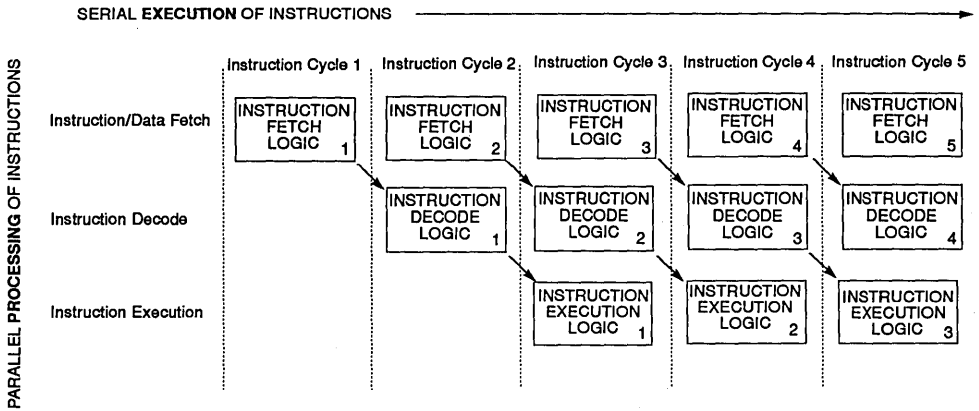
The program control unit uses a three-level pipelined architecture in which concurrent instruction fetch, decode, and execution occur. This pipelined operation remains essentially hidden from the user and makes programming straightforward. The pipeline is illustrated in Figure 5-3, which shows the operations of each of the execution units and all initial conditions necessary to follow the execution of the instruction sequence shown in the figure. The pipeline is described in more detail in Section 7.2.1 Instruction Pipeline.

The first instruction, I1, should be interpreted as follows: multiply the contents of X0 by the contents of Y0, add the product to the contents already in accumulator A, round the result to the "nearest even," store the result back in accumulator A, move the contents in X data memory (pointed to by R0) into X0 and postincrement R0, and move the contents in Y data memory (pointed to by R4) into Y1 and postincrement R4. The second instruction, I2, should be interpreted as follows: clear accumulator A, move the contents in X0 into the location in X data memory pointed to by R0 and postincrement R0. Before the clear oper-

EXAMPLE PROGRAM SEGMENT

```
Instruction 1   MACR    X0,Y1,A    X:(R0)+,X0    Y:(R4)+,Y1
Instruction 2   CLR     A          X0,X:(R0)+    A,Y:(R4)-
Instruction 3   MAC     X0,Y1,A    X:(R0)+,X0    Y:(R4)+,Y1
```

SEQUENCE OF OPERATIONS



EXECUTION OF EXAMPLE PROGRAM

		Instruction Cycle 1	Instruction Cycle 2	Instruction Cycle 3	Instruction Cycle 4	Instruction Cycle 5
INSTRUCTION FETCH → INSTRUCTION DECODE → INSTRUCTION EXECUTION →		I1	I2 I1	I3 I2 I1	I4 I3 I2	I5 I4 I3
PARALLEL OPERATIONS	INITIAL CONDITIONS					
ADDRESS UPDATE (AGU)	R0=$0005 R4=$0008			R0=5+1 R4=8+1	R0=6+1 R4=9-1	R0=7+1 R4=8+1
INSTRUCTION EXECUTION	A: A2=$00 A1=$000066 A0=$000000			A: A2=$00 A1=$0000A2 A0=$000000	A: A2=$00 A1=$000000 A0=$000000	A: A2=$00 A1=$000000 A0=$000050
(DATA ALU)	X0=$400000 Y1=$000077			X0=$000005 Y1=$000008	X0=$000005 Y1=$000008	X0=$000007 Y1=$000008
X MEMORY AT ADDRESS $0005 $0006 $0007	DATA $000005 $000006 $000007			$000005 $000006 $000007	$000005 $000005 $000007	$000005 $000005 $000007
Y MEMORY AT ADDRESS $0008 $0009	DATA $000008 $000009			$000008 $000009	$000008 $0000A2	$000008 $0000A2

Figure 5-3 Three-Stage Pipeline

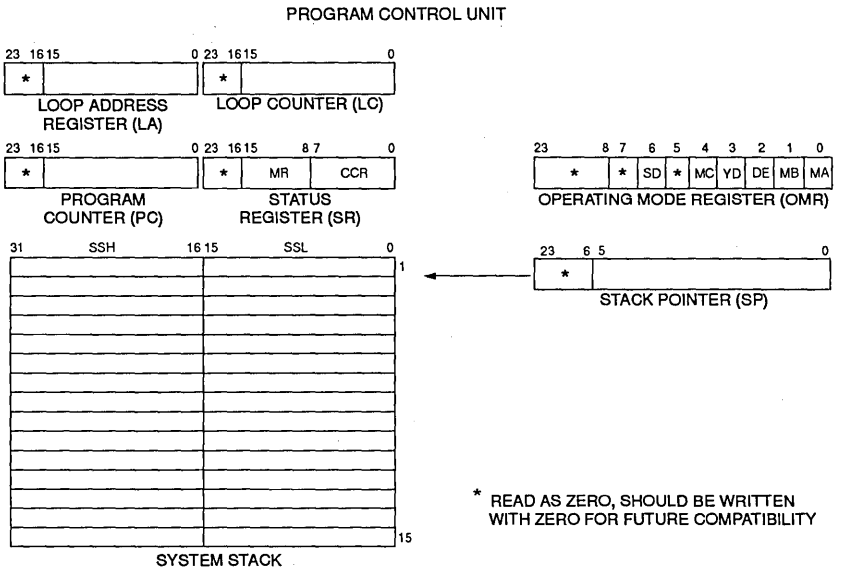


Figure 5-4 Program Control Unit Programming Model

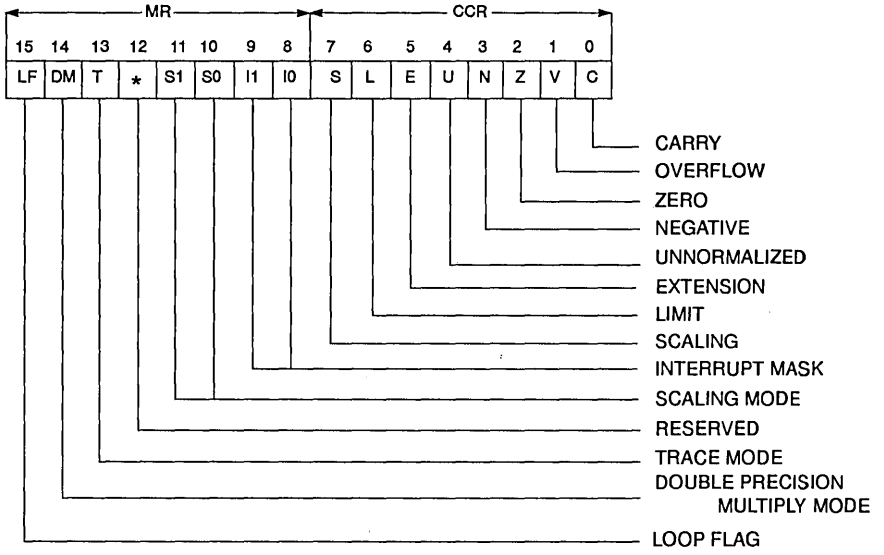
ation, move the contents in accumulator A into the location in Y data memory pointed to by R4 and postdecrement R4. The third instruction, I3, is the same as I1, except the rounding operation is not performed.

5.4 PROGRAMMING MODEL

The program control unit features LA and LC registers which support the DO loop instruction and the standard program flow-control resources, such as a PC, complete SR, and SS. With the exception of the PC, all registers are read/write to facilitate system debugging. Figure 5-4 shows the program control unit programming model with the six registers and SS. The following paragraphs give a detailed description of each register.

5.4.1 Program Counter

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



All bits are cleared after hardware reset except bits 8 and 9 which are set to ones.
Bits 12 and 16 to 23 are reserved, read as zero and should be written with zero for future compatibility

Figure 5-5 Status Register Format

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

5.4.2 Status Register

The 16-bit SR consists of a mode register (MR) in the high-order eight bits and a condition code register (CCR) in the low-order eight bits, as shown in Figure 5-5. 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 MR is a special purpose control register which defines the current system state of the processor. The MR bits are affected by processor reset, exception processing, the DO, end current DO loop (ENDDO), return from interrupt (RTI), and SWI instructions and by instructions that directly reference the MR register, such as OR immediate to control register (ORI) and AND immediate to control register (ANDI). During processor reset, the interrupt mask bits of the MR will be set. The scaling mode bits, loop flag, and trace bit will be cleared.

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

5.4.2.1 Carry (Bit 0)

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

5.4.2.2 Overflow (Bit 1)

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

5.4.2.3 Zero (Bit 2)

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

5.4.2.4 Negative (Bit 3)

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

5.4.2.5 Unnormalized (Bit 4)

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

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

5.4.2.6 Extension (Bit 5)

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

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

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

5.4.2.7 Limit (Bit 6)

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

5.4.2.8 Scaling Bit (Bit 7)

The scaling bit (S) is used to detect data growth, which is required in Block Floating Point FFT operation. Typically, the bit is tested after each pass of a radix 2 FFT and, if it is set, the 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." This bit is computed according to the following logical equations when the result of accumulator A or B is moved to XDB or YDB. It is a "sticky" bit, cleared only by an instruction that specifically clears it.

If S1=0 and S0=0 (no scaling)
then S = (A46 XOR A45) OR (B46 XOR B45)

If S1=0 and S0=1 (scale down)
then S = (A47 XOR A46) OR (B47 XOR B46)

If S1=1 and S0=0 (scale up)
then S = (A45 XOR A44) OR (B45 XOR B44)

If S1=1 and S0=1 (reserved)
then the S flag is undefined.

where Ai and Bi means bit i in accumulator A or B.

5.4.2.9 Interrupt Masks (Bits 8 and 9)

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

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

5.4.2.10 Scaling Mode (Bits 10 and 11)

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

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

The scaling mode affects data read from the A or B accumulator registers out to the XDB and YDB. Different scaling modes can occur with the same program code to allow dynamic scaling. Dynamic scaling facilitates 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.

5.4.2.11 Reserved Status (Bit 12)

This bits is reserved for future expansion and will read as zero during DSP read operations.

5.4.2.12 Trace Mode (Bit 13)

The trace mode (T) bit specifies the tracing function of the DSP56000/56001 *only*. (With other members of the DSP56K family, use the OnCE trace mode described in Section 10.5.) For the DSP56000/56001, if the T bit is set at the beginning of any instruction execution, a trace exception will be generated after the instruction execution is completed. If the T bit is cleared, tracing is disabled and instruction execution proceeds normally. If a long interrupt is executed during a trace exception, the SR with the trace bit set will be stacked, and the trace bit in the SR is cleared (see SECTION 7 – PROCESSING STATES for a complete description of a long interrupt operation). The T bit is also cleared during processor reset.

5.4.2.13 Double Precision Multiply Mode (Bit 14)

The processor is in double precision multiply mode when this bit is set. (See Section 3.4 for detailed information on the double precision multiply mode.) When the DM bit is set, the operations performed by the MPY and MAC instructions change so that a double precision 48-bit by 48-bit double precision multiplication can be performed in six instructions. The DSP56K software simulator accurately shows how the MPY, MAC, and other Data ALU instructions operate while the processor is in the double precision multiply mode.

5.4.2.14 Loop Flag (Bit 15)

The loop flag (LF) bit is set when a program loop is in progress. It detects the end of a program loop. The LF is the only SR bit that is restored when a program loop is terminated. Stacking and restoring the LF when initiating and exiting a program loop, respectively, allow the nesting of program loops. At the start of a long interrupt service routine, the SR (including the LF) is pushed on the SS and the SR LF is cleared. When returning from the long interrupt with an RTI instruction, the SS is pulled and the LF is restored. During a processor reset, the LF is cleared.

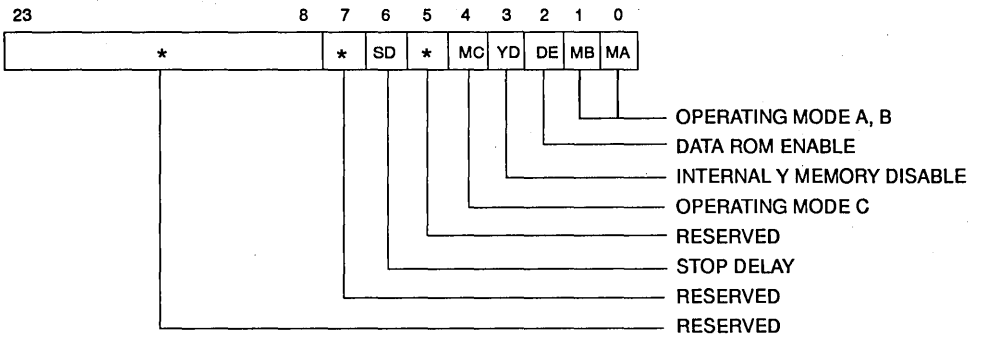


Figure 5-6 OMR Format

5.4.3 Operating Mode Register

The OMR is a 24-bit register (only six bits are defined) that sets the current operating mode of the processor. Each chip in the DSP56K family of processors has its own set of operating modes which determine the memory maps for program and data memories, and the startup procedure that occurs when the chip leaves the reset state. The OMR bits are only affected by processor reset and by the ANDI, ORI, and MOVEC instructions, which directly reference the OMR.

The OMR format with all of its defined bits is shown in Figure 5-6. For product-specific OMR bit definitions, see the individual chip's user manual for details on its respective operating modes.

5.4.4 System Stack

The SS is a separate 15X32-bit internal memory divided into two banks, the SSH and the SSL, each 16 bits wide. The SSH stores the PC contents, and the SSL stores the SR contents for subroutine calls, long interrupts, and program looping. The SS will also store the LA and LC registers. The SS is in stack memory space; its address is always inherent and implied by the current instruction.

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

The SS is also used to implement no-overhead nested hardware DO loops. When the DO instruction is executed, the LA:LC are pushed on the SS, then the PC:SR are pushed on

the SS. Since each SS location can be addressed as separate 16-bit registers (SSH and SSL), software stacks can be created for unlimited nesting.

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

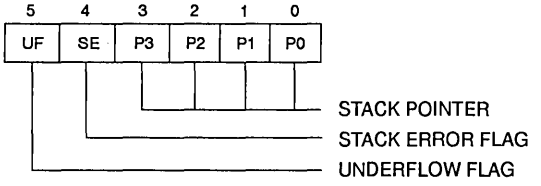


Figure 5-7 Stack Pointer Register Format

5.4.5 Stack Pointer Register

The 6-bit SP register indicates the location of the top of the SS and the status of the SS (underflow, empty, full, and overflow). The SP register is referenced implicitly by some instructions (DO, REP, JSR, RTI, etc.) or directly by the MOVEC instruction. The SP register format is shown in Figure 5-7. The SP register works as a 6-bit counter that addresses (selects) a 15-location stack with its four LSBs. The possible SP values are shown in Figure 5-8 and described in the following paragraphs.

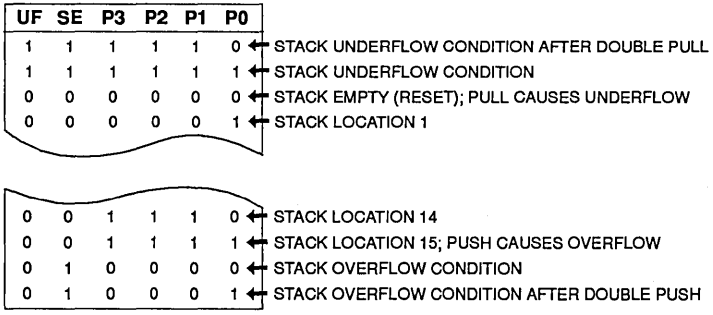


Figure 5-8 SP Register Values

5.4.5.1 Stack Pointer (Bits 0–3)

The SP points to the last location used 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 to which the SP points. An item is pulled off the stack by copying it from that location and then by decrementing the SP.

5.4.5.2 Stack Error Flag (Bit 4)

The stack error flag indicates that a stack error has occurred, and the transition of the stack error flag from zero to one causes a priority level-3 stack error exception.

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

Any implied pull operation with SP equal to zero will cause a stack error exception, and the SP will read 111111 (or 111110 if an implied double pull occurs).

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

Note: When SP is zero (stack empty), instructions that read the stack without SP post-decrement and instructions that write to the stack without SP preincrement do not cause a stack error exception (i.e., 1) DO SSL,xxxx 2) REP SSL 3) MOVEC or move peripheral data (MOVEP) when SSL is specified as a source or destination).

5.4.5.3 Underflow Flag (Bit 5)

The underflow flag is set when a stack underflow occurs. The underflow flag is a "sticky bit" when the stack error flag is set. That is, when the stack error flag is set, the underflow flag will not change state. The combination of "underflow=1" and "stack error=0" is an illegal combination and will not occur unless it is forced by the user. If this condition is forced by the user, the hardware will correct itself based on the result of the next stack operation.

5.4.5.4 Reserved Stack Pointer Registration (Bits 6–23)

SP register bits 6 through 23 are reserved for future expansion and will read as zero during read operations.

5.4.6 Loop Address Register

The LA is a read/write register which is stacked into the SSH by a DO instruction and is unstacked by end-of-loop processing or by an ENDDO instruction. The contents of the LA register indicate the location of the last instruction word in a program loop. When that last instruction is fetched, the processor checks the contents of the LC register (see the following section). If the contents are not one, the processor decrements the LC and takes the next instruction from the top of the SS. If the LC is one, the PC is incremented, the loop flag is restored (pulled from the SS), the SS is purged, the LA and LC registers are pulled from the SS and restored, and instruction execution continues normally.

5.4.7 Loop Counter Register

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

5.4.8 Programming Model Summary

The complete programming model for the DSP56K central processing module is shown in Figure 5-9. Programming models for the peripherals are shown in the appropriate user manuals.

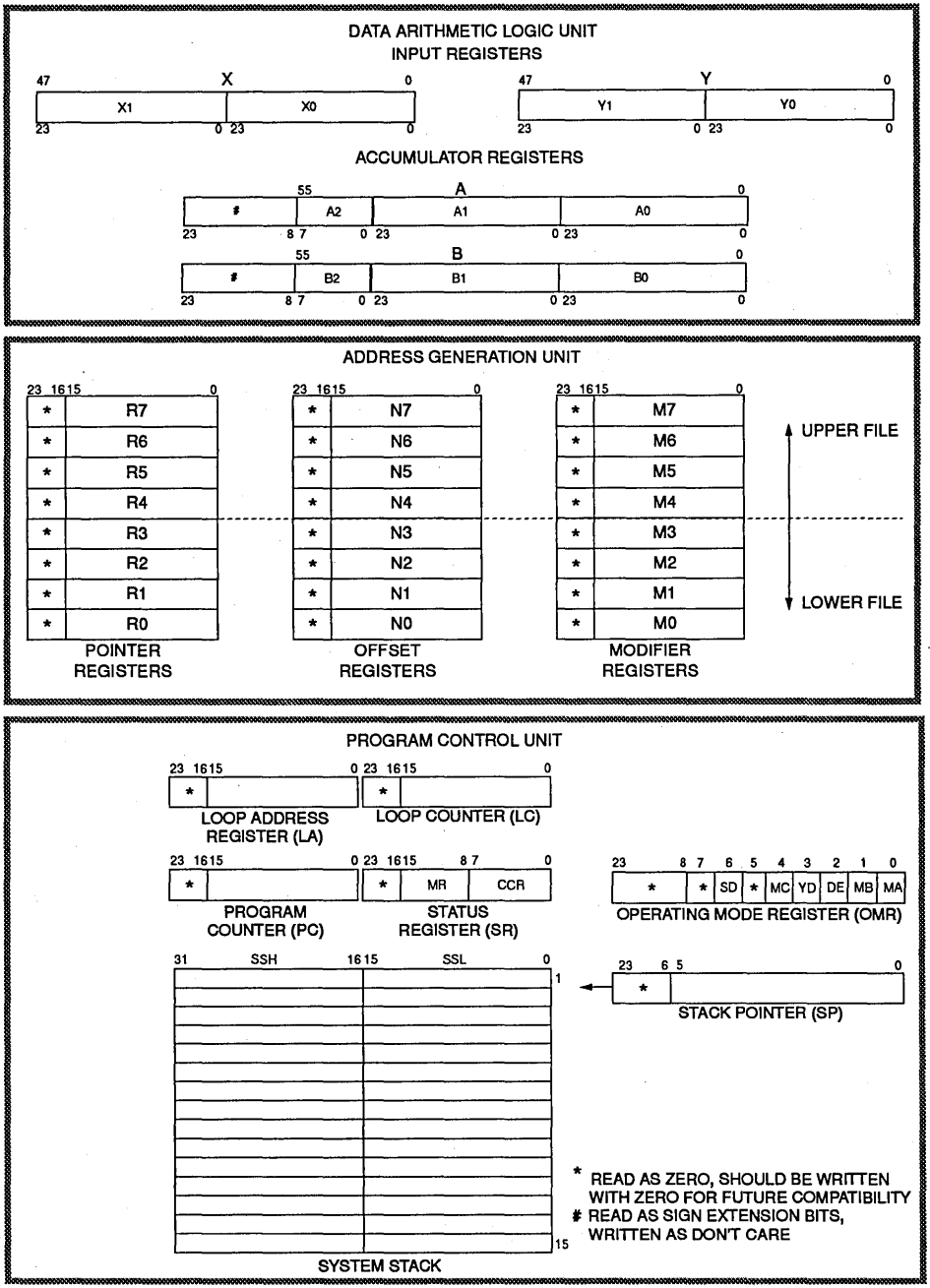
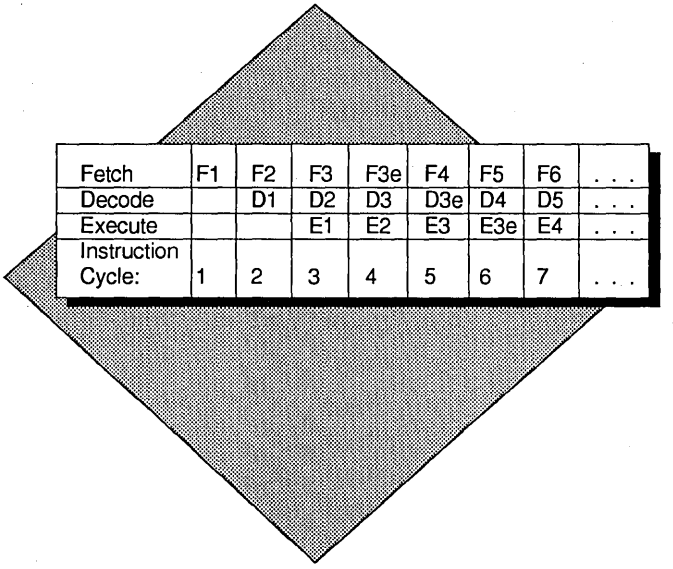


Figure 5-9 DSP56K Central Processing Module Programming Model

SECTION 6
INSTRUCTION SET INTRODUCTION



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

SECTION CONTENTS

SECTION 6.1 INSTRUCTION SET INTRODUCTION 3

SECTION 6.2 SYNTAX 3

SECTION 6.3 INSTRUCTION FORMATS 3
- 6.3.1 Operand Sizes 5
- 6.3.2 Data Organization in Registers 6
  - 6.3.2.1 Data ALU Registers 6
  - 6.3.2.2 AGU Registers 7
  - 6.3.2.3 Program Control Registers 8
- 6.3.3 Data Organization in Memory 9
- 6.3.4 Operand References 11
  - 6.3.4.1 Program References 11
  - 6.3.4.2 Stack References 11
  - 6.3.4.3 Register References 11
  - 6.3.4.4 Memory References 11
    - 6.3.4.4.1 X Memory References 11
    - 6.3.4.4.2 Y Memory References 12
    - 6.3.4.4.3 L Memory References 12
    - 6.3.4.4.4 YX Memory References 12
- 6.3.5 Addressing Modes 12
  - 6.3.5.1 Register Direct Modes 13
    - 6.3.5.1.1 Data or Control Register Direct 13
    - 6.3.5.1.2 Address Register Direct 13
  - 6.3.5.2 Address Register Indirect Modes 13
  - 6.3.5.3 Special Addressing Modes 14
    - 6.3.5.3.1 Immediate Data 14
    - 6.3.5.3.2 Absolute Address 14
    - 6.3.5.3.3 Immediate Short 14
    - 6.3.5.3.4 Short Jump Address 14
    - 6.3.5.3.5 Absolute Short 14
    - 6.3.5.3.6 I/O Short 16
    - 6.3.5.3.7 Implicit Reference 16
  - 6.3.5.4 Addressing Modes Summary 20

SECTION 6.4 INSTRUCTION GROUPS 20
- 6.4.1 Arithmetic Instructions 22
- 6.4.2 Logical Instructions 23
- 6.4.3 Bit Manipulation Instructions 24
- 6.4.4 Loop Instructions 24
- 6.4.5 Move Instructions 26
- 6.4.6 Program Control Instructions 27

6.1 INSTRUCTION SET INTRODUCTION

The programming model shown in Figure 6-1 suggests that the DSP56K central processing module architecture can be viewed as three functional units which operate in parallel: data arithmetic logic unit (data ALU), address generation unit (AGU), and program control unit (PCU). The instruction set keeps each of these units busy throughout each instruction cycle, achieving maximal speed and maintaining minimal program size.

This section introduces the DSP56K instruction set and instruction format. The complete range of instruction capabilities combined with the flexible addressing modes used in this processor provide a very powerful assembly language for implementing digital signal processing (DSP) algorithms. The instruction set has been designed to allow efficient coding for DSP high-level language compilers such as the C compiler. Execution time is minimized by the hardware looping capabilities, use of an instruction pipeline, and parallel moves.

6.2 SYNTAX

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

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

The opcode column indicates the data ALU, AGU, or program control unit operation to be performed and must always be included in the source code. The operands column specifies the operands to be used by the opcode. The XDB and YDB columns specify optional data transfers over the XDB and/or YDB and the associated addressing modes. The address space qualifiers (X:, Y:, and L:) indicate which address space is being referenced. Parallel moves are allowed in 30 of the 62 instructions. Additional information is presented in APPENDIX A - INSTRUCTION SET DETAILS.

6.3 INSTRUCTION FORMATS

The DSP56K instructions consist of one or two 24-bit words – an operation word and an optional effective address extension word. The general format of the operation word is

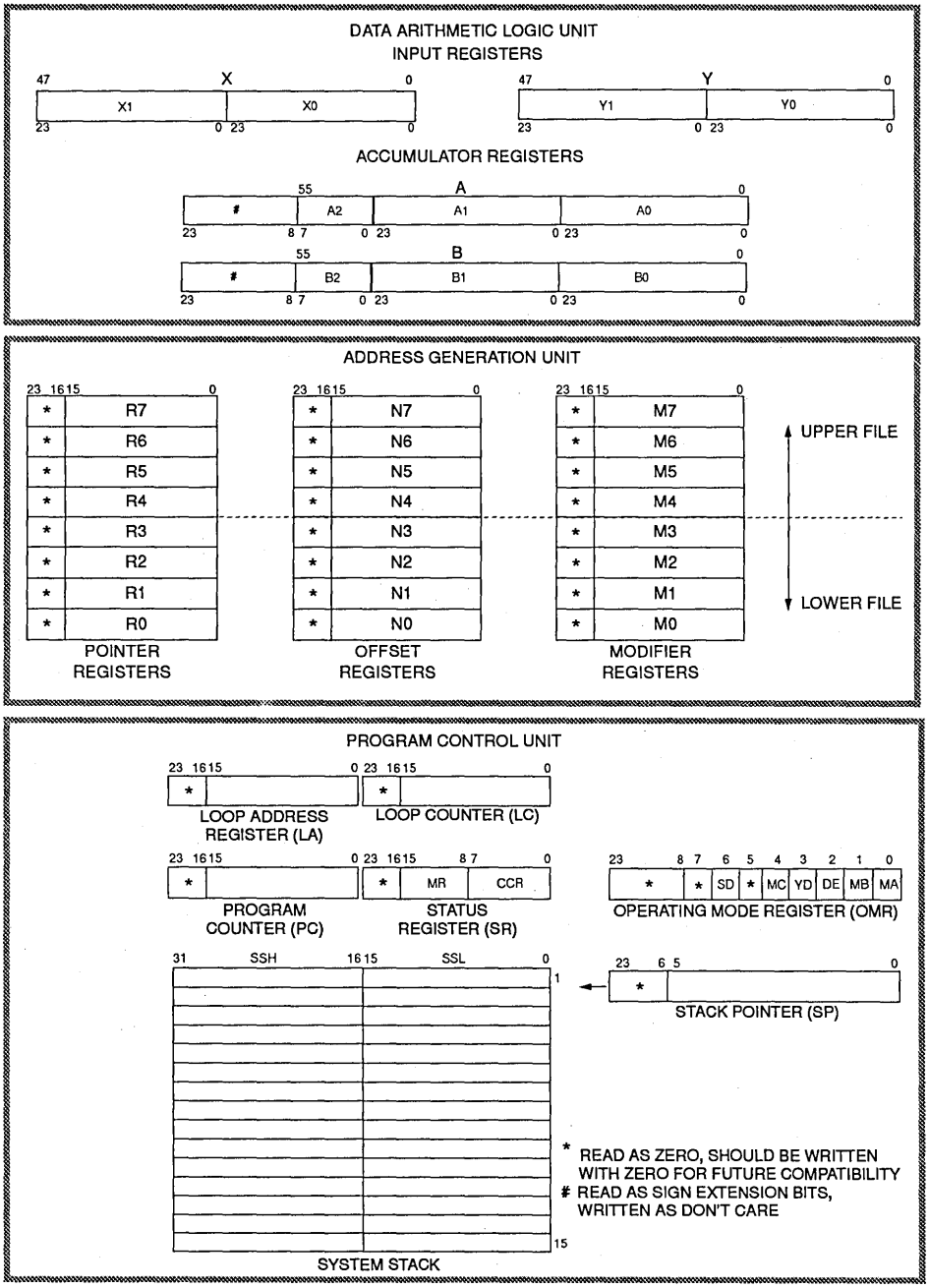


Figure 6-1 DSP56K Central Processing Module Programming Model

shown in Figure 6-2. Most instructions specify data movement on the XDB, YDB, and data ALU operations in the same operation word. The DSP56K performs each of these operations in parallel.

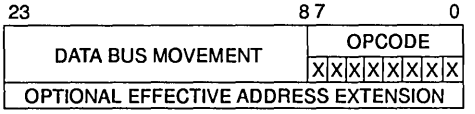


Figure 6-2 General Format of an Instruction Operation Word

The data bus movement field provides the operand reference type. It selects the type of memory or register reference to be made, the direction of transfer, and the effective address(es) for data movement on the XDB and YDB. This field may require additional information to fully specify the operand for certain addressing modes. An effective address extension word following the operation word provides an immediate data address or an absolute address if required (see Section 6.3.5.3 for the description of special addressing modes). Examples of operations that may include the extension word include the move operations X:, X:R, Y:, R:Y, and L:. Additional information is presented in APPENDIX A - INSTRUCTION SET DETAILS.

The opcode field of the operation word specifies the data ALU operation or the program control unit operation to be performed, and any additional operands required by the instruction. Only those data ALU and program control unit operations that can accompany data bus movement will be specified in the opcode field of the instruction. Other data ALU, program control unit, and all address ALU operations will be specified in an instruction word with a different format. These formats include operation words which contain short immediate data or short absolute addresses (see Section 6.3.5.3 for the description of special addressing modes).

6.3.1 Operand Sizes

Operand sizes are defined as follows: a byte is 8 bits long, a short word is16 bits long, a word is 24 bits long, a long word is 48 bits long, and an accumulator is 56 bits long (see Figure 6-3). The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Implicit instructions support some subset of the five sizes shown in Figure 6-3.

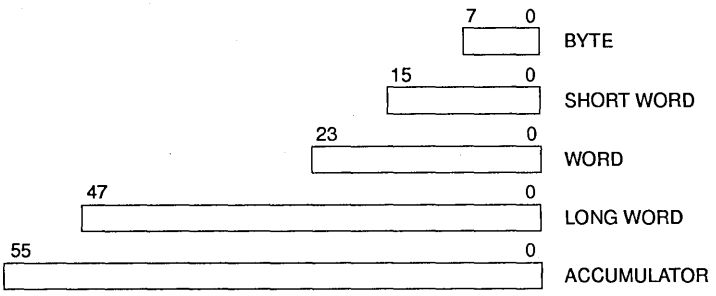


Figure 6-3 Operand Sizes

6.3.2 Data Organization in Registers

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

6.3.2.1 Data ALU Registers

The eight main data ALU registers are 24 bits wide. Word operands occupy one register; long-word operands occupy two concatenated registers. The least significant bit (LSB) is the right-most bit (bit 0) and the most significant bit (MSB) is the left-most bit (bit 23 for word operands and bit 47 for long-word operands). The two accumulator extension registers are eight bits wide.

When an accumulator extension register acts as a source operand, it occupies the low-order portion (bits 0–7) of the word and the high-order portion (bits 8–23) is sign extended (see Figure 6-4). When used as a destination operand, this register receives the low-order portion of the word, and the high-order portion is not used. Accumulator operands occupy an entire group of three registers (i.e., A2:A1:A0 or B2:B1:B0). The LSB is the right-most bit (bit 0), and the MSB is the left-most bit (bit 55).

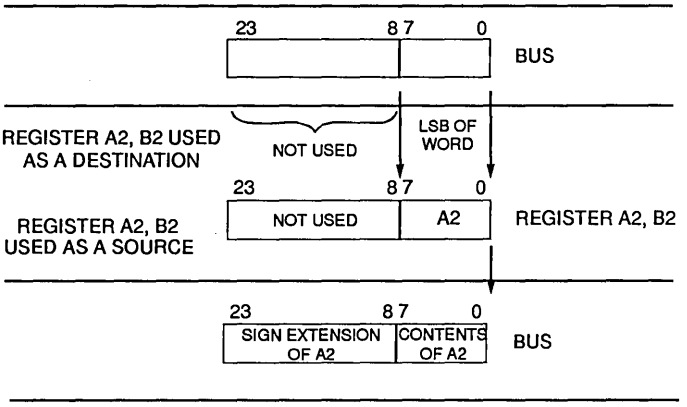


Figure 6-4 Reading and Writing the ALU Extension Registers

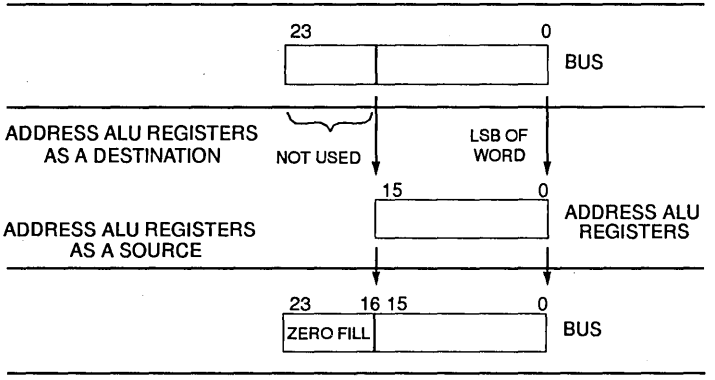


Figure 6-5 Reading and Writing the Address ALU Registers

6.3.2.2 AGU Registers

The 24 AGU registers are 16 bits wide. They may be accessed as word operands for address, address modifier, and data storage. When used as a source operand, these registers occupy the low-order portion of the 24-bit word; the high-order portion is read as zeros (see Figure 6-5). When used as a destination operand, these registers receive the low-order portion of the word; the high-order portion is not used. The notation "Rn" designates one of the eight address registers, R0–R7; the notation "Nn" designates one of the eight address offset registers, N0–N7; and the notation "Mn" designates one of the eight address modifier registers, M0–M7.

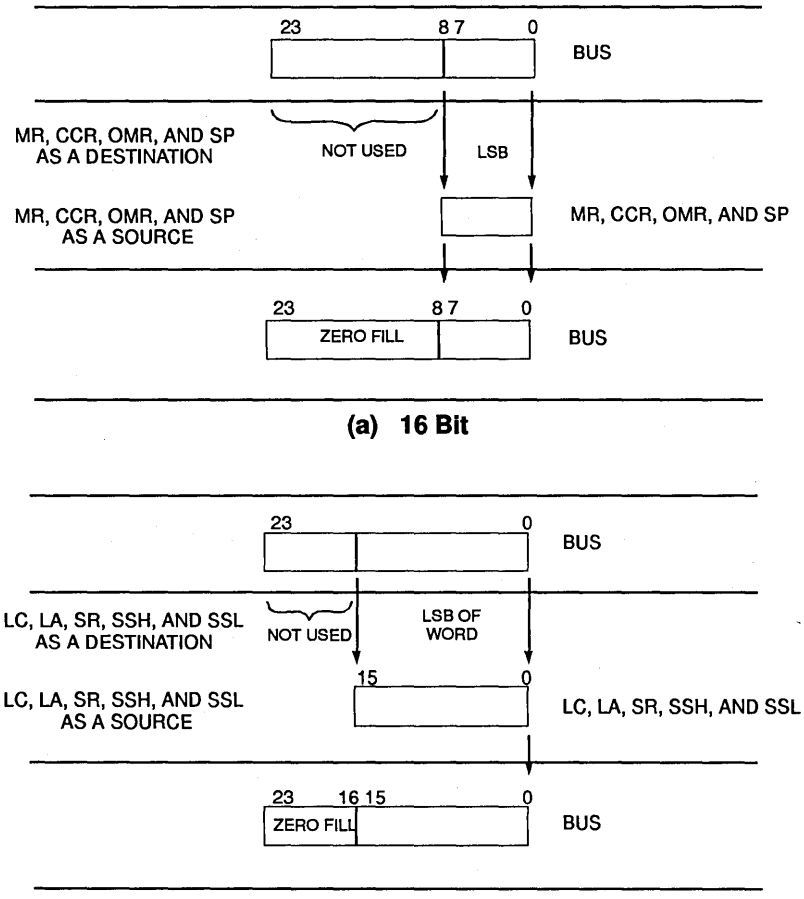


Figure 6-6 Reading and Writing Control Registers

6.3.2.3 Program Control Registers

The 8-bit operating mode register (OMR) may be accessed as a word operand. However, not all eight bits are defined, and those that are defined will vary depending on the DSP56K family member. In general, undefined bits are written as "don't care" and read as zero.

The 16-bit SR has the system mode register (MR) occupying the high-order eight bits and the user condition code register (CCR) occupying the low-order eight bits. The SR may be accessed as a word operand.

The MR and CCR may be accessed individually as word operands (see Figure 6-6(b)). The LC, LA, system stack high (SSH), and system stack low (SSL) registers are 16 bits wide and may be accessed as word operands (see Figure 6-6(a)). When used as a source operand, these registers occupy the low-order portion of the 24-bit word; the high-order portion is zero. When used as a destination operand, they receive the low-order portion of the 24-bit word; the high-order portion is not used. The system stack pointer (SP) is a 6-bit register that may be accessed as a word operand.

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

6.3.3 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 24-bit-wide X and Y memories can store word, short-word, and byte operands. Short-word and byte operands, which usually occupy the low-order portion of the X or Y memory word, are either zero extended or sign extended on the XDB or YDB.

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

Data ALU

Xn	Input Registers X1, X0 (24 Bits)
Yn	Input Registers Y1, Y0 (24 Bits)
An	Accumulator Registers A2 (8 Bits), A1, A0 (24 Bits)
Bn	Accumulator Registers B2 (8 Bits), B1, B0 (24 Bits)
X	Input Register X (X1:X0, 48 Bits)
Y	Input Register Y (Y1:Y0, 48 Bits)
A	Accumulator A (A2:A1:A0, 56 Bits)*
B	Accumulator B (B2:B1:B0, 56 Bits)*
AB	Accumulators A and B (A1:B1, 48 Bits)*
BA	Accumulators B and A (B1:A1, 48 Bits)*
A10	Accumulator A (A1:A0, 48 Bits)
B10	Accumulator B (B1:B0, 48 Bits)

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

Address ALU

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

Program Control Unit

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

Addresses

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

Miscellaneous

#xx	Immediate Short Data (8 Bits)
#xxx	Immediate Short Data (12 Bits)
#xxxxxx	Immediate Data (24 Bits)
#n	Immediate Short Data (5 Bits)
S,Sn	Source Operand Register
D,Dn	Destination Operand Register
D[n]	Bit n of D Affected
r	Rounding Constant
I1,I0	Interrupt Priority Level in SR
LF	Loop Flag in SR

6.3.4 Operand References

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

6.3.4.1 Program References

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

6.3.4.2 Stack References

Stack (S) references, which are references to the System Stack (SS), a separate 32-bit-wide internal memory space, are used implicitly to store the PC and SR for subroutine calls, interrupts, and returns. In addition to the PC and SR, the LA and LC registers are stored on the stack when a program loop is initiated. S references are always implied by the instruction. Data is written to the stack memory to save the processor state and is read from the stack memory to restore the processor state. In contrast to S references, references to SSL and SSH are always explicit.

6.3.4.3 Register References

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

6.3.4.4 Memory References

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

6.3.4.4.1 X Memory References

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

6.3.4.4.2 Y Memory References

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

6.3.4.4.3 L Memory References

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

6.3.4.4.4 YX Memory References

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

6.3.5 Addressing Modes

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

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

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

The addressing modes are grouped into three categories: register direct, address register indirect, and special. These addressing modes are described in the following paragraphs. Refer to Table 6-1 for a summary of the addressing modes and allowed operand references.

6.3.5.1 Register Direct Modes

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

6.3.5.1.1 Data or Control Register Direct

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

6.3.5.1.2 Address Register Direct

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

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

6.3.5.2 Address Register Indirect Modes

The address register indirect mode description is presented in SECTION 4 - ADDRESS GENERATION UNIT.

6.3.5.3 Special Addressing Modes

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

6.3.5.3.1 Immediate Data

Classified as a program reference, this addressing mode requires one word of instruction extension containing the immediate data. Figure 6-7 shows three examples. Example A moves immediate data to register A0 without affecting A1 or A2. Examples B and C zero fill register A0 and sign extend register A2.

6.3.5.3.2 Absolute Address

This addressing mode requires one word of instruction extension containing the absolute address. Figure 6-8 shows that MOVE Y:$5432,B0 copies the contents of address $5432 into B0 without changing memory location $5432, register B1, or register B2. This addressing mode is classified as both a memory reference and program reference. The 16-bit absolute address is stored in the 16 LSBs of the extension word; the eight MSBs are zero filled.

6.3.5.3.3 Immediate Short

The 8- or 12-bit operand, which is in the instruction operation word, is classified as a program reference. The immediate data is interpreted as an unsigned integer (low-order portion) or signed fraction (high-order portion), depending on the destination register. Figure 6-9 shows the use of immediate short addressing in four examples.

6.3.5.3.4 Short Jump Address

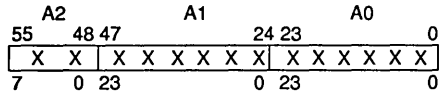
The operand occupies 12 bits in the instruction operation word, which allows addresses $0000–$0FFF to be accessed (see Figure 6-10). The address is zero extended to 16 bits when used to address program memory. This addressing mode is classified as a program reference.

6.3.5.3.5 Absolute Short

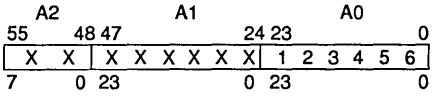
The address of the operand occupies six bits in the instruction operation word, allowing addresses $0000–$003F to be accessed (see Figure 6-11). Classified as both a memory reference and program reference, the address is zero extended to 16 bits when used to address an operand or program memory.

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

BEFORE EXECUTION

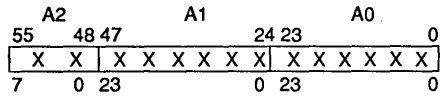


AFTER EXECUTION

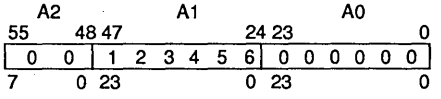


EXAMPLE B:POSITIVE IMMEDIATE INTO 56-BIT REGISTER
(MOVE #$123456,A)

BEFORE EXECUTION

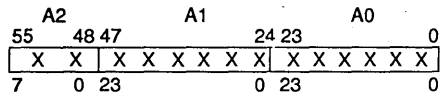


AFTER EXECUTION

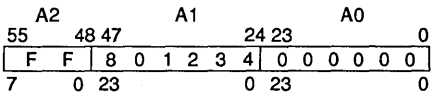


EXAMPLE C: NEGATIVE IMMEDIATE INTO 56-BIT REGISTER
(MOVE #$801234,A)

BEFORE EXECUTION



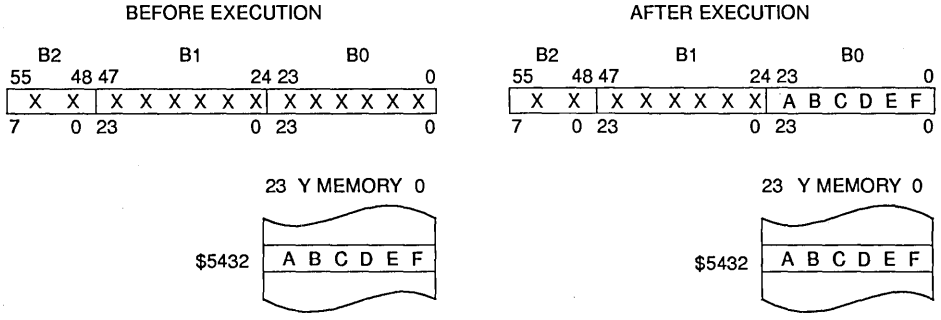
AFTER EXECUTION



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

Figure 6-7 Special Addressing – Immediate Data

EXAMPLE: MOVE Y:$5432,B0



Assembler Syntax: XXXX or aa
Memory Spaces: P:
Additional Instruction Execution Time (Clocks): 2
Additional Effective Address Words: 1

Figure 6-8 Special Addressing – Absolute Addressing

6.3.5.3.6 I/O Short

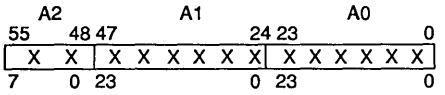
Classified as a memory reference, the I/O short addressing mode is similar to absolute short addressing. The address of the operand occupies six bits in the instruction operation word. I/O short is used with the bit manipulation and MOVEP instructions. The I/O short address is ones extended to 16 bits to address the I/O portion of X and Y memory (addresses $FFC0–$FFFF – see Figure 6-12).

6.3.5.3.7 Implicit Reference

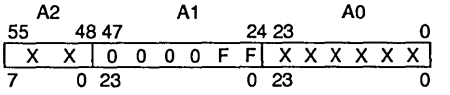
Some instructions make implicit reference to PC, SS, LA, LC, or SR. For example, the jump instruction (JMP) implicitly references the PC; whereas, the repeat next instruction (REP) implicitly references LC. The registers implied and their uses are defined by the individual instruction descriptions (see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

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

BEFORE EXECUTION

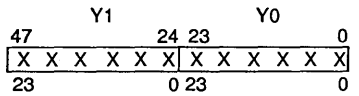


AFTER EXECUTION

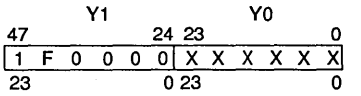


EXAMPLE B:POSITIVE IMMEDIATE SHORT INTO X0, X1, Y0, Y1, A, B
(MOVE #$1F, Y1)

BEFORE EXECUTION

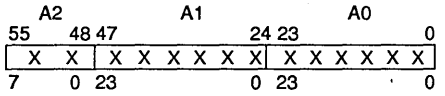


AFTER EXECUTION

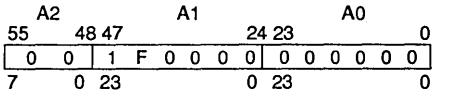


EXAMPLE C: POSITIVE IMMEDIATE SHORT INTO X, Y, A, B
(MOVE #$1F, A)

BEFORE EXECUTION

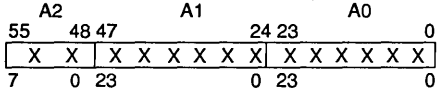


AFTER EXECUTION

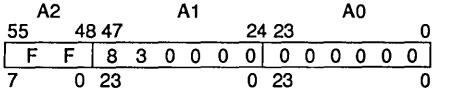


EXAMPLE D: NEGATIVE IMMEDIATE INTO 56-BIT REGISTER
(MOVE #$801234,A)

BEFORE EXECUTION



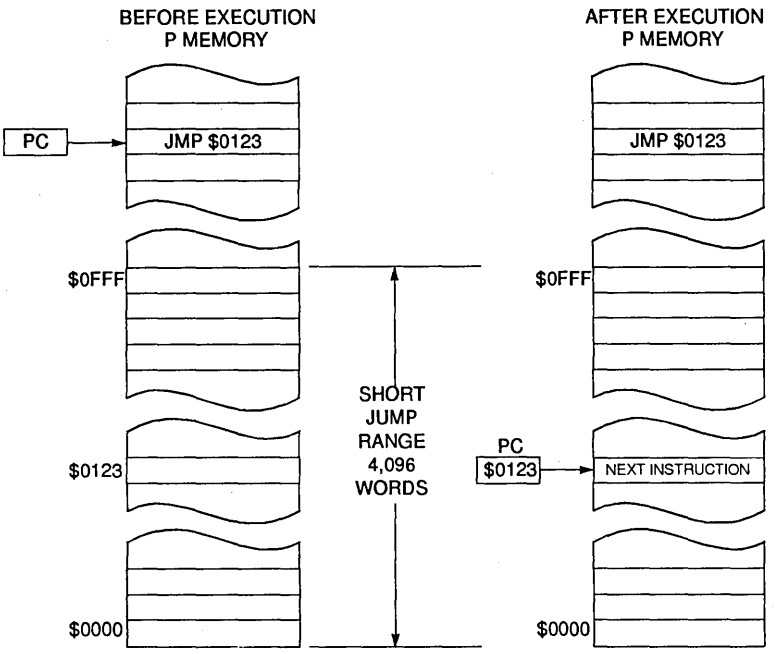
AFTER EXECUTION



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

Figure 6-9 Special Addressing – Immediate Short Data

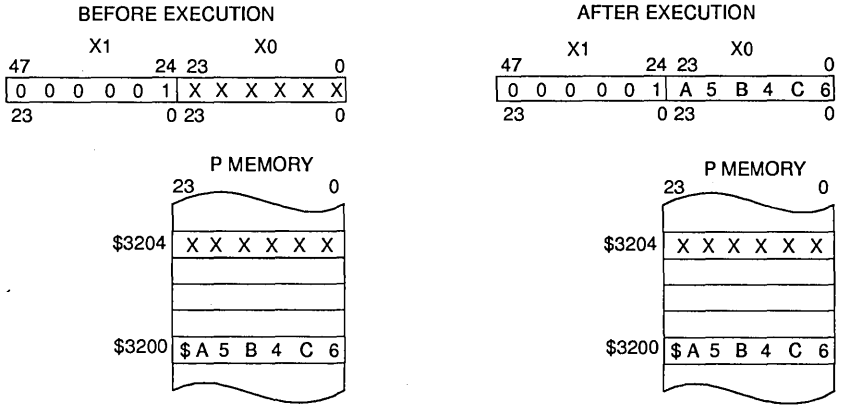
EXAMPLE: JMP $123



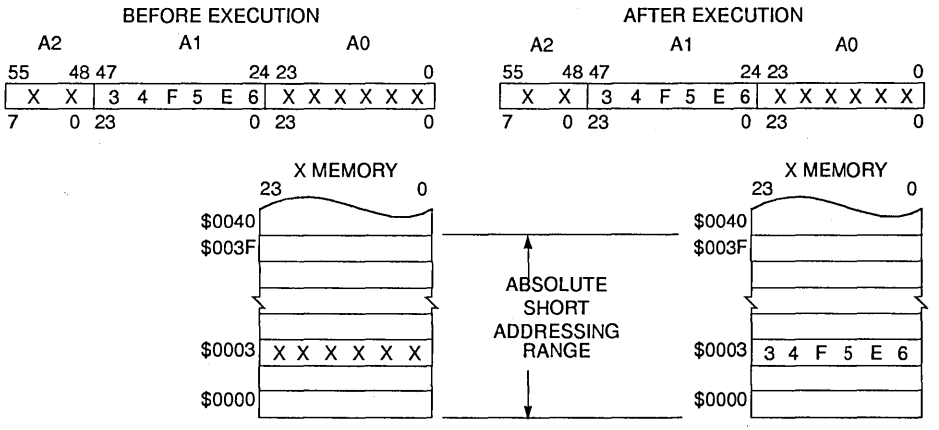
Assembler Syntax: XXX
Memory Spaces: P:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 6-10 Special Addressing – Short Jump Address

EXAMPLE A: MOVE P: $3200,X0



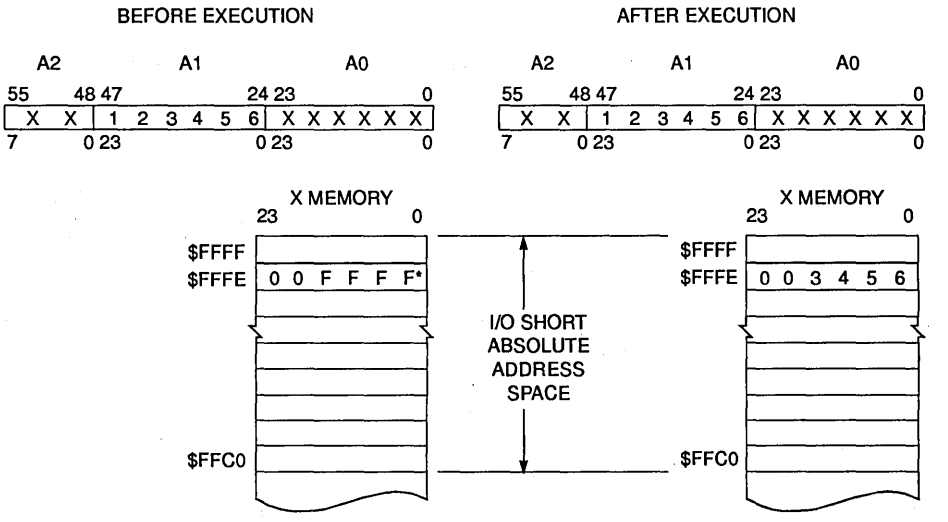
EXAMPLE B: MOVE A1, X: $3



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

Figure 6-11 Special Addressing – Absolute Short Address

EXAMPLE: MOVEP A1, X:<<$FFFE



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

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

Figure 6-12 Special Addressing – I/O Short Address

6.3.5.4 Addressing Modes Summary

Table 6-1 is a summary of the addressing modes discussed in the previous paragraphs.

6.4 INSTRUCTION GROUPS

The instruction set is divided into the following groups:

- Arithmetic
- Bit Manipulation
- Move
- Logical
- Loop
- Program Control

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

Table 6-1 Addressing Modes Summary

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

Where: MMMM = Address Modifier
P = Program Reference
S = Stack Reference
C = Program Control Unit Register Reference
D = Data ALU Register Reference
A = AGU Register Reference
X = X Memory Reference
Y = Y Memory Reference
L = L Memory Reference

6.4.1 Arithmetic Instructions

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

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

*These instructions do not allow parallel data moves.
**Certain applications of these instructions do not allow parallel data moves.

6.4.2 Logical Instructions

The logical instructions execute in one instruction cycle and 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.

Logical instructions are the only instructions that allow apparent duplicate destinations, such as:

```
AND X0,A          X:(R0):A0
```

A logical instruction uses only the MSP portion of the A and B registers (A1 and B1). Therefore, the instruction actually ignores what appears to be a duplicate destination and logically ANDs the value in the X0 register with the bits in the A1 portion (bits 47-24) of the A accumulator. The parallel move shown above can simultaneously write to either of the other two portions of the A or the B accumulator without conflict. Avoid confusion by explicitly stating A1 or B1 in the original instruction.

Optional data transfers may be specified with most logical instructions, allowing parallel data movement over the XDB and YDB or over the GDB during a data ALU operation. This parallel movement allows new data to be prefetched for use in subsequent instructions and allows results calculated in previous instructions to be stored. The following list includes the logical instructions:

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

*These instructions do not allow parallel data moves.

6.4.3 Bit Manipulation Instructions

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

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

6.4.4 Loop Instructions

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

DO	Start Hardware Loop
ENDDO	Exit from Hardware Loop

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

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

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

The operation of a DO loop is shown in Figure 6-13. When a program loop is initiated with the execution of a DO instruction, the following events occur:

1. The stack is pushed.
   A. The SP is incremented.
   B. The current 16-bit LA and 16-bit LC registers are pushed onto the SS to allow nested loops.
   C. The LC register is initiated with the loop count value specified in the DO instruction.

START OF LOOP

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

END OF LOOP

1)SSL(LF) ⬧ SR
2)SP−1 ⬧ SP; SSH ⬧ LA; SSL ⬧ LC; SP−1 ⬧ SP
3)PC + 1 ⬧ PC

NOTE:
#xxx=Loop Count Number
Expr=Expression

Figure 6-13 Hardware DO Loop

2. The stack is pushed again.
   A. The SP is incremented.
   B. The address of the first instruction in the program loop (PC) and the current SR contents are pushed onto the SS.
   C. The LA register is initialized with the value specified in the DO instruction decremented by one.
3. The LF bit in the SR is set. The LF bit is set when a program loop is in progress and enables the end-of-loop detection.

The program loop continues execution until the program address fetched equals the LA register contents (last address of program loop). The contents of the LC are then tested for one. If the LC is not one, it is decremented, and the top location in the stack RAM is read (but not pulled) into the PC to return to the start of the loop. If the LC is one, the program loop is terminated by the following sequence:

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

The LF bit (pulled from the SS when a loop is terminated) indicates if the terminated loop was a nested loop. Figure 6-14 shows two DO loops, one nested inside the other. If the stack is managed to prevent a stack overflow, DO loops can be stacked indefinitely.

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

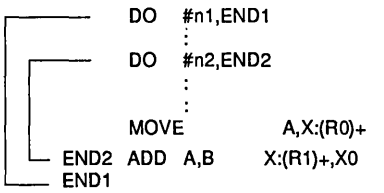


Figure 6-14 Nested DO Loops

6.4.5 Move Instructions

The move instructions perform data movement over the XDB and YDB or over the GDB. Move instructions only affect the CCR bits S and L The S bit is affected if data growth is detected when the A or B registers are moved onto the bus. The L bit is affected if limiting is performed when reading a data ALU accumulator register. An address ALU instruction (LUA) is also included in the following move instructions. The MOVE instruction is the parallel move with a data ALU no-operation (NOP).

6

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

Note: Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with a MOVE-type instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in SECTION 7 - PROCESSING STATES on page 7-10.

There are nine classifications of parallel data moves between registers and memory. Figure 6-15 shows seven parallel moves. The source of the data to be moved and the destination are separated by a comma.

Examples of the other two classifications, XY and long (L) moves, are shown in Figure 6-16. Example A illustrates the following steps: 1) register X0 is added to register A and the result is placed in register A; 2) register X0 is moved to the X memory register location pointed to by R3, and R3 is incremented; and 3) the contents of the Y memory location pointed to by R7 is moved to the B register, and R7 is decremented.

Example B depicts the following sequence: 1) register X0 is added to register A and the result is placed in register A; and 2) registers A and B are moved, respectively, to the locations in memories X and Y pointed to by R2, and then R2 is incremented by N2. The

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

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

Figure 6-15 Classifications of Parallel Data Moves

contents of the 56-bit registers A and B were rounded to 24 bits before moving to the 24-bit memory registers.

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

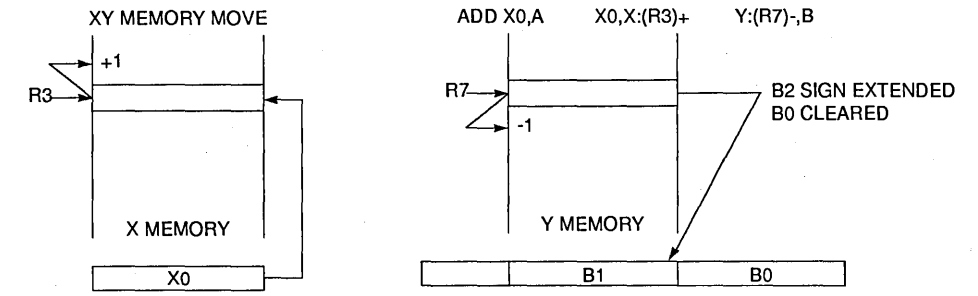
6.4.6 Program Control Instructions

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. The following list contains the program control instructions:

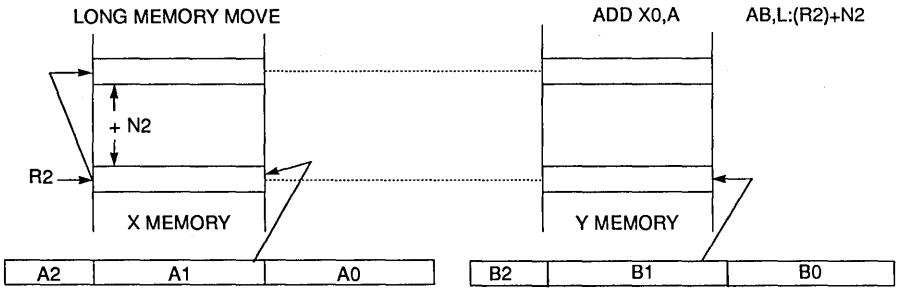
DEBUG	Enter Debug Mode
DEBUGcc	Enter Debug Mode Conditionally
Ill	Illegal Instruction
Jcc	Jump Conditionally
JMP	Jump

JCLR	Jump if Bit Clear
JSET	Jump if Bit Set
JScc	Jump to Subroutine Conditionally
JSR	Jump to Subroutine
JSCLR	Jump to Subroutine if Bit Clear
JSSET	Jump to Subroutine if Bit Set
NOP	No Operation
REP	Repeat Next Instruction
RESET	Reset On-Chip Peripheral Devices
RTI	Return from Interrupt
RTS	Return from Subroutine
STOP	Stop Processing (Low-Power Standby)
SWI	Software Interrupt
WAIT	Wait for Interrupt (Low-Power Standby)

Example A



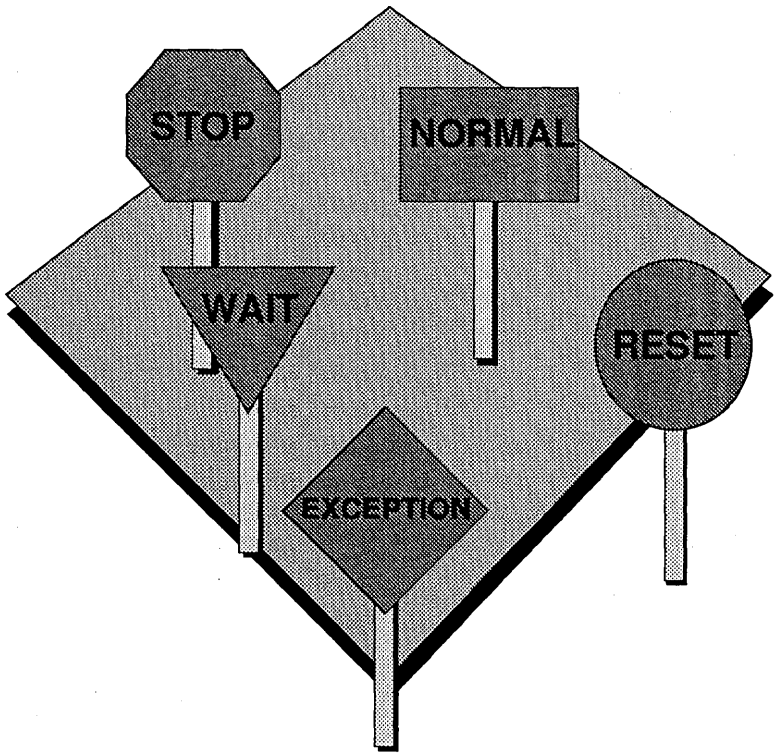
Example B



A,B ARE SHIFTED AND LIMITED

Figure 6-16 Parallel Move Examples

SECTION 7
PROCESSING STATES



SECTION CONTENTS

SECTION 7.1 PROCESSING STATES 3

SECTION 7.2 NORMAL PROCESSING STATE 3
7.2.1 Instruction Pipeline 3
7.2.2 Summary of Pipeline-Related Restrictions 8

SECTION 7.3 EXCEPTION PROCESSING STATE 10
7.3.1 Interrupt Types 12
7.3.2 Interrupt Priority Structure 12
7.3.2.1 Interrupt Priority Levels 14
7.3.2.2 Exception Priorities Within an IPL 15
7.3.3 Interrupt Sources 16
7.3.3.1 Hardware Interrupt Sources 16
7.3.3.2 Software Interrupt Sources 17
7.3.3.3 Other Interrupt Sources 22
7.3.4 Interrupt Arbitration 24
7.3.5 Interrupt Instruction Fetch 24
7.3.6 Instructions Preceding the Interrupt Instruction Fetch 25
7.3.7 Interrupt Instruction Execution 26

SECTION 7.4 RESET PROCESSING STATE 33

SECTION 7.5 WAIT PROCESSING STATE 36

SECTION 7.6 STOP PROCESSING STATE 37

7.1 PROCESSING STATES

The DSP56K processor is always in one of five processing states: normal, exception, reset, wait, or stop. This section describes each of the processing states.

7.2 NORMAL PROCESSING STATE

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

7.2.1 Instruction Pipeline

DSP56K instruction execution occurs in a three-stage pipeline, which allows most instructions to execute at a rate of one instruction per instruction cycle. However, certain instructions require additional time to execute: instructions longer than one word, instructions using an addressing mode that requires more than one cycle, and instructions that cause a control-flow change. In the latter case, a cycle is needed to clear the pipeline.

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

Table 7-1 demonstrates pipelining. F1, D1, and E1 refer to the fetch, decode, and execute operations, respectively, of the first instruction. The third instruction, which contains an instruction extension word, takes two instruction cycles to execute. The extension word will be either an absolute address or immediate data. Although it takes three instruction cycles for the pipeline to fill and the first instruction to execute, an instruction usually executes on each instruction cycle thereafter.

Table 7-1 Instruction Pipelining

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

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

The pipeline is normally transparent to the user. However, there are certain instruction-sequence dependent situations where the pipeline will affect the program execution. Such situations are best described by case studies. Most of these restricted sequences occur because 1) all addresses are formed during instruction decode, or 2) they are the result of contention for an internal resource such as the status register (SR). If the execution of an instruction depends on the relative location of the instruction in a sequence of instructions, there is a pipeline effect. To test for a suspected pipeline effect, compare between the execution of the suspect instruction 1) when it directly follows the previous instruction and 2) when four NOPs are inserted between the two. If there is a difference, it is caused by a pipeline effect. The DSP56K assembler flags instruction sequences with potential pipeline effects so that the user can determine if the operation will execute as expected.

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

1. No pipeline effect:

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

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

2. Instruction that started execution during decode:

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

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

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

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

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

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

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

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

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

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

second instruction of the downloaded code at P:$0001 of the internal RAM. However, the ANDI instruction allows the OMR to be changed before the JMP instruction uses it, and the JMP fetches P:$0000 of the internal RAM.

Case 4: An interrupt has two additional control cycles that are executed in the interrupt controller concurrently with the fetch, decode, and execute cycles (see Section 7.3 and Figure 7-4). During these two control cycles, the interrupt is arbitrated by comparing the interrupt mask level with the interrupt priority level (IPL) of the interrupt and allowing or disallowing the interrupt. Therefore, if the interrupt mask is changed after an interrupt is arbitrated and accepted as pending but before the interrupt is executed, the interrupt will be executed, regardless of what the mask was changed to. The following examples show that the old interrupt mask is in effect for up to four additional instruction cycles after the interrupt mask is changed. All instructions shown in the examples here are one-word instructions; however, one two-word instruction can replace two one-word instructions except where noted.

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

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

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

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

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

Note 2: II=Interrupt instruction from maskable interrupt.

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

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

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

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

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

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

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

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

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

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

7.2.2 Summary of Pipeline-Related Restrictions

The following paragraphs give a summary of the instruction sequences that cause pipeline effects. Additional information about the individual instructions can be found in APPENDIX A - INSTRUCTION SET DETAILS.

DO instruction restrictions:

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

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

The DO instruction cannot specify SSH as a source register, as in the following example:

```
DO SSH,xxxx
```

Restrictions near the end of DO loops:

Proper DO loop operation is guaranteed if no instruction starting at address LA-2, LA-1, or LA specifies the program controller registers SR, SP, SSL, LA, LC, or (implicitly) PC as a destination register, or specifies SSH as a source or a destination register.

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

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

The restricted instructions at LA include the following:

Any two-word instruction
Jcc, JMP, JScc, JSR,
REP, RESET, RTI, RTS, STOP, WAIT

Another restriction is shown below:

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

ENDDO instruction restrictions:

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

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

RTI and RTS instruction restrictions:

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

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

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

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

SP and SSH/SSL register manipulation restrictions:

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

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

and

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

and

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

and

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

Also, the instruction MOVEC SSH,SSH is illegal.

Rn, Nn, and Mn register restrictions:

Due to pipelining, if an address register Rn is the destination of a MOVE-type instruction except MOVEP (MOVE, MOVEC, MOVEM, LUA, Tcc), the new contents will not be available for use as an address pointer until the second following instruction cycle.

Likewise, if an offset register Nn or a modifier register Mn is the destination of a MOVE-type instruction except MOVEP, the new contents will not be available for use in address calculations until the second following instruction cycle.

However, if the processor is in the No Update addressing mode (where Mn and Nn are ignored) and register Mn or Nn is the destination of a MOVE instruction, the next instruction may use the corresponding Rn register as an address pointer. Also, if the processor is in the Postincrement by 1, Postdecrement by 1, or Predecrement by 1 addressing mode (where Nn is ignored), a MOVE to Nn may be immediately followed by an instruction that uses Rn as an address pointer.

Fast interrupt routines:

SWI, STOP, and WAIT may not be used in a fast interrupt routine. (Fast interrupts are described in Section 7.3.1.)

7.3 EXCEPTION PROCESSING STATE (INTERRUPT PROCESSING)

The exception processing state is associated with interrupts that can be generated by conditions inside the DSP or from external sources. 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 minimal overhead is ideal. A fast interrupt accomplishes a limited context switch. The processor relies on a long interrupt when it must accomplish a more complex task to service the interrupt. Fast interrupts and long interrupts are described in more detail in Section 7.3.1.

There are many sources for interrupts on the DSP56K family of chips, and some of these sources can generate more than one interrupt. The DSP56K family of processors features a prioritized interrupt vector scheme with 32 vectors to provide fast interrupt service. The interrupt priority structure is discussed in Section 7.3.2. The following list outlines how the DSP56K processes interrupts:

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

If neither instruction is a change of program-flow instruction (i.e., a JSR), the state of the machine is not saved on the stack, and a fast interrupt is executed. A long interrupt occurs if one of the interrupt instructions fetched is a JSR instruction. The PC is immediately released, the SR and the PC are saved in the stack, and the jump instruction controls where the next instruction shall be fetched. While either an unconditional jump or a conditional jump can be used to form a long interrupt, they do not store the PC on the stack. Therefore, there is no return path.

Activities 2 and 3 listed above require two additional control cycles, which effectively make the **interrupt** pipeline five levels deep.

7.3.1 Interrupt Types

The DSP56K relies on two types of interrupt routines: fast and long. The fast interrupt fetches only two words and then automatically resumes execution of the main program; whereas, the long interrupt must be told to return to the main program by executing an RTI instruction. The fast routine consists of two automatically inserted interrupt instruction words. These words can contain any unrestricted, single two-word instruction or any two one-word instructions (see Section A.9 in APPENDIX A - INSTRUCTION SET DETAILS for a list of restrictions). Fast interrupt routines are never interruptible.

CAUTION

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

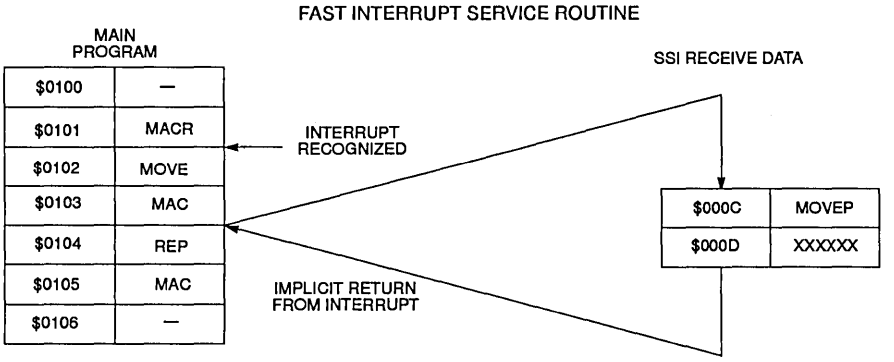
If one of the instructions in the fast routine is a JSR, then a long interrupt routine is formed. The following actions occur during execution of the JSR instruction when it occurs in the interrupt starting address or in the interrupt starting address +1:

1. The PC (containing the return address) and the SR are stacked.
2. The loop flag is reset.
3. The scaling mode bits are reset.
4. The IPL is raised to disallow further interrupts at the same or lower levels (except that hardware $\overline{\text{RESET}}$, NMI, stack error, trace, and SWI can always interrupt).
5. The trace bit in the SR is cleared (in the DSP56000/56001 only).

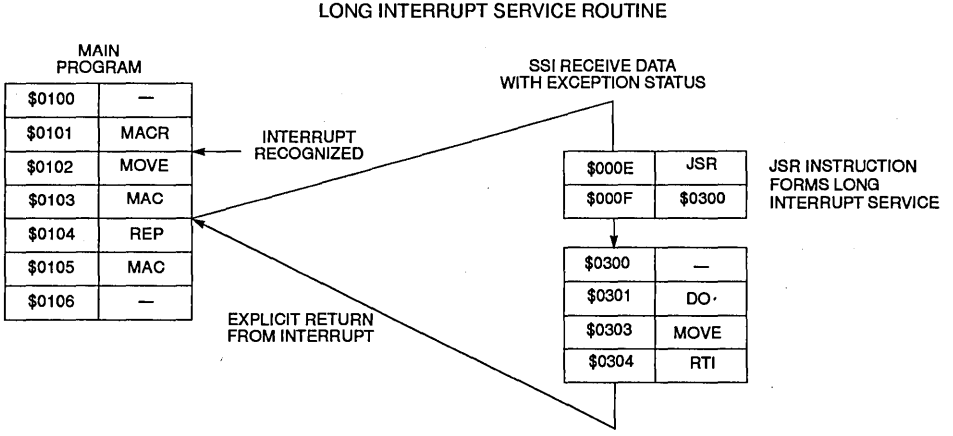
The long interrupt routine should be terminated by an RTI. Long interrupt routines are interruptible by higher priority interrupts. Figure 7-1 shows examples of fast and long interrupts.

7.3.2 Interrupt Priority Structure

Interrupts are organized in a flexible priority structure. Each interrupt has an associated interrupt priority level (IPL) that can range from zero to three. Levels 0 (lowest level), 1, and 2 are maskable. Level 3 is the highest IPL and is not maskable. The only IPL 3 interrupts are $\overline{\text{RESET}}$, illegal instruction interrupt (III), nonmaskable interrupt ($\overline{\text{NMI}}$), stack error, trace, and software interrupt (SWI). The interrupt mask bits (I1, I0) in the SR reflect the current priority level and indicate the IPL needed for an interrupt source to interrupt the processor (see Table 7-2). Interrupts are inhibited for all priority levels below the cur-



(a) DSP56K Fast Interrupt



(b) DSP56K Long Interrupt

Figure 7-1 Fast and Long Interrupt Examples

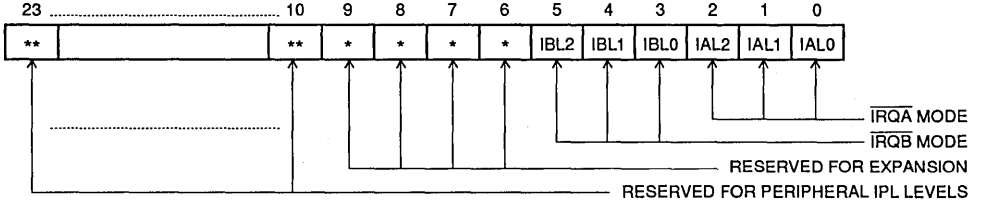
Table 7-2 Status Register Interrupt Mask Bits

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

rent processor priority level. However, level 3 interrupts are not maskable and therefore can always interrupt the processor. DSP56K Family central processor interrupt sources and their IPLs are listed in Table 7-6. For information on on-chip peripheral interrupt priority levels, see the individual DSP56K family member's User's Manual.

7.3.2.1 Interrupt Priority Levels

The IPL for each on-chip peripheral device (HI, SSI, SCI) and for each external interrupt source ($\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$) can be programmed to one of the three maskable priority levels (IPL 0, 1, or 2) under software control. IPLs are set by writing to the interrupt priority register shown in Figure 7-2. This read/write register is located in program memory at address $FFFF. It specifies the IPL for each of the interrupting devices including IRQA, IRQB and each peripheral device. (For specific peripheral information, see the specific DSP56K family member's User's Manual.) In addition, it specifies the trigger mode of the external interrupt sources and is used to enable or disable the individual external interrupts. The interrupt priority register is cleared on $\overline{\text{RESET}}$ or by the reset instruction. Table 7-3 defines the IPL bits. Table 7-4 defines the external interrupt trigger mode bits.



Bits 6 to 9 are reserved, read as zero and should be written with zero for future compatibility.

Figure 7-2 Interrupt Priority Register (Addr X:$FFFF)

Table 7-3 Interrupt Priority Level Bits

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

Table 7-4 External Interrupt

IxL2	Trigger Mode
0	Level
1	Negative Edge

7.3.2.2 Exception Priorities Within an IPL

If more than one interrupt is pending when an instruction is executed, the processor will service the interrupt with the highest priority level first. When multiple interrupt requests with the same IPL are pending, a second fixed-priority structure within that IPL determines which interrupt the processor will service. The fixed priority of interrupts within an IPL and the interrupt enable bits for all interrupts are shown in Table 7-5.

Table 7-5 Central Processor Interrupt Priorities Within an IPL

Priority	Exception	Enabled By	Bit No.	X Data Memory Address
Level 3 (Nonmaskable)				
Highest	Hardware $\overline{\text{RESET}}$	—	—	—
	III	—	—	—
	NMI	—	—	—
	Stack Error	—	—	—
	Trace	—	—	—
Lowest	SWI	—	—	—
Levels 0, 1, 2 (Maskable)				
Higher	$\overline{\text{IRQA}}$ (External Interrupt)	$\overline{\text{IRQA}}$ Mode Bits	0 and 1	$FFFF
Lower	$\overline{\text{IRQB}}$ (External Interrupt)	$\overline{\text{IRQB}}$ Mode Bits	3 and 4	$FFFF

7.3.3 Interrupt Sources

Interrupts can originate from any of the vector addresses listed in Table 7-6, which shows the corresponding interrupt starting address for each interrupt source. These addresses are located in the first 64 locations of program memory.

Table 7-6 Interrupt Sources

Interrupt Starting Address	IPL	Interrupt Source
$0000	3	Hardware $\overline{\text{RESET}}$
$0002	3	Stack Error
$0004	3	Trace
$0006	3	SWI
$0008	0 - 2	$\overline{\text{IRQA}}$
$000A	0 - 2	$\overline{\text{IRQB}}$
:	:	Vectors available for peripherals
$001E	3	NMI
:	:	Vectors available for peripherals
$003E	3	Illegal Instruction

When an interrupt occurs, 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 DSP56K can be described as "vectored". A vectored interrupt structure has low execution overhead. If it is known beforehand that certain interrupts will not be used, those interrupt vector locations can be used for program or data storage.

7.3.3.1 Hardware Interrupt Sources

There are two types of hardware interrupts in the DSP56K: internal and external. The internal interrupt sources include all of the on-chip peripheral devices. For further information on a device's internal interrupt sources, see the device's individual User's Manual.

The external hardware interrupt sources are the $\overline{\text{RESET}}$, $\overline{\text{NMI}}$, $\overline{\text{IRQA}}$, and $\overline{\text{IRQB}}$ pins on the program interrupt controller in the Program Control Unit.

The level sensitive $\overline{\text{RESET}}$ interrupt is the highest priority interrupt with an IPL of 3. $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ can be programmed to one of three priority levels: 0, 1, or 2 - all of which are maskable. $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ have independent enable control and can be programmed to be level sensitive or edge sensitive. Since level-sensitive interrupts will not be cleared

automatically when they are serviced, they must be cleared by other means to prevent multiple interrupts. 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.

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

The processor begins interrupt service by fetching the instruction word in the first vector location. The interrupt is considered finished when the processor fetches the instruction word in the second vector location.

In an edge-triggered interrupt, the internal latch is automatically cleared when the second vector location is fetched. The fetch of the first vector location does not guarantee that the second location will be fetched. Figure 7-3 illustrates the one case where the second vector location is not fetched. The SWI instruction in the figure discards the fetch of the first interrupt vector to ensure that the SWI vectors will be fetched. Instruction n4 is decoded as an SWI while ii1 is being fetched. Execution of the SWI requires that ii1 be discarded and the two SWI vectors (ii3 and ii4) be fetched instead.

CAUTION

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

The edge sensitive NMI is a priority 3 interrupt and cannot be masked. Only $\overline{\text{RESET}}$ and illegal instruction have higher priority than NMI.

7.3.3.2 Software Interrupt Sources

There are two software interrupt sources — software interrupt (SWI) and illegal instruction interrupt (III).

SWI is a nonmaskable interrupt (IPL 3), which is serviced immediately following the SWI instruction execution, usually using a long interrupt service routine. The difference between an SWI and a JSR instruction is that the SWI sets the interrupt mask to prevent interrupts below IPL 3 from being serviced. The SWI's ability to mask out lower level

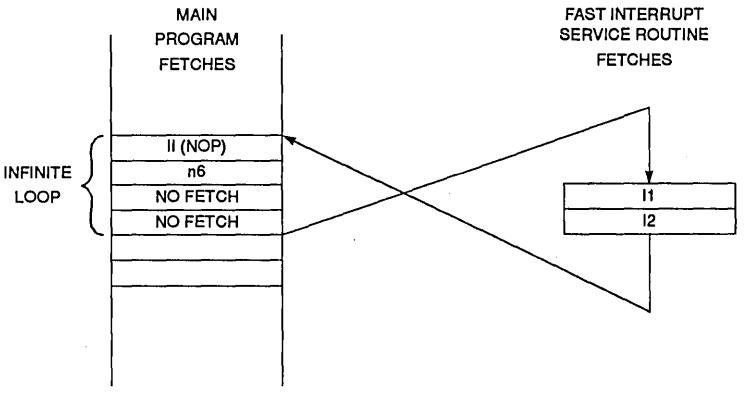
interrupts makes it very useful for setting breakpoints in monitor programs. The JSR instruction does not affect the interrupt mask.

The III is also a nonmaskable interrupt (IPL 3). It is serviced immediately following the execution or the attempted execution of an illegal instruction (any undefined operation code). IIIs are fatal errors. Only a long interrupt routine should be used for the III routine. RTI or RTS should not be used at the end of the interrupt routine because, during the III service, the JSR located in the III vector will normally stack the address of the illegal instruction (see Figure 7-4). Returning from the interrupt routine would cause the processor to attempt to execute the illegal interrupt again and cause an infinite loop which can only be broken by cycling power. This long interrupt (see Figure 7-4) can be used as a diagnostic tool to allow the programmer to examine the stack (MOVE SSH, dest) and locate the illegal instruction, or the application program can be restarted with the hope that the failure was a soft error. The illegal instruction is useful for triggering the illegal interrupt service routine to see if the III routine can recover from illegal instructions.

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

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

Figure 7-3 Interrupting an SWI



(a) Instruction Fetches from Memory

ILLEGAL INSTRUCTION INTERRUPT RECOGNIZED AS PENDING

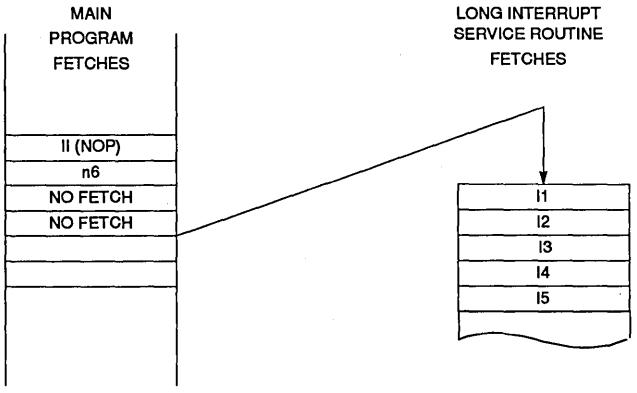
ILLEGAL INSTRUCTION INTERRUPT RECOGNIZED AS PENDING

INTERRUPT CONTROL CYCLE 1								i						
INTERRUPT CONTROL CYCLE 2									i					
FETCH		n1	n2	n3	n4	n5	n6	—	—	ii1	ii2	n5		
DECODE			n1	n2	n3	n4	II	—	—	—	ii	ii2	II	
EXECUTE				n1	n2	n3	n4	NOP	—	—	—	ii1	ii2	NOP
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14

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

(b) Program Controller Pipeline

Figure 7-4 Illegal Instruction Interrupt Serviced by a Fast Interrupt



(a) Instruction Fetches from Memory

ILLEGAL INSTRUCTION INTERRUPT RECOGNIZED AS PENDING

ILLEGAL INSTRUCTION INTERRUPT RECOGNIZED AS PENDING

INTERRUPT CONTROL CYCLE 1								i						
INTERRUPT CONTROL CYCLE 2									i					
FETCH		n1	n2	n3	n4	n5	n6	—	—	ii1	ii2	ii3	ii4	ii5
DECODE			n1	n2	n3	n4	II	—	—	—	ii1	ii2	ii3	ii4
EXECUTE				n1	n2	n3	n4	NOP	—	—	—	ii1	ii2	ii3
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14

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

(b) Program Controller Pipeline

Figure 7-5 Illegal Instruction Interrupt Serviced by a Long Interrupt

ILLEGAL INSTRUCTION INTERRUPT
RECOGNIZED AS PENDING

INTERRUPT CONTROL CYCLE 1										i					
INTERRUPT CONTROL CYCLE 2											i				
FETCH		n1	n2	n3	n4	n5	n6	n7	—	—	—	ii1	ii2	n8	
DECODE			n1	n2	n3	n4	REP	II	—	—	—	—	ii1	ii2	n8
EXECUTE				n1	n2	n3	n4	REP	REP	NOP	—	—	—	ii1	ii2	n8
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

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

Figure 7-6 Repeated Illegal Instruction

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

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

2. If the illegal instruction follows an REP instruction (see Figure 7-6), the processor will effectively execute the illegal instruction as a repeated NOP and the interrupt vector will then be inserted in the pipeline. The next instruction will be fetched but will not be decoded or executed. The processor will stack the address of the next sequential instruction, which is two instructions after the illegal instruction.

In DO loops, if the illegal instruction is in the loop address (LA) location and the instruction preceding it (i.e., at LA-1) is being interrupted, the loop counter (LC) will be decremented as if the loop had reached the LA instruction. When the interrupt service ends and the instruction flow returns to the loop, the illegal instruction will be refetched (since it is the next sequential instruction in the flow). The loop state machine will again decrement LC because the LA instruction is being executed. At this point, the illegal instruction will trigger the III. The result is that the loop state machine decrements LC twice in one loop due to the presence of the illegal opcode at the LA location.

7.3.3.3 Other Interrupt Sources

Other interrupt sources include the stack error interrupt and trace interrupt (DSP56000/56001) which are IPL3 interrupts.

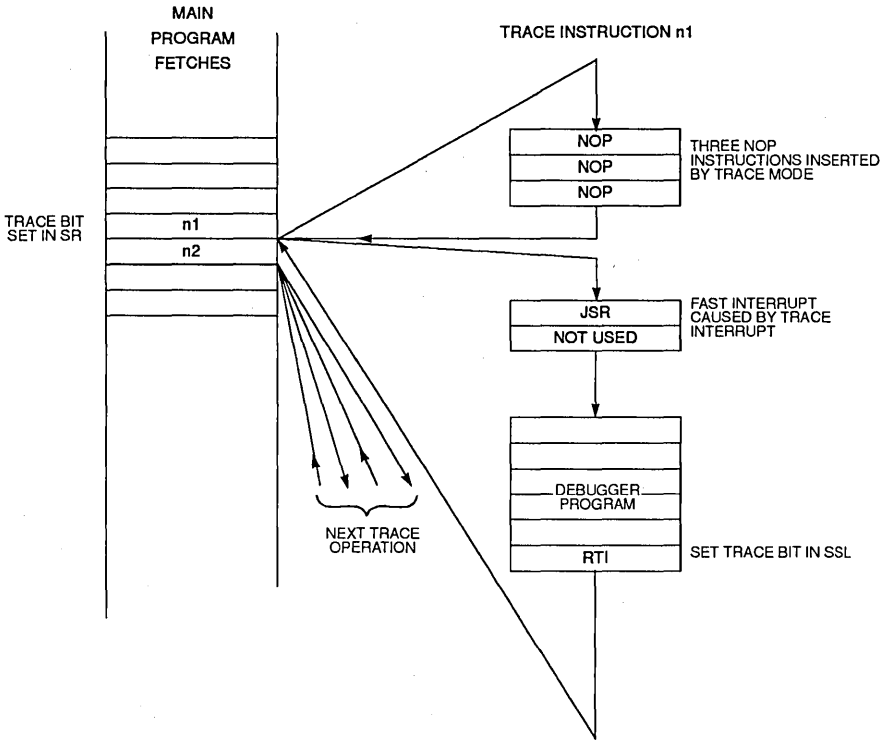
An overflow or underflow of the system stack (SS) causes a stack error interrupt which is vectored to P:$0002 (see SECTION 5 - PROGRAM CONTROL UNIT for additional information on the stack error flag). Since the stack error is nonrecoverable, a long interrupt should be used to service it. The service routine should not end in an RTI because executing an RTI instruction "pops" the stack, which has been corrupted.

The DSP56000/56001 includes a facility for instruction-by-instruction tracing as a program development aid. This trace mode generates a trace exception after each instruction executed (see Figure 7-7), which can be used by a debugger program to monitor the execution of a program. (With members of the DSP56K family other than DSP56000/56001, use the OnCE trace mode described in 10.5.)

The trace bit in the SR defines the trace mode. In the trace mode, the processor will generate a trace exception after it executes each instruction. When the processor is servicing the trace exception, it expects to encounter a JSR in the trace vector locations, thereby forming a long interrupt routine. The JSR stacks the SR and clears the trace bit to prevent tracing while executing the trace exception service routine. This service routine should end with an RTI instruction, which restores the SR (with the trace bit set) from the SS, and causes the next instruction to be traced. The pipeline must be flushed to allow each sequential instruction to be traced. The tracing facility appends three instruction cycles to the end of each instruction traced (see the three NOP instructions shown in Figure 7-7) to flush the pipeline and allow the next trace interrupt to follow the next sequential interrupt.

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

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



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPT CONTROL CYCLE 1	i												i					
INTERRUPT CONTROL CYCLE 2		i													i			
FETCH		n1	NOP	NOP	NOP	JSR	—	TRACE PROGRAM			RTI	—	n2	NOP	NOP	NOP		
DECODE			n1	NOP	NOP	NOP	JSR	NOP	TRACE PROGRAM			RTI	NOP	n2	NOP	NOP	NOP	
EXECUTE				n1	NOP	NOP	NOP	JSR	NOP	TRACE PROGRAM			RTI	NOP	n2	NOP	NOP	NOP
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

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

(b) Program Controller Pipeline

Figure 7-7 Trace Exception

7.3.4 Interrupt Arbitration

Interrupt arbitration and control, which occurs concurrently with the fetch-decode-execute cycle, takes two instruction cycles. External interrupts are internally synchronized with the processor clock before their interrupt-pending flags are set. Each external and internal interrupt has its own flag. After each instruction is executed, the DSP arbitrates all interrupts. During arbitration, each 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 IPLs shown in Table 7-5, 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.

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

The internal interrupt acknowledge signal clears the edge-triggered interrupt flags and the internal latches of the NMI, SWI, and trace interrupts. The stack error bit in the stack pointer register is "sticky" and requires a "MOVE" or a bit clear operation directly on the stack pointer register. Some peripheral interrupts may also be cleared by the internal interrupt acknowledge signal, as defined in their specifications. Peripheral interrupt requests that need a read/write action to some register do not receive the internal interrupt acknowledge signal, and they will remain pending until their registers are read/written. Further, level-triggered interrupts will not be cleared. The acknowledge signal will be generated after the interrupt vectors have been generated, not before.

7.3.5 Interrupt Instruction Fetch

The interrupt controller generates an interrupt instruction fetch address, which points to the first instruction word of a two-word interrupt routine. This address is used for the next instruction fetch, instead of the contents of the PC, and the interrupt instruction fetch address +1 is used for the subsequent instruction fetch. While the interrupt instructions are being fetched, the PC cannot be updated. After the two interrupt words have been fetched, the PC is used for any subsequent instruction fetches.

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

7.3.6 Instructions Preceding the Interrupt Instruction Fetch

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

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

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

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

1. If a long interrupt routine is used to service an SWI, then the processor priority level is set to 3. Thus, all interrupts except other level-3 interrupts are disabled until the SWI service routine terminates with an RTI (unless the SWI service routine software lowers the processor priority level).
2. While servicing an interrupt, the next interrupt service will be delayed according to the following rule: after the first interrupt instruction word reaches the instruction decoder, at least three more instructions will be decoded before decoding the next first interrupt instruction word. If any one pair of instructions being counted is the REP instruction followed by an instruction to be repeated, then the combination is counted as two instructions independent of the number of repeats done. Sequential REP combinations will cause pending interrupts to be rejected and can not be interrupted until the sequence of REP combinations ends.
3. The following instructions are not interruptible: SWI, STOP, WAIT, and RESET.
4. The REP instruction and the instruction being repeated are not interruptible.
5. If the trace bit in the SR (DSP56000/56001 only) is set, the only interrupts that will be processed are the hardware RESET, III,NMI, stack error, and trace. Peripheral and external interrupt requests will be ignored. The interrupt generated by the SWI instruction will be ignored.

7.3.7 Interrupt Instruction Execution

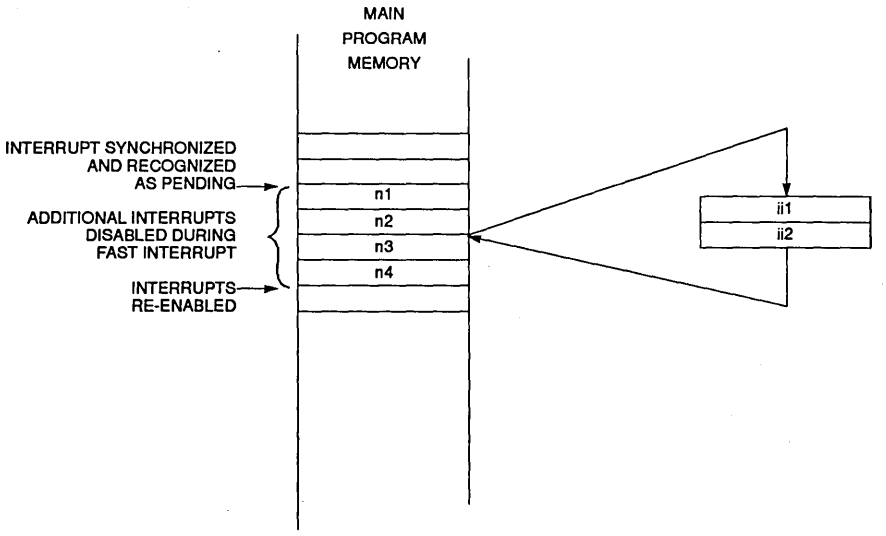
Interrupt instruction execution is considered "fast" if neither of the instructions of the interrupt service routine causes a change of flow. A JSR within a fast interrupt routine forms a long interrupt, which is terminated with an RTI instruction to restore the PC and SR from the stack and return to normal program execution. Reset is a special exception, which will normally contain only a JMP instruction at the exception start address. At the programmer's option, almost any instruction can be used in the fast interrupt routine. The restricted instructions include SWI, STOP, and WAIT. Figure 7-8 and Figure 7-10 show the fast and the long interrupt service routines. The fast interrupt executes only two instructions and then automatically resumes execution of the main program; whereas, the long interrupt must be told to return to the main program by executing an RTI instruction.

Figure 7-8 illustrates the effect of a fast interrupt routine in the stream of instruction fetches.

Figure 7-9 shows the sequence of instruction decodes between two fast interrupts. Four decodes occur between the two interrupt decodes (two after the first interrupt and two preceding the second interrupt). The requirement for these four decodes establishes the maximum rate at which the DSP56K will respond to interrupts — namely, one interrupt every six instructions (six instruction cycles if all six instructions are one instruction cycle each). Since some instructions take more than one instruction cycle, the minimum number of instructions between two interrupts may be more than six instruction cycles.

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

1. A JSR to the starting address of the interrupt service routine is not located at one of the two interrupt vector addresses.
2. The processor status is not saved.
3. The fast interrupt routine may (but should not) modify the status of the normal instruction stream.
4. The fast interrupt routine may contain any single two-word instruction or any two one-word instructions except SWI, STOP, and WAIT.
5. The PC, which contains the address of the next instruction to be executed in normal processing remains unchanged during a fast interrupt routine.



ii = INTERRUPT INSTRUCTION
n = NORMAL INSTRUCTION

(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

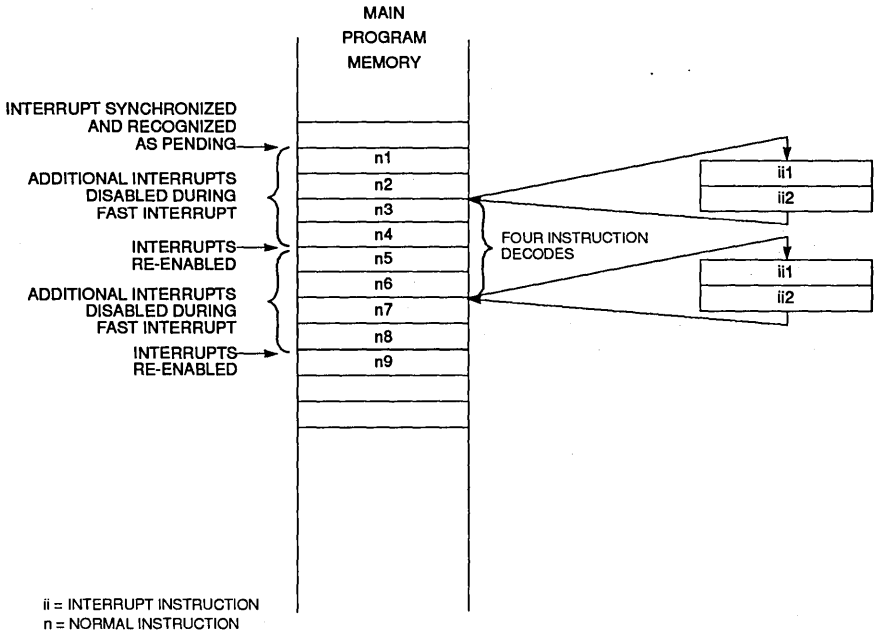
INTERRUPTS RE-ENABLED

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

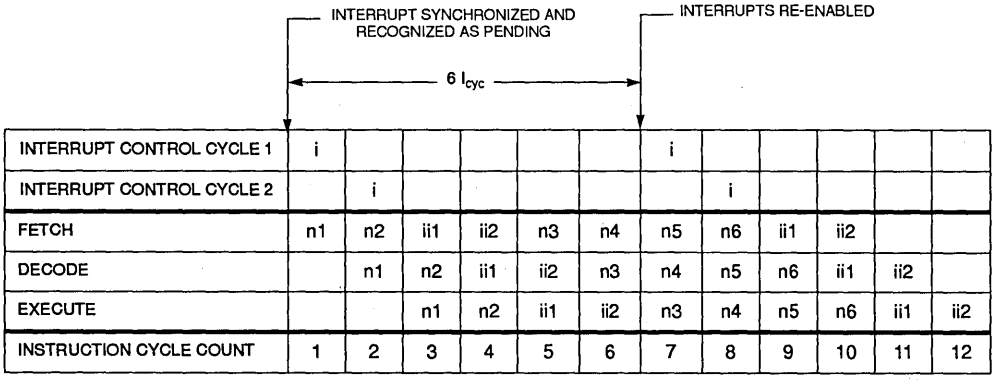
i = INTERRUPT
ii = INTERRUPT INSTRUCTION WORD
n = NORMAL INSTRUCTION WORD

(b) Program Controller Pipeline

Figure 7-8 Fast Interrupt Service Routine



(a) Instruction Fetches from Memory



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

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

(b) Program Controller Pipeline

Figure 7-9 Two Consecutive Fast Interrupts

6. The fast interrupt returns without an RTI.
7. Normal instruction fetching resumes using the PC following the completion of the fast interrupt routine.
8. A fast interrupt is not interruptible.
9. A JSR instruction within the fast interrupt routine forms a long interrupt routine.
10. The primary application is to move data between memory and I/O devices.

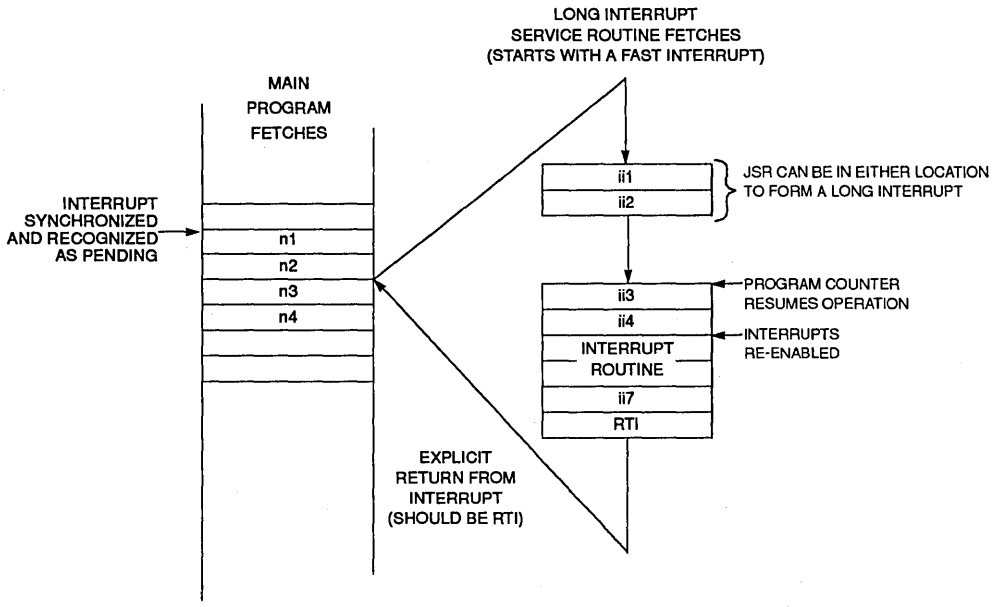
The execution of a long interrupt routine always conforms to the following rules:

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

Figure 7-10 illustrates the effect of a long interrupt routine on the instruction pipeline. A short JSR (a JSR with 12-bit absolute address) is used to form the long interrupt routine. For this example, word 6 of the long interrupt routine is an RTI. The point at which interrupts are re-enabled and subsequent interrupts are allowed is shown to illustrate the non-interruptible nature of the early instructions in the long interrupt service routine.

Either one of the two instructions of the fast interrupt can be the JSR instruction that forms the long interrupt. Figure 7-11 and Figure 7-12 show the two possible cases. If the first fast interrupt vector instruction is the JSR, the second instruction is never used.

A REP instruction and the instruction that follows it are treated as a single two-word instruction, regardless of how many times it repeats the second instruction of the pair. Instruction fetches are suspended and will be reactivated only after the LC is decre-



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

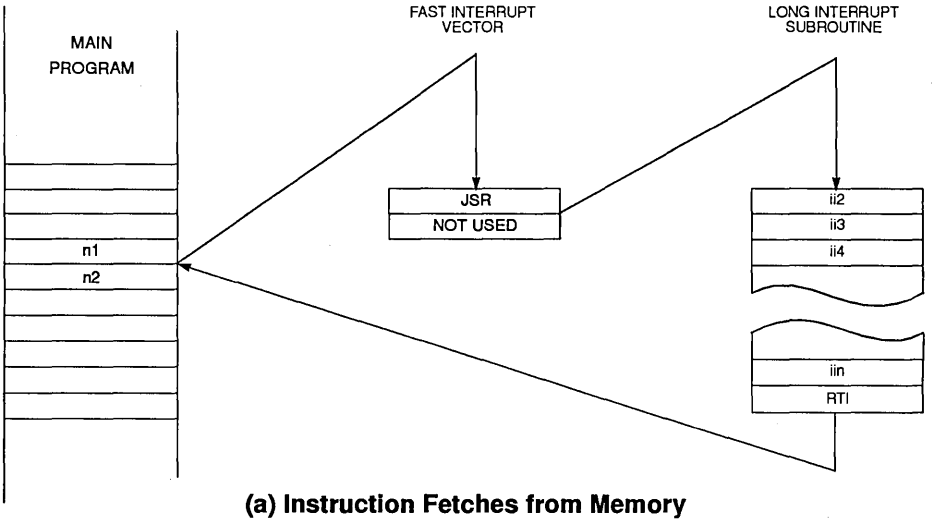
INTERRUPTS RE-ENABLED

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

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

(b) Program Controller Pipeline

Figure 7-10 Long Interrupt Service Routine



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

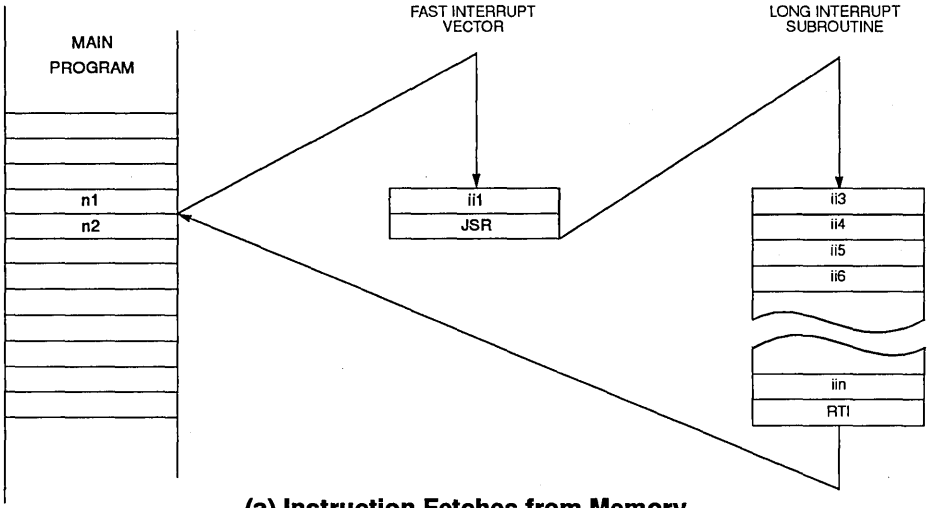
INTERRUPTS RE-ENABLED

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

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

(b) Program Controller Pipeline

Figure 7-11 JSR First Instruction of a Fast Interrupt



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPTS RE-ENABLED

INTERRUPT CONTROL CYCLE 1	i														
INTERRUPT CONTROL CYCLE 2		i													
FETCH		n1	ii1	JSR	—	ii3	ii4	ii5		iin	RTI	—	n2		
DECODE			n1	ii1	JSR	NOP	ii3	ii4	ii5	ii6	iin	RTI	NOP	n2	
EXECUTE				n1	ii1	JSR	NOP	ii3	ii4	ii5	ii6	iin	RTI	NOP	n2
INSTRUCTION CYCLE COUNT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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

(b) Program Controller Pipeline

Figure 7-12 JSR Second Instruction of a Fast Interrupt

mented to one (see Figure 7-13). During the execution of n2 in Figure 7-13, no interrupts will be serviced. When LC finally decrements to one, the fetches are reinitiated, and pending interrupts can be serviced.

Sequential REP packages will cause pending interrupts to be rejected until the sequence of REP packages ends. REP packages are not interruptible because the instruction being repeated is not refetched. While that instruction is repeating, no instructions are fetched or decoded, and an interrupt can not be inserted. For example, in Figure 7-14, if n1, n3, and n5 are all REP instructions, no interrupts will be serviced until the last REP instruction (n5 and its repeated instruction, n6) completes execution.

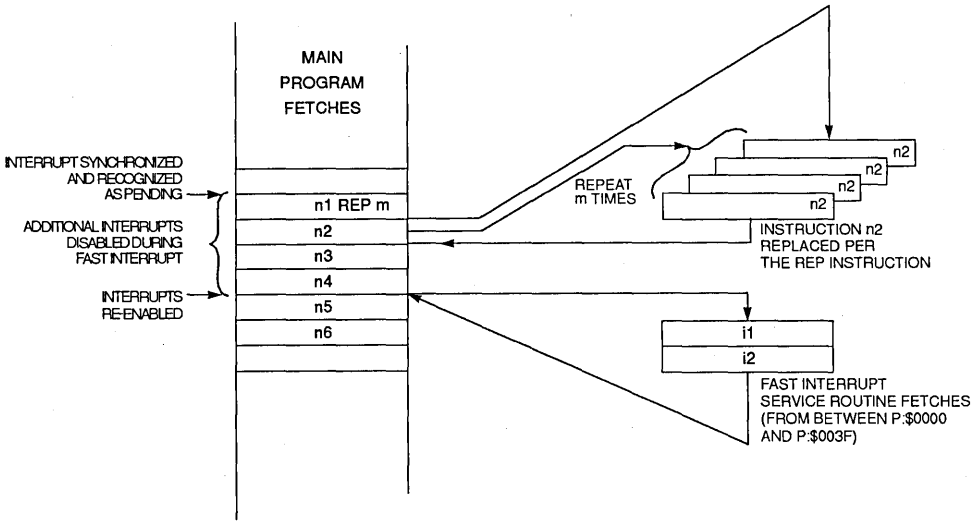
7.4 RESET PROCESSING STATE

The processor enters the reset processing state when a hardware reset occurs and the external RESET pin is asserted. The reset state:

1. resets internal peripheral devices;
2. sets the modifier registers to $FFFF;
3. clears the interrupt priority register;
4. sets the BCR to $FFFF, thereby inserting 15 wait states in all external memory accesses;
5. clears the stack pointer;
6. clears the scaling mode, trace mode, loop flag, double precision multiply mode, and condition code bits of the SR, and sets the interrupt mask bits of the SR;
7. clears the data ROM enable bit, the stop delay bit, and the internal Y memory disable bit, and
8. the DSP remains in the reset state until the RESET pin is deasserted.

When the processor deasserts the reset state:

9. it loads the chip operating mode bits of the OMR from the external mode select pins (MODA, MODB, MODC), and
10. begins program execution at program memory address defined by the state of bits MODA, MODB, and MODC in the OMR. The first instruction must be fetched and then decoded before executing. Therefore, the first instruction execution is two instruction cycles after the first instruction fetch.



i = INTERRUPT INSTRUCTION
n = NORMAL INSTRUCTION

(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

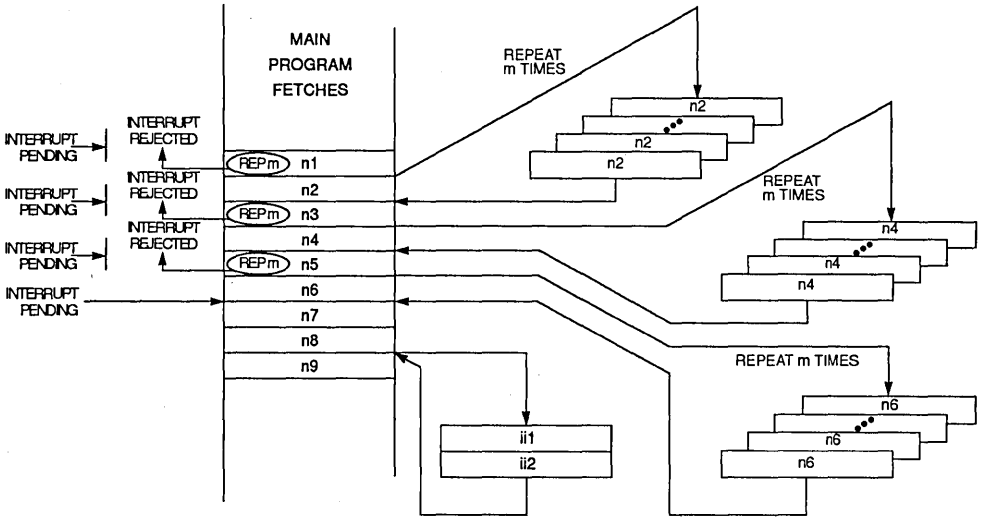
INTERRUPTS RE-ENABLED

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

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

(b) Program Controller Pipeline

Figure 7-13 Interrupting an REP Instruction



(a) Instruction Fetches from Memory

INTERRUPT SYNCHRONIZED AND RECOGNIZED AS PENDING

INTERRUPTS RE-ENABLED

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

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

(b) Program Controller Pipeline

Figure 7-14 Interrupting Sequential REP Instructions

7.5 WAIT PROCESSING STATE

The WAIT instruction brings the processor into the wait processing state which is one of two low power-consumption states. Asserting the OnCE's debug request pin releases the DSP from the wait state. In the wait state, the internal clock is disabled from all internal circuitry except the internal peripherals. All internal processing is halted until an unmasked interrupt occurs, the Debug Request pin of the OnCE is asserted, or the DSP is reset.

Figure 7-15 shows a WAIT instruction being fetched, decoded, and executed. It is fetched as n3 in this example and, during decode, is recognized as a WAIT instruction. The following instruction (n4) is aborted, and the internal clock is disabled from all internal circuitry except the internal peripherals. The processor stays in this state until an interrupt or reset is recognized. The response time is variable due to the timing of the interrupt with respect to the internal clock. Figure 7-15 shows the result of a fast interrupt bringing the processor out of the wait state. The two appropriate interrupt vectors are fetched and put in the instruction pipe. The next instruction fetched is n4, which had been aborted earlier. Instruction execution proceeds normally from this point.

7

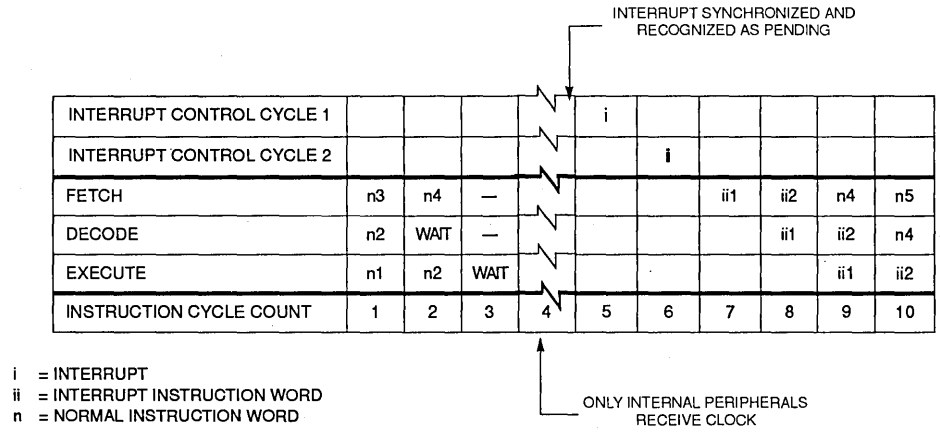
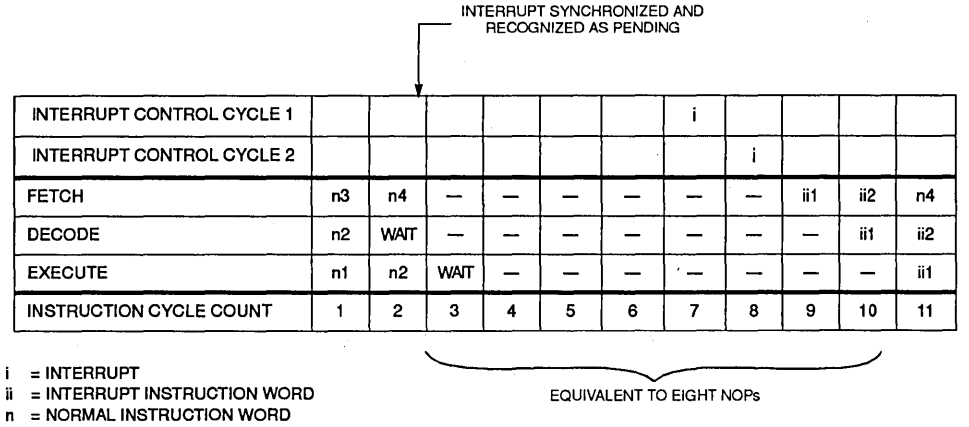


Figure 7-15 Wait Instruction Timing

Figure 7-16 shows an example of the WAIT instruction being executed at the same time that an interrupt is pending. Instruction n4 is aborted as before. The WAIT instruction causes a five-instruction-cycle delay from the time it is decoded, after which the interrupt is processed normally. The internal clocks are not turned off, and the net effect is that of executing eight NOP instructions between the execution of n2 and ii1.



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

Figure 7-16 Simultaneous Wait Instruction and Interrupt

7.6 STOP PROCESSING STATE

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

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

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

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

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

The STOP instruction is fetched in stop cycle 1 of Figure 7-17, decoded in stop cycle 2 (which is where it is first recognized as a stop command), and executed in stop cycle 3. The next instruction (n4) is fetched during stop cycle 2 but is not decoded in stop cycle 3 because, by that time, the STOP instruction prevents the decode. The processor stops the clock and enters the stop mode. The processor will stay in the stop mode until it is restarted.

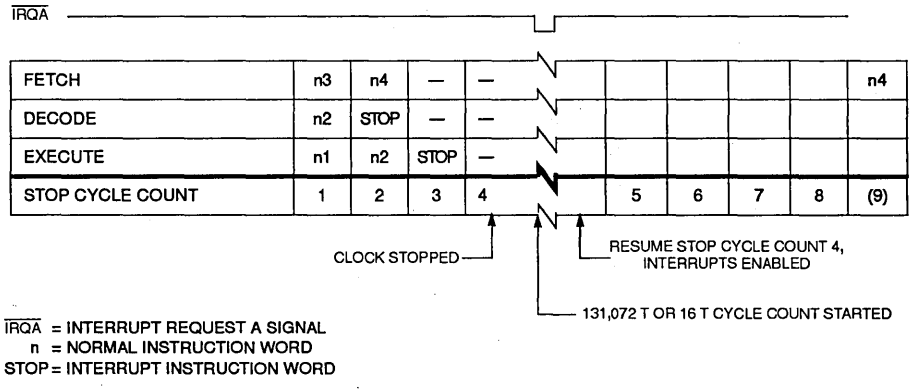


Figure 7-17 STOP Instruction Sequence

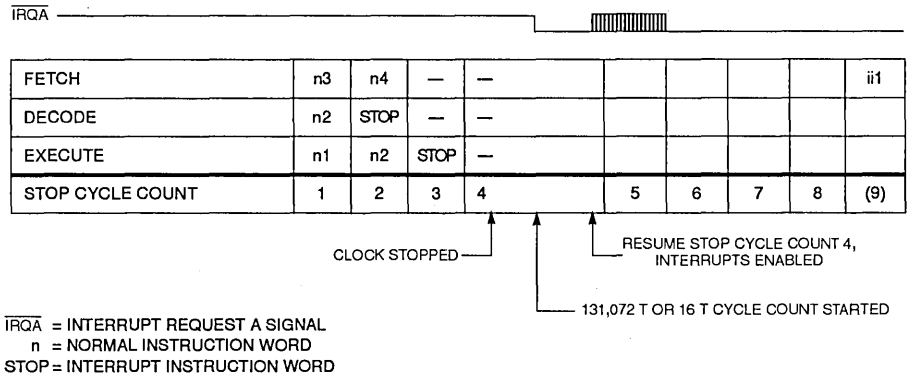


Figure 7-18 STOP Instruction Sequence Followed by $\overline{\text{IRQA}}$

7

Figure 7-18 shows the system being restarted by asserting the IRQA signal. If the exit from stop state was caused by a low level on the $\overline{\text{IRQA}}$ pin, then the processor will service the highest priority pending interrupt. If no interrupt is pending, then the processor resumes at the instruction following the STOP instruction that brought the processor into the stop state.

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

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

For example, assume that SD=0 so that the 128K T counter is used. During the 128K T count, the processor ignores interrupts until the last few counts and, at that time, begins to synchronize them. At the end of the 128K T cycle delay period, the chip restarts instruction processing, completes stop cycle 4 (interrupt arbitration occurs at this time), and executes stop cycles 5, 6, 7, and 8 (it takes 17T from the end of the 128K T delay to

the first instruction fetch). If the $\overline{\text{IRQA}}$ signal is released (pulled high) after a minimum of 4T but less than 128K T cycles, no $\overline{\text{IRQA}}$ interrupt will occur, and the instruction fetched after stop cycle 8 will be the next sequential instruction (n4 in Figure 7-18). An $\overline{\text{IRQA}}$ interrupt will be serviced as shown in Figure 7-18 if 1) the $\overline{\text{IRQA}}$ signal had previously been initialized as level sensitive, 2) $\overline{\text{IRQA}}$ is held low from the end of the 128K T cycle delay counter to the end of stop cycle count 8, and 3) no interrupt with a higher interrupt level is pending. If $\overline{\text{IRQA}}$ is not asserted during the last part of the STOP instruction sequence (6, 7, and 8) and if no interrupts are pending, the processor will refetch the next sequential instruction (n4). Since the $\overline{\text{IRQA}}$ signal is asserted (see Figure 7-18), the processor will recognize the interrupt and fetch and execute the instructions at P:$0008 and P:$0009 (the $\overline{\text{IRQA}}$ interrupt vector locations).

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

1. Define $\overline{\text{IRQA}}$ as level sensitive – an edge-triggered interrupt will not be serviced.
2. Define $\overline{\text{IRQA}}$ priority as higher than the other sources and higher than the program priority.
3. Ensure that no stack error or trace interrupts are pending.
4. Execute the STOP instruction and enter the stop state.
5. Recover from the stop state by asserting the $\overline{\text{IRQA}}$ pin and holding it asserted for the whole clock recovery time. If it is low, the IRQA vector will be fetched. Also, the user must ensure that NMI will not be asserted during these last three cycles; otherwise, NMI will be serviced before $\overline{\text{IRQA}}$ because NMI priority is higher.
6. The exact elapsed time for clock recovery is unpredictable. The external device that asserts $\overline{\text{IRQA}}$ must wait for some positive feedback, such as specific memory access or a change in some predetermined I/O pin, before deasserting $\overline{\text{IRQA}}$.

The STOP sequence totals 131,104 T cycles (if SD=0) or 48 T cycles (if SD=1) in addition to the period with no clocks from the stop fetch to the $\overline{\text{IRQA}}$ vector fetch (or next instruction). However, there is an additional delay if the internal oscillator is used. An indeterminate period of time is needed for the oscillator to begin oscillating and then stabilize its amplitude. The processor will still count 131,072 T cycles (or 16 T cycles), but

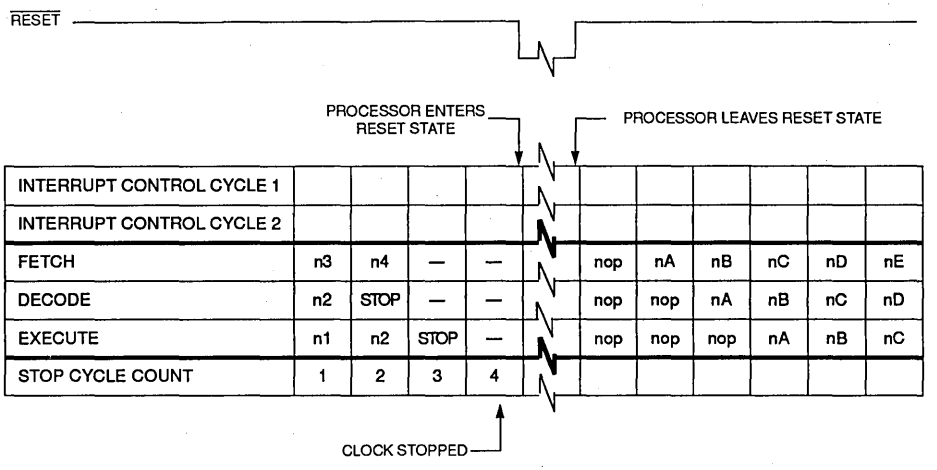
the period of the first oscillator cycles will be irregular; thus, an additional period of 19,000 T cycles should be allowed for oscillator irregularity (the specification recommends a total minimum period of 150,000 T cycles for oscillator stabilization). If an external oscillator is used that is already stabilized, no additional time is needed.

The PLL may be disabled or not when the chip enters the STOP state. If it is disabled and will not be re-enabled when the chip leaves the STOP state, the number of T cycles will be much greater because the PLL must regain lock.

If the STOP instruction is executed when the $\overline{\text{IRQA}}$ signal is asserted, the clock generator will not be stopped, but the four-phase clock will be disabled for the duration of the 128K T cycle (or 16 T cycle) delay count. In this case, the STOP looks like a 131,072 T + 35 T cycle (or 51 T cycle) NOP, since the STOP instruction itself is eight instruction cycles long (32 T) and synchronization of $\overline{\text{IRQA}}$ is 3T, which equals 35T.

A trace or stack error interrupt pending before entering the stop state is not cleared and will remain pending. During the clock stabilization delay, all peripheral and external interrupts are cleared and ignored (includes all SCI, SSI, HI, $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$, and NMI interrupts, but not trace or stack error). If the SCI, SSI, or HI have interrupts enabled in 1) their respective control registers and 2) in the interrupt priority register, then interrupts like SCI transmitter empty will be immediately pending after the clock recovery delay and will be serviced before continuing with the next instruction. If peripheral interrupts must be disabled, the user should disable them with either the control registers or the interrupt priority register before the STOP instruction is executed.

If $\overline{\text{RESET}}$ is used to restart the processor (see Figure 7-19), the 128K T cycle delay counter would not be used, all pending interrupts would be discarded, and the processor would immediately enter the reset processing state as described in Section 7.4. For example, the stabilization time recommended in the *DSP56001 Technical Data Sheet* for the clock ($\overline{\text{RESET}}$ should be asserted for this time) is only 50 T for a stabilized external clock but is the same 150,000 T for the internal oscillator. These stabilization times are recommended and are not imposed by internal timers or time delays. The DSP fetches instructions immediately after exiting reset. If the user wishes to use the 128K T (or 16 T) delay counter, it can be started by asserting $\overline{\text{IRQA}}$ for a short time (about two clock cycles).



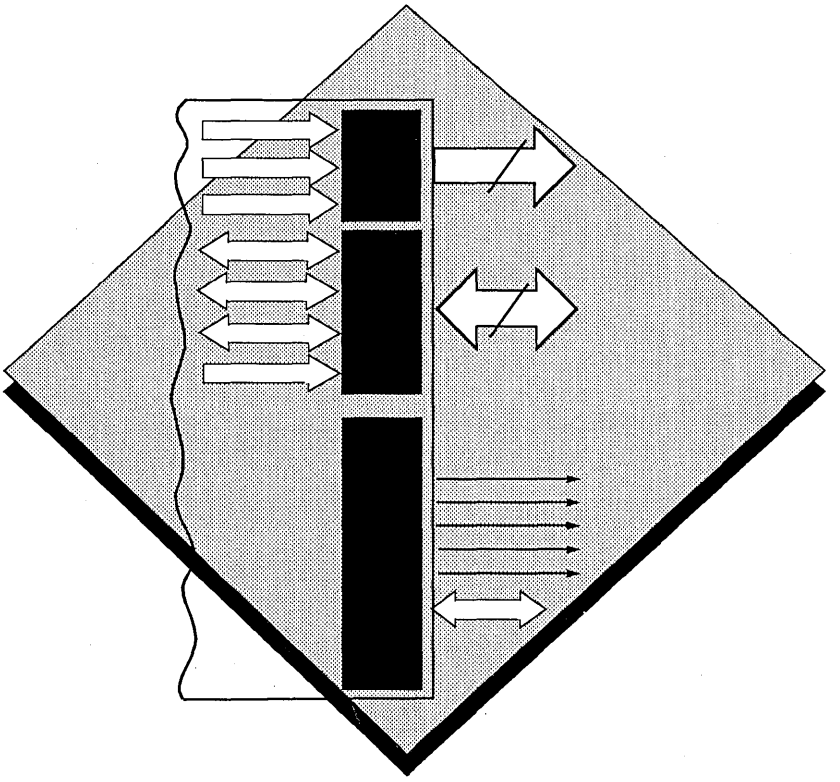
INTERRUPT CONTROL CYCLE 1										
INTERRUPT CONTROL CYCLE 2										
FETCH	n3	n4	—	—	nop	nA	nB	nC	nD	nE
DECODE	n2	STOP	—	—	nop	nop	nA	nB	nC	nD
EXECUTE	n1	n2	STOP	—	nop	nop	nop	nA	nB	nC
STOP CYCLE COUNT	1	2	3	4						

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

7

Figure 7-19 STOP Instruction Sequence Recovering with $\overline{\text{RESET}}$

SECTION 8
PORT A



SECTION CONTENTS

SECTION 8.1 PORT A OVERVIEW 3

SECTION 8.2 PORT A INTERFACE 3
8.2.1 Read/Write Control Signals 5
8.2.1.1 Program Memory Select (PS) 5
8.2.1.2 Data Memory Select (DS) 5
8.2.1.3 X/Y Select (X/Y) 5
8.2.2 Port A Address and Data Bus Signals 5
8.2.2.1 Address (A0–A15) 6
8.2.2.2 Data (D0–D23) 6
8.2.3 Port A Bus Control Signals 6
8.2.3.1 Read Enable (RD) 6
8.2.3.2 Write Enable (WR) 6
8.2.3.3 Port A Access Control Signals 6
8.2.4 Interrupt and Mode Control 6
8.2.5 Port A Wait States 6

8.1 PORT A OVERVIEW

Port A provides a versatile interface to external memory, allowing economical connection with fast memories, slow memories/devices, and multiple bus master systems. This section introduces the signals associated with this memory expansion port that are common among the members of the DSP56K family of processors which feature Port A. Certain characteristics, such as signaling, timing, and bus arbitration, vary between members of the processor family and are detailed in each device's own User's Manual.

Port A has two power-reduction features. It can access internal memory spaces, toggling only the external memory signals that need to change, and eliminate unneeded switching current. Also, if conditions allow the processor to operate at a lower memory speed, wait states can be added to the external memory access to significantly reduce power while the processor accesses those memories.

8.2 PORT A INTERFACE

The DSP56K processor can access one or more of its memory sources (X data memory, Y data memory, and program memory) while it executes an instruction. The memory sources may be either internal or external to the DSP. Three address buses (XAB, YAB, and PAB) and four data buses (XDB, YDB, PDB, and GDB) are available for internal memory accesses during one instruction cycle. Port A's one address bus and one data bus are available for external memory accesses. If all memory sources are internal to the DSP, one or more of the three memory sources may be accessed in one instruction cycle (i.e., program memory access or program memory access plus an X, Y, XY, or L memory reference). However, when one or more of the memories are external to the chip, memory references may require additional instruction cycles because only one external memory access can occur per instruction cycle.

If an instruction cycle requires more than one external access, the processor will make the accesses in the following priority: X memory, Y memory, and program memory. It takes one instruction cycle for each external memory access – i.e., one access can be executed in one instruction cycle, two accesses take two instruction cycles, etc. Since the external bus is only 24 bits wide, one XY or long external access will take two instruction cycles.

The port A external data bus shown in Figure 8-1 is 24 bits wide. The 16-bit address bus can sustain a rate of one memory access per instruction cycle (using no-wait-state memory which is discussed in Section 8.2.5.)

Figure 8-1 shows the port A signals divided into their three functional groups: address bus signals (A0-A15), data bus signals (D0-D15), and bus control. The bus control signals can

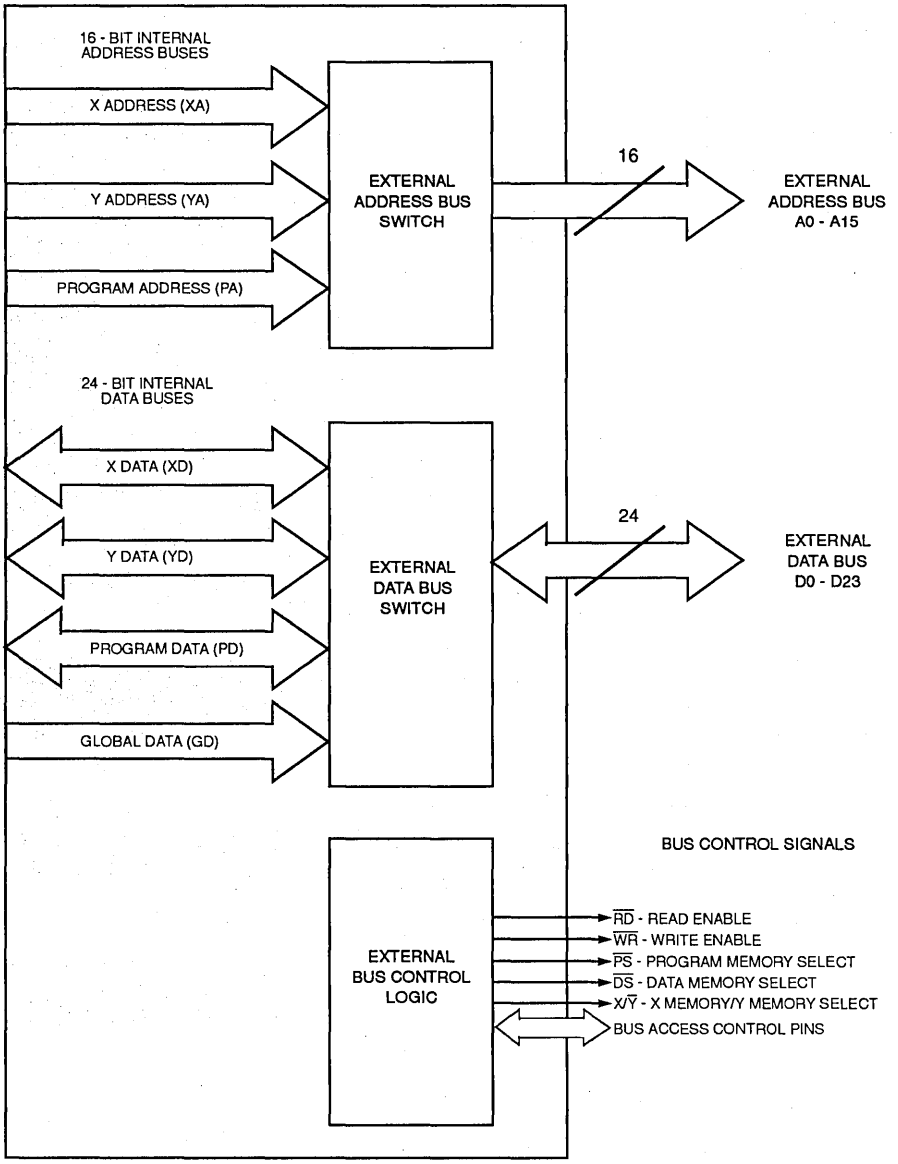


Figure 8-1 Port A Signals

be subdivided into three additional groups: read/write control ($\overline{RD}$ and $\overline{WR}$), address space selection (including program memory select ($\overline{PS}$), data memory select ($\overline{DS}$), and X/$\overline{Y}$ select) and bus access control.

The read/write controls are self-descriptive. They can be used as decoded read and write controls, or, the write signal can be used as the read/write control and the read signal can be used as an output enable (or data enable) control for the memory. Decoding in this fashion simplifies the connection to high-speed random-access memories (RAMs). The address space selection signals can be considered as additional address signals, which extend the addressable memory from 64K words to 192K words

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

8.2.1 Read/Write Control Signals

The following paragraphs describe the Port A read/write control signals. These pins are three-stated during reset and may require pullup resistors to prevent erroneous operation of a memory device or other external components.

8.2.1.1 Program Memory Select ($\overline{PS}$)

This three-state output is asserted only when external program memory is referenced.

8.2.1.2 Data Memory Select ($\overline{DS}$)

This three-state output is asserted only when external data memory is referenced.

8.2.1.3 X/$\overline{Y}$ Select (X/$\overline{Y}$)

This three-state output selects which external data memory space (X or Y) is referenced by $\overline{DS}$.

8.2.2 Port A Address and Data Bus Signals

The following paragraphs describe the Port A address and data bus signals. These pins are three-stated during reset and may require pullup resistors to prevent erroneous operation.

8.2.2.1 Address (A0–A15)

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

8.2.2.2 Data (D0–D23)

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

8.2.3 Port A Bus Control Signals

The following paragraphs describe the Port A bus control signals. The bus control signals provide the means to connect additional bus masters (which may be additional DSPs, microprocessors, direct memory access (DMA) controllers, etc.) to the port A bus. They are three-stated during reset and may require pullup resistors to prevent erroneous operation.

8.2.3.1 Read Enable ($\overline{RD}$)

This three-state output is asserted to read external memory on the data bus (D0–D23).

8.2.3.2 Write Enable ($\overline{WR}$)

This three-state output is asserted to write external memory on the data bus (D0–D23).

8.2.3.3 Port A Access Control Signals

Port A features a group of configurable pins that perform bus arbitration and bus access control. The pins, such as Bus Needed ($\overline{BN}$), Bus Request. ($\overline{BR}$), Bus Grant ($\overline{BG}$), Bus Wait ($\overline{WT}$), and Bus Strobe ($\overline{BS}$), and their designations differ between members of the DSP56K family and are explained in the respective devices' user manuals.

8.2.4 Interrupt and Mode Control

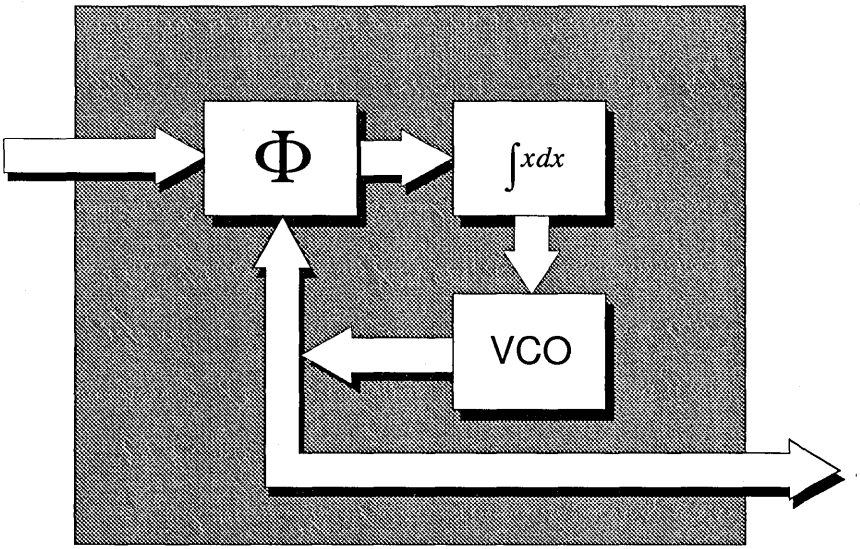
Port A features a pin set that selects the chip's operating mode and receives interrupt requests from external sources. The pins and their designations vary between members of the DSP56K family and are explained in the respective devices' user manuals.

8.2.5 Port A Wait States

The DSP56K processor features two methods to allow the user to accommodate slow memory by changing the port A bus timing. The first method uses the16-bit bus control register (BCR), which resides in X Data memory space. The BCR allows a fixed number of wait states to be inserted in a given memory access to all locations in any one of the four memory spaces: X, Y, P, and I/O. The second method uses the bus strobe/wait ($\overline{BS}$/

$\overline{WT}$) facility, which allows an external device to insert an arbitrary number of wait states when accessing either a single location or multiple locations of external memory or I/O space. Wait states are executed until the external device releases the DSP to finish the external memory cycle. An internal wait-state generator can be programmed using the BCR to insert up to15 wait states if it is known ahead of time that access to slower memory or I/O devices is required. A bus wait signal allows an external device to control the number of wait states (not limited to 15) inserted in a bus access operation.

SECTION 9
PLL CLOCK OSCILLATOR



SECTION CONTENTS

SECTION 9.1 PLL CLOCK OSCILLATOR INTRODUCTION 3

SECTION 9.2 PLL COMPONENTS 3
9.2.1 Phase Detector and Charge Pump Loop Filter 4
9.2.2 Voltage Controlled Oscillator (VCO) 5
9.2.3 Frequency Multiplier 5
9.2.4 Low Power Divider (LPD) 5
9.2.5 PLL Control Register (PCTL) 5
9.2.5.1 PCTL Multiplication Factor Bits (MF0-MF11) - Bits 0-11 5
9.2.5.2 PCTL Division Factor Bits (DF0-DF3) - Bits 12-15 6
9.2.5.3 PCTL XTAL Disable Bit (XTLD) - Bit 16 7
9.2.5.4 PCTL STOP Processing State Bit (PSTP) - Bit 17 7
9.2.5.5 PCTL PLL Enable Bit (PEN) - Bit 18 8
9.2.5.6 PCTL Clock Output Disable Bits (COD0-COD1) - Bits 19-20 8
9.2.5.7 PCTL Chip Clock Source Bit (CSRC) - Bit 21 9
9.2.5.8 PCTL CKOUT Clock Source Bit (CKOS) - Bit 22 9
9.2.5.9 PCTL Reserved Bit - Bit 23 9

SECTION 9.3 PLL PINS 9

SECTION 9.4 PLL OPERATION CONSIDERATIONS 11
9.4.1 Operating Frequency 11
9.4.2 Hardware Reset 11
9.4.3 Operation with PLL Disabled 12
9.4.4 Changing the MF0-MF11 Bits 12
9.4.5 Change of DF0-DF3 Bits 13
9.4.6 Loss of Lock 13
9.4.7 STOP Processing State 13
9.4.8 CKOUT Considerations 14
9.4.9 Synchronization Among EXTAL, CKOUT, and the Internal Clock 14

9.1 PLL CLOCK OSCILLATOR INTRODUCTION

The DSP56K family of processors (with the exception of the DSP56000 and DSP56001) features a PLL (phase-locked loop) clock oscillator in its central processing module, shown in Figure 9-2. The PLL allows the processor to operate at a high internal clock frequency using a low frequency clock input, a feature which offers two immediate benefits. Lower frequency clock inputs reduce the overall electromagnetic interference generated by a system, and the ability to oscillate at different frequencies reduces costs by eliminating the need to add additional oscillators to a system.

The PLL performs frequency multiplication to allow the processor to use almost any available external system clock for full speed operation, while also supplying an output clock synchronized to a synthesized internal core clock. It also improves the synchronous timing of the processor's external memory port, significantly reducing the timing skew between EXTAL and the internal chip phases. The PLL is unusual in that it provides a low power divider on its output, which can reduce or restore the chip operating frequency without losing the PLL lock

A DSP56K processor uses a four-phase clock for instruction execution which runs at the instruction execution rate. It can accept an external clock through the EXTAL input, or it can run on an internal oscillator, bypassing the PLL function, when the user connects an external crystal between XTAL and EXTAL. (The PLL need not be disabled when the processor accepts an external clock.)

9.2 PLL COMPONENTS

The PLL block diagram is shown below in Figure 9-1. The components of the PLL are described in the following sections.

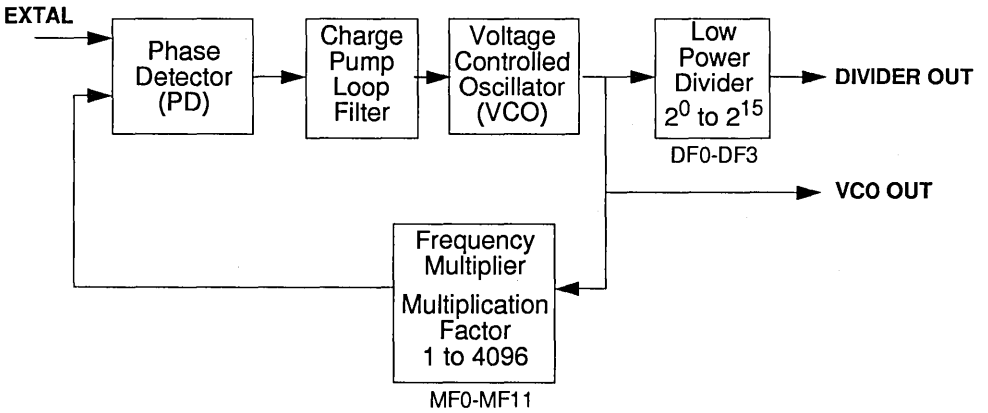


Figure 9-1 PLL Block Diagram

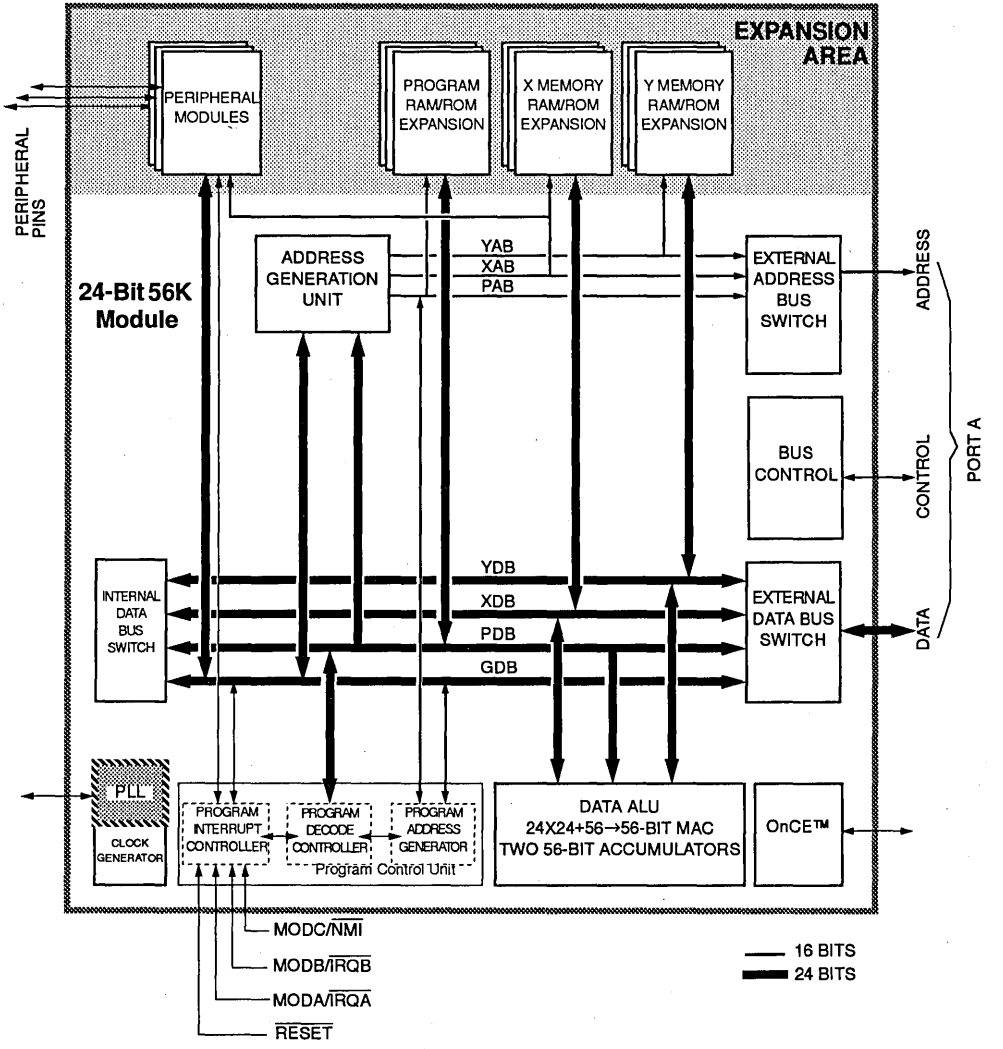


Figure 9-2 DSP56K Block Diagram

9.2.1 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 (described in Section 9.3) and determines the PLL operation. (See the appropriate Technical Data Sheet for more detailed information about a particular device's phase and frequency.)

After the PLL locks on to the proper phase/frequency, it reverts to the narrow bandwidth mode, which is useful for tracking small changes due to frequency drift of the EXTAL clock.

9.2.2 Voltage Controlled Oscillator (VCO)

The VCO can oscillate at frequencies from the minimum speed specified in a device's Technical Data Sheet (typically10 MHz) up to the device's maximum allowed clock input frequency.

9.2.3 Frequency Multiplier

Inside the PLL, the frequency multiplier divides the VCO output frequency by its division factor (n). If the frequency multiplier's output frequency is different from the EXTAL frequency, the charge pump loop filter generates an error signal. The error signal causes the VCO to adjust its frequency until the two input signals to the phase detector have the same phase and frequency. At this point (phase lock) the VCO will be running at n times the EXTAL frequency, where n is the multiplication factor for the frequency multiplier. The programmable multiplication factor ranges from 1 to 4096

9.2.4 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^{15}. Since the LPD is not in the closed loop of the PLL, changes in the divide factor will 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 the low power mode, it can immediately do so with no time needed for clock recovery or PLL lock.

9.2.5 PLL Control Register (PCTL)

The PLL control register (PCTL) is a 24-bit read/write register which directs the operation of the on-chip PLL. It is mapped into the processor's internal X memory at X:$FFFD. The PCTL control bits are described in the following sections.

9.2.5.1 PCTL Multiplication Factor Bits (MF0-MF11) - Bits 0-11

The Multiplication Factor Bits MF0-MF11 define the multiplication factor (MF) that will be applied to the PLL input frequency. The MF can be any integer from 1 to 4096. Table 9-1

11	10	9	8	7	6	5	4	3	2	1	0
MF11	MF10	MF9	MF8	MF7	MF6	MF5	MF4	MF3	MF2	MF1	MF0

23	22	21	20	19	18	17	16	15	14	13	12
**	CKOS	CSRC	COD1	COD0	PEN	PSTP	XTLD	DF3	DF2	DF1	DF0

** Reserved bits, read as zero, should be written with zero for future compatibility.

Figure 9-3 PLL Control Register (PCTL)

shows how to program the MF0-MF11 bits. The VCO will oscillate at a frequency of MF x F_{ext}, where F_{ext} is the EXTAL clock frequency. The multiplication factor must be chosen to ensure that the resulting VCO output frequency will lay in the range specified in the device's Technical Data Sheet. Any time a new value is written into the MF0-MF11 bits, the PLL will lose the lock condition. After a time delay, the PLL will relock. The MF0-MF11 bits are set to a pre-determined value during hardware reset; the value is implementation dependent and may be found in each DSP56K family member's user manual.

Table 9-1 Multiplication Factor Bits MF0-MF11

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

9

9.2.5.2 PCTL Division Factor Bits (DF0-DF3) - Bits 12-15

The Division Factor Bits DF0-DF3 define the divide factor (DF) of the low power divider. These bits specify any power of two divide factor in the range from 2^0 to 2^{15}. Table 9-2

shows the programming of the DF0-DF3 bits. Changing the value of the DF0-DF3 bits will not cause a loss of lock condition. Whenever possible, changes of the operating frequency of the chip (for example, to enter a low power mode) should be made by changing the value of the DF0-DF3 bits rather than changing the MF0-MF11 bits. For MF≤4, changing DF0-DF3 may lengthen 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 such synchronization is not guaranteed and the instruction cycle is not lengthened. Note that CKOUT is synchronized with the internal clock in all cases. The DF bits are cleared (division by one) by hardware reset.

Table 9-2 Division Factor Bits DF0-DF3

DF3-DF0	Division Factor DF
$0	2^0
$1	2^1
$2	2^2
•	•
•	•
$E	2^{14}
$F	2^{15}

9.2.5.3 PCTL XTAL Disable Bit (XTLD) - Bit 16

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 in the high ("1") state, 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 that XTLD be set (disabling XTAL) to minimize RFI noise and power dissipation. The XTLD bit is cleared by hardware reset.

9.2.5.4 PCTL STOP Processing State Bit (PSTP) - Bit 17

The PSTP bit controls the behavior of the PLL and of the on-chip crystal oscillator during the STOP processing state. When PSTP is set, the PLL and the on-chip crystal oscillator will remain operating while the chip is in the STOP processing state, enabling rapid recovery from the STOP state. When PSTP is cleared, the PLL and the on-chip crystal oscillator will be disabled when the chip enters the STOP processing. For minimal power consumption during the STOP state, at the cost of longer recovery time, PSTP should be

cleared. To enable rapid recovery when exiting the STOP state, at the cost of higher power consumption in the STOP state, PSTP should be set. PSTP is cleared by hardware reset.

9.2.5.5 PCTL PLL Enable Bit (PEN) - Bit 18

The PEN bit enables the PLL operation. When this bit is set, the PLL is enabled and the internal clocks will be 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 does not operate in order to minimize power consumption. The PLOCK pin is asserted when PEN is cleared. The PEN bit may be set by software but it cannot be reset by software. During hardware reset this bit receives the value of the PINIT pin. The only way to clear PEN is to hold the PINIT pin low during hardware reset.

A relationship exists between PSTP and PEN where PEN adjusts PSTP's control of the PLL operation. When PSTP is set and PEN (see Table 9-3) is cleared, the on-chip crystal oscillator remains operating in the STOP state, but the PLL is disabled. 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 9-3 PSTP and PEN Relationship

		Operation during STOP			
PSTP	**PEN**	**PLL**	**Oscillator**	**Recovery**	**Power Consumption**
0	x	Disabled	Disabled	long	minimal
1	0	Disabled	Enabled	rapid	lower
1	1	Enabled	Enabled	rapid	higher

9

9.2.5.6 PCTL Clock Output Disable Bits (COD0-COD1) - Bits 19-20

The COD0-COD1 bits control the output buffer of the clock at the CKOUT pin. Table 9-4 specifies the effect of COD0-COD1 on the CKOUT pin. When both COD0 and COD1 are set, the CKOUT pin is held in the high ("1") state. If the CKOUT pin is not connected to external circuits, it is recommended that both COD1 and COD0 be set (disabling clock output) to minimize RFI noise and power dissipation. If the CKOUT output is low at the moment the COD0-COD1 bits are set, it will complete the low cycle and then be disabled high. If the programmer re-enables the CKOUT output before it reaches the high logic level during the disabling process, the CKOUT operation will be unaffected. The COD0--COD1 bits are cleared by hardware reset.

Table 9-4 Clock Output Disable Bits COD0-COD1

COD1	COD0	CKOUT Pin
0	0	Clock Out Enabled, Full Strength Output Buffer
0	1	Clock Out Enabled, 2/3 Strength Output Buffer
1	0	Clock Out Enabled, 1/3 Strength Output Buffer
1	1	Clock Out Disabled

9.2.5.7 PCTL Chip Clock Source Bit (CSRC) - Bit 21

The CSRC bit specifies whether the clock for the chip is taken from the output of the VCO or is taken from the output of the Low Power Divider (LPD). When CSRC is set, the clock for the chip is taken from the VCO. When CSRC is cleared, the clock for the chip is taken from the output of the LPD. See Section 9.4.8 for restrictions. CSRC is cleared by hardware reset.

9.2.5.8 PCTL CKOUT Clock Source Bit (CKOS) - Bit 22

The CKOS bit specifies whether the CKOUT clock output is taken from the output of the VCO or is taken from the output of the Low Power Divider (LPD). When CKOS is set, the CKOUT clock output is taken from the VCO. When CKOS is cleared, the CKOUT clock output is taken from the output of the LPD. If the PLL is disabled (PEN=0), CKOUT is taken from EXTAL. See Section 9.4.8 for restrictions. CKOS is cleared by hardware reset.

9.2.5.9 PCTL Reserved Bit - Bit 23

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

9.3 PLL PINS

Some of the PLL pins need not be implemented. The specific PLL pin configuration for each DSP56K chip implementation is available in the respective device's user's manual. The following pins are dedicated to the PLL operation:

PVCC VCC dedicated to the analog PLL circuits. The voltage should be well regulated and the pin should be provided with an extremely low impedance path to the VCC power rail. PVCC should be bypassed to PGND by a 0.1µF capacitor located as close as possible to the chip package.

PGND GND dedicated to the analog PLL circuits. The pin should be provided with an extremely low impedance path to ground. PVCC should be bypassed to PGND by a 0.1µF capacitor located as close as possible to the chip package.

CLVCC VCC for the CKOUT output. The voltage should be well regulated and the pin should be provided with an extremely low impedance path to the VCC power rail. CLVCC should be bypassed to CLGND by a 0.1µF capacitor located as close as possible to the chip package.

CLGND GND for the CKOUT output. The pin should be provided with an extremely low impedance path to ground. CLVCC should be bypassed to CLGND by a 0.1µF capacitor located as close as possible to the chip package.

PCAP Off-chip capacitor for the PLL filter. One terminal of the capacitor is connected to PCAP while the other terminal is connected to PVCC. The capacitor value is specified in the particular device's Technical Data Sheet.

CKOUT This 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 CKOUT is derived from, and has the same frequency and duty cycle as, EXTAL.

Note: If the PLL is enabled and the multiplication factor is less than or equal to 4, then CKOUT is synchronized to EXTAL.

CKP This input pin defines the polarity of the CKOUT signal. Strapping CKP through a resistor to GND will make the CKOUT polarity the same as the EXTAL polarity. Strapping CKP through a resistor to VCC will make the CKOUT polarity the inverse of the EXTAL polarity. The CKOUT clock polarity is internally latched at the end of the hardware reset, so that any changes of the CKP pin logic state after deassertion of $\overline{\text{RESET}}$ will not affect the CKOUT clock polarity.

9

PINIT During the assertion of hardware reset, the value at the PINIT input pin is written into the PEN bit of the PLL control register. After hardware reset is deasserted, the PINIT pin is ignored.

PLOCK The PLOCK output originates from the Phase Detector. The chip asserts PLOCK when the PLL is enabled and has locked on the proper phase and frequency of EXTAL. The PLOCK output is deasserted by the chip if the PLL is enabled and has not locked on the proper phase and frequency. PLOCK is asserted if the PLL is disabled. PLOCK is a reliable indicator of the PLL lock state only after exiting the hardware reset state.

9.4 PLL OPERATION CONSIDERATIONS

The following paragraphs discuss PLL operation considerations.

9.4.1 Operating Frequency

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

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

where: DF is the division factor defined by the DF0-DF3 bits

F_{CHIP} is the chip operating frequency

F_{EXT} is the external input frequency to the chip at the EXTAL pin

F_{VCO} is the output frequency of the VCO

MF is the multiplication factor defined by the MF0-MF11 bits

The chip frequency is derived from the output of the low power divider. If the low power divider is bypassed, the equation is the same but the division factor should be assumed to be equal to one.

9.4.2 Hardware Reset

Hardware reset causes the initialization of the PLL. The following considerations apply:

1. The MF0-MF11 bits in the PCTL register are set to their pre-determined hardware reset value. The DF0-DF3 bits and the Chip Clock Source bit in the PCTL register are cleared. This causes the chip clock frequency to be equal to the external input frequency (EXTAL) multiplied by the multiplication factor defined by MF0-MF11.

2. During hardware reset assertion, the PINIT pin value is written into the PEN bit in the PCTL register. If the PINIT pin is asserted (setting PEN), the PLL acquires the proper phase/frequency. While hardware reset is asserted, the internal chip clock will be driven by the EXTAL pin until the PLL achieves lock (if enabled). If the PINIT pin is deasserted during hardware reset assertion, the PEN bit is cleared, the PLL is deactivated and the internal chip clock is driven by the EXTAL pin.

3. PLOCK is a reliable indicator of the PLL lock state only after exiting the hardware reset state.

4. For all input frequencies which would result in a VCO output frequency lower than the minimum specified in the device's Technical Data Sheet (typically 10 MHz), PINIT must be cleared during hardware reset, disabling PLL operation. Otherwise, proper operation of the PLL cannot be guaranteed. If the resulting VCO clock frequency would be less than the minimum and the user wishes to operate with the PLL enabled, the user must issue an instruction which loads the PCTL control register with a multiplication factor that would bring the VCO frequency above 10 MHz and would enable the PLL operation. Until this instruction is executed, the PLL is disabled, which may cause a large skew (<15nsec) between the external input clock and the internal processor clock. If internal low frequency of operation is desired with the PLL enabled, the VCO output frequency may be divided down by using the internal low power divider.
5. The CKP pin only affects the CKOUT clock polarity during the hardware reset state. At the end of the hardware reset state, the CKP state is internally latched.

9.4.3 Operation with PLL Disabled

1. If the PLL is disabled, the PLOCK pin is asserted.
2. If the PLL is disabled, the internal chip clock and CKOUT are driven from the EXTAL input.

9.4.4 Changing the MF0-MF11 Bits

Changes to the MF0-MF11 bits cause the following to occur:

1. The PLL will lose the lock condition, the PLOCK pin will be deasserted.
2. The PLL acquires the proper phase/frequency. Until this occurs the internal chip clock phases will be frozen. This ensures that the clock used by the chip is a clock that has reached a stable frequency.
3. When lock occurs, PLOCK is asserted and the PLL drives the internal chip clock and CKOUT.
4. While PLL has not locked, CKOUT is held low if CKP is cleared. CKOUT is held high if CKP is set.

9.4.5 Change of DF0-DF3 Bits

Changes to the DF0-DF3 bits do not cause a loss of lock condition. The internal clocks will immediately revert to the frequency prescribed by the new divide factor. For MF≤4, changing DF0-DF3 may lengthen the instruction cycle or CKOUT pulse following the PLL control register update in order to keep synchronization between EXTAL and the internal

chip clock. (Here, T3 is equal to the phase described by the new divide factor plus the time required to wait for a synchronizing pulse, which is less than 1.5ETc.) For MF>4, such synchronization is not guaranteed and the instruction cycle is not lengthened.

If the DF0-DF3 bits are changed by the same instruction that changes the MF0-MF11 bits, the LPD divider factor changes before the detection of the change in the multiplication factor. This means that the detection of loss of lock will occur after the LPD has started dividing by the new division factor.

9.4.6 Loss of Lock

The PLL distinguishes between cases where MF>4 and cases where MF≤4. If MF≤4, the PLL will detect loss of lock if a skew of 2.5 to 4.5 ns develops between the two clock inputs to the phase detector.

If MF>4, the PLL will detect loss of lock when there is a discrepancy of one clock cycle between the two clock inputs to the phase detector. When either of these two conditions occurs, the following also occur:

1. PLOCK will be deasserted, indicating that loss of lock condition has occurred.
2. The PLL will re-acquire the proper phase/frequency. When lock occurs, PLOCK will be asserted.

9.4.7 STOP Processing State

If the PSTP bit is cleared, executing the STOP instruction will disable the on-chip crystal oscillator and the PLL. In this state the chip consumes the least possible power. When recovering from the STOP state, the recovery time will be 16 or 64k external clock cycles (according to bit 6 in the Operating Mode Register) plus the time needed for the PLL to achieve lock.

If the PSTP bit is set, executing the STOP instruction will leave the on-chip crystal oscillator (if XTLD=0) and the PLL loop (if PEN=1) operating, but will disable the clock to the LPD and the rest of the DSP. When recovering from the STOP state, the recovery time will be only three clock cycles.

9.4.8 CKOUT Considerations

The CKOUT clock output is held high while disabled, which is also while the COD0-COD1 bits are set. If the CKOUT clock output is low at the moment the COD0-COD1 bits are set, then the CKOUT clock output will complete the low cycle and then be disabled high. If the programmer re-enables the CKOUT clock output before it reaches the high logic level during the disabling process, the CKOUT operation will be unaffected.

While the PLL is regaining lock, the CKOUT clock output remains at the same logic level it held when the PLL lost lock, which is when the clocks were frozen in the DSP.

When the chip enters the WAIT processing state, the core phases are disabled but CKOUT continues to operate. When PLL is disabled, CKOUT will be fed from EXTAL.

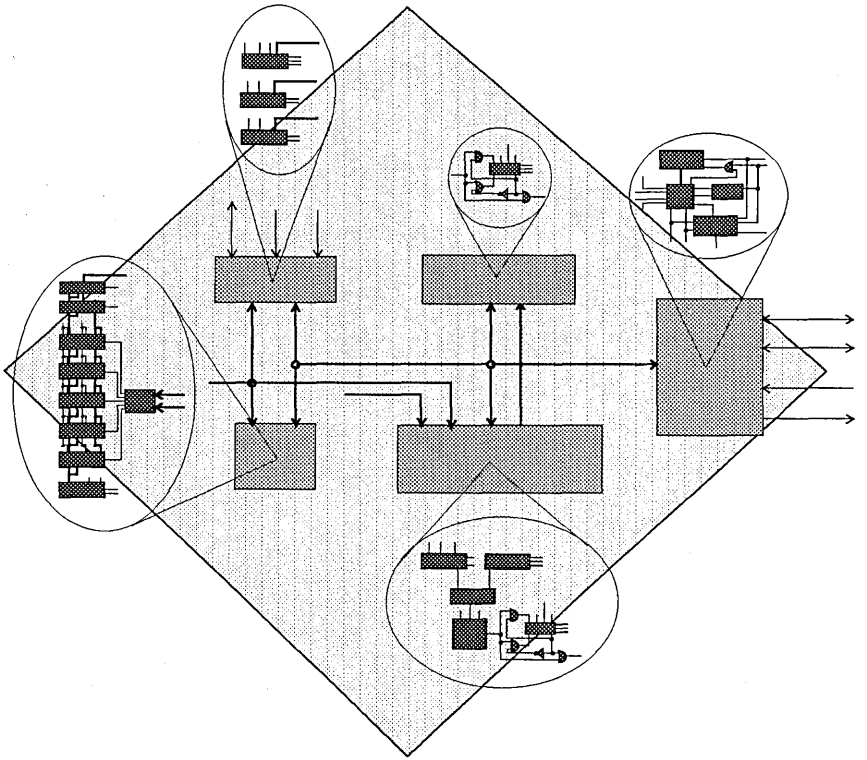
If DF>1 and CKOS≠CSRC, then the programmer must change either CKOS or CSRC before taking any action that causes the PLL to lose and subsequently regain lock, such as changing the multiplication factor, enabling PLL operation, or recovering from the STOP state with PSTP=0.

Any change of the CKOS or CSRC bits must be done while DF=1.

9.4.9 Synchronization Among EXTAL, CKOUT, and the Internal Clock

Low clock skew between EXTAL and CKOUT is guaranteed only if MF≤4. The synchronization between CKOUT and the internal chip activity and Port A timing is guaranteed in all cases where CKOS=CSRC and the bits have never differed from one another.

SECTION 10
ON-CHIP EMULATION (OnCE)



SECTION CONTENTS

SECTION 10.1 INTRODUCTION 3

SECTION 10.2 ON-CHIP EMULATION (OnCE) PINS 3

SECTION 10.3 OnCE CONTROLLER AND SERIAL INTERFACE 6

SECTION 10.4 OnCE MEMORY BREAKPOINT LOGIC 11

SECTION 10.5 OnCE TRACE LOGIC 13

SECTION 10.6 METHODS OF ENTERING THE DEBUG MODE 14

SECTION 10.7 PIPELINE INFORMATION AND GLOBAL DATA BUS REGISTER 16

SECTION 10.8 PROGRAM ADDRESS BUS HISTORY BUFFER 18

SECTION 10.9 SERIAL PROTOCOL DESCRIPTION 19

SECTION 10.10 DSP56K TARGET SITE DEBUG SYSTEM REQUIREMENTS 19

SECTION 10.11 USING THE OnCE 20

10.1 ON-CHIP EMULATION INTRODUCTION

The DSP56K on-chip emulation (OnCE) circuitry provides a sophisticated debugging tool that allows simple, inexpensive, and speed independent access to the processor's internal registers and peripherals. OnCE tells application programmers exactly what the status is within the registers, memory locations, buses, and even the last five instructions that were executed. OnCE capabilities are accessible through a standard set of pins which are the same on all of the members of the DSP56K processor family. Figure 10-1 shows the components of the OnCE circuitry. OnCE is shown as part of the DSP56K central processing module in Figure 10-2.

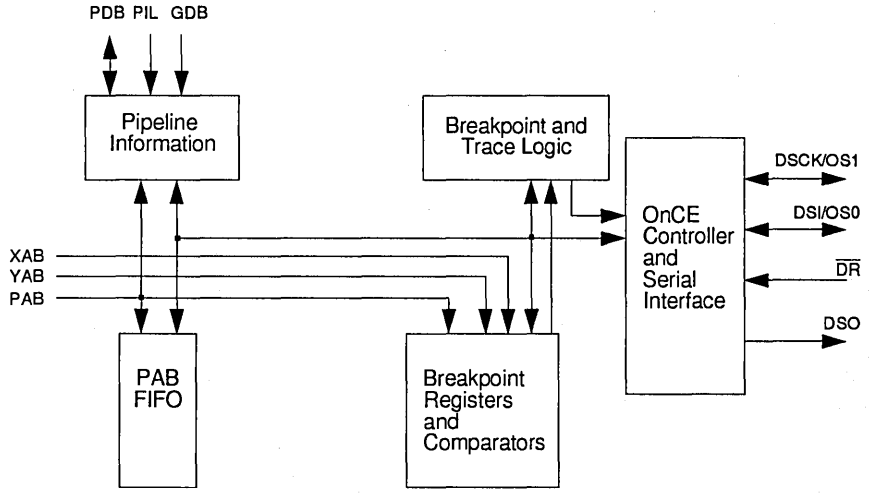


Figure 10-1 OnCE Block Diagram

10

10.2 ON-CHIP EMULATION (OnCE) PINS

The following paragraphs describe the OnCE pins associated with the OnCE controller and serial interface component shown in Figure 10-1.

10.2.1 Debug Serial Input/Chip Status 0 (DSI/OS0)

Serial data or commands are provided to the OnCE controller through the DSI/OS0 pin when it is an input. The data received on the DSI pin will be recognized only when the DSP56K has entered the debug mode of operation. Data is latched on the falling edge of the DSCK serial clock (described in Section 10.2.2). Data is always shifted into the OnCE serial port most significant bit (MSB) first. When the DSI/OS0 pin is an output, it works in conjunction with the OS1 pin to provide chip status information (see Table 10-1). The

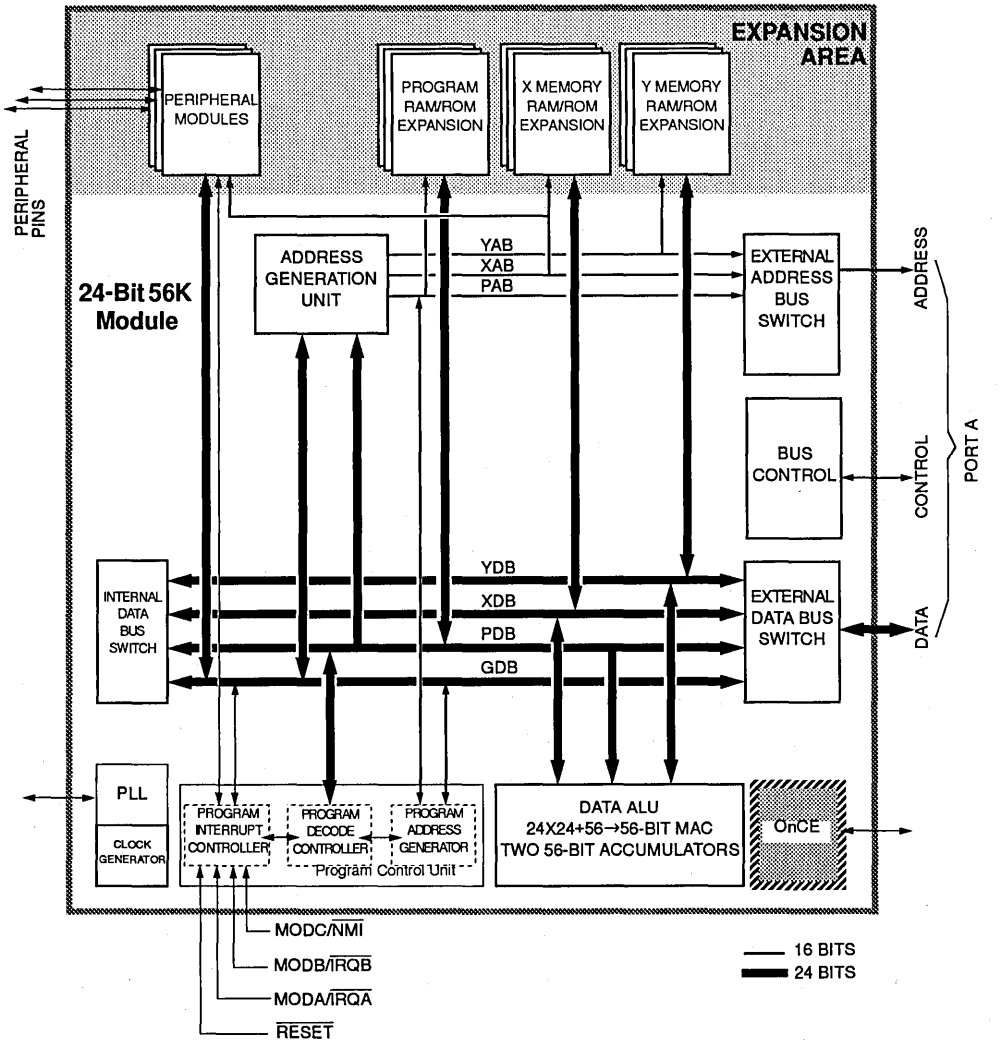


Figure 10-2 DSP56K Block Diagram

DSI/OS0 pin is an output when the processor is not in debug mode. When switching from output to input, the pin is three-stated. During hardware reset, this pin is defined as an output and it is driven low.

Note: To avoid possible glitches, an external pull-down resistor should be attached to this pin.

10.2.2 Debug Serial Clock/Chip Status 1 (DSCK/OS1)

The DSCK/OS1 pin supplies the serial clock to the OnCE when it is an input. The serial clock provides pulses required to shift data into and out of the OnCE serial port. (Data is clocked into the OnCE on the falling edge and is clocked out of the OnCE serial port on the rising edge.) The debug serial clock frequency must be no greater than 1/8 of the processor clock frequency. When an output, this pin, in conjunction with the OS0 pin, provides information about the chip status (see Table 10-1). The DSCK/OS1 pin is an output when the chip is not in debug mode. When switching from output to input, the pin is three-stated. During hardware reset, this pin is defined as an output and it is driven low.

Note: To avoid possible glitches, an external pull-down resistor should be attached to this pin.

Table 10-1 Chip Status Information

OS1	OS0	Status
0	0	Normal State
0	1	Stop or Wait State
1	0	Chip waits for bus mastership
1	1	Chip waits for end of memory wait states (due to $\overline{WT}$ assertion or BCR)

10.2.3 Debug Serial Output (DSO)

Serial data is read from the OnCE through the DSO pin, as specified by the last command received from the external command controller. Data is always shifted out the OnCE serial port most significant bit (MSB) first. Data is clocked out of the OnCE serial port on the rising edge of DSCK.

The DSO pin also provides acknowledge pulses to the external command controller. When the chip enters the debug mode, the DSO pin will be pulsed low to indicate (acknowledge) that the OnCE is waiting for commands. After receiving a read command, the DSO pin will be pulsed low to indicate that the requested data is available and the OnCE serial port is ready to receive clocks in order to deliver the data. After receiving a write command, the DSO pin will be pulsed low to indicate that the OnCE serial port is ready to receive the data to be written; after the data is written, another acknowledge pulse will be provided.

During hardware reset and when the processor is idle, the DSO pin is held high.

10.2.4 Debug Request Input ($\overline{DR}$)

The debug request input ($\overline{DR}$) allows the user to enter the debug mode of operation from the external command controller. When $\overline{DR}$ is asserted, it causes the DSP56K 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 DSI line. While in debug mode, the $\overline{DR}$ pin lets the user reset the OnCE controller by asserting it and deasserting it after receiving acknowledge. It may be necessary to reset the OnCE controller in cases where synchronization between the OnCE controller and external circuitry is lost. $\overline{DR}$ must be deasserted after the OnCE responds with an acknowledge on the DSO pin and before sending the first OnCE command. Asserting $\overline{DR}$ will cause the chip to exit the STOP or WAIT state.

10.3 OnCE CONTROLLER AND SERIAL INTERFACE

The OnCE Controller and Serial Interface contains the following blocks: OnCE command register, bit counter, OnCE decoder, and the status/control register. Figure 10-3 illustrates a block diagram of the OnCE controller and serial interface

10.3.1 OnCE Command Register (OCR)

The OCR is an 8-bit shift register that receives its serial data from the DSI pin. It holds the 8-bit commands to be used as input for the OnCE Decoder. The Command Register is shown in Figure 10-4.

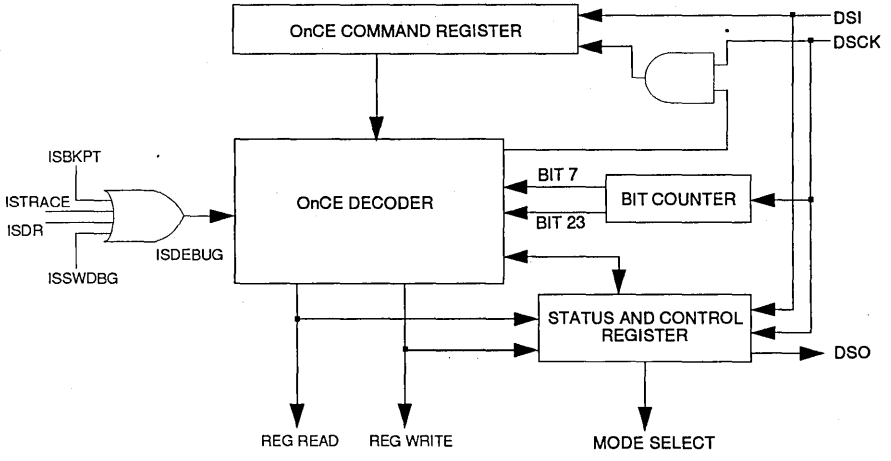


Figure 10-3 OnCE Controller and Serial Interface

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

Figure 10-4 OnCE Command Register

10.3.1.1 Register Select (RS4-RS0) Bits 0-4

The Register Select bits define which register is source (destination) for the read (write) operation. Table 10-2 indicates the OnCE register addresses.

Table 10-2 OnCE Register Addressing

RS4-RS0	Register Selected
00000	OnCE Status and Control Register (OSCR)
00001	Memory Breakpoint Counter (OMBC)
00010	Reserved
00011	Trace Counter (OTC)
00100	Reserved
00101	Reserved
00110	Memory Upper Limit Register (OMULR)
00111	Memory Lower Limit Register (OMLLR)
01000	GDB Register (OGDBR)
01001	PDB Register (OPDBR)
01010	PAB Register for Fetch (OPABFR)
01011	PIL Register (OPILR)
01100	Clear Memory Breakpoint Counter (OMBC)
01101	Reserved
01110	Clear Trace Counter (OTC)
01111	Reserved
10000	Reserved
10001	Program Address Bus FIFO and Increment Counter
10010	Reserved
10011	PAB Register for Decode (OPABDR)
101xx	Reserved
11xx0	Reserved
11x0x	Reserved
110xx	Reserved
11111	No Register Selected

10.3.1.2 Exit Command (EX) Bit 5

If the EX bit is set, the processor will leave the 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.

EX	Action
0	Remain in debug mode
1	Leave debug mode

10.3.1.3 Go Command (GO) Bit 6

If the GO bit is set, the chip will execute the instruction which resides in the PIL register. To execute the instruction, the processor leaves the debug mode, and the status is reflected in the OS0-OS1 pins. The processor will return to the debug mode immediately after executing the instruction if the EX bit is cleared. The processor 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.

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

10.3.1.4 Read/Write Command (R/W) Bit 7

The R/W bit specifies the direction of data transfer. The table below describes the options defined by the R/W bit.

R/W	Action
0	Write the data associated with the command into the register specified by RS4-RS0
1	Read the data contained in the register specified by RS4-RS0

10.3.2 OnCE Bit Counter (OBC)

The OBC is a 5-bit counter associated with shifting in and out the data bits. The OBC is incremented by the falling edges of the DSCK. The OBC is cleared during hardware reset and whenever the DSP56K acknowledges that the debug mode has been entered. The OBC supplies two signals to the OnCE Decoder: one indicating that the first 8 bits were

shifted in (so a new command is available) and the second indicating that 24 bits were shifted in (the data associated with that command is available) or that 24 bits were shifted out (the data required by a read command was shifted out).

10.3.3 OnCE Decoder (ODEC)

The ODEC supervises the entire OnCE activity. It receives as input the 8-bit command from the OCR, two signals from OBC (one indicating that 8 bits have been received and the other that 24 bits have been received), and two signals indicating that the processor was halted. The ODEC generates all the strobes required for reading and writing the selected OnCE registers.

10.3.4 OnCE Status and Control Register (OSCR)

The Status and Control Register is a 16-bit register used to select the events that will put the chip in debug mode and to indicate the reason for entering debug mode. The control bits are read/write while the status bits are read only. See Figure 10-5.

10.3.4.1 Memory Breakpoint Control (BC0-BC3) Bits 0-3

These control bits enable memory breakpoints. They allow memory breakpoints to occur when a memory address is within the low and high memory address registers and will select whether the breakpoint will be recognized for read, write, or fetch (program space) accesses. These bits are cleared on hardware reset. See Table 10-3 for the definition of the BC0-BC3 bits.

When BC3-BC0=0001, program memory breakpoints are enabled for any **fetch** access to the program space (true and false fetches, fetches of 2nd word, etc.). Explicit program memory accesses resulting from MOVEP and MOVEM instructions to/from program memory space are ignored.

When BC3-BC0=0010, program memory breakpoints are enabled for any **read** access to the Program space (MOVEP and MOVEM instructions from P: memory space, true and false fetches, fetches of 2nd word, etc.). Explicit program memory write accesses resulting from MOVEP and MOVEM instructions to P: memory space are ignored.

15 11	10	9	8	7	6	5	4	3	2	1	0
*	TO	MBO	SWO	*	*	*	TME	BC3	BC2	BC1	BC0

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

Figure 10-5 OnCE Status and Control Register (OSCR)

When BC3-BC0=0011, program memory breakpoints are enabled for any **read or write** access to the Program space (any kind of MOVE, true and false fetches, fetches of second word, etc.).

When BC3-BC0=0100, program memory breakpoints are enabled only for **fetches of the first instruction word** of instructions that are actually executed. Aborted instructions and prefetched instructions that are discarded (such as jump targets that are not taken) are ignored by the breakpoint logic.

When BC3-BC0=0101, 0110 or 0111, program memory breakpoints are enabled only for explicit program memory access resulting from MOVEP or MOVEM instructions to/from P: memory space.

Table 10-3 Memory Breakpoint Control Table

BC3	BC2	BC1	BC0	DESCRIPTION
0	0	0	0	Breakpoint disabled
0	0	0	1	Breakpoint on any fetch (including aborted instructions)
0	0	1	0	Breakpoint on any P read (any fetch or move)
0	0	1	1	Breakpoint on any P access (any fetch, P move R/W)
0	1	0	0	Breakpoint on executed fetches only
0	1	0	1	Breakpoint on P space write
0	1	1	0	Breakpoint on P space read (no fetches)
0	1	1	1	Breakpoint on P space write or read (no fetches)
1	0	0	0	Reserved
1	0	0	1	Breakpoint on X space write
1	0	1	0	Breakpoint on X space read
1	0	1	1	Breakpoint on X space write or read
1	1	0	0	Reserved
1	1	0	1	Breakpoint on Y space write
1	1	1	0	Breakpoint on Y space read
1	1	1	1	Breakpoint on Y space write or read

10.3.4.2 Trace Mode Enable (TME) Bit 4

The TME control bit, when set, enables the Trace Mode of operation (see Section 10.5). This bit is cleared on hardware reset.

10.3.4.3 Reserved (Bits 5-7, 11-15)

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

10.3.4.4 Software Debug Occurrence (SWO) Bit 8

This read-only status bit is set when the processor enters debug mode of operation as a result of the execution of the DEBUG or DEBUGcc instruction with condition true. This bit is cleared on hardware reset or when leaving the debug mode with the GO and EX bits set.

10.3.4.5 Memory Breakpoint Occurrence (MBO) Bit 9

This read-only status bit is set when a memory breakpoint occurs. This bit is cleared on hardware reset or when leaving the debug mode with the GO and EX bits set.

10.3.4.6 Trace Occurrence (TO) Bit 10

This read-only status bit is set when the processor enters debug mode of operation, when the trace counter is zero and the trace mode has been armed. This bit is cleared on hardware reset or when leaving the debug mode with the GO and EX bits set.

10.4 OnCE MEMORY BREAKPOINT LOGIC

Memory breakpoints may be set on program memory or data memory locations. Also, the breakpoint does not have to be in a specific memory address but within an 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 contains a latch for the addresses, registers that store the upper and lower address limit, comparators, and a breakpoint counter. Figure 10-6 illustrates the block diagram of the OnCE Memory Breakpoint Logic.

Address comparators help to determine where a program may be getting lost or when data is being written to areas that should not be written to. 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 in any operating mode. Memory accesses are monitored according to the contents of the OSCR.

The low address comparator will generate a logic true signal when the address on the bus is greater than or equal to the contents of the lower limit register. The high address comparator will generate a logic true signal when the address on the bus is less than or equal to the contents of the upper limit register. If the low address comparator and high address comparator both issue a logic true signal, the address is within the address range and the breakpoint counter is decremented if the contents are greater than zero. If zero, the counter is not decremented and the breakpoint exception occurs (ISBKPT asserted).

10.4.1 Memory Address Latch (OMAL)

The Memory Address Latch is a 16-bit register that latches the PAB, XAB or YAB on every instruction cycle according to the BC3-BC0 bits in OSCR.

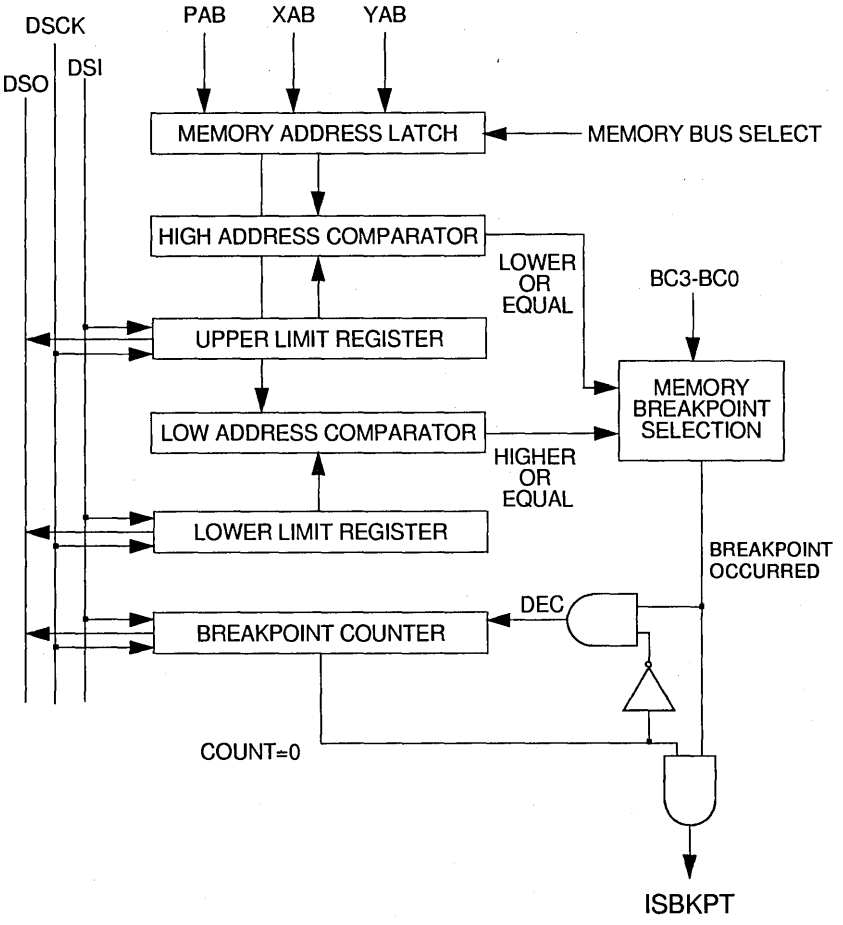


Figure 10-6 OnCE Memory Breakpoint Logic

10

10.4.2 Memory Upper Limit Register (OMULR)

The 16-bit Memory Upper Limit Register stores the memory breakpoint upper limit. The OMULR can be read or written through the OnCE serial interface. Before enabling breakpoints, OMULR must be loaded by the external command controller.

10.4.3 Memory Lower Limit Register (OMLLR)

The 16-bit Memory Lower Limit Register stores the memory breakpoint lower limit. The OMLLR can be read or written through the OnCE serial interface. Before enabling breakpoints, OMLLR must be loaded by the external command controller.

10.4.4 Memory High Address Comparator (OMHC)

The OMHC compares the current memory address (stored in OMAL) with the OMULR contents. If OMULR is higher than or equal to OMAL then the comparator delivers a signal indicating that the address is lower than or equal to the upper limit.

10.4.5 Memory Low Address Comparator (OMLC)

The OMLC compares the current memory address (stored in OMAL) with the OMLLR contents. If OMLLR is lower than or equal to OMAL then the comparator delivers a signal indicating that the address is higher than or equal to the lower limit.

10.4.6 Memory Breakpoint Counter (OMBC)

The 24-bit OMBC 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 BC3-BC0 bits in the OSCR register and by the memory upper and lower limit registers. On each occurrence of the memory access event, the breakpoint counter is decremented. When the counter has reached the value of zero and a new occurrence takes place, the chip will enter the debug mode. The OMBC can be read, written, or cleared through the OnCE serial interface.

Anytime the upper or lower limit registers are changed, or a different breakpoint event is selected in the OSCR, the breakpoint counter must be written afterward. This assures that the OnCE breakpoint logic is reset and that no previous events will affect the new breakpoint event selected.

The breakpoint counter is cleared by hardware reset.

10.5 OnCE TRACE LOGIC

The OnCE trace logic allows the user to execute instructions in single or multiple steps before the chip returns to the debug mode and awaits OnCE commands from the debug serial port. (The OnCE trace logic is independent of the trace facility of the DSP56000/56001, which is operated through the trace interrupt discussed in Section 7.3.3.3, and started by setting the trace bit in the processor's status register discussed in Section 5.4.2.12). The OnCE trace logic block diagram is shown in Figure 10-7.

The trace counter allows more than one instruction to be executed in real time before the chip returns to the debug mode of operation. This feature helps the software developer debug sections of code which do not have a normal flow or are getting hung up in infinite loops. The trace counter also enables the user to count the number of instructions executed in a code segment.

To initiate 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 processor 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) will decrement the trace counter.

Upon decrementing the trace counter to zero, the processor will re-enter the debug mode, the trace occurrence bit TO in the OSCR will be set, and the DSO pin will be toggled to indicate that the processor has entered debug mode and is requesting service (ISTRACE asserted).

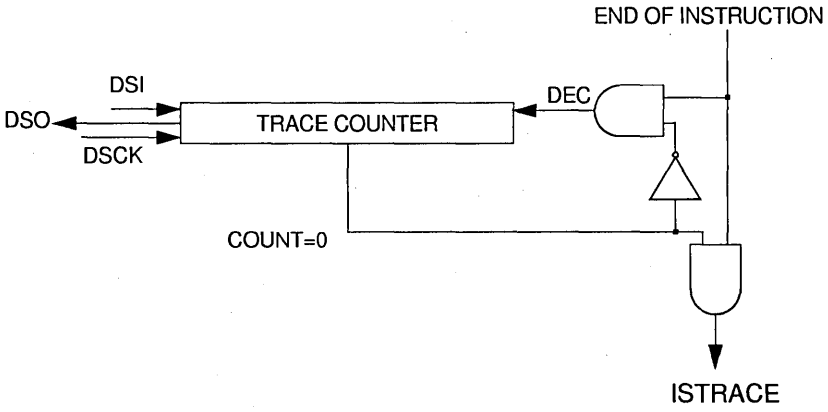


Figure 10-7 OnCE Trace Logic Block Diagram

10

10.5.1 Trace Counter (OTC)

The OTC is a 24-bit counter that can be read, written, or cleared through the OnCE serial interface. 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.

10.6 METHODS OF ENTERING THE DEBUG MODE

The chip acknowledges having entered the debug mode by pulsing low the DSO line, informing the external command controller that the chip has entered the debug mode and is waiting for commands.The following paragraphs discuss conditions that bring the processor into the debug mode.

10.6.1 External Debug Request During RESET

Holding the $\overline{DR}$ line asserted during the assertion of $\overline{RESET}$ causes the chip to enter the debug mode. After receiving the acknowledge, the external command controller must deassert the $\overline{DR}$ line before sending the first command. Note that in this case the chip does not execute any instruction before entering the debug mode.

10.6.2 External Debug Request During Normal Activity

Holding the $\overline{DR}$ line asserted during normal chip activity causes the chip to finish the execution of the current instruction and then enter the debug mode. After receiving the acknowledge, the external command controller must deassert the $\overline{DR}$ line before sending the first command. Note that in this case the chip completes the execution of the current instruction and stops after the newly fetched instruction enters the instruction latch. This process is the same for any newly fetched instruction including instructions fetched by the interrupt processing, or those that will be aborted by the interrupt processing.

10.6.3 External Debug Request During STOP

Asserting $\overline{DR}$ when the chip is in the stop state (i. e., has executed a STOP instruction) and keeping it asserted until an acknowledge pulse in DSO is produced causes the chip to exit the stop state and enter the debug mode. After receiving the acknowledge, the external command controller must deassert $\overline{DR}$ before sending the first command. Note that in this case, the chip completes the execution of the STOP instruction and halts after the next instruction enters the instruction latch.

10.6.4 External Debug Request During WAIT

Asserting $\overline{DR}$ when the chip is in the wait state (i. e., has executed a WAIT instruction) and keeping it asserted until an acknowledge pulse in DSO is produced causes the chip to exit the wait state and enter the debug mode. After receiving the acknowledge, the external command controller must deassert $\overline{DR}$ before sending the first command. Note that in this case, the chip completes the execution of the WAIT instruction and halts after the next instruction enters the instruction latch.

10.6.5 Software Request During Normal Activity

Upon executing the DEBUG or DEBUGcc instruction when the specified condition is true, the chip enters the debug mode after the instruction following the DEBUG instruction has entered the instruction latch.

10.6.6 Enabling Trace Mode

When the trace mode mechanism is enabled and the trace counter is greater than zero,

the trace counter is decremented after each instruction execution. The completed execution of an instruction when the trace counter is zero will cause the chip to enter the debug mode.

Note: Only instructions actually executed cause the trace counter to decrement, i.e. an aborted instruction will not decrement the trace counter and will not cause the chip to enter the debug mode.

10.6.7 Enabling Memory Breakpoints

When the memory breakpoint mechanism is enabled with a breakpoint counter value of zero, the chip enters the debug mode after completing the execution of the instruction that caused the memory breakpoint to occur. In case of breakpoints on executed program memory fetches, the breakpoint will be acknowledged immediately after the execution of the fetched instruction. In case of breakpoints on data memory addresses (accesses to X, Y or P memory spaces by MOVE instructions), the breakpoint will be acknowledged after the completion of the instruction following the instruction that accessed the specified address.

10.7 PIPELINE INFORMATION AND GLOBAL DATA BUS REGISTER

A number of on-chip registers store the chip pipeline status to restore the pipeline and resume normal chip activity upon return from the debug mode. Figure 10-8 shows the block diagram of the pipeline information registers with the exception of the program address bus (PAB) registers, which are shown in Figure 10-9.

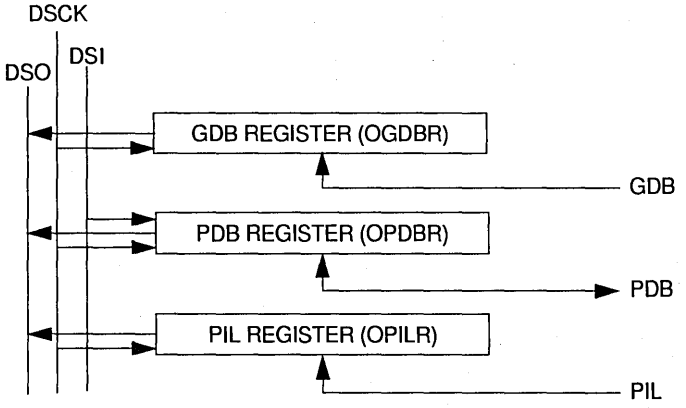


Figure 10-8 OnCE Pipeline Information and GDB Registers

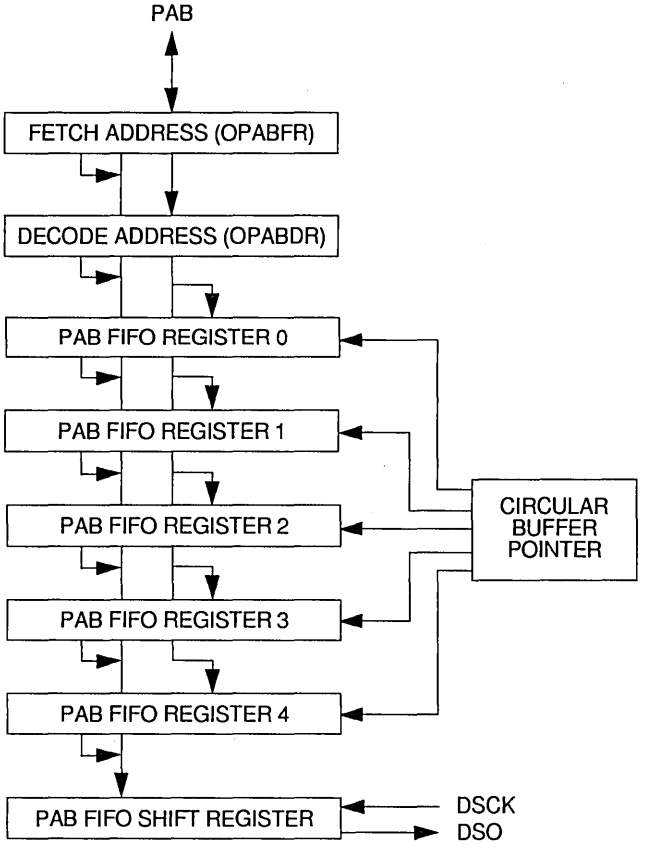


Figure 10-9 OnCE PAB FIFO

10.7.1 Program Data Bus Register (OPDBR)

The OPDBR is a 24-bit latch that stores the value of the program data bus which was generated by the last program memory access before the chip entered the debug mode. OPDBR can be read or written through the OnCE serial interface. It is affected by the operations performed during the debug mode and must be restored by the external command controller when the chip returns to normal mode.

10.7.2 Pipeline Instruction Latch Register (OPILR)

The 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 OnCE serial interface. 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. Since there is no direct write access to this register, this task is accomplished in the first write to OPDBR after entering the debug mode or after executing the GO command; the data from OPDBR is transferred to OPILR only in these cases.

10.7.3 Global Data Bus Register (OGDBR)

The OGDBR is a 24-bit latch that can only be read through the OnCE serial interface. OGDBR is not actually required from a pipeline status restore point of view but is required as a means of passing information between the chip and the external command controller. 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 will force the chip to execute an instruction that brings that information to OGDBR. Then, the contents of OGDBR will be delivered serially to the external command controller by the command "READ GDB REGISTER".

10.8 PROGRAM ADDRESS BUS HISTORY BUFFER

There are two read-only PAB registers which give pipeline information when the debug mode is entered. The OPABFR register tells which opcode address is in the fetch stage of the pipeline and OPABDR tells which opcode is in the decode stage. To ease debugging activity and keep track of program flow, a First-In-First-Out (FIFO) buffer stores the addresses of the last five instructions that were executed.

10.8.1 PAB Register for Fetch (OPABFR)

The OPABFR is a 16-bit register that stores the address of the last instruction that was fetched before the debug mode was entered. The OPABFR can only be read through the OnCE serial interface. This register is not affected by the operations performed during the debug mode.

10.8.2 PAB Register for Decode (OPABDR)

The OPABDR 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 if the chip would not have entered the debug mode. OPABDR can only be read through the serial interface. This register is not affected by the operations performed during the debug mode.

10.8.3 PAB FIFO

The PAB FIFO stores the addresses of the last five instructions that were executed. The FIFO is implemented as a circular buffer containing five 16-bit registers and one 3-bit counter. All the registers have the same address but any read access to the FIFO address will cause the counter to increment, making it point to the next FIFO register. The registers

are serially available to the external command controller through their common FIFO address. Figure 10-9 shows the block diagram of the PAB FIFO. The FIFO is not affected by the operations performed during the debug mode except for the FIFO pointer increment when reading the FIFO. When entering the debug mode, the FIFO counter will be pointing to the FIFO register containing the address of the oldest of the five executed instructions. The first FIFO read will obtain the oldest address and the following FIFO reads will get the other addresses from the oldest to the newest (the order of execution).

To ensure FIFO coherence, a complete set of five reads of the FIFO must be performed because each read increments the FIFO pointer, thus making it point to the next location. After five reads the pointer will point to the same location it pointed to before starting the read procedure.

10.9 SERIAL PROTOCOL DESCRIPTION

The following protocol permits an efficient means of communication between the OnCE's external command controller and the DSP56K chip. Before starting any debugging activity, the external command controller must wait for an acknowledge on the DSO line, indicating that the chip has entered the debug mode. The external command controller communicates with the chip by sending 8-bit commands that may be accompanied by 24 bits of data. Both commands and data are sent or received most significant bit first. After sending a command, the external command controller must wait for the processor to acknowledge execution of the command before it may send a new command.

When accessing OnCE 16-bit registers, the register contents appear in the 16 most significant bits in the 24-bit data field, and the 8 least significant bits are zeroed.

10.9.1 OnCE Commands

The OnCE commands may be classified as follows:

- read commands (when the chip will deliver the required data).
- write commands (when the chip will receive data and write the data in one of the OnCE registers).
- commands that do not have data transfers associated with them.

The commands are 8 bits long and have the format shown in Figure 10-4.

10.10 DSP56K TARGET SITE DEBUG SYSTEM REQUIREMENTS

A typical DSP56K debug environment consists of a target system where the DSP56K resides in the user defined hardware. The debug serial port interfaces to the external command controller over a 6-wire link which includes the 4 OnCE wires, a ground, and a reset wire. The reset wire is optional and is only used to reset the DSP56K and its associated circuitry.

The external command controller acts as the medium between the DSP56K target system and a host computer. The external command controller circuit acts as a DSP56K serial debug port driver and host computer command interpreter. The controller issues commands based on the host computer inputs from a user interface program which communicates with the user.

10.11 USING THE OnCE

The following notations are used:

ACK = Wait for acknowledge on the DSO pin

CLK = Issue 24 clocks to read out data from the selected register

10.11.1 Begin Debug Activity

Most of the debug activities have the following beginning:

1. ACK
2. Save pipeline information:
   a. Send command READ PDB REGISTER (10001001)
   b. ACK
   c. CLK
   d. Send command READ PIL REGISTER (10001011)
   e. ACK
   f. CLK
3. Read PAB FIFO and fetch/decode info (this step is optional):
   a. Send command READ PAB address for fetch (10001010)
   b. ACK
   c. CLK
   d. Send command READ PAB address for decode (10010011)
   e. ACK
   f. CLK
   g. Send command READ FIFO REGISTER and increment counter (10010001)
   h. ACK
   i. CLK
   j. Send command READ FIFO REGISTER and increment counter (10010001)

k. ACK

l. CLK

m. Send command READ FIFO REGISTER and increment counter (10010001)

n. ACK

o. CLK

p. Send command READ FIFO REGISTER and increment counter (10010001)

q. ACK

r. CLK

s. Send command READ FIFO REGISTER and increment counter (10010001)

t. ACK

u. CLK

10.11.2 Displaying A Specified Register

1. Send command WRITE PDB REGISTER, GO, no EX (01001001). The OnCE controller selects PDB as destination for serial data.
2. ACK
3. Send the 24-bit DSP56K opcode: "MOVE reg,x:OGDB"
   After 24 bits have been received, the PDB register drives the PDB. The OnCE controller releases the chip from the debug mode, the chip executes the MOVE instruction, and the contents of the register specified in the instruction are loaded in the GDB REGISTER. The signal that marks the end of the instruction returns the chip to the debug mode.
4. ACK
5. Send command READ GDB REGISTER (10001000)
   The OnCE controller selects GDB as source for serial data.
6. ACK
7. CLK

10.11.3 Displaying X Memory Area Starting From Address XXXX

This command uses R0 to minimize serial traffic.

1. Send command WRITE PDB REGISTER, GO, no EX (01001001).
   The OnCE controller selects PDB as destination for serial data.
2. ACK
3. Send the 24-bit DSP56K opcode: "MOVE R0,x:OGDB"

After 24 bits have been received the PDB register drives the PDB. The OnCE controller releases the chip from the debug mode and the contents of R0 are loaded in the GDB REGISTER. The signal that marks the end of the instruction returns the chip to the debug mode.

4. ACK
5. Send command READ GDB REGISTER (10001001)
   The OnCE controller selects GDB as source for serial data.
6. ACK
7. CLK
   The external command controller generates 24 clocks that shift out the contents of the GDB register. The value of R0 is thus saved and should be restored before exiting the debug mode.
8. Send command WRITE PDB REGISTER, no GO, no EX (00001001)
   OnCE controller selects PDB as destination for serial data.
9. ACK
10. Send the 24-bit DSP56K opcode: "MOVE #$xxxx,R0"
    After 24 bits have been received, the PDB register drives the PDB. The OnCE controller causes the processor to load the opcode.
11. ACK
12. Send command WRITE PDB REGISTER, GO, no EX (01001001)
    The OnCE controller selects PDB as destination for serial data.
13. ACK
14. Send the 24-bit 2nd word of: "MOVE #$xxxx,R0" (the xxxx field).
    After 24 bits have been received, the PDB register drives the PDB. The OnCE controller releases the chip from the debug mode and the instruction starts execution. The signal that marks the end of the instruction returns the chip to the debug mode.
15. ACK
16. Send command WRITE PDB REGISTER, GO, no EX (01001001)
    The OnCE controller selects PDB as destination for serial data.
17. ACK
18. Send the 24-bit DSP56K opcode: "MOVE X:(R0)+,x:OGDB"
    After 24 bits have been received, the PDB register drives the PDB. The OnCE controller releases the chip from the debug mode and the contents of X:(R0) are loaded in the GDB REGISTER. The signal that marks the end of the instruction returns the chip to the debug mode.

19. ACK
20. Send command READ GDB REGISTER (10001000)
    The OnCE controller selects GDB as source for serial data.
21. ACK
22. CLK
23. Send command NO REGISTER SELECTED, GO, no EX (01011111)
    The OnCE controller releases the chip from the debug mode and the instruction is executed again in a "REPEAT-like" fashion. The signal that marks the end of the instruction returns the chip to the debug mode.
24. ACK
25. Send command READ GDB REGISTER (10001000)
    The OnCE controller selects GDB as source for serial data.
26. ACK
27. CLK
28. Repeat from step 23 until the entire memory area is examined.
29. After finishing reading the memory, R0 should to be restored as follows.
30. Send command WRITE PDB REGISTER, no GO, no EX (00001001)
    OnCE controller selects PDB as destination for serial data.
31. ACK
32. Send the 24-bit DSP56K opcode: "MOVE #saved_r0,R0"
    After 24 bits have been received, the PDB register drives the PDB. The OnCE controller causes the processor to load the opcode.
33. ACK
34. Send command WRITE PDB REGISTER, GO, no EX (01001001)
    The OnCE controller selects PDB as destination for serial data.
35. ACK
36. Send the 24-bit second word of: "MOVE #saved_r0,R0" (the saved_r0 field).
    After 24 bits have been received, the PDB register drives the PDB. The OnCE controller releases the chip from the debug mode and the instruction starts execution. The signal that marks the end of the instruction returns the chip to the debug mode.
37. ACK

10.11.4 Executing a Single-Word DSP56K Instruction While in Debug Mode

1. Send command WRITE PDB REGISTER, GO, no EX (01001001).
   The OnCE controller selects PDB as destination for serial data.
2. ACK
3. Send the single-word 24-bit DSP56K opcode to be executed.
   After 24 bits have been received, the PDB register drives the PDB. The OnCE controller releases the chip from the debug mode and the chip executes the instruction. The signal that marks the end of the instruction returns the chip to the debug mode. Some DSP56K instructions should not be executed in this state: DO, REP, ILLEGAL or any opcode that is considered illegal, and DEBUG.
4. ACK

10.11.5 Executing a Two-Word DSP56K Instruction While in Debug Mode

1. Send command WRITE PDB REGISTER, no GO, no EX (00001001).
   The OnCE controller selects PDB as destination for serial data.
2. ACK
3. Send the first instruction word (24-bit DSP56K opcode)
   After 24 bits have been received, the PDB register drives the PDB. The OnCE controller causes the processor to load the opcode.
   Some DSP56K instructions should not be executed in this state: DO, REP, ILLEGAL or any opcode that is considered illegal, and DEBUG.
4. ACK
5. Send command WRITE PDB REGISTER, GO, no EX (01001001)
   The OnCE controller selects PDB as destination for serial data.
6. ACK
7. Send the second 24-bit instruction word.
   After 24 bits have been received, the PDB register drives the PDB. The OnCE controller releases the chip from the debug mode and the instruction starts execution. The signal that marks the end of the instruction returns the chip to the debug mode.
8. ACK

10.11.6 Returning from Debug Mode to Normal Mode

There are two cases for returning from the debug mode in a single processor:

- Control is returned to the program that was running before debug was initiated.
- Jump to a different program location is executed.

10.11.6.1 Case 1: Return To The Previous Program (Return To Normal Mode)

1. Send command WRITE PDB REGISTER, no GO, no EX (00001001)
   The OnCE controller selects the PDB as the destination for serial data. Also, the OnCE controller selects the on-chip PAB register as the source for the PAB bus.
2. ACK
3. Send the 24 bits of the saved PIL (instruction latch) value.
   After the 24 bits have been received, the PDB register drives the PDB. The OnCE controller causes the PIL to latch the PDB value. In this way, the PIL is restored to the same state as before entering the debug mode.
4. ACK
5. Send command WRITE PDB REGISTER, GO, EX (01101001)
   The OnCE controller selects PDB as destination for the serial data to follow.
6. ACK
7. Send the 24 bits of the saved PDB value.
   After the 24 bits have been received, the PDB register drives the PDB. In this way, the PDB is restored to the same state as before entering the debug mode. The EX bit causes the OnCE controller to release the chip from the debug mode and the status bits in OSCR are cleared. The GO bit causes the chip to start executing instructions.

10.11.6.2 Case 2: Jump To A New Program (Go From Address $xxxx)

1. Send command WRITE PDB REGISTER, no GO, no EX (00001001)
   The OnCE controller selects PDB as destination for serial data. Also, the OnCE controller selects the on-chip PAB register as the source for the PAB bus.
2. ACK
3. Send 24 bits of the opcode of a two-word jump instruction instead of the saved PIL value. After the 24 bits have been received, the PDB register drives the PDB. The OnCE controller causes the PIL to latch the PDB value. In this way, the instruction latch will contain the opcode of the jump instruction which will cause the change in the program flow.
4. ACK
5. Send command WRITE PDB REGISTER, GO, EX (01101001)
   The OnCE controller selects PDB as destination for serial data.
6. ACK
7. Send 24 bits of the jump target absolute address ($xxxxxx).
   After 24 bits have been received, the PDB register drives the PDB. In this way, the

PDB contains the second word of the jump as required for the jump instruction execution. The EX bit causes the OnCE controller to release the chip from the debug mode and the status bits in OSCR are cleared. The GO bit causes the chip to start executing the jump instruction which will then cause the chip to continue instruction execution from the target address. Note that the trace counter will count the jump instruction so the current trace counter may need to be corrected if the trace mode is enabled.

10.11.7 Debugging Multiprocessor Systems With a Single External Command Controller

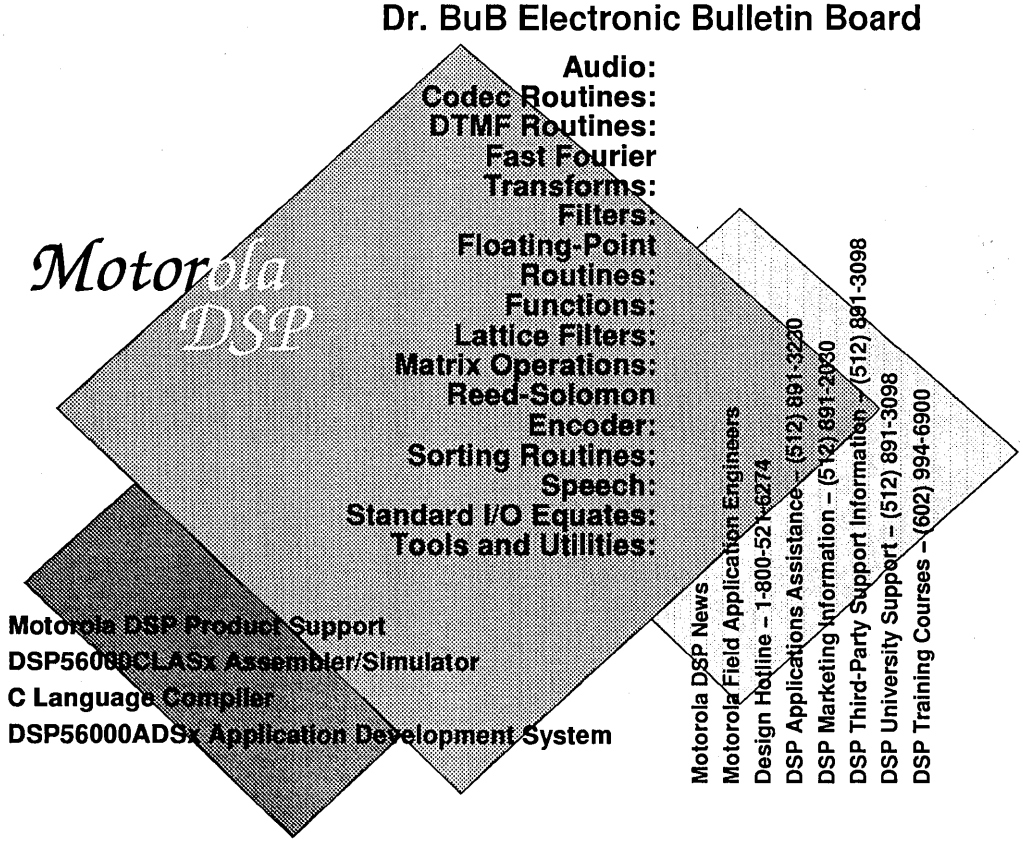
In multiprocessor systems, each processor may be individually debugged as described above. When simultaneous exit of the debug state is desired for more than one processor, each processor must first be loaded with the required PIL and PDB values where processing should proceed. This is accomplished by the following sequence as applied to each processor:

1. Send command WRITE PDB REGISTER, no GO, no EX (00001001)
   The OnCE controller selects PDB as destination for serial data. Also, the OnCE controller selects the on-chip PAB register as the source for the PAB bus.
2. ACK
3. Send 24 bits of either the opcode of a 2-word jump instruction or the saved PIL value. After the 24 bits have been received, the PDB register drives the PDB. The OnCE controller causes the PIL to latch the PDB value.
4. ACK
5. Send command WRITE PDB REGISTER, no GO, no EX (00001001)
   The OnCE controller selects PDB as destination for serial data.
6. ACK
7. Send 24 bits of either the jump target absolute address ($xxxxxx) or the saved PDB value. After 24 bits have been received, the PDB register drives the PDB.
8. ACK

At this point, all processors should have the required PIL and PDB values while still in debug mode. To return all processors to the normal execution state simultaneously, the following command should be issued to all processors in parallel:

9. Send command NO REGISTER SELECTED, GO, EX (01111111)
   The OnCE controller releases the chips from the debug mode and instruction execution is resumed.

SECTION 11
ADDITIONAL SUPPORT



11

SECTION CONTENTS

SECTION 11.1 USER SUPPORT 3

SECTION 11.2 MOTOROLA DSP PRODUCT SUPPORT 4
11.2.1 DSP56000CLASx Assembler/Simulator 4
11.2.2 Macro Cross Assembler Features: 4
11.2.3 Simulator Features: 5
11.2.4 DSP56KCCx Language Compiler Features: 5

SECTION 11.3 DSP56KADSx APPLICATION DEVELOPMENT SYSTEM 6
11.3.1 DSP56KADS Application Development System Hardware Features: 6
11.3.2 DSP56KADSx Application Development System Software Features: 6
11.3.3 Support Integrated Circuits: 7

SECTION 11.4 Dr. BuB ELECTRONIC BULLETIN BOARD 7

SECTION 11.5 MOTOROLA DSP NEWS 16

SECTION 11.6 MOTOROLA FIELD APPLICATION ENGINEERS 16

SECTION 11.7 DESIGN HOTLINE– 1-800-521-6274 16

SECTION 11.8 DSP HELP LINE – (512) 891-3230 16

SECTION 11.9 MARKETING INFORMATION– (512) 891-2030 16

SECTION 11.10 THIRD-PARTY SUPPORT INFORMATION – (512) 891-3098 16

SECTION 11.11 UNIVERSITY SUPPORT – (512) 891-3098 16

SECTION 11.12 TRAINING COURSES – (602) 897-3665 or (800) 521-6274 .17

SECTION 11.13 REFERENCE BOOKS AND MANUALS 17

11.1 USER SUPPORT

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

	Motorola	Third Party
Design	Data Sheets Application Notes Application Bulletins Software Examples	Data Acquisition Packages Filter Design Packages Operating System Software Simulator
Prototyping	Assembler Linker C Compiler Simulator Application Development System (ADS) In-Circuit Emulator Cable for ADS	Logic Analyzer with DSP56000/DSP56001 ROM Packages In-Circuit Emulators Data Acquisition Cards DSP Development System Cards Operating System Software Debug Software
Design Verification	Application Development System (ADS) In-Circuit Emulator Simulator	Data Acquisition Packages Logic Analyzer with DSP56000/DSP56001 ROM Packages Data Acquisition Cards DSP Development System Cards Application-Specific Development Tools Debug Software

The following is a partial list of the support available for the DSP56000/DSP56001. Additional information can be obtained through Dr. BuB or the appropriate support telephone service.

11.2 MOTOROLA DSP PRODUCT SUPPORT

- DSP56000CLASx Design-In Software Package which includes:
  - Relocatable Macro Assembler
  - Linker
  - Simulator (simulates single or multiple DSP56K processors))
  - Librarian
- DSP56KCCx GNU C Compiler
- DSP56000/DSP56001 Applications Development System (ADS)
- Support Integrated Circuits
- DSP Bulletin Board (Dr. BuB)
- Motorola DSP Newsletter
- Motorola Field Application Engineers (FAEs)
  See your local telephone directory for the Motorola Semiconductor Sector sales office telephone number.
- Design Hotline
- Applications Assistance
- Marketing Information
- Third-Party Support Information
- University Support Information

11.2.1 DSP56000CLASx Assembler/Simulator

The Macro Cross Assembler and Simulator run on:

1. IBM™ PCs (-386 or higher) under DOS 2.x and 3.x
2. Macintosh™ II under MAC OS 4.1 or later
3. SUN-4™ under UNIX™ BSD 4.2
4. NeXT™ under Mach

11

11.2.2 Macro Cross Assembler Features:

- Production of relocatable object modules compatible with linker program when in relocatable mode
- Production of absolute files compatible with simulator program when in absolute mode
- Supports full instruction set, memory spaces, and parallel data transfer fields of

the DSP56K family of processors

- Modular programming features: local labels, sections, and external definition/reference directives
- Nested macro processing capability with support for macro libraries
- Complex expression evaluation including boolean operators
- Built-in functions for data conversion, string comparison, and common transcendental math functions
- Directives to define circular and bit-reversed buffers
- Extensive error checking and reporting

11.2.3 Simulator Features:

- Simulation of all DSP56K family members
- Simulation of multiple DSP56Ks
- Linkable object code modules:
  - –Nondisplay simulator library
  - –Display simulator library
- C language source code for:
  - –Screen management functions
  - –Terminal I/O functions
  - –Simulation examples
- Single stepping through object programs
- Up to 99 conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Instruction, clock cycle, and histogram counters
- Session and/or command logging for later reference
- ASCII input/output files for peripherals
- Help-file and help-line display of simulator commands
- Loading and saving of files to/from simulator memory
- Macro command definition and execution
- Display enable/disable of registers and memory
- Hexadecimal/decimal/binary calculator

11.2.4 DSP56KCCx Language Compiler Features:

- GNU - ANSI Standard
- Structures/Unions
- Floating Point

- In-line assembler language code compatibility
- Full Function preprocessor for:
  - –Macro definition/expansion
  - –File Inclusion
  - –Conditional compilation
- Full error detection and reporting

11.3 DSP56KADSx APPLICATION DEVELOPMENT SYSTEM

11.3.1 DSP56KADS Application Development System Hardware Features:

- Processor speed independent
- Multiple (up to 8) application development module (ADM) support with programmable ADM addresses
- 8K/32Kx24 user-configurable RAM for DSP56K code development
- 1Kx24 monitor ROM expandable to 4Kx24
- 96-pin Euro-card connector making all DSP56K pins accessible
- In-circuit emulation capabilities when used with the DSP56KEMULTRCABL cable
- Separate berg pin connectors for alternate accessing of serial or host/DMA ports
- ADM can be used in stand-alone configuration
- No external power supply needed when connected to a host platform

11.3.2 DSP56KADSx Application Development System Software Features:

- Single/multiple stepping through DSP56K object programs
- Up to 99 conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Session and/or command logging for later reference
- Loading and saving files to/from ADM memory
- Macro command definition and execution
- Display enable/disable of registers and memory
- Debug commands supporting multiple ADMs
- Hexadecimal/decimal/binary calculator
- Host operating system commands from within ADS user interface program
- Multiple OS I/O file access from DSP56K object programs
- Fully compatible with the DSP56KCLASx design-in software package
- On-line help screens for each command and DSP56K register

11.3.3 Support Integrated Circuits:

- 8Kx24 Static RAM – MC56824
- DSP56ADC16 16-bit, sigma-delta 100-kHz analog-to-digital converter
- DSP56401 AES/EBU processor
- DSP56200 FIR filter

11.4 Dr. BuB ELECTRONIC BULLETIN BOARD

Dr. BuB is an electronic bulletin board which provides free source code for a large variety of topics that can be used to develop applications with Motorola DSP products. The software library contains files including FFTs, FIR filters, IIR filters, lattice filters, matrix algebra routines, companding routines, floating-point routines, and others. In addition, the latest product information and documentation (including information on new products and improvements to existing products) is posted. Questions about Motorola DSP products posted on Dr. BuB are answered promptly. Access to Dr. BuB is through calling **(512) 891-3771** using a modem set to **8 data bits, no parity, and 1 stop bit**. Dr. BuB will automatically set the data transfer rate to match your modem (9600, 4800, 2400, 1200 or 300 BPS).

A partial list of the software available on Dr. BuB follows.

Document ID	Version	Synopsis	Size
Audio:			
rvb1.asm	1.0	Easy-to-read reverberation routine	17056
rvb2.asm	1.0	Same as RVB1.ASM but optimized	15442
stereo.hlp	1.0	Help file for STEREO.ASM	620
dge.asm	1.0	Digital Graphic Equalizer code from	14880
Codec Routines:			
loglin.asm	1.0	Companded CODEC to linear PCM data conversion	4572
loglin.hlp		Help for loglin.asm	1479
loglint.asm	1.0	Test program for loglin.asm	2184
loglint.hlp		Help for loglint.asm	1993
linlog.asm	1.1	Linear PCM to companded CODEC data conversion	4847
linlog.hlp		Help for linlog.asm	1714
DTMF Routines:			
clear.cmd	1.0	Explained in read.me file	119
data.lod	1.0		421
det.asm	1.0	Subroutine used in IIR DTMF	5923
dtmf.asm	1.0	Main routine used in IIR DTMF	10685
dtmf.mem	1.0	Memory for DTMF routine	48
dtmfmstr.asm	1.0	Main routine for multichannel DTMF	7409
dtmfmstr.mem	1.0	Memory for multichannel DTMF routine	41
dtmftwo.asm	1.0		10256
ex56.bat	1.0		94
genxd.lod	1.0	Data file	183
genyd.lod	1.0	Data file	180
goertzel.asm	1.0	Goertzel routine	4393
goertzel.lnk	1.0	Link file for Goertzel routine	6954
goertzel.lst	1.0	List file for Goertzel routine	11600
load.cmd	1.0		46
tstgoert.mem	1.0	Memory for Goertzel routine	384

Document ID	Version	Synopsis	Size
sub.asm	1.0	Subroutine linked for use in IIR DTMF	2491
read.me	1.0	Instructions	738
Fast Fourier Transforms:			
sincos.asm	1.2	Sine-Cosine Table Generator for FFTs	1185
sincos.hlp		Help for sincos.asm	887
sinewave.asm	1.1	Full-Cycle Sine wave Table Generator Generator Macro	1029
sinewave.hlp		for sinewave.asm	1395
fftr2a.asm	1.1	Radix 2, In-Place, DIT FFT (smallest)	3386
fftr2a.hlp		Help for fftr2a.asm	2693
fftr2at.asm	1.1	Test Program for FFTs (fftr2a.asm)	999
fftr2at.hlp		Help for fftr2at.asm	563
fftr2b.asm	1.1	Radix 2, In-Place, DIT FFT (faster)	4290
fftr2b.hlp		Help for fftr2b.asm	3680
fftr2c.asm	1.2	Radix 2, In-Place, DIT FFT (even faster)	5991
fftr2c.hlp		Help for fftr2c.asm	3231
fftr2d.asm	1.0	Radix 2, In-Place, DIT FFT (using DSP56001 sine-cosine ROM tables)	3727
fftr2d.hlp		Help for fftr2d.asm	3457
fftr2dt.asm	1.0	Test program for fftr2d.asm	1287
fftr2dt.hlp		Help for fftr2dt.asm	614
fftr2e.asm	1.0	1024 Point, Non-In-Place, FFT (3.39ms)	8976
fftr2e.hlp		Help for fftr2e.asm	5011
fftr2et.asm	1.0	Test program for fftr2e.asm	984
fftr2et.hlp		Help for fftr2et.asm	408
dct1.asm	1.1	Discrete Cosine Transform using FFT	5493
dct1.hlp	1.1	Help file for dct1.asm	970
fftr2cc.asm	1.0	Radix 2, In-place Decimation-in-time complex FFT macro	6524
fftr2cc.hlp	1.0	Help file for fftr2cc.asm	3533

Document ID	Version	Synopsis	Size
fftr2cn.asm	1.0	Radix 2, Decimation-in-time Complex FFT macro with normally ordered input/output	6584
fftr2cn.hlp	1.0	Help file for fftr2cn.asm	2468
fftr2en.asm	1.0	1024 point, not-in-place, complex FFT macro with normally ordered input/output	9723
fftr2en.hlp	1.0	Help file for fftr2en.asm	4886
dhit1.asm	1.0	Routine to compute Hilbert transform in the frequency domain	1851
dhit1.hlp	1.0	Help file for dhit1.asm	1007
fftr2bf.asm	1.0	Radix-2, decimation-in-time FFT with block floating point	13526
fftr2bf.hlp	1.0	Help file for fftr2bf.asm	1578
fftr2aa.asm	1.0	FFT program for automatic scaling	3172
Filters:			
fir.asm	1.0	Direct Form FIR Filter	545
fir.hlp		Help for fir.asm	2161
firt.asm	1.0	Test program for fir.asm	1164
iir1.asm	1.0	Direct Form Second Order All Pole IIR Filter	656
iir1.hlp		Help for iir1.asm	1786
iir1t.asm	1.0	Test program for iir1.asm	1157
iir2.asm	1.0	Direct Form Second Order All Pole IIR Filter with Scaling	801
iir2.hlp		Help for iir2.asm	2286
iir2t.asm	1.0	Test program for iir2.asm	1311
iir3.asm	1.0	Direct Form Arbitrary Order All Pole IIR Filter	776
iir3.hlp		Help for iir3.asm	2605
iir3t.asm	1.0	Test program for iir3.asm	1309
iir4.asm	1.0	Second Order Direct Canonic IIR Filter (Biquad IIR Filter)	713
iir4.hlp		Help for iir4.asm	2255
iir4t.asm	1.0	Test program for iir4.asm	1202

Document ID	Version	Synopsis	Size
iir5.asm	1.0	Second Order Direct Canonic IIR Filter with Scaling (Biquad IIR Filter)	842
iir5.hlp		Help for iir5.asm	2803
iir5t.asm	1.0	Test program for iir5.asm	1289
iir6.asm	1.0	Arbitrary Order Direct Canonic IIR Filter	923
iir6.hlp		Help for iir6.asm	3020
iir6t.asm	1.0	Test program for iir6.asm	1377
iir7.asm	1.0	Cascaded Biquad IIR Filters	900
iir7.hlp		Help for iir7.asm	3947
iir7t.asm	1.0	Test program for iir7.asm	1432
lms.hlp	1.0	LMS Adaptive Filter Algorithm	5818
transiir.asm	1.0	Implements the transposed IIR filter	1981
transiir.hlp	1.0	Help file for transiir.asm	974
Floating-Point Routines:			
fpdef.hlp	2.0	Storage format and arithmetic representation definition	10600
fpcalls.hlp	2.1	Subroutine calling conventions	11876
fplist.asm	2.0	Test file that lists all subroutines	1601
fprevs.hlp	2.0	Latest revisions of floating-point lib	1799
fpinit.asm	2.0	Library initialization subroutine	2329
fpadd.asm	2.0	Floating point add	3860
fpsub.asm	2.1	Floating point subtract	3072
fpcmp.asm	2.1	Floating point compare	2605
fpmpy.asm	2.0	Floating point multiply	2250
fpmac.asm	2.1	Floating point multiply-accumulate	2712
fpdiv.asm	2.0	Floating point divide	3835
fpsqrt.asm	2.0	Floating point square root	2873
fpneg.asm	2.0	Floating point negate	2026
fpabs.asm	2.0	Floating point absolute value	1953
fpscale.asm	2.0	Floating point scaling	2127

Document ID	Version	Synopsis	Size
fpfix.asm	2.0	Floating to fixed point conversion	3953
fpfloat.asm	2.0	Fixed to floating point conversion	2053
fpceil.asm	2.0	Floating point CEIL subroutine	1771
fpfloor.asm	2.0	Floating point FLOOR subroutine	2119
durbin.asm	1.0	Solution for LPC coefficients	5615
durbin.hlp	1.0	Help file for DURBIN.ASM	2904
fpfrac.asm	2.0	Floating point FRACTION subroutine	1862
Functions:			
log2.asm	1.0	Log base 2 by polynomial approximation	1118
log2.hlp		Help for log2.asm	719
log2t.asm	1.0	Test program for log2.asm	1018
log2nrm.asm	1.0	Normalizing base 2 logarithm macro	2262
log2nrm.hlp		Help for log2nrm.asm	676
log2nrmt.asm	1.0	Test program for log2nrm.asm	1084
exp2.asm	1.0	Exponential base 2 by polynomial approximation	926
exp2.hlp		Help for exp2.asm	759
exp2t.asm	1.0	Test program for exp2.asm	1019
sqrt1.asm	1.0	Square Root by polynomial approximation, 7 bit accuracy	991
sqrt1.hlp		Help for sqrt1.asm	779
sqrt1t.asm	1.0	Test program for sqrt1.asm	1065
sqrt2.asm	1.0	Square Root by polynomial approximation, 10 bit accuracy	899
sqrt2.hlp		Help for sqrt2.asm	776
sqrt2t.asm	1.0	Test program for sqrt2.asm	1031
sqrt3.asm	1.0	Full precision Square Root Macro	1388
sqrt3.hlp		Help for sqrt3.asm	794
sqrt3t.asm	1.0	Test program for sqrt3.asm	1053
tli.asm	1.1	Linear table lookup/interpolation routine for function generation	3253
tli.hlp	1.1	Help for tli.asm	1510

Document ID	Version	Synopsis	Size
bingray.asm	1.0	Binary to Gray code conversion macro	601
bingrayt.asm	1.0	Test program for bingray.asm	991
rand1.asm	1.1	Pseudo Random Sequence Generator	2446
rand1.hlp		Help for rand1.asm	704
Lattice Filters:			
latfir1.asm	1.0	Lattice FIR Filter Macro	1156
latfir1.hlp		Help for latfir1.asm	6327
latfir1t.asm	1.0	Test program for latfir1.asm	1424
latfir2.asm	1.0	Lattice FIR Filter Macro (modified modulo count)	1174
latfir2.hlp		Help for latfir2.asm	1295
latfir2t.asm	1.0	Test program for latfir2.asm	1423
latiir.asm	1.0	Lattice IIR Filter Macro	1257
latiir.hlp		Help for latiir.asm	6402
latiirt.asm	1.0	Test program for latiir.asm	1407
latgen.asm	1.0	Generalized Lattice FIR/IIR Filter Macro	1334
latgen.hlp		Help for latgen.asm	5485
latgent.asm	1.0	Test program for latgen.asm	1269
latnrm.asm	1.0	Normalized Lattice IIR Filter Macro	1407
latnrm.hlp		Help for latnrm.asm	7475
latnrmt.asm	1.0	Test program for latnrm.asm	1595
Matrix Operations:			
matmul1.asm	1.0	[1x3][3x3]=[1x3] Matrix Multiplication	1817
matmul1.hlp		Help for matmul1.asm	527
matmul2.asm	1.0	General Matrix Multiplication, C=AB	2650
matmul2.hlp		Help for matmul2.asm	780
matmul3.asm	1.0	General Matrix Multiply-Accumulate, C=AB+Q	2815
matmul3.hlp	1.0	Help for matmul3.asm	865

Document ID	Version	Synopsis	Size
Reed-Solomon Encoder:			
readme.rs	1.0	Instructions for Reed-Solomon coding	5200
rscd.asm	1.0	Reed-Solomon coder for DSP56000 simulator	5822
newc.c	1.0	Reed-Solomon coder coded in C	4075
table1.asm	1.0	Include file for R-S coder	7971
table2.asm	1.0	Include file for R-S coder	4011
Sorting Routines:			
sort1.asm	1.0	Array Sort by Straight Selection	1312
sort1.hlp		Help for sort1.asm	1908
sort1t.asm	1.0	Test program for sort1.asm	689
sort2.asm	1.1	Array Sort by Heapsort Method	2183
sort2.hlp		Help for sort2.asm	2004
sort2t.asm	1.0	Test program for sort2.asm	700
Speech:			
lgsol1.asm	2.0	Leroux-Gueguen solution for PARCOR (LPC) coefficients	4861
lgsol1.hlp		Help for lgsol1.asm	3971
durbin1.asm	1.2	Durbin Solution for PARCOR (LPC) coefficients	6360
durbin1.hlp		Help for durbin1.asm	3616
adpcm.asm	1.0	32 kbps CCITT ADPCM Speech Coder	120512
adpcm.hlp	1.0	Help file for adpcm.asm	14817
adpcmns.asm	1.0	Nonstandard ADPCM source code	54733
adpcmns.hlp	1.0	Help file for adpcmns.asm	9952
Standard I/O Equates:			
ioequ.asm	1.1	Motorola Standard I/O Equate File	8774
ioequlc.asm	1.1	Lower Case Version of ioequ.asm	8788
intequ.asm	1.0	Standard Interrupt Equate File	1082
intequlc.asm	1.0	Lower Case Version of intequ.asm	1082

Document ID	Version	Synopsis	Size
Tools and Utilities:			
srec.c	4.10	Utility to convert DSP56000 OMF format to SREC.	38975
srec.doc	4.10	Manual page for srec.c.	7951
srec.h	4.10	Include file for srec.c	3472
srec.exe	4.10	Srec executable for IBM PC	22065
sloader.asm	1.1	Serial loader from the SCI port for the DSP56001	3986
sloader.hlp	1.1	Help for sloader.asm	2598
sloader.p	1.1	Serial loader s-record file for download to EPROM	736
parity.asm	1.0	Parity calculation of a 24-bit number in accumulator A	1641
parity.hlp	1.0	Help for parity.asm	936
parityt.asm	1.0	Test program for parity.asm	685
parityt.hlp	1.0	Help for parityt.asm	259
dspbug		Ordering information for free debug monitor for DSP56000/DSP56001	882
The following is a list of current DSP56200 related software:			
p1	1.0	Information on 56200 Filter Software	6343
p2	1.0	Interrupt Driven Adaptive Filter Flowchart.	10916
p3	1.0	"C" code implementation of p2	25795
p4	1.0	Polled I/O Adaptive Filter Flowchart	10361
p5	1.0	"C" code implementation of p4	24806
p6	1.1	Interrupt Driven Dual FIR Filter Flowchart.	9535
p7	1.0	"C" code implementation of p6	28489
p8	1.0	Polled I/O Dual FIR Filter Flowchart	9656
p9	1.0	"C" code implementation of p8	28525

11.5 MOTOROLA DSP NEWS

The Motorola DSP News is a quarterly newsletter providing information on new products, application briefs, questions and answers, DSP product information, third-party product news, etc. This newsletter is free and is available upon request by calling the marketing information phone number listed below.

11.6 MOTOROLA FIELD APPLICATION ENGINEERS

Information and assistance for DSP applications is available through the local Motorola field office. See your local telephone directory for telephone numbers or call (512)891-2030.

11.7 DESIGN HOTLINE– 1-800-521-6274

This is the Motorola number for information about any Motorola product.

11.8 DSP HELP LINE – (512) 891-3230

Design assistance for specific DSP applications is available by calling this number.

11.9 MARKETING INFORMATION– (512) 891-2030

Marketing information, including brochures, application notes, manuals, price quotes, etc., for Motorola DSP-related products is available by calling this number.

11.10 THIRD-PARTY SUPPORT INFORMATION – (512) 891-3098

Information about third-party manufacturers who use and support Motorola DSP products is available by calling this number. Third-party support includes:

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

Additional information is available on Dr. BuB and in DSP News.

11.11 UNIVERSITY SUPPORT – (512) 891-3098

Information concerning university support programs and university discounts for all Motorola DSP products is available by calling this number.

11.12 TRAINING COURSES – (602) 897-3665 or (800) 521-6274

There are two DSP56000 Family training courses available:

1. Introduction to the DSP5600X (MTTA5) is a 4.5-hour audio-tape course on the DSP56K Family architecture and programming.

2. Introduction to the DSP5600X (MTT31) is a four-day instructor-led course and laboratory which covers the details of the DSP5600X architecture and programming.

Additional information is available by writing to:

Motorola SPS Training and Technical Operations
Mail Drop EL524
P. O. Box 21007
Phoenix, Arizona 85036

or by calling the number above. A technical training catalog is available which describes these courses and gives the current training schedule and prices.

11.13 REFERENCE BOOKS AND MANUALS

A list of DSP-related books is included here as an aid for the engineer who is new to the field of DSP. This is a partial list of DSP references intended to help the new user find useful information in some of the many areas of DSP applications. Many of the books could be included in several categories but are not repeated.

General DSP:

ADVANCED TOPICS IN SIGNAL PROCESSING
Jae S. Lim and Alan V. Oppenheim
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

APPLICATIONS OF DIGITAL SIGNAL PROCESSING
A. V. Oppenheim
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978

DISCRETE-TIME SIGNAL PROCESSING
A. V. Oppenheim and R. W. Schafer
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989

DIGITAL PROCESSING OF SIGNALS THEORY AND PRACTICE
Maurice Bellanger
New York, NY: John Wiley and Sons, 1984

DIGITAL SIGNAL PROCESSING
Alan V. Oppenheim and Ronald W. Schafer
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975

DIGITAL SIGNAL PROCESSING: A SYSTEM DESIGN APPROACH
David J. DeFatta, Joseph G. Lucas, and William S. Hodgkiss
New York, NY: John Wiley and Sons, 1988

FOUNDATIONS OF DIGITAL SIGNAL PROCESSING AND DATA ANALYSIS
J. A. Cadzow
New York, NY: MacMillan Publishing Company, 1987

HANDBOOK OF DIGITAL SIGNAL PROCESSING
D. F. Elliott
San Diego, CA: Academic Press, Inc., 1987

INTRODUCTION TO DIGITAL SIGNAL PROCESSING
John G. Proakis and Dimitris G. Manolakis
New York, NY: Macmillan Publishing Company, 1988

MULTIRATE DIGITAL SIGNAL PROCESSING
R. E. Crochiere and L. R. Rabiner
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983

SIGNAL PROCESSING ALGORITHMS
S. Stearns and R. Davis
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

SIGNAL PROCESSING HANDBOOK
C.H. Chen
New York, NY: Marcel Dekker, Inc., 1988

SIGNAL PROCESSING – THE MODERN APPROACH
James V. Candy
New York, NY: McGraw-Hill Company, Inc., 1988

THEORY AND APPLICATION OF DIGITAL SIGNAL PROCESSING
Rabiner, Lawrence R., Gold and Bernard
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975

11

Digital Audio and Filters:

ADAPTIVE FILTER AND EQUALIZERS
B. Mulgrew and C. Cowan
Higham, MA: Kluwer Academic Publishers, 1988

ADAPTIVE SIGNAL PROCESSING
B. Widrow and S. D. Stearns
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985

ART OF DIGITAL AUDIO, THE
John Watkinson
Stoneham. MA: Focal Press, 1988

DESIGNING DIGITAL FILTERS
Charles S. Williams
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986

DIGITAL AUDIO SIGNAL PROCESSING AN ANTHOLOGY
John Strawn
William Kaufmann, Inc., 1985

DIGITAL CODING OF WAVEFORMS
N. S. Jayant and Peter Noll
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL FILTERS: ANALYSIS AND DESIGN
Andreas Antoniou
New York, NY: McGraw-Hill Company, Inc., 1979

DIGITAL FILTERS AND SIGNAL PROCESSING
Leland B. Jackson
Higham, MA: Kluwer Academic Publishers, 1986

DIGITAL SIGNAL PROCESSING
Richard A. Roberts and Clifford T. Mullis
New York, NY: Addison-Welsey Publishing Company, Inc., 1987

INTRODUCTION TO DIGITAL SIGNAL PROCESSING
Roman Kuc
New York, NY: McGraw-Hill Company, Inc., 1988

INTRODUCTION TO ADAPTIVE FILTERS
Simon Haykin
New York, NY: MacMillan Publishing Company, 1984

MUSICAL APPLICATIONS OF MICROPROCESSORS (Second Edition)
H. Chamberlin
Hasbrouck Heights, NJ: Hayden Book Co., 1985

C Programming Language:

C: A REFERENCE MANUAL
Samuel P. Harbison and Guy L. Steele
Prentice-Hall Software Series, 1987.

PROGRAMMING LANGUAGE - C
American National Standards Institute,
ANSI Document X3.159-1989
American National Standards Institute, inc., 1990

THE C PROGRAMMING LANGUAGE
Brian W. Kernighan, and Dennis M. Ritchie
Prentice-Hall, Inc., 1978.

Controls:

ADAPTIVE CONTROL
K. Astrom and B. Wittenmark
New York, NY: Addison-Welsey Publishing Company, Inc., 1989

ADAPTIVE FILTERING PREDICTION & CONTROL
G. Goodwin and K. Sin
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

AUTOMATIC CONTROL SYSTEMS
B. C. Kuo
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987

COMPUTER CONTROLLED SYSTEMS: THEORY & DESIGN
K. Astrom and B. Wittenmark
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL CONTROL SYSTEMS
B. C. Kuo
New York, NY: Holt, Reinholt, and Winston, Inc., 1980

DIGITAL CONTROL SYSTEM ANALYSIS & DESIGN
C. Phillips and H. Nagle
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

ISSUES IN THE IMPLEMENTATION OF DIGITAL FEEDBACK COMPENSATORS
P. Moroney
Cambridge, MA: The MIT Press, 1983

Graphics:

CGM AND CGI
D. B. Arnold and P. R. Bono
New York, NY: Springer-Verlag, 1988

COMPUTER GRAPHICS (Second Edition)
D. Hearn and M. Pauline Baker
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986

FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS
J. D. Foley and A. Van Dam
Reading MA: Addison-Wesley Publishing Company Inc., 1984

GEOMETRIC MODELING
Michael E. Morteson
New York, NY: John Wiley and Sons, Inc.

GKS THEORY AND PRACTICE
P. R. Bono and I. Herman (Eds.)
New York, NY: Springer-Verlag, 1987

ILLUMINATION AND COLOR IN COMPUTER GENERATED IMAGERY
Roy Hall
New York, NY: Springer-Verlag

POSTSCRIPT LANGUAGE PROGRAM DESIGN
Glenn C. Reid - Adobe Systems, Inc.
Reading MA: Addison-Wesley Publishing Company, Inc., 1988

MICROCOMPUTER DISPLAYS, GRAPHICS, AND ANIMATION
Bruce A. Artwick
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985

PRINCIPLES OF INTERACTIVE COMPUTER GRAPHICS
William M. Newman and Roger F. Sproull
New York, NY: McGraw-Hill Company, Inc., 1979

PROCEDURAL ELEMENTS FOR COMPUTER GRAPHICS
David F. Rogers
New York, NY: McGraw-Hill Company, Inc., 1985

RENDERMAN INTERFACE, THE
Pixar
San Rafael, CA. 94901

Image Processing:

DIGITAL IMAGE PROCESSING
William K. Pratt
New York, NY: John Wiley and Sons, 1978

DIGITAL IMAGE PROCESSING (Second Edition)
Rafael C. Gonzales and Paul Wintz
Reading, MA: Addison-Wesley Publishing Company, Inc., 1977

DIGITAL IMAGE PROCESSING TECHNIQUES
M. P. Ekstrom
New York, NY: Academic Press, Inc., 1984

DIGITAL PICTURE PROCESSING
Azriel Rosenfeld and Avinash C. Kak
New York, NY: Academic Press, Inc., 1982

SCIENCE OF FRACTAL IMAGES, THE
M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, H. O. Peitgen,
D. Saupe, and R. F. Voss
New York, NY: Springer-Verlag

Motorola DSP Manuals:

MOTOROLA DSP56000 LINKER/LIBRARIAN REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP56000 MACRO ASSEMBLER REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP56000 SIMULATOR REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP56000/DSP56001 USER'S MANUAL
Motorola, Inc.,1990.

Numerical Methods:

ALGORITHMS (THE CONSTRUCTION, PROOF, AND ANALYSIS OF PROGRAMS)
P. Berliout and P. Bizard
New York, NY: John Wiley and Sons, 1986

MATRIX COMPUTATIONS
G. H. Golub and C. F. Van Loan
John Hopkins Press, 1983

NUMERICAL RECIPES IN C - THE ART OF SCIENTIFIC PROGRAMMING
William H. Press, Brian P. Flannery,
Saul A. Teukolsky, and William T. Vetterling
Cambridge University Press, 1988

NUMBER THEORY IN SCIENCE AND COMMUNICATION
Manfred R. Schroeder
New York, NY: Springer-Verlag, 1986

Pattern Recognition:

PATTERN CLASSIFICATION AND SCENE ANALYSIS
R. O. Duda and P. E. Hart
New York, NY: John Wiley and Sons, 1973

CLASSIFICATION ALGORITHMS
Mike James
New York, NY: Wiley-Interscience, 1985
Spectral Analysis:

STATISTICAL SPECTRAL ANALYSIS, A NONPROBABILISTIC THEORY
William A. Gardner
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

THE FAST FOURIER TRANSFORM AND ITS APPLICATIONS
E. Oran Brigham
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

THE FAST FOURIER TRANSFORM AND ITS APPLICATIONS
R. N. Bracewell
New York, NY: McGraw-Hill Company, Inc., 1986

Speech:

ADAPTIVE FILTERS - STRUCTURES, ALGORITHMS, AND APPLICATIONS
Michael L. Honig and David G. Messerschmitt
Higham, MA: Kluwer Academic Publishers, 1984

DIGITAL CODING OF WAVEFORMS
N. S. Jayant and P. Noll
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL PROCESSING OF SPEECH SIGNALS
Lawrence R. Rabiner and R. W. Schafer
Englwood Cliffs, NJ: Prentice-Hall, Inc., 1978

LINEAR PREDICTION OF SPEECH
J. D. Markel and A. H. Gray, Jr.
New York, NY: Springer-Verlag, 1976

SPEECH ANALYSIS, SYNTHESIS, AND PERCEPTION
J. L. Flanagan
New York, NY: Springer-Verlag, 1972

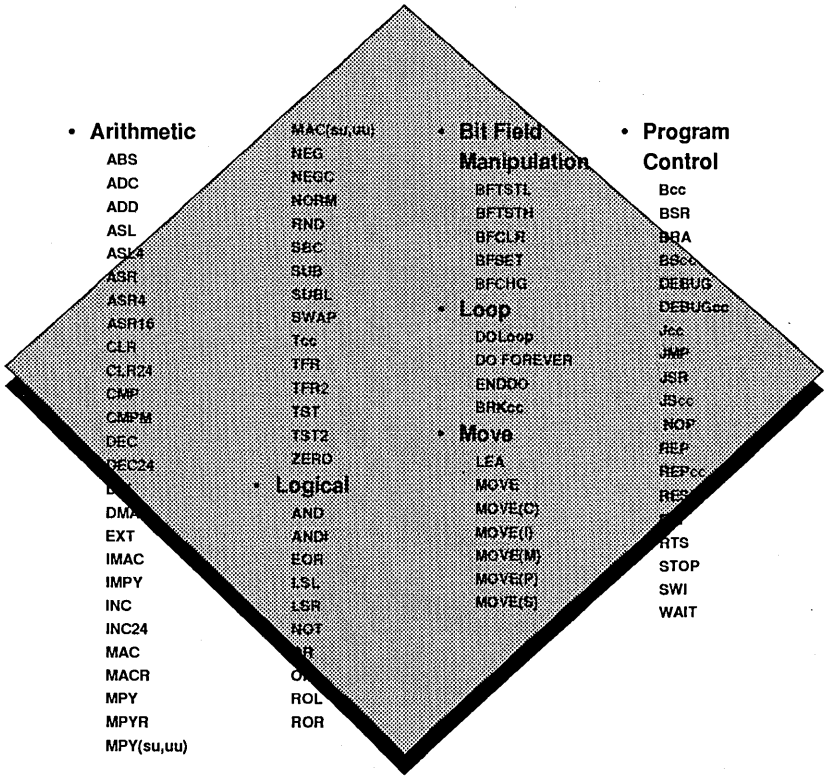
SPEECH COMMUNICATION – HUMAN AND MACHINE
D. O'Shaughnessy
Reading, MA: Addison-Wesley Publishing Company, Inc., 1987

Telecommunications:

DIGITAL COMMUNICATION
Edward A. Lee and David G. Messerschmitt
Higham, MA: Kluwer Academic Publishers, 1988

DIGITAL COMMUNICATIONS
John G. Proakis
New York, NY: McGraw-Hill Publishing Co., 1983

APPENDIX A
INSTRUCTION SET DETAILS



A

SECTION CONTENTS

SECTION A.1 APPENDIX A INTRODUCTION 3

SECTION A.2 INSTRUCTION GUIDE 3

SECTION A.3 NOTATION 4

SECTION A.4 ADDRESSING MODES 10
A.4.1 Addressing Mode Modifiers 13

SECTION A.5 CONDITION CODE COMPUTATION 14

SECTION A.6 PARALLEL MOVE DESCRIPTIONS 15

SECTION A.7 INSTRUCTION DESCRIPTIONS 17

SECTION A.8 INSTRUCTION TIMING 224

SECTION A.9 INSTRUCTION SEQUENCE RESTRICTIONS 235
A.9.1 Restrictions Near the End of DO Loops 236
A.9.2 Other DO Restrictions 237
A.9.3 ENDDO Restrictions 237
A.9.4 RTI and RTS Restrictions 238
A.9.5 SP and SSH/SSL Manipulation Restrictions 238
A.9.6 R, N, and M Register Restrictions 240
A.9.7 Fast Interrupt Routines 240
A.9.8 REP Restrictions 241

SECTION A.10 INSTRUCTION ENCODING 241
A.10.1 Partial Encodings for Use in Instruction Encoding 242
A.10.2 Instruction Encoding for the Parallel Move Portion of an Instruction 246
A.10.3 Instruction Encoding for Instructions Which Do Not Allow Parallel Moves 248
A.10.4 Parallel Instruction Encoding of the Operation Code 259

A

A.1 APPENDIX A INTRODUCTION

This appendix contains detailed information about each instruction in the DSP56K instruction set. It presents an instruction guide to help the user understand the individual instruction descriptions and follows with sections on notation and addressing modes. The instructions are then discussed in alphabetical order.

A.2 INSTRUCTION GUIDE

The following information is included in each instruction description with the goal of making each description self-contained:

1. **Name and Mnemonic:** The mnemonic is highlighted in **bold** type for easy reference.
2. **Assembler Syntax and Operation:** For each instruction syntax, the corresponding operation is symbolically described. If there are several operations indicated on a single line in the operation field, those operations do not necessarily occur in the order shown but are generally assumed to occur in parallel. If a parallel data move is allowed, it will be indicated in parenthesis in both the assembler syntax and operation fields. If a letter in the mnemonic is optional, it will be shown in parenthesis in the assembler syntax field.
3. **Description:** A complete text description of the instruction is given together with any special cases and/or condition code anomalies of which the user should be aware when using that instruction.
4. **Example:** An example of the use of the instruction is given. The example is shown in DSP56K assembler source code format. Most arithmetic and logical instruction examples include one or two parallel data moves to illustrate the many types of parallel moves that are possible. The example includes a complete explanation, which discusses the contents of the registers referenced by the instruction (but not those referenced by the parallel moves) both before and after the execution of the instruction. Most examples are designed to be easily understood without the use of a calculator.
5. **Condition Codes:** The status register is depicted with the condition code bits which can be affected by the instruction highlighted in **bold** type. Not all bits in the status register are used. Those which are reserved are indicated with a double asterisk and are read as zeros.
6. **Instruction Format:** The instruction fields, the instruction opcode, and the instruction extension word are specified for each instruction syntax. When the extension

word is optional, it is so indicated. The values which can be assumed by each of the variables in the various instruction fields are shown under the instruction field's heading. Note that the symbols used in decoding the various opcode fields of an instruction are **completely arbitrary**. Furthermore, the opcode symbols used in one instruction are **completely independent** of the opcode symbols used in a different instruction.

7. **Timing:** The number of oscillator clock cycles required for each instruction syntax is given. This information provides the user a basis for comparison of the execution times of the various instructions in oscillator clock cycles. Refer to Table A-1 and Section A.8 for a complete explanation of instruction timing, including the meaning of the symbols "aio", "ap", "ax", "ay", "axy", "ea", "jx", "mv", "mvb", "mvc", "mvm", "mvp", "rx", "wio", "wp", "wx", and "wy".

8. **Memory:** The number of program memory words required for each instruction syntax is given. This information provides the user a basis for comparison of the number of program memory locations required for each of the various instructions in 24-bit program memory words. Refer to Table A-1 and Section A.8 for a complete explanation of instruction memory requirements, including the meaning of the symbols "ea" and "mv".

A.3 NOTATION

Each instruction description contains symbols used to abbreviate certain operands and operations. Table A-1 lists the symbols used and their respective meanings. Depending on the context, registers refer to either the register itself or the contents of the register.

Table A-1 Instruction Description Notation

Data ALU Registers Operands

Xn	Input Register X1 or X0 (24 Bits)
Yn	Input Register Y1 or Y0 (24 Bits)
An	Accumulator Registers A2, A1, A0 (A2 — 8 Bits, A1 and A0 — 24 Bits)
Bn	Accumulator Registers B2, B1, B0 (B2 — 8 Bits, B1 and B0 — 24 Bits)
X	Input Register X = X1: X0 (48 Bits)
Y	Input Register Y = Y1: Y0 (48 Bits)
A	Accumulator A = A2: A1: A0 (56 Bits)*
B	Accumulator B = B2: B1: B0 (56 Blts)*
AB	Accumulators A and B = A1: B1 (48 Bits)*
BA	Accumulators B and A = B1: A1 (48 Bits)*
A10	Accumulator A = A1: A0 (48 Bits)
B10	Accumulator B= B1:B0 (48 bits)
* **NOTE:** In data move operations, shifting and limiting are performed when this register is specified as a source operand. When specified as a destination operand, sign extension and possibly zeroing are performed.	

Address ALU Registers Operands

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

Table A-1 Instruction Description Notation (Continued)

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)
OMR	Operating Mode Register (8 Bits)
LA	Hardware Loop Address Register (16 Bits)
LC	Hardware Loop Counter Register (16 Bits)
SP	System Stack Pointer Register (6 Bits)
SSH	Upper Portion of the Current Top of the Stack (16 Bits)
SSL	Lower Portion of the Current Top of the Stack (16 Bits)
SS	System Stack RAM = SSH: SSL (15 Locations by 32 Bits)

Address Operands

ea	Effective Address
eax	Effective Address for X Bus
eay	Effective Address for Y Bus
xxxx	Absolute Address (16 Bits)
xxx	Short Jump Address (12 Bits)
aa	Absolute Short Address (6 Bits, Zero Extended)
pp	I/O Short Address (6 Bits, Ones Extended)
<. . .>	Specifies the Contents of the Specified Address
X:	X Memory Reference
Y:	Y Memory Reference
L:	Long Memory Reference = X:Y
P:	Program Memory Reference

A

Table A-1 Instruction Description Notation (Continued)

Miscellaneous Operands

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

Unary Operators

-	Negation Operator
—	Logical NOT Operator (Overbar)
PUSH	Push Specified Value onto the System Stack (SS) Operator
PULL	Pull Specified Value from the System Stack (SS) Operator
READ	Read the Top of the System Stack (SS) Operator
PURGE	Delete the Top Value on the System Stack (SS) Operator
| |	Absolute Value Operator

Binary Operators

+	Addition Operator
-	Subtraction Operator
*	Multiplication Operator
÷, /	Division Operator
+	Logical Inclusive OR Operator
•	Logical AND Operator
⊕	Logical Exclusive OR Operator
→	"Is Transferred To" Operator
:	Concatenation Operator

A

Table A-1 Instruction Description Notation (Continued)

Addressing Mode Operators

<<	I/O Short Addressing Mode Force Operator
<	Short Addressing Mode Force Operator
>	Long Addressing Mode Force Operator
#	Immediate Addressing Mode Operator
#>	Immediate Long Addressing Mode Force Operator
#<	Immediate Short Addressing Mode Force Operator

Mode Register (MR) Symbols

DM	Double Precision Multiply Bit Indicating if the Chip is in Double Precision Multiply Mode
LF	Loop Flag Bit Indicating When a DO Loop is in Progress
T	Trace Mode Bit Indicating if the Tracing Function has been Enabled
S1, S0	Scaling Mode Bits Indicating the Current Scaling Mode
I1, I0	Interrupt Mask Bits Indicating the Current Interrupt Priority Level

Condition Code Register (CCR) Symbols

Standard Definitions (Table A-5 in Section A.5 Describes Exceptions)

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 A or B is in Use
U	Unnormalized Bit Indicating if the A or B Result is Unnormalized
N	Negative Bit Indicating if Bit 55 of the A or B Result is Set
Z	Zero Bit Indicating if the A or B Result Equals Zero
V	Overflow Bit Indicating if Arithmetic Overflow has Occurred in A or B
C	Carry Bit Indicating if a Carry or Borrow Occurred in A or B Result

Table A-1 Instruction Description Notation (Continued)

Instruction Timing Symbols

aio	Time Required to Access an I/O Operand
ap	Time Required to Access a P Memory Operand
ax	Time Required to Access an X Memory Operand
ay	Time Required to Access a Y Memory Operand
axy	Time Required to Access XY Memory Operands
ea	Time or Number of Words Required for an Effective Address
jx	Time Required to Execute Part of a Jump-Type Instruction
mv	Time or Number of Words Required for a Move-Type Operation
mvb	Time Required to Execute Part of a Bit Manipulation Instruction
mvc	Time Required to Execute Part of a MOVEC Instruction
mvm	Time Required to Execute Part of a MOVEM Instruction
mvp	Time Required to Execute Part of a MOVEP Instruction
rx	Time Required to Execute Part of an RTI or RTS Instruction
wio	Number of Wait States Used in Accessing External I/O
wp	Number of Wait States Used in Accessing External P Memory
wx	Number of Wait States Used in Accessing External X Memory
wy	Number of Wait States Used in Accessing External Y Memory

Other Symbols

( )	Optional Letter, Operand, or Operation
(...)	Any Arithmetic or Logical Instruction Which Allows Parallel Moves
EXT	Extension Register Portion of an Accumulator (A2 or B2)
LS	Least Significant
LSP	Least Significant Portion of an Accumulator (A0 or B0)
MS	Most Significant
MSP	Most Significant Portion of a n Accumulator (A1 or B1)
r	Rounding constant
S/L	Shifting and/or Limiting on a Data ALU Register
Sign Ext	Sign Extension of a Data ALU Register
Zero	Zeroing of a Data ALU Register

A.4 ADDRESSING MODES

The addressing modes are grouped into three categories: register direct, address register indirect, and special. These addressing modes are summarized in Table A-2. All address calculations are performed in the address ALU to minimize execution time and loop overhead. Addressing modes, which specify whether the operands are in registers, in memory, or in the instruction itself (such as immediate data), provide the specific address of the operands.

The register direct addressing mode can be subclassified according to the specific register addressed. The data registers include X1, X0, Y1, Y0, X, Y, A2, A1, A0, B2, B1, B0, A, and B. The control registers include SR, OMR, SP, SSH, SSL, LA, LC, CCR, and MR.

Address register indirect modes use an address register Rn (R0–R7) to point to locations in X, Y, and P memory. The contents of the Rn address register (Rn) is the effective address (ea) of the specified operand, except in the "indexed by offset" mode where the effective address (ea) is (Rn+Nn). Address register indirect modes use an address modifier register Mn to specify the type of arithmetic to be used to update the address register Rn. If an addressing mode specifies an address offset register Nn, the given address offset register is used to update the corresponding address register Rn. The Rn address register may only use the corresponding address offset register Nn and the corresponding address modifier register Mn. For example, the address register R0 may only use the N0 address offset register and the M0 address modifier register during actual address computation and address register update operations. This unique implementation allows the user to easily address a wide variety of DSP-oriented data structures. All address register indirect modes use at least one set of address registers (Rn, Nn, and Mn), and the XY memory reference uses two sets of address registers, one for the X memory space and one for the Y memory space.

The special addressing modes include immediate and absolute addressing modes as well as implied references to the program counter (PC), the system stack (SSH or SSL), and program (P) memory.

Addressing modes may also be categorized by the ways in which they can be used. Table A-2 and Table A-3 show the various categories to which each addressing mode belongs. These addressing mode categories may be combined so that additional, more restrictive classifications may be defined. For example, the instruction descriptions may use a **memory alterable** classification, which refers to addressing modes that are **both** memory addressing modes **and** alterable addressing modes. Thus, memory alterable addressing modes use address register indirect and absolute addressing modes.

Table A-2 DSP56K Addressing Modes

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

NOTE:

S = System Stack Reference
C = Program Controller Register Reference
D = Data ALU Register Reference
A = Address ALU Register Reference
P = Program Memory Reference
X = X Memory Reference
Y = Y Memory Reference
L = L Memory Reference
XY = XY Memory Reference

A

Table A-3 DSP56K Addressing Mode Encoding

Addressing Mode	Mode MMM	Reg RRR	Addressing Categories				Assembler Syntax
			U	P	M	A	
Register Direct							
Data or Control Register	—	—				X	(See Table A-1)
Address Register	—	—				X	Rn
Address Offset Register	—	—				X	Nn
Address Modifier Register	—	—				X	Mn
Address Register Indirect							
No Update	100	Rn		X	X	X	(Rn)
Postincrement by 1	011	Rn	X	X	X	X	(Rn) +
Postdecrement by 1	010	Rn	X	X	X	X	(Rn) -
Postincrement by Offset Nn	001	Rn	X	X	X	X	(Rn) + Nn
Postdecrement by Offset Nn	000	Rn	X		X	X	(RN) - Nn
Indexed by Offset Nn	101	Rn			X	X	(Rn + Nn)
Predecrement by 1	111	Rn			X	X	- (Rn)
Special							
Immediate Data	110	100			X		#xxxxxx
Absolute Address	110	000			X	X	xxxx
Immediate Short Data	—	—					#xx
Short Jump Address	—	—				X	xxx
Absolute Short Address	—	—				X	aa
I/O Short Address	—	—				X	pp
Implicit	—	—				X	

Update Mode (U) – Modifies address registers without any associated data move.
Parallel Mode (P) – Used in instructions where two effective addresses are required.
Memory Mode (M) – Refers to operands in memory using an effective addressing field.
Alterable Mode (A) – Refers to alterable or writable registers or memory.

A

The address register indirect addressing modes require that the offset register number be the same as the address register number. The assembler syntax "N" may be used instead of "Nn" in the address register indirect memory addressing modes. If "N" is specified, the offset register number is the same as the address register number.

A.4.1 Addressing Mode Modifiers

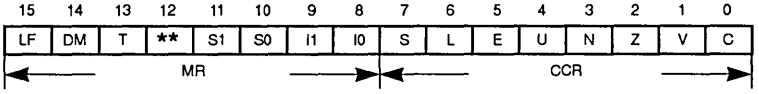
The addressing mode selected in the instruction word is further specified by the contents of the address modifier register Mn. The addressing mode update modifiers (M0–M7) are shown in Table A-4. There are no restrictions on the use of modifier types with any address register indirect addressing mode.

Table A-4 Addressing Mode Modifier Summary

Binary M0-M7	Hex M0-M7	Addressing Mode Arithmetic
0000 0000 0000 0000	0000	Reverse Carry (Bit Reverse)
0000 0000 0000 0001	0001	Modulo 2
00000000 0000 0010	0002	Modulo 3
:	:	:
0111 1111 1111 1110	7FFE	Modulo 32767
0111 1111 1111 1111	7FFF	Modulo 32768
1000 0000 0000 0000	8000	Reserved
1000 0000 0000 0001	8001	Multiple Wrap-Around Modulo 2
1000 0000 0000 0010	8002	Reserved
1000 0000 0000 0011	8003	Multiple Wrap-Around Modulo 4
:	:	Reserved
1000 0000 0000 0111	8007	Multiple Wrap-Around Modulo 8
:	:	Reserved
1000 0000 0000 1111	800F	Multiple Wrap-Around Modulo 2^4
:	:	Reserved
1000 0000 0001 1111	801F	Multiple Wrap-Around Modulo 2^5
:	:	Reserved
1000 0000 0011 1111	803F	Multiple Wrap-Around Modulo 2^6
:	:	Reserved
1000 0000 0111 1111	807F	Multiple Wrap-Around Modulo 2^7
:	:	Reserved
1000 0000 1111 1111	80FF	Multiple Wrap-Around Modulo 2^8
:	:	Reserved
1000 0001 1111 1111	81FF	Multiple Wrap-Around Modulo 2^9
:	:	Reserved
1000 0011 1111 1111	83FF	Multiple Wrap-Around Modulo 2^{10}
:	:	Reserved
1000 0111 1111 1111	87FF	Multiple Wrap-Around Modulo 2^{11}
:	:	Reserved
1000 1111 1111 1111	8FFF	Multiple Wrap-Around Modulo 2^{12}
:	:	Reserved
1001 1111 1111 1111	9FFF	Multiple Wrap-Around Modulo 2^{13}
:	:	Reserved
1011 1111 1111 1111	BFFF	Multiple Wrap-Around Modulo 2^{14}
:	:	Reserved
1111 1111 1111 1111	FFFF	Linear (Modulo 2^{15})

A

A.5 CONDITION CODE COMPUTATION



The condition code register (CCR) portion of the status register (SR) consists of eight defined bits:

S — Scaling Bit	N — Negative Bit
L — Limit Bit	Z — Zero Bit
E — Extension Bit	V — Overflow Bit
U — Unnormalized Bit	C — Carry Bit

The E, U, N, Z, V, and C bits are **true** condition code bits that reflect the condition of the **result of a data ALU operation**. These condition code bits are **not latched** and are **not affected by address ALU calculations or by data transfers** over the X, Y, or global data buses. The L bit is a **latching overflow bit** which indicates that an overflow has occurred in the data ALU or that data limiting has occurred when moving the contents of the A and/or B accumulators. The S bit is a **latching** bit used in block floating point operations to indicate the need to scale the number in A or B. See SECTION 5 – PROGRAM CONTROL UNIT for information on the MR portion of the status register.

The standard definition of the condition code bits follows. Exceptions to these standard definitions are given in the notes which follow Table A-5.

S (Scaling Bit)

The scaling bit (S) is used to detect data growth, which is required in Block Floating Point FFT operation. 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." This bit is computed according to the logical equations below when an instruction or a parallel move moves the result of accumulator A or B to XDB or YDB. It is a "sticky" bit, cleared only by an instruction that specifically clears it.

The following logical equations are used to compute the scaling bit based upon the scaling mode bits:

If S1=0 and S0=0 (no scaling)
then S = (A46 XOR A45) OR (B46 XOR B45)

If S1=0 and S0=1 (scale down)
then S = (A47 XOR A46) OR (B47 XOR B46)

If S1=1 and S0=0 (scale up)
then S = (A45 XOR A44) OR (B45 XOR B44)

If S1=1 and S0=1 (reserved)
then the S flag is undefined.

where Ai and Bi means bit i in accumulator A or B.

L (Limit Bit)

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. Not affected otherwise. This bit is **latched** and must be reset by the user.

E (Extension Bit)

Cleared if all the bits of the **signed integer portion** of the A or B result are the **same** – i.e., the bit patterns are either 00 . . . 00 or 11 . . . 11. Set otherwise. The signed integer portion is defined by the scaling mode as shown in the following table:

A

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

Note that the **signed integer portion** of an accumulator **IS NOT** necessarily the same as the **extension register portion** of that accumulator. The signed integer portion of an accumulator consists of the MS 8, 9, or 10 bits of that accumulator, depending on the scaling mode being used. The extension register portion of an accumulator (A2 or B2) is always the MS 8 bits of that accumulator. **The E bit refers to the signed integer portion of an accumulator and NOT the extension register portion of that accumulator.** For example, if the current scaling mode is set for no scaling (i.e., S1=S0=0), the signed integer portion of the A or B accumulator consists of **bits 47 through 55**. If the A accumulator contained the signed 56-bit value $00:800000:000000 as a **result of a data ALU operation**, the E bit **would** be set (E=1) since the **9** MS bits of that accumulator were not all the same (i.e., neither 00 . . 00 nor 11 . . 11). This means that data limiting **will** occur if that 56-bit value is specified as a **source** operand in a move-type operation. This limiting operation will result in either a positive or negative, 24-bit or 48-bit saturation constant being stored in the specified destination. The **only** situation in which the signed integer portion of an accumulator and the extension register portion of an accumulator are the same is in the "Scale Down" scaling mode (i.e., S1=0 and S0=1).

U (Unnormalized Bit) — Set if the two MS bits of the MSP portion of the A or B result are the same. Cleared otherwise. The MSP portion is defined by the scaling mode. The U bit is computed as follows:

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

N (Negative Bit) — Set if the MS bit 55 of the A or B result is set. Cleared otherwise.

Z (Zero Bit) — Set if the A or B result equals zero. Cleared otherwise.

V (Overflow Bit) — Set if an arithmetic overflow occurs in the 56-bit A or B result. This indicates that the result cannot be represented in the 56-bit accumulator; thus, the accumulator has overflowed. Cleared otherwise.

A

C (Carry Bit) — Set if a carry is generated out of the MS bit of the A or B result of an addition or if a borrow is generated out of the MS bit of the A or B result of a subtraction. The carry or borrow is generated out of bit 55 of the A or B result. Cleared otherwise.

Table A-5 shows how each condition code bit is affected by each instruction. Exceptions to the standard definitions given above are indicated by a number or a "?". Consult the corresponding note for the special definition that applies in each particular case. Although many of the instructions allow optional parallel moves, Table A-5 applies when there are **no** parallel moves associated with an instruction. With this restriction, the states of the condition code bits are determined only by the execution of the instruction itself. However, the S and L bits may be determined differently than shown in the table when a parallel move is associated with the instruction. When using an optional parallel move, refer to the individual instruction's detailed description in Section A.7 to see how the S and L bits are determined.

Table A-5 Condition Code Computations for Instructions (No Parallel Move)

Mnemonic	S	L	E	U	N	Z	V	C	Notes	Mnemonic	S	L	E	U	N	Z	V	C	Notes
ABS	—	✓	✓	✓	✓	✓	✓	—		LSR	—	—	—	—	1	9	1	11	
ADC	—	✓	✓	✓	✓	✓	✓	✓		LUA	—	—	—	—	—	—	—	—	
ADD	—	✓	✓	✓	✓	✓	✓	✓		MAC	—	✓	✓	✓	✓	✓	✓	—	
ADDL	—	✓	✓	✓	✓	✓	2	✓		MACR	—	✓	✓	✓	✓	✓	✓	—	
ADDR	—	✓	✓	✓	✓	✓	✓	✓		MOVE	✓	✓	—	—	—	—	—	—	
AND	—	—	—	—	8	9	1	—		MOVEC	?	?	?	?	?	?	?	?	13
ANDI	?	?	?	?	?	?	?	?	3	MOVEM	?	?	?	?	?	?	?	?	13
ASL	—	✓	✓	✓	✓	✓	2	4		MOVEP	?	?	?	?	?	?	?	?	13
ASR	—	—	✓	✓	✓	✓	1	5		MPY	—	—	✓	✓	✓	✓	1	—	
BCHG	?	?	?	?	?	?	?	?	14	MPYR	—	—	✓	✓	✓	✓	1	—	
BCLR	?	?	?	?	?	?	?	?	14	NEG	—	✓	✓	✓	✓	✓	✓	—	
BSET	?	?	?	?	?	?	?	?	14	NOP	—	—	—	—	—	—	—	—	
BTST	?	?	—	—	—	—	—	?	14	NORM	—	✓	✓	✓	✓	✓	2	—	
CLR	—	—	✓	✓	✓	✓	1	—		NOT	—	—	—	—	8	9	1	—	
CMP	—	✓	✓	✓	✓	✓	✓	✓		OR	—	—	—	—	8	9	1	—	
CMPM	—	✓	✓	✓	✓	✓	✓	✓		ORI	?	?	?	?	?	?	?	?	6
DEBUG	—	—	—	—	—	—	—	—		REP	✓	✓	—	—	—	—	—	—	
DEBUGcc	—	—	—	—	—	—	—	—		RESET	—	—	—	—	—	—	—	—	
DEC	—	✓	✓	✓	✓	✓	✓	✓		RND	—	✓	✓	✓	✓	✓	✓	—	
DIV	—	✓	—	—	—	—	2	7		ROL	—	—	—	—	8	9	1	10	
DO	✓	✓	—	—	—	—	—	—		ROR	—	—	—	—	8	9	1	11	
ENDDO	—	—	—	—	—	—	—	—		RTI	?	?	?	?	?	?	?	?	12
EOR	—	—	—	—	8	9	1	—		RTS	—	—	—	—	—	—	—	—	
ILLEGAL	—	—	—	—	—	—	—	—		SBC	—	✓	✓	✓	✓	✓	✓	✓	
INC	—	✓	✓	✓	✓	✓	✓	✓		STOP	—	—	—	—	—	—	—	—	
Jcc	—	—	—	—	—	—	—	—		SUB	—	✓	✓	✓	✓	✓	✓	✓	
JCLR	?	?	—	—	—	—	—	—	14	SUBL	—	✓	✓	✓	✓	✓	2	✓	
JMP	—	—	—	—	—	—	—	—		SUBR	—	✓	✓	✓	✓	✓	✓	✓	
JScc	—	—	—	—	—	—	—	—		SWI	—	—	—	—	—	—	—	—	
JSCLR	?	?	—	—	—	—	—	—	14	Tcc	—	—	—	—	—	—	—	—	
JSET	?	?	—	—	—	—	—	—	14	TFR	—	—	—	—	—	—	—	—	
JSR	—	—	—	—	—	—	—	—		TST	—	—	✓	✓	✓	✓	1	—	
JSSET	?	?	—	—	—	—	—	—	14	WAIT	—	—	—	—	—	—	—	—	
LSL	—	—	—	—	8	9	1	10											

where: ✓ Set according to the **standard** definition of the operation
— Not affected by the operation
? or # Set according to a **special** definition (refer to the following notes) and can be a 0 or 1

A

The following notes apply to Table A-5:

1. The bit is cleared.
2. V — Set if an arithmetic overflow occurs in the 56-bit A or B result or if the MS bit of the destination operand is changed as a result of the left shift. Cleared otherwise.
3. For destination operand CCR, the bits are cleared if the corresponding bits in the immediate data are cleared. Otherwise they are not affected. For other destination operands, the bits are not affected.
4. C — Set if bit 55 of the source operand was set prior to instruction execution. Cleared otherwise.
5. C — Set if bit 0 of the source operand was set prior to instruction execution. Cleared otherwise.
6. For destination operand CCR, the bits are set if the corresponding bits in the immediate data are set. Otherwise, they are not affected. For other destination operands, the bits are not affected.
7. C — Set if bit 55 of the result is cleared. Cleared otherwise.
8. N — Set if bit 47 of the A or B result is set. Cleared otherwise.
9. Z — Set if bits 47 - 24 of the A or B result are zero. Cleared otherwise.
10. C — Set if bit 47 of the source operand was set prior to instruction execution. Cleared otherwise.
11. C — Set if bit 24 of the source operand was set prior to instruction execution. Cleared otherwise.
12. Set according to the value pulled from the stack.
13. For destination operand SR, the bits are set according to the corresponding bit of the source operand. If SR is not specified as a destination operand, the L bit is set if data limiting occurred and the S bit is computed according to the definition. (See Section A.5.) Otherwise, the bits are unaffected.
14. Due to complexity, refer to the detailed description of the instruction.

A.6 PARALLEL MOVE DESCRIPTIONS

Many of the instructions in the DSP56K instruction set allow optional parallel data bus movement. Section A.7 indicates the parallel move option in the instruction syntax with the statement "'parallel move)". The MOVE instruction is equivalent to a NOP with parallel moves. Therefore, a detailed description of each parallel move is given with the MOVE instruction details in Section A.7, beginning on page A-160.

A.7 INSTRUCTION DESCRIPTIONS

The following section describes each instruction in the DSP56K instruction set in complete detail. The format of each instruction description is given in Section A.2. Instructions which allow parallel moves include the notation "(parallel move)" in both the **Assembler Syntax** and the **Operation** fields. The example given with each instruction discusses the contents of all the registers and memory locations referenced by the opcode-operand portion of that instruction but not those referenced by the parallel move portion of that instruction. Refer to page A-160 for a complete discussion of parallel moves, including examples which discuss the contents of all the registers and memory locations referenced by the parallel move portion of an instruction.

Note: Whenever an instruction uses an accumulator as both a destination operand for a data ALU operation and as a source for a parallel move operation, the parallel move operation occurs **first** and will use the data that exists in the accumulator before the execution of the data ALU operation has occurred.

Whenever a bit in the condition code register is defined according to the **standard** definition given in Section A.5, a brief definition will be given in **normal** text in the **Condition Code** section of that instruction description. Whenever a bit in the condition code register is defined according to a **special** definition for some particular instruction, the special definition of that bit will be given in the **Condition Code** section of that instruction in **bold** text to alert the user to any special conditions concerning its use.

The definition and thus the computation of both the E (extension) and U (unnormalized) bits of the condition code register (CCR) varies according to the scaling mode being used. Refer to Section A.5 for complete details.

Note: The signed integer portion of an accumulator is NOT necessarily the same as either the A2 or B2 extension register portion of that accumulator. The signed integer portion of an accumulator is defined according to the scaling mode being used and can consist of the MS 8, 9, or 10 bits of an accumulator. Refer to Section A.5 for complete details.

ABS — Absolute Value — ABS

Operation:
| D | → D (parallel move)

Assembler Syntax:
ABS D (parallel move)

Description: Take the absolute value of the destination operand D and store the result in the destination accumulator.

Example:

```
        :
ABS   A1    #$123456,X0    A,Y0      ;take abs. value, set up X0, save value
        :
```

	Before Execution		After Execution
A	$FF:FFFFFF:FFFFF2	A	$00:000000:00000E

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $FF:FFFFFF:FFFFF2. Since this is a negative number, the execution of the ABS instruction takes the twos complement of that value and returns $00:000000:00000E.

Note: For the case in which the D operand equals $80:000000:000000 (-256.0), the ABS instruction will cause an overflow to occur since the result cannot be correctly expressed using the standard 56-bit, fixed-point, twos-complement data representation. Data limiting does not occur (i.e., A is not set to the limiting value of $7F:FFFFFF:FFFFFF).

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION.
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

ABS — Absolute Value — ABS

Instruction Format:

ABS D

Opcode:

23 — 8	7	6	5	4	3	2	1	0
DATA BUS MOVE FIELD	0	0	1	0	d	1	1	0
OPTIONAL EFFECTIVE ADDRESS EXTENSION								

Instruction Fields:

D d

A 0

B 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

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 (48 bits) may be added to the (56-bit) destination accumulator.

Note: The carry bit is set correctly for multiple precision arithmetic using long-word operands if the extension register of the destination accumulator (A2 or B2) is the sign extension of bit 47 of the destination accumulator (A or B).

Example:

```
        :
MOVE L:<$0,X          ;get a 48-bit LS long-word operand in X
MOVE L:<$1,A          ;get other LS long word in A (sign ext.)
MOVE L:<$2,Y          ;get a 48-bit MS long-word operand in Y
ADD X,A L:<$3,B       ;add LS words; get other MS word in B
ADC Y,B A10,L:<$4     ;add MS words with carry, save LS sum
MOVE B10,L:<$5        ;save MS sum
        :
```

	Before Execution		After Execution
A	$FF:800000:000000	A	$FF:000000:000000
X	$800000:000000	X	$800000:000000
B	$00:000000:000001	B	$00:000000:000003
Y	$000000:000001	Y	$000000:000001

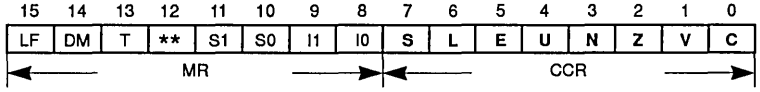
Explanation of Example: This example illustrates long-word double-precision (96-bit) addition using the ADC instruction. Prior to execution of the ADD and ADC instructions, the double-precision 96-bit value $000000:000001:800000:000000 is loaded into the Y and X registers (Y:X), respectively. The other double-precision 96-bit value $000000:000001:800000:000000 is loaded into the B and A accumulators (B:A), respectively. Since the 48-bit value loaded into the A accumulator is automatically sign extended to 56 bits and the other 48-bit long-word operand is internally sign extended to 56 bits during instruction execution, the carry bit will be set correctly after the execution of the ADD X,A instruction. The ADC Y,B instruction then produces the correct MS 56-bit

A

ADC — Add Long with Carry — ADC

result. The actual 96-bit result is stored in memory using the A10 and B10 operands (instead of A and B) because shifting and limiting is not desired.

Condition Codes:



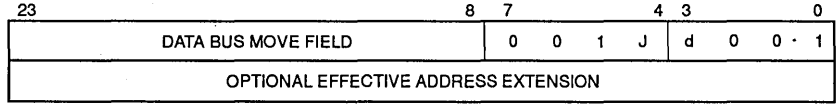
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format:

ADC S,D

Opcode:



Instruction Fields:

S,D	J d
X,A	0 0
X,B	0 1
Y,A	1 0
Y,B	1 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

ADD Add ADD

Operation:
S+D→D (parallel move

Assembler Syntax:
ADD S,D (parallel move)

Description: Add the source operand S to the destination operand D and store the result in the destination accumulator. Words (24 bits), long words (48 bits), and accumulators (56 bits) may be added to the destination accumulator.

Note: The carry bit is set correctly using word or long-word source operands if the extension register of the destination accumulator (A2 or B2) is the sign extension of bit 47 of the destination accumulator (A or B). Thus, the carry bit is always set correctly using accumulator source operands, but can be set incorrectly if A1, B1, A10, or B10 are used as source operands and A2 and B2 are not replicas of bit 47.

Example:

```
        :
ADD  X0,A A,X1     A,Y:(R1)+I     ;24-bit add, set up X1, save prev. result
        :
```

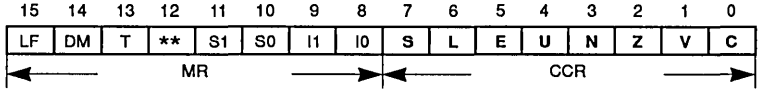
	Before Execution	After Execution
X0	$FFFFFF	$FFFFFF
A	$00:000100:000000	$00:0000FF:000000

Explanation of Example: Prior to execution, the 24-bit X0 register contains the value $FFFFFF and the 56-bit A accumulator contains the value $00:000100:000000. The ADD instruction automatically appends the 24-bit value in the X0 register with 24 LS zeros, sign extends the resulting 48-bit long word to 56 bits, and adds the result to the 56-bit A accumulator. Thus, 24-bit operands are added to the MSP portion of A or B (A1 or B1) because all arithmetic instructions assume a fractional, twos complement data representation. Note that 24-bit operands can be added to the LSP portion of A or B (A0 or B0) by loading the 24-bit operand into X0 or Y0, forming a 48-bit word by loading X1 or Y1 with the sign extension of X0 or Y0 and executing an ADD X,A or ADD Y,A instruction.

A

ADD — Add — ADD

Condition Codes:



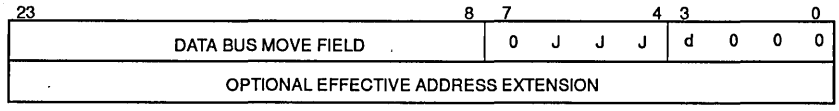
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format:

ADD S,D

Opcode:



Instruction Fields:

S,D	J J J d	S,D	J J J d	S,D	J J J d
B,A	0 0 1 0	X0,A	1 0 0 0	Y1,A	1 1 1 0
A,B	0 0 1 1	X0,B	1 0 0 1	Y1,B	1 1 1 1
X,A	0 1 0 0	Y0,A	1 0 1 0		
X,B	0 1 0 1	Y0,B	1 0 1 1		
Y,A	0 1 1 0	X1,A	1 1 0 0		
Y,B	0 1 1 1	X1,B	1 1 0 1		

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

ADDL Shift Left and Add Accumulators ADDL

Operation:
S+2*D→D (parallel move)

Assembler Syntax:
ADDL S,D (parallel move)

Description: Add the source operand S to two times the destination operand D and store the result in the destination accumulator. The destination operand D is arithmetically shifted one bit to the left, and a zero is shifted into the LS bit of D prior to the addition operation. The carry bit is set correctly if the source operand does not overflow as a result of the left shift operation. The overflow bit may be set as a result of either the shifting or addition operation (or both). This instruction is useful for efficient divide and decimation in time (DIT) FFT algorithms.

Example:

```
        :
ADDL A,B #$0,R0          ;A+2*B→B, set up addr. reg. R0
        :
```

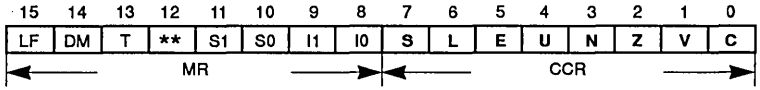
	Before Execution		After Execution
A	$00:000000:000123	A	$00:000000:000123
B	$00:005000:000000	B	$00:00A000:000123

Explanation of Example: Prior to execution, the 56-bit accumulator contains the value $00:000000:000123, and the 56-bit B accumulator contains the value $00:005000:000000. The ADDL A,B instruction adds two times the value in the B accumulator to the value in the A accumulator and stores the 56-bit result in the B accumulator.

A

ADDL Shift Left and Add Accumulators ADDL

Condition Codes:



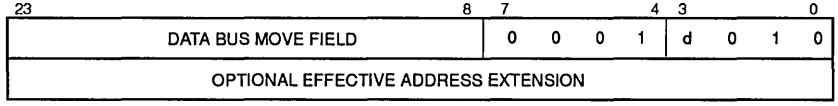
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — **Set if overflow has occurred in A or B result or if the MS bit of the destination operand is changed as a result of the instruction's left shift**
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format:

ADDL S,D

Opcode:



Instruction Fields:

S,D	d
B,A	0
A,B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

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 is always set correctly, and the overflow bit can only be set by the addition operation and not by an overflow due to the initial shifting operation. This instruction is useful for efficient divide and decimation in time (DIT) FFT algorithms.

Example:

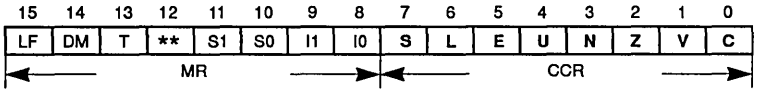
```
        :
ADDR B,A X0,X:(R1)+N1 Y0,Y:(R4)-     ;B+A / 2→A, save X0 and Y0
        :
```

	Before Execution		After Execution
A	$80:000000:2468AC	A	$C0:013570:123456
B	$00:013570:000000	B	$00:013570:000000

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $80:000000:2468AC, and the 56-bit B accumulator contains the value $00:013570:000000. The ADDR B,A instruction adds one-half the value in the A accumulator to the value in the B accumulator and stores the 56-bit result in the A accumulator.

ADDR Shift Right and Add Accumulators ADDR

Condition Codes:



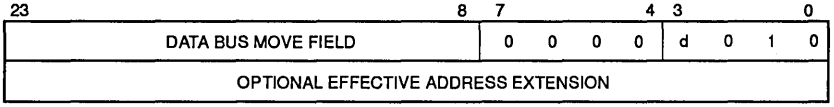
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format:

ADDR S,D

Opcode:



Instruction Fields:

S,D	d
B,A	0
A,B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

AND — Logical AND — AND

Operation:
S • D[47:24]→D[47:24] (parallel move)
where •denotes the logical AND operator

Assembler Syntax:
AND S,D (parallel move)

Description: Logically AND the source operand S with bits 47–24 of the destination operand D and store the result in bits 47–24 of the destination accumulator. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

Example:

```
        :
AND X0,A1 (R5)-N5          ;AND X0 with A1, update R5 using N5
        :
```

	Before Execution	After Execution
X0	$FF0000	$FF0000
A	$00:123456:789ABC	$00:120000:789ABC

Explanation of Example: Prior to execution, the 24-bit X0 register contains the value $FF0000, and the 56-bit A accumulator contains the value $00:123456:789ABC. The AND X0,A instruction logically ANDs the 24-bit value in the X0 register with bits 47–24 of the A accumulator (A1) and stores the result in the A accumulator with bits 55–48 and 23–0 unchanged.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

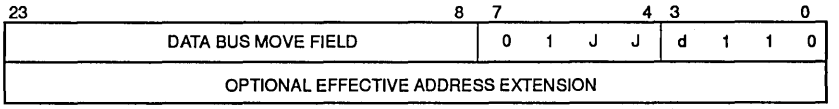
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting occurs during parallel move
N — **Set if bit 47 of A or B result is set**
Z— **Set if bits 47–24 of A or B result are zero**
V — **Always cleared**

AND — Logical AND — AND

Instruction Format:

AND S,D

Opcode:



Instruction Fields:

S	J J	D d
X0	0 0	A 0 (only A1 is changed)
X1	1 0	B 1 (only B1 is changed)
Y0	0 1	
Y1	1 1	

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

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.

Restrictions: The ANDI #xx,MR instruction cannot be used **immediately before** an ENDDO or RTI instruction and cannot be one of the **last three** instructions in a DO loop (at LA-2, LA-1, or LA).

The ANDI #xx,CCR instruction cannot be used **immediately before** an RTI instruction.

Example:

```
        :
AND #$FE,CCR        ;clear carry bit C in cond. code register
        :
```

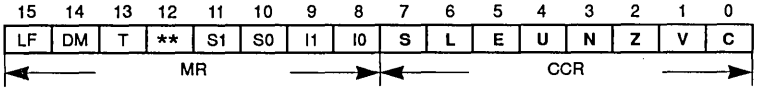
	Before Execution		After Execution
CCR	$31	CCR	$30

Explanation of Example: Prior to execution, the 8-bit condition code register (CCR) contains the value $31. The AND #$FE,CCR instruction logically ANDs the immediate 8-bit value $FE with the contents of the condition code register and stores the result in the condition code register.

A

ANDI AND Immediate with Control Register ANDI

Condition Codes:



For CCR Operand:

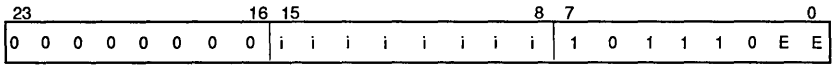
S — **Cleared if bit 7 of the immediate operand is cleared**
L — **Cleared if bit 6 of the immediate operand is cleared**
E — **Cleared if bit 5 of the immediate operand is cleared**
U — **Cleared if bit 4 of the immediate operand is cleared**
N — **Cleared if bit 3 of the immediate operand is cleared**
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 Format:

AND(I) #xx,D

Opcode:



Instruction Fields:

#xx=8-bit Immediate Short Data — iiiiiiii

D	E E
MR	0 0
CCR	0 1
OMR	1 0

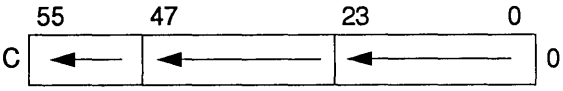
Timing: 2 oscillator clock cycles

Memory: 1 program word

A

ASL — Arithmetic Shift Accumulator Left — ASL

Operation:



Assembler Syntax: ASL D (parallel move)

Description: Arithmetically shift the destination operand D one bit to the left and store the result in the destination accumulator. The MS bit of D prior to instruction execution is shifted into the carry bit C and a zero is shifted into the LS bit of the destination accumulator D. If a zero shift count is specified, the carry bit is cleared. The difference between ASL and LSL is that ASL operates on the entire 56 bits of the accumulator and therefore sets the V bit if the number overflowed.

Example:

```
        :
ASL A   (R3)-           ;multiply A by 2, update R3
        :
```

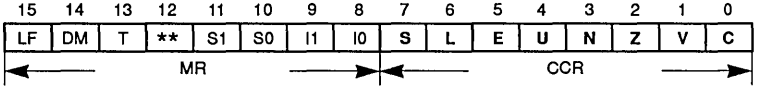
	Before Execution		After Execution
A	$A5:012345:012345	A	$4A:02468A:02468A
SR	$0300	SR	$0373

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $A5:012345:012345. The execution of the ASL A instruction shifts the 56-bit value in the A accumulator one bit to the left and stores the result back in the A accumulator.

A

ASL — Arithmetic Shift Accumulator Left — ASL

Condition Codes:



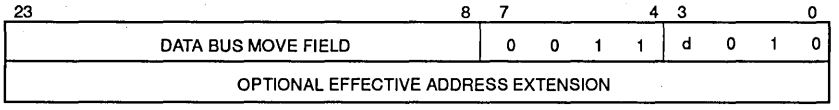
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — **Set if bit 55 of A or B result is changed due to left shift**
C — **Set if bit 55 of A or B was set prior to instruction execution**

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format:

ASL D

Opcode:



Instruction Fields:

D	d
A	0
B	1

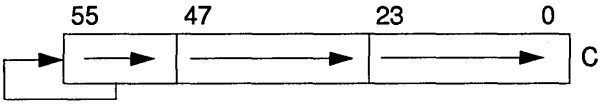
Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

ASR — Arithmetic Shift Accumulator Right — ASR

Operation:



Assembler Syntax: ASR D (parallel move)

Description: Arithmetically shift the destination operand D one bit to the right and store the result in the destination accumulator. The LS bit of D prior to instruction execution is shifted into the carry bit C, and the MS bit of D is held constant.

Example:

```
        :
ASR B   X:-(R3),R3      ;divide B by 2, update R3, load R3
        :
```

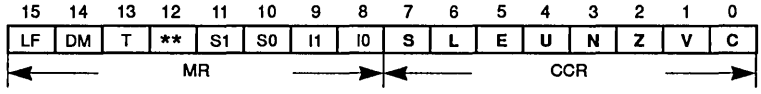
	Before Execution		After Execution
B	$A8:A86420:A86421	B	$D4:543210:543210
SR	$0300	SR	$0329

Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $A8:A86420:A86421. The execution of the ASR B instruction shifts the 56-bit value in the B accumulator one bit to the right and stores the result back in the B accumulator.

A

ASR — Arithmetic Shift Accumulator Right — ASR

Condition Codes:



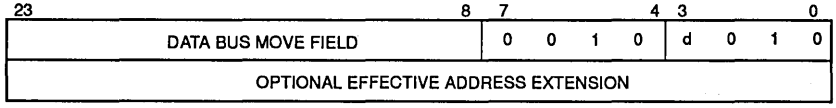
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting occurs during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — **Always cleared**
C — **Set if bit 0 of A or B was set prior to instruction execution**

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format:

ASR D

Opcode:



Instruction Fields:

D	d
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

BCHG — Bit Test and Change — BCHG

Operation:

$\underline{D[n]} \rightarrow C;$
$\overline{D[n]} \rightarrow D[n]$

Assembler Syntax:

BCHG	#n,X:ea
BCHG	#n,X:aa
BCHG	#n,X:pp
BCHG	#n,Y:ea
BCHG	#n,Y:aa
BCHG	#n,Y:pp
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 condition code 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.

Example:

```
        :
BCHG    #$7,X:<<$FFE2    ;test and change bit 7 in I/O Port B DDR
        :
```

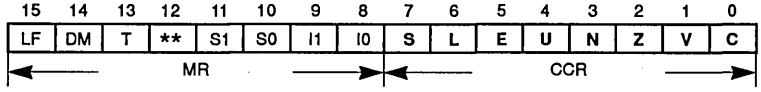
	Before Execution		After Execution
X:$FFE2	$000000	X;$FFE2	$000080
SR	$0300	SR	$0300

A

BCHG Bit Test and Change BCHG

Explanation of Example: Prior to execution, the 24-bit X location X:$FFE2 (I/O port B data direction register) contains the value $000000. The execution of the BCHG #$7,X:<<$FFE2 instruction tests the state of the 7th bit in X:$FFE2, sets the carry bit C accordingly, and then complements the 7th bit in X:$FFE2.

Condition Codes:



CCR Condition Codes:

For destination operand SR:

C — **Changed if bit 0 is specified. Not affected otherwise.**
V — **Changed if bit 1 is specified. Not affected otherwise.**
Z — **Changed if bit 2 is specified. Not affected otherwise.**
N — **Changed if bit 3 is specified. Not affected otherwise.**
U — **Changed if bit 4 is specified. Not affected otherwise.**
E — **Changed if bit 5 is specified. Not affected otherwise.**
L — **Changed if bit 6 is specified. Not affected otherwise.**
S — **Changed if bit 7 is specified. Not affected otherwise.**

For destination operand A or B:

S —Computed according to the definition. See Notes on page A-47.
L — Set if data limiting has occurred. See Notes on page A-47.
E — **Not affected**
U — **Not affected**
N — **Not affected**
Z — **Not affected**
V — **Not affected**
C — **Set if bit tested is set. Cleared otherwise.**

A

BCHG Bit Test and Change BCHG

For other destination operands:

S — **Not affected**
L — **Not affected**
E — **Not affected**
U — **Not affected**
N — **Not affected**
Z — **Not affected**
V — **Not affected**
C — **Set if bit tested is set. Cleared otherwise.**

MR Status Bits:

For destination operand SR:

I0 — Changed if bit 8 is specified. Not affected otherwise.
I1 — Changed if bit 9 is specified. Not affected otherwise.
S0 — Changed if bit 10 is specified. Not affected otherwise.
S1 — Changed if bit 11 is specified. Not affected otherwise.
T — Changed if bit 13 is specified. Not affected otherwise.
DM — Changed if bit 14 is specified. Not affected otherwise
LF — Changed if bit 15 is specified. Not affected otherwise.

For other destination operands:

I0 — Not affected
I1 — Not affected
S0 — Not affected
S1 — Not affected
T — Not affected
DM — Not affected
LF — Not affected

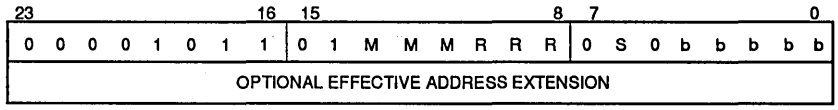
A

BCHG Bit Test and Change BCHG

Instruction Format:

BCHG #n,X:ea
BCHG #n,Y:ea

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M M M R R R	Memory Space	S	Bit Number bbbbb
(Rn)-Nn	0 0 0 r r r	X Memory	0	00000
(Rn)+Nn	0 0 1 r r r	Y Memory	1	•
(Rn)-	0 1 0 r r r			•
(Rn)+	0 1 1 r r r			•
(Rn)	1 0 0 r r r			10111
(Rn+Nn)	1 0 1 r r r			
-(Rn)	1 1 1 r r r			
Absolute address	1 1 0 0 0 0			

where "rrr" refers to an address register R0-R7

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

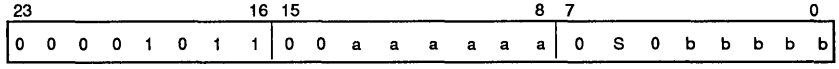
A

BCHG Bit Test and Change BCHG

Instruction Format:

BCHG #n,X:aa
BCHG #n,Y:aa

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

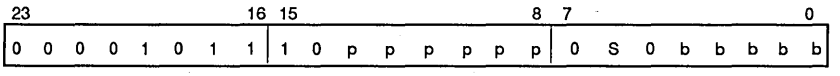
A

BCHG Bit Test and Change BCHG

Instruction Format:

BCHG #n,X:pp
BCHG #n,Y:pp

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit I/O Short Address=pppppp

I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

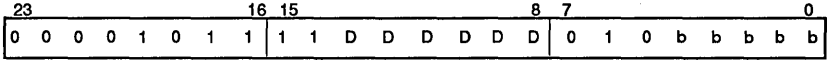
A

BCHG — Bit Test and Change — BCHG

Instruction Format:

BCHG #n,D

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
D=destination register=DDDDDD
xxxx=16-bit Absolute Address in extension word

Destination Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See Section A.10 and Table A-18 for specific register encodings.

A

BCHG Bit Test and Change BCHG

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The resulting 24 bit value is placed back into A1 or B1. A0 or B0 is cleared and the sign of A1 or B1 is extended into A2 or B2.
5. The bit test and change is performed on A1 or B1, and the C bit is set if the bit tested is set.

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

BCLR — Bit Test and Clear — BCLR

Operation:

D[n] → C;
0 → D[n]

Assembler Syntax:

BCLR #n,X:ea
BCLR #n,X:aa
BCLR #n,X:pp
BCLR #n,Y:ea
BCLR #n,Y:aa
BCLR #n,Y:pp
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 condition code 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.

Example:

```
        :
BCLR    #$E,X:<<$FFE4    ;test and clear bit 14 in I/O Port B Data Reg.
        :
```

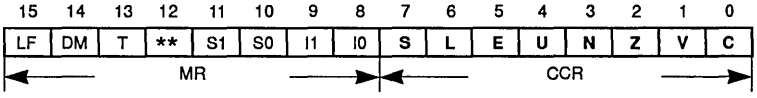
	Before Execution		After Execution
X:$FFE4	$FFFFFF	X:$FFE4	$FFBFFF
SR	$0300	SR	$0301

A

BCLR Bit Test and Clear BCLR

Explanation of Example: Prior to execution, the 24-bit X location X:$FFE4 (I/O port B data register) contains the value $FFFFFF. The execution of the BCLR #$E,X:<<$FFE4 instruction tests the state of the 14th bit in X:$FFE4, sets the carry bit C accordingly, and then clears the 14th bit in X:$FFE4.

Condition Codes:



CCR Condition Codes:

For destination operand SR:

C — **Cleared if bit 0 is specified. Not affected otherwise.**
V — **Cleared if bit 1 is specified. Not affected otherwise.**
Z — **Cleared if bit 2 is specified. Not affected otherwise.**
N — **Cleared if bit 3 is specified. Not affected otherwise.**
U — **Cleared if bit 4 is specified. Not affected otherwise.**
E — **Cleared if bit 5 is specified. Not affected otherwise.**
L — **Cleared if bit 6 is specified. Not affected otherwise.**
S — **Cleared if bit 7 is specified. Not affected otherwise.**

For destination operand A or B:

S —Computed according to the definition. See Notes on page A-55.
L — Set if data limiting has occurred. See Notes on page A-55.
E — **Not affected**
U — **Not affected**
N — **Not affected**
Z — **Not affected**
V — **Not affected**
C — **Set if bit tested is set. Cleared otherwise.**

A

BCLR Bit Test and Clear BCLR

For other destination operands:

C — **Set if bit tested is set. Cleared otherwise.**
V — **Not affected**
Z — **Not affected**
N — **Not affected**
U — **Not affected**
E — **Not affected**
L — **Not affected**
S — **Not affected**

MR Status Bits:

For destination operand SR:

I0 — Cleared if bit 8 is specified. Not affected otherwise.
I1 — Cleared if bit 9 is specified. Not affected otherwise.
S0 — Cleared if bit 10 is specified. Not affected otherwise.
S1 — Cleared if bit 11 is specified. Not affected otherwise.
T — Cleared if bit 13 is specified. Not affected otherwise.
DM — Cleared if bit 14 is specified. Not affected otherwise
LF — Cleared if bit 15 is specified. Not affected otherwise.

For other destination operands:

I0 — Not affected
I1 — Not affected
S0 — Not affected
S1 — Not affected
T — Not affected
DM — Not affected
LF — Not affected

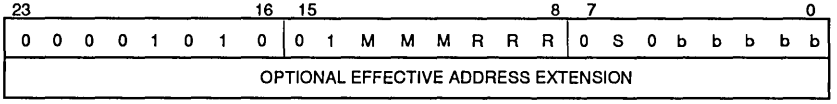
A

BCLR — Bit Test and Clear — BCLR

Instruction Format:

BCLR #n,X:ea
BCLR #n,Y:ea

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r
Absolute address	1 1 0 0 0 0

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where "rrr" refers to an address register R0-R7

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

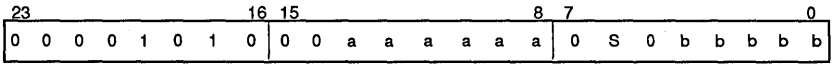
A

BCLR — Bit Test and Clear — BCLR

Instruction Format:

BCLR #n,X:aa
BCLR #n,Y:aa

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

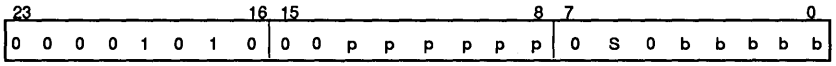
A

BCLR Bit Test and Clear BCLR

Instruction Format:

BCLR #n,X:pp
BCLR #n,Y:pp

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit I/O Short Address=pppppp

I/O Short Address pppppp	Memory Space S		Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

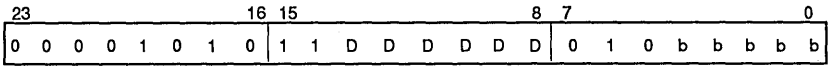
A

BCLR — Bit Test and Clear — BCLR

Instruction Format:

BCLR #n,D

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
D=destination register=DDDDDD
xxxx=16-bit Absolute Address in extension word

Destination Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See Section A.10 and Table A-18 for specific register encodings.

A

BCLR Bit Test and Clear BCLR

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The resulting 24 bit value is placed back into A1 or B1. A0 or B0 is cleared and the sign of A1 or B1 is extended into A2 or B2.
5. The bit test and clear is performed on A1 or B1, and the C bit is set if the bit tested is set.

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

BSET — Bit Test and Set — BSET

Operation:

D[n] → C;
1 → D[n]

Assembler Syntax:

BSET	#n,X:ea
BSET	#n,X:aa
BSET	#n,X:pp
BSET	#n,Y:ea
BSET	#n,Y:aa
BSET	#n,Y:pp
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 condition code 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.

Example:

```
        :
BSET    #$0,X:<<$FFE5    ;test and clear bit 14 in I/O Port B Data Reg.
        :
```

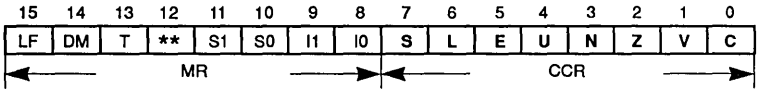
	Before Execution		After Execution
X:$FFE5	$000000	X:$FFE5	$000001
SR	$0300	SR	$0300

A

BSET Bit Test and Set BSET

Explanation of Example: Prior to execution, the 24-bit X location X:$FFE5 (I/O port C data register) contains the value $000000. The execution of the BSET #$0,X:<<$FFE5 instruction tests the state of the 0th bit in X:$FFE5, sets the carry bit C accordingly, and then sets the 0th bit in X:$FFE5.

Condition Codes:



CCR Condition Codes:

For destination operand SR:

C — **Set if bit 0 is specified. Not affected otherwise.**
V — **Set if bit 1 is specified. Not affected otherwise.**
Z — **Set if bit 2 is specified. Not affected otherwise.**
N — **Set if bit 3 is specified. Not affected otherwise.**
U — **Set if bit 4 is specified. Not affected otherwise.**
E — **Set if bit 5 is specified. Not affected otherwise.**
L — **Set if bit 6 is specified. Not affected otherwise.**
S — **Set if bit 7 is specified. Not affected otherwise.**

For destination operand A or B:

S —Computed according to the definition. See Notes on page A-63.
L — Set if data limiting has occurred. See Notes on page A-63.
E — **Not affected**
U — **Not affected**
N — **Not affected**
Z — **Not affected**
V — **Not affected**
C — **Set if bit tested is set. Cleared otherwise.**

A

BSET Bit Test and Set BSET

For other destination operands:

C — **Set if bit tested is set. Cleared otherwise.**
V — **Not affected**
Z — **Not affected**
N — **Not affected**
U — **Not affected**
E — **Not affected**
L — **Not affected**
S — **Not affected**

MR Status Bits:

For destination operand SR:

I0 — Set if bit 8 is specified. Not affected otherwise.
I1 — Set if bit 9 is specified. Not affected otherwise.
S0 — Set if bit 10 is specified. Not affected otherwise.
S1 — Set if bit 11 is specified. Not affected otherwise.
T — Set if bit 13 is specified. Not affected otherwise.
DM — Set if bit 14 is specified. Not affected otherwise
LF — Set if bit 15 is specified. Not affected otherwise.

For other destination operands:

I0 — Not affected
I1 — Not affected
S0 — Not affected
S1 — Not affected
T — Not affected
DM — Not affected
LF — Not affected

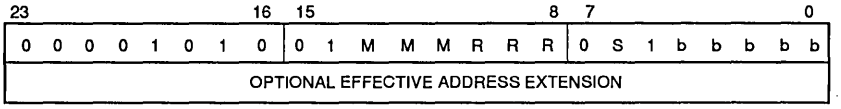
A

BSET — Bit Test and Set — BSET

Instruction Format:

BSET #n,X:ea
BSET #n,Y:ea

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r
Absolute address	1 1 0 0 0 0

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where "rrr" refers to an address register R0-R7

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

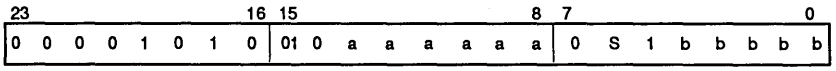
A

BSET — Bit Test and Set — BSET

Instruction Format:

BSET #n,X:aa
BSET #n,Y:aa

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

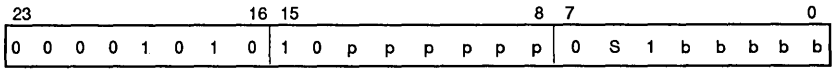
A

BSET — Bit Test and Set — BSET

Instruction Format:

BSET #n,X:pp
BSET #n,Y:pp

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit I/O Short Address=pppppp

I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

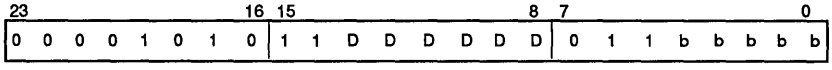
A

BSET — Bit Test and Set — BSET

Instruction Format:

BSET #n,D

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
D=destination register=DDDDDD
xxxx=16-bit Absolute Address in extension word

Destination Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See Section A.10 and Table A-18 for specific register encodings.

A

BSET Bit Test and Set BSET

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The resulting 24 bit value is placed back into A1 or B1. A0 or B0 is cleared and the sign of A1 or B1 is extended into A2 or B2.
5. The bit test and set is performed on A1 or B1, and the C bit is set if the bit tested is set.

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

BTST Bit Test BTST

Operation:

D[n] → C;
D[n] → C;
D[n] → C;
D[n] → C;
D[n] → C;
D[n] → C;
D[n] → C;

Assembler Syntax:

BTST #n,X:ea
BTST #n,X:aa
BTST #n,X:pp
BTST #n,Y:ea
BTST #n,Y:aa
BTST #n,Y:pp
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 condition code register. The bit to be tested is selected by an immediate bit number from 0–23. This instruction is useful for performing serial to parallel conversion when used with the appropriate rotate instructions. This instruction can use all memory alterable addressing modes.

Example:

```
        :
BTST    #$0,X:<<$FFEE   ;read SSI serial input flag IF1 into C bit
ROL     A               ;rotate carry bit C into LSB of A1
        :
```

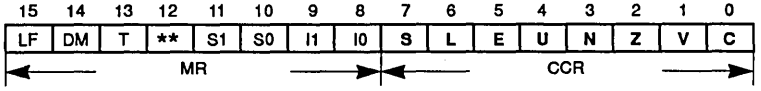
	Before Execution		After Execution
X:$FFEE	$000002	X:$FFEE	$000002
SR	$0300	SR	$0301

Explanation of Example: Prior to execution, the 24-bit X location X:$FFEE (I/O SSI status register) contains the value $000002. The execution of the BTST #$1,X:<<$FFEE instruction tests the state of the 1st bit (serial input flag IF1) in X:$FFEE and sets the carry bit C accordingly. This instruction sequence illustrates serial to parallel conversion using the carry bit C and the 24-bit A1 register.

A

BTST Bit Test BTST

Condition Codes:



CCR Condition Codes:

For destination operand A or B:

C — **Set if bit tested is set. Cleared otherwise.**
V — **Not affected**
Z — **Not affected**
N — **Not affected**
U — **Not affected**
E — **Not affected**
L — **Set if data limiting has occurred.** See Notes on page A-69.
S — **Computed according to the definition.** See Notes on page A-69.

For other destination operands:

C — **Set if bit tested is set. Cleared otherwise.**
V — **Not affected**
Z — **Not affected**
N — **Not affected**
U — **Not affected**
E — **Not affected**
L — **Not affected**
S — **Not affected**

MR Status bits are not affected.

SP — Stack Pointer:

For destination operand SSH: SP — Decrement by 1.
For other destination operands: Not affected

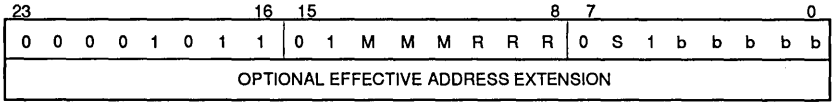
A

BTST Bit Test BTST

Instruction Format:

BTST #n,X:ea
BTST #n,Y:ea

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r
Absolute address	1 1 0 0 0 0

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where "rrr" refers to an address register R0-R7

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

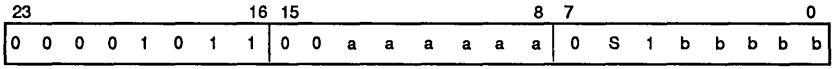
A

BTST — Bit Test — BTST

Instruction Format:

BTST #n,X:aa
BTST #n,Y:aa

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

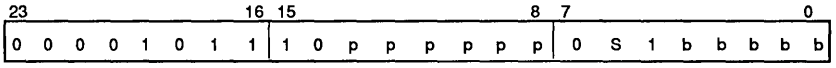
A

BTST — Bit Test — BTST

Instruction Format:

BTST #n,X:pp
BTST #n,Y:pp

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit I/O Short Address=pppppp

I/O Short Address pppppp	Memory Space S	Bit Number bbbbb
000000	X Memory 0	00000
•	Y Memory 1	•
•		•
111111		10111

Timing: 4+mvb oscillator clock cycles

Memory: 1+ea program words

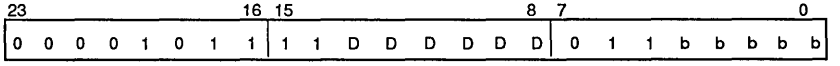
A

BTST Bit Test BTST

Instruction Format:

BTST #n,D

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
D=destination register=DDDDDD,
xxxx=16-bit Absolute Address in extension word

Destination Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See Section A.10 and Table A-18 for specific register encodings.

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The bit test is performed on the resulting 24-bit value and the C bit is set if the bit tested is set. The original contents of A or B are not changed.

Timing: 4+mvb oscillator clock cycles
Memory: 1+ea program words

CLR — Clear Accumulator — CLR

Operation:
0 →D (parallel move)

Assembler Syntax:
CLR D (parallel move)

Description: Clear the destination accumulator. This is a 56-bit clear instruction.

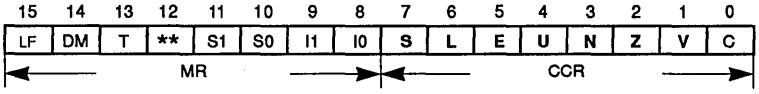
Example:

```
        :
CLR  A    #$7F,N          ;clear A, set up N0 addr. reg.
        :
```

	Before Execution		After Execution
A	$12:345678:9ABCDE	A	$00:000000:000000

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $12:345678:9ABCDE. The execution of the CLR A instruction clears the 56-bit A accumulator to zero.

Condition Codes:



S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
E — **Always cleared**
U — **Always set**
N — **Always cleared**
Z— **Always set**
V — **Always cleared**

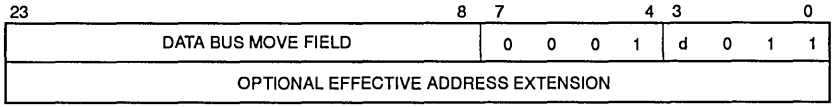
A

CLR — Clear Accumulator — CLR

Instruction Format:

CLR D

Opcode:



Instruction Fields:

D	d
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

CMP Compare CMP

Operation:
S2 – S1(parallel move)

Assembler Syntax:
CMP S1, S2 (parallel move)

Description: Subtract the source one operand, S1, from the source two accumulator, S2, and update the condition code register. The result of the subtraction operation is not stored.

Note: This instruction subtracts 56-bit operands. When a word is specified as S1, it is sign extended and zero filled to form a valid 56-bit operand. For the carry to be set correctly as a result of the subtraction, S2 must be properly sign extended. S2 can be improperly sign extended by writing A1 or B1 explicitly prior to executing the compare so that A2 or B2, respectively, may not represent the correct sign extension. This note particularly applies to the case where it is extended to compare 24-bit operands such as X0 with A1.

Example:

```
        :
CMP  Y0,B    X0,X:(R6)+N6    Y1,Y:(R0)-    ;comp. Y0 and B, save X0, Y1
        :
```

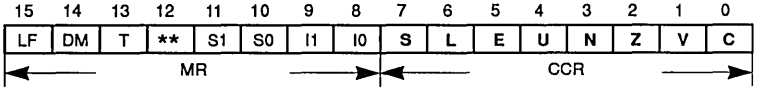
	Before Execution	After Execution
B	$00:000020:000000	$00:000020:000000
Y0	$000024	$000024
SR	$0300	$0319

Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $00:000020:000000 and the 24-bit Y0 register contains the value $000024. The execution of the CMP Y0,B instruction automatically appends the 24-bit value in the Y0 register with 24 LS zeros, sign extends the resulting 48-bit long word to 56 bits, subtracts the result from the 56-bit B accumulator and updates the condition code register.

A

CMP Compare CMP

Condition Codes:



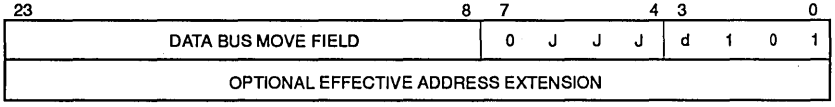
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format:

CMP S1, S2

Opcode:



Instruction Fields:

S1,S2	J J J d	S1,S2	J J J d
B,A	0 0 0 0	Y0,B	1 0 1 1
A,B	0 0 0 1	X1,A	1 1 0 0
X0,A	1 0 0 0	X1,B	1 1 0 1
X0,B	1 0 0 1	Y1,A	1 1 1 0
Y0,A	1 0 1 0	Y1,B	1 1 1 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

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 condition code register. The result of the subtraction operation is not stored.

Note: This instruction subtracts 56-bit operands. When a word is specified as S1, it is sign extended and zero filled to form a valid 56-bit operand. For the carry to be set correctly as a result of the subtraction, S2 must be properly sign extended. S2 can be improperly sign extended by writing A1 or B1 explicitly prior to executing the compare so that A2 or B2, respectively, may not represent the correct sign extension. This note particularly applies to the case where it is extended to compare 24-bit operands such as X0 with A1.

Example:

```
        :
CMPM    X1,A    BA,L:-(R4)      ;comp. Y0 and B, save X0, Y1
        :
```

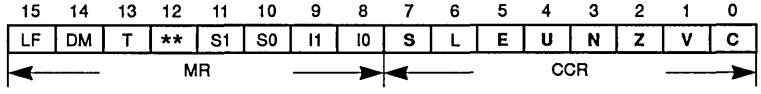
	Before Execution	After Execution
A	$00:000006:000000	$00:000006:000000
X1	$FFFFF7	$FFFFF7
SR	$0300	$0319

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:000006:000000, and the 24-bit X1 register contains the value $FFFFF7. The execution of the CMPM X1,A instruction automatically appends the 24-bit value in the X1 register with 24 LS zeros, sign extends the resulting 48-bit long word to 56 bits, takes the absolute value of the resulting 56-bit number, subtracts the result from the absolute value of the contents of the 56-bit A accumulator, and updates the condition code register.

A

CMPM Compare Magnitude CMPM

Condition Codes:



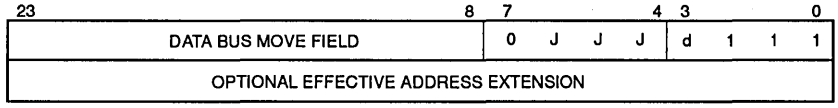
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during a parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result.

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format:

CMPM S1, S2

Opcode:



Instruction Fields:

S1,S2	J J J d	S1,S2	J J J d	S1,S2	J J J d
B,A	0 0 0 0	X0,B	1 0 0 1	X1,A	1 1 0 0
A,B	0 0 0 1	Y0,A	1 0 1 0	X1,B	1 1 0 1
X0,A	1 0 0 0	Y0,B	1 0 1 1	Y1,A	1 1 1 0
				Y1,B	1 1 1 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

DEBUG — Enter Debug Mode — DEBUG

Operation:

Enter the debug mode

Assembler Syntax:

DEBUG

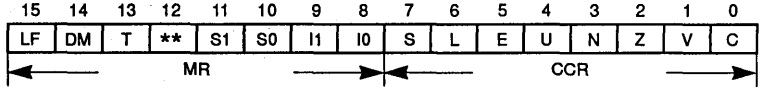
Description: Enter the debug mode and wait for OnCE commands.

Example:

```
        :
DEBUG           ;enter the debug mode
        :
```

Explanation of Example: Upon executing the DEBUG instruction, the chip enters the debug mode after the instruction following the DEBUG instruction has entered the instruction latch. Entering the debug mode is acknowledged by the chip by pulsing low the DSO line. This informs the external command controller that the chip has entered the debug mode and is waiting for commands.

Condition Codes:



The condition codes are not affected by this instruction

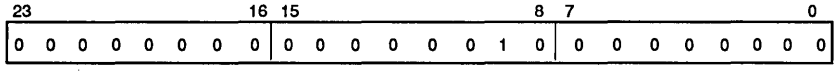
Instruction Format:

DEBUG

A

DEBUG Enter Debug Mode DEBUG

Opcode:



Timing: 4 oscillator clock cycles

Memory: 1 program word

A

DEBUGcc Enter Debug Mode Conditionally DEBUGcc

Operation:

If cc, then enter the debug mode

Assembler Syntax:

DEBUGcc

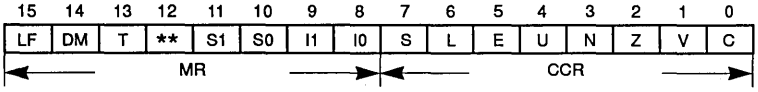
Description: If the specified condition is true, enter the debug mode and wait for OnCE commands. If the specified condition is false, continue with the next instruction.

The term "cc" may specify the following conditions:

	"cc" Mnemonic	Condition
CC (HS)	— carry clear (higher or same)	C=0
CS (LO)	— carry set (lower)	C=1
EC	— extension clear	E=0
EQ	— equal	Z=1
ES	— extension set	E=1
GE	— greater than or equal	N ⊕ V=0
GT	— greater than	Z+(N ⊕ V)=0
LC	— limit clear	L=0
LE	— less than or equal	Z+(N ⊕ V)=1
LS	— limit set	L=1
LT	— less than	N ⊕ V=1
MI	— minus	N=1
NE	— not equal	Z=0
NR	— normalized	Z+($\overline{U}$•$\overline{E}$)=1
PL	— plus	N=0
NN	— not normalized	Z+($\overline{U}$•$\overline{E}$)=0

where
$\overline{U}$ denotes the logical complement of U,
+ denotes the logical OR operator,
• denotes the logical AND operator, and
⊕ denotes the logical Exclusive OR operator

Condition Codes:



The condition codes are not affected by this instruction.

DEBUGcc Enter Debug Mode Conditionally DEBUGcc

Example:

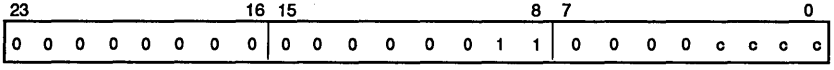
```
        :
CMP   Y0,   B          ; Compare register Y0 with the B accumulator.
DEBUGge                ; Enter the debug mode if
                       ; the previous test result is "greater than".
        :
```

Explanation of Example: The results of the comparison between Y0 and B will be recorded in the status register bits. The conditional debug instruction looks at the conditions (for greater than or equal in this case) and if they are met (N ⊕ V=0) then the DEBUG instruction will be executed. The chip enters the debug mode after the instruction following the DEBUG instruction has entered the instruction latch. The chip pulses low the DSO line to inform the external command controller that it has entered the debug mode and that the chip is waiting for commands.

Instruction Format:

DEBUGcc

Opcode:



Instruction Fields:

Mnemonic	c c c c	Mnemonic	c c c c
CC (HS)	0 0 0 0	CS (LO)	1 0 0 0
GE	0 0 0 1	LT	1 0 0 1
NE	0 0 1 0	EQ	1 0 1 0
PL	0 0 1 1	MI	1 0 1 1
NN	0 1 0 0	NR	1 1 0 0
EC	0 1 0 1	ES	1 1 0 1
LC	0 1 1 0	LS	1 1 1 0
GT	0 1 1 1	LE	1 1 1 1

Timing: 4 oscillator clock cycles

Memory: 1 program word

A

DEC — Decrement by One — DEC

Operation:
D−1 → D

Assembler Syntax:
DEC D

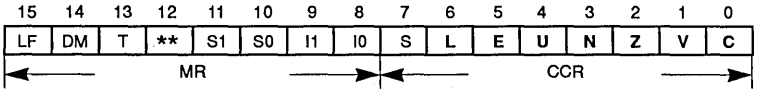
Description: Decrement by one the specified operand and store the result in the destination accumulator. One is subtracted from the LSB of D.

Example:

```
        :
DEC     A       ;Decrement the content of A accumulator by one
        :
```

Explanation of Example: One is subtracted from the content of the A accumulator.

Condition Codes:



L — Set if overflow has occurred in result. Not affected otherwise
E — Set if the signed integer portion of result is in use
U— Set if result is unnormalized
N — Set if bit 55 of result is set
Z — Set if result equals zero
V — Set if overflow has occurred in result
C — Set if a borrow occurs from bit 55 of result

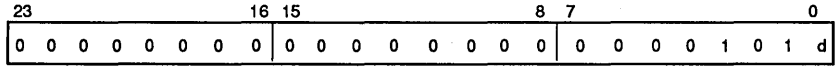
A

DEC — Decrement by One — DEC

Instruction Format:

DEC D

Opcode:



Instruction Fields:

D	d
A	0
B	1

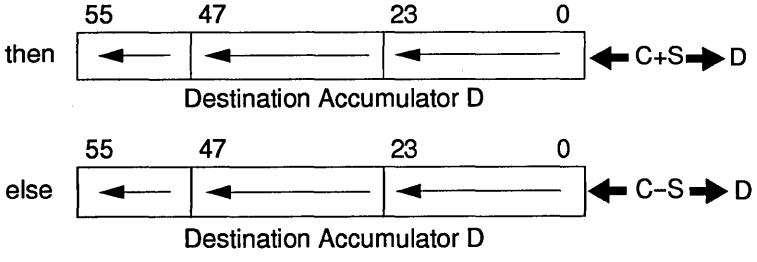
Timing: 2 oscillator clock cycles

Memory: 1 program word

A

DIV — Divide Interation — DIV

Operation: If D[55]⊕S[23]=1,



where ⊕ denotes the logical exclusive OR operator

Assembler Syntax: DIV S,D

Description:
Divide the destination operand D by the source operand S and store the result in the destination accumulator D. **The 48-bit dividend must be a positive fraction which has been sign extended to 56-bits and is stored in the full 56-bit destination accumulator D. The 24-bit divisor is a signed fraction and is stored in the source operand S.** Each DIV iteration calculates one quotient bit using a nonrestoring fractional division algorithm (see description on the next page). After the execution of the first DIV instruction, the destination operand holds both the partial remainder and the formed quotient. The partial remainder occupies the high-order portion of the destination accumulator D and is a signed fraction. The formed quotient occupies the low-order portion of the destination accumulator D (A0 or B0) and is a positive fraction. One bit of the formed quotient is shifted into the LS bit of the destination accumulator at the start of each DIV iteration. The formed quotient is the true quotient if the true quotient is positive. If the true quotient is negative, the formed quotient must be negated. **Valid results are obtained only when |D| < |S| and the operands are interpreted as fractions.** Note that this condition ensures that the magnitude of the quotient is less than one (i.e., is fractional) and precludes division by zero.

A

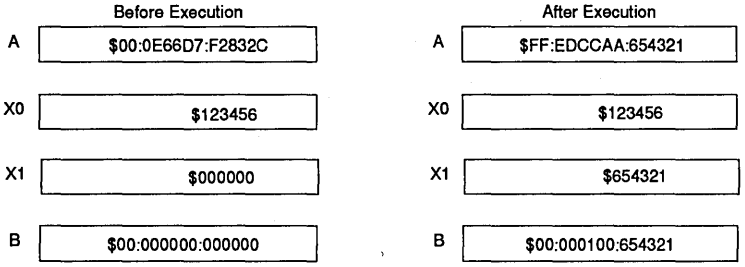
The DIV instruction calculates one quotient bit based on the divisor and the previous partial remainder. To produce an N-bit quotient, the DIV instruction is executed N times where N is the number of bits of precision desired in the quotient, 1;leN;le24. Thus, for a full-precision (24 bit) quotient, 24 DIV iterations are required. In general, executing the DIV instruction N times produces an N-bit quotient and a 48-bit remainder which has (48–N) bits of precision and whose N MS bits are zeros. The partial remainder is not a true remainder and must be corrected due to the nonrestoring nature of the division algorithm before it may be used. Therefore, once the divide is complete, it is necessary to reverse the last DIV operation and restore the remainder to obtain the true remainder.

The DIV instruction uses a nonrestoring fractional division algorithm which consists of the following operations (see the previous **Operation** diagram):

1. **Compare the source and destination operand sign bits:** An exclusive OR operation is performed on bit 55 of the destination operand D and bit 23 of the source operand S;
2. **Shift the partial remainder and the quotient:** The 55-bit destination accumulator D is shifted one bit to the left. The carry bit C is moved into the LS bit (bit 0) of the accumulator;
3. **Calculate the next quotient bit and the new partial remainder:** The 24-bit source operand S (signed divisor) is either added to or subtracted from the MSP portion of the destination accumulator (A1 or B1), and the result is stored back into the MSP portion of that destination accumulator. If the result of the exclusive OR operation previously described was a "1" (i.e., the sign bits were different), the source operand S is added to the accumulator. If the result of the exclusive OR operation was a "0" (i.e., the sign bits were the same), the source operand S is subtracted from the accumulator. Due to the automatic sign extension of the 24-bit signed divisor, the addition or subtraction operation correctly sets the carry bit C of the condition code register with the next quotient bit.

Example: (4-Quadrant division, 24-bit signed quotient, 48-bit signed remainder)

```
        ABS A A,B           ;make dividend positive, copy A1 to B1
        EOR X0,B B,X:$0     ;save rem. sign in X:$0, quo. sign in N
        AND #$FE,CCR        ;clear carry bit C (quotient sign bit)
        REP #$18            ;form a 24-bit quotient
        DIV X0,A            ;form quotient in A0, remainder in A1
        TFR A,B             ;save quotient and remainder in B1,B0
        JPL SAVEQUO         ;go to SAVEQUO if quotient is positive
        NEG B               ;complement quotient if N bit set
SAVEQUO TFR X0,B B0,X1      ;save quo. in X1, get signed divisor
        ABS B               ;get absolute value of signed divisor
        ADD A,B             ;restore remainder in B1
        JCLR #23,X:$0,DONE  ;go to DONE if remainder is positive
        MOVE #$0,B0         ;clear LS 24 bits of B
        NEG B               ;complement remainder if negative
DONE    .......
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the 56-bit, sign-extended fractional dividend D (D=$00.0E66D7:F2832C=0.112513535894635 approx.) and the 24-bit X0 register contains the 24-bit, signed fractional divisor S (S=$123456=0.142222166061401). Since |D|<|S|, the execution of the previous divide routine stores the correct 24-bit signed quotient in the 24-bit X1 register (A/X0=0.791111111164093=$654321=X1). The partial remainder is restored by reversing the last DIV operation and adding back the absolute value of the signed divisor in X0 to the partial remainder in A1. This produces the correct LS 24 bits of the 48-bit signed remained in the 24-bit B1 register. Note that the remainder is really a 48-bit value which has 24 bits of precision. Thus, the correct 48-bit remainder is $000000:000100 which equals 0.0000000000018190 approximately.

A

Note that the divide routine used in the previous example assumes that the sign-extended 56-bit signed fractional dividend is stored in the A accumulator and that the 24-bit signed fractional divisor is stored in the X0 register. This routine produces a **full 24-bit signed quotient and a 48-bit signed remainder.**

This routine may be greatly simplified for the case in which only positive, fractional operands are used to produce a 24-bit positive quotient and a 48-bit positive remainder, as shown in the following example:

```
1-Quadrant division, 24-bit unsigned quotient, 48-bit unsigned remainder
AND #$FE,CCR      ;clear carry bit C (quotient sign bit)
REP #$18          ;form a 24-bit quotient and remainder
DIV X0,A          ;form quotient in A0, remainder in A1
ADD X0,A          ;restore remainder in A1
```

Note that this routine assumes that the 56-bit positive, fractional, sign-extended dividend is stored in the A accumulator and that the 24-bit positive, fractional divisor is stored in the X0 register. After execution, the 24-bit positive fractional quotient is stored in the A0 register; the LS 24 bits of the 48-bit positive fractional remainder are stored in the A1 register.

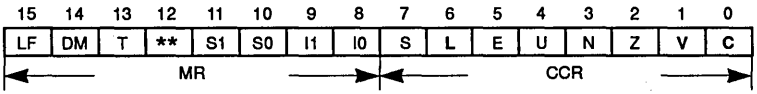
There are many variations possible when choosing a suitable division routine for a given application. The selection of a suitable division routine normally involves specification of the following items:

1. the number of bits of precision in the dividend;
2. the number of bits of precision N in the quotient;
3. whether the value of N is fixed or is variable;
4. whether the operands are unsigned or signed;
5. whether or not the remainder is to be calculated.

A complete discussion of the various division routines is beyond the scope of this manual. For a more complete discussion of these routines, refer to the application note entitled *Fractional and Integer Arithmetic Using the DSP56001.*

For extended precision division (i.e., for N-bit quotients where N>24), the DIV instruction is no longer applicable, and a user-defined N-bit division routine is required. For further information on division algorithms, refer to pages 524–530 of *Theory and Application of Digital Signal Processing* by Rabiner and Gold (Prentice-Hall, 1975), pages 190–199 of *Computer Architecture and Organization* by John Hayes (McGraw-Hill, 1978), pages 213–223 of *Computer Arithmetic: Principles, Architecture, and Design* by Kai Hwang (John Wiley and Sons, 1979), or other references as required.

Condition Codes:



L — Set if overflow bit V is set

V — **Set if the MS bit of the destination operand is changed as a result of the instruction's left shift operation**

C — **Set if bit 55 of the result is cleared.**

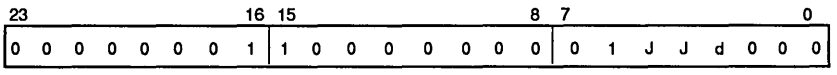
A

DIV — Divide Interation — DIV

Instruction Format:

DIV S,D

Opcode:



Instruction Fields:

S,D	J J d	S,D	J J d
X0,A	0 0 0	X1,A	1 0 0
X0,B	0 0 1	X1,B	1 0 1
Y0,A	0 1 0	Y1,A	1 1 0
Y0,B	0 1 1	Y1,B	1 1 1

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

Operation:	Assembler Syntax:	
SP+1 → SP;LA → SSH;LC → SSL;X:ea → LC SP+1 → SP;PC → SSH;SR → SSL;expr -1 → LA 1 → LF	DO	X:ea,expr
SP+1 → SP;LA → SSH;LC → SSL;X:aa → LC SP+1 → SP;PC → SSH;SR → SSL;expr -1 → LA 1 → LF	DO	X:aa,expr
SP+1 → SP;LA → SSH;LC → SSL;Y:ea → LC SP+1 → SP;PC → SSH;SR → SSL;expr -1 → LA 1 → LF	DO	Y:ea,expr
SP+1 → SP;LA → SSH;LC → SSL;Y:aa → LC SP+1 → SP;PC → SSH;SR → SSL;expr -1 → LA 1 → LF	DO	Y:aa,expr
SP+1 → SP;LA → SSH;LC → SSL;#xxx → LC SP+1 → SP;PC → SSH;SR → SSL;expr -1 → LA 1 → LF	DO	#xxx,expr
SP+1 → SP;LA → SSH;LC → SSL;S → LC SP+1 → SP;PC → SSH;SR → SSL;expr -1 → LA 1 → LF	DO	S,expr

End of Loop:
SSL(LF) → SR;SP-1 → SP
SSH → LA;SSL → LC;SP - 1 → SP

Description: Begin a hardware DO loop that is to be repeated the number of times specified in the instruction's source operand and whose range of execution is terminated by the destination operand (previously shown as "expr"). No overhead other than the execution of this DO instruction is required to set up this loop. DO loops can be nested and the loop count can be passed as a parameter.

A

During the first instruction cycle, the current contents of the loop address (LA) and the loop counter (LC) registers are pushed onto the system stack. The DO instruction's source operand is then loaded into the loop counter (LC) register. The LC register contains the remaining number of times the DO loop will be executed and can be accessed from inside the DO loop subject to certain restrictions. If LC equals zero, the DO loop is

executed 65,536 times. All address register indirect addressing modes may be used to generate the effective address of the source operand. If immediate short data is specified, the 12 LS bits of LC are loaded with the 12-bit immediate value, and the four MS bits of LC are cleared.

During the second instruction cycle, the current contents of the program counter (PC) register and the status register (SR) are pushed onto the system stack. The stacking of the LA, LC, PC, and SR registers is the mechanism which permits the nesting of DO loops. The DO instruction's destination operand (shown as "expr") is then loaded into the loop address (LA) register. This 16-bit operand is located in the instruction's 24-bit absolute address extension word as shown in the opcode section. The value in the program counter (PC) register pushed onto the system stack is the address of the first instruction following the DO instruction (i.e., the first actual instruction in the DO loop). This value is read (i.e., copied but not pulled) from the top of the system stack to return to the top of the loop for another pass through the loop.

During the third instruction cycle, the loop flag (LF) is set. This results in the PC being repeatedly compared with LA to determine if the last instruction in the loop has been fetched. If LA equals PC, the last instruction in the loop has been fetched and the loop counter (LC) is tested. If LC is not equal to one, it is decremented by one and SSH is loaded into the PC to fetch the first instruction in the loop again. If LC equals one, the "end-of-loop" processing begins.

When executing a DO loop, the instructions are actually fetched each time through the loop. Therefore, a DO loop can be interrupted. DO loops can also be nested. When DO loops are nested, the end-of-loop addresses must also be nested and are not allowed to be equal. The assembler generates an error message when DO loops are improperly nested. Nested DO loops are illustrated in the example.

Note: The assembler calculates the end-of-loop address to be loaded into LA (the absolute address extension word) by evaluating the end-of-loop expression "expr" and subtracting one. This is done to accommodate the case where the last word in the DO loop is a two-word instruction. Thus, the end-of-loop expression "expr" in the source code must represent the address of the instruction AFTER the last instruction in the loop as shown in the example.

During the "end-of-loop" processing, the loop flag (LF) from the lower portion (SSL) of SP is written into the status register (SR), the contents of the loop address (LA) register are restored from the upper portion (SSH) of (SP–1), the contents of the loop counter (LC) are restored from the lower portion (SSL) of (SP–1) and the stack pointer (SP) is decre-

mented by two. Instruction fetches now continue at the address of the instruction following the last instruction in the DO loop. Note that LF is the only bit in the status register (SR) that is restored after a hardware DO loop has been exited.

Note: The loop flag (LF) is cleared by a hardware reset.

Restrictions: The "end-of-loop" comparison previously described actually occurs at instruction fetch time. That is, LA is being compared with PC when the instruction at LA–2 is being executed. Therefore, instructions which access the program controller registers and/or change program flow cannot be used in locations LA–2, LA–1, or LA.

Proper DO loop operation is not guaranteed if an instruction **starting** at address **LA–2, LA–1**, or **LA** specifies one of the **program controller registers** SR, SP, SSL, LA, LC, or (implicitly) PC as a **destination** register. Similarly, the SSH program controller register may not be specified as a **source or destination** register in an instruction starting at address LA–2, LA–1, or LA. Additionally, the SSH register cannot be specified as a **source** register in the DO instruction itself and LA cannot be used as a **target** for **jumps to subroutine** (i.e., JSR, JScc, JSSET, or JSCLR to LA). A DO instruction cannot be repeated using the REP instruction.

The following instructions cannot **begin** at the indicated position(s) near the end of a DO loop:

At LA–2, LA–1, and LA	DO MOVEC from SSH MOVEM from SSH MOVEP from SSH MOVEC to LA, LC, SR, SP, SSH, or SSL MOVEM to LA, LC, SR, SP, SSH, or SSL MOVEP to LA, LC, SR, SP, SSH, or SSL ANDI MR ORI MR Two-word instructions which read LC, SP, or SSL
At LA–1	Single-word instructions (except REP) which read LC, SP, or SSL, JCLR, JSET, two-word JMP, two-word Jcc

A

At LA

any two-word instruction*

Jcc	REP
JCLR	RESET
JSET	RTI
JMP	RTS
JScc	STOP
JSR	WAIT

*This restriction applies to the situation in which the DSP56K simulator's single-line assembler is used to change the **last** instruction in a DO loop from a one-word instruction to a two-word instruction.

Other Restrictions:

DO SSH,xxxx
JSR to (LA) whenever the loop flag (LF) is set
JScc to (LA) whenever the loop flag (LF) is set
JSCLR to (LA) whenever the loop flag (LF) is set
JSSET to (LA) whenever the loop flag (LF) is set

A DO instruction cannot be repeated using the REP instruction.

Note: Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with a MOVE-type instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.This restriction also applies to the situation in which the **last** instruction in a **DO** loop changes an address register and the first instruction at the top of the **DO** loop uses that same address register. The **top** instruction becomes the **following** instruction because of the loop construct.

Similarly, since the DO instruction accesses the program controller registers, the DO instruction must not be immediately preceded by any of the following instructions:

Immediately before DO

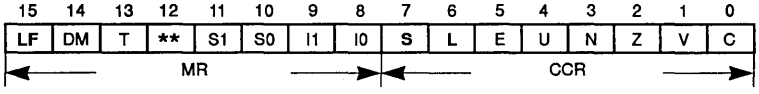
MOVEC to LA, LC, SSH, SSL, or SP
MOVEM to LA, LC, SSH, SSL, or SP
MOVEP to LA, LC, SSH, SSL, or SP
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH

Example:

```
        :
        DO #cnt1, END1          ;begin outer DO loop
              :
        DO #cnt2, END2          ;begin inner DO loop
              :
              :
        MOVE A,X:(R0)+
              :
END2                            ;last instruction in inner loop
        ADD  A,B X:(R1)+,X0     ;(in outer loop)
END1          :                 ;last instruction in outer loop
              :                 ;first instruction after outer loop
```

Explanation of Example: This example illustrates a nested DO loop. The outer DO loop will be executed "cnt1" times while the inner DO loop will be executed ("cnt1" * "cnt2") times. Note that the labels END1 and END2 are located at the first instruction past the end of the DO loop, as mentioned above, and are nested properly.

Condition Codes:



For source operand A or B:
LF — Set when a DO loop is in progress
S — Computed according to the definition. See Notes on page A-97.
L — Set if data limiting occurred. See Notes on page A-97.

For other source operands:

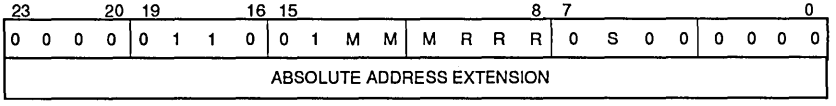
LF — Set when a DO loop is in progress

A

Instruction Format:

DO X:ea, expr
DO Y:ea, expr

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR,
expr=16-bit Absolute Address in 24-bit extension word

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r

Memory Space	S
X Memory	0
Y Memory	1

where "rrr" refers to an address register R0-R7

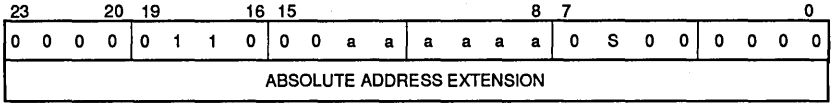
Timing: 6+mv oscillator clock cycles

Memory: 2 program words

Instruction Format:

DO X:aa, expr
DO Y:aa, expr

Opcode:



Instruction Fields:

ea=6-bit Effective Short Address=aaaaaa,
expr=16-bit Absolute Address in 24-bit extension word

Absolute Short Address aaaaaa	Memory Space S	
000000	X Memory	0
•	Y Memory	1
•		
111111		

Timing: 6+mv oscillator clock cycles

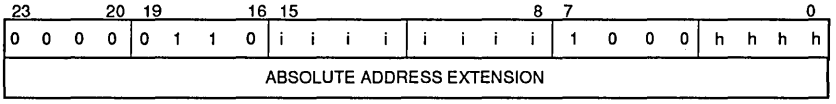
Memory: 2 program words

A

Instruction Format:

DO #xxx, expr

Opcode:



Instruction Fields:

#xxx=12-bit Immediate Short Data = hhhhiiiiiiii,
expr=16-bit Absolute Address in 24-bit extension word

Immediate Short Data hhhh iiiiiiii

000000000000
•
•
111111111111

Timing: 6+mv oscillator clock cycles

Memory: 2 program words

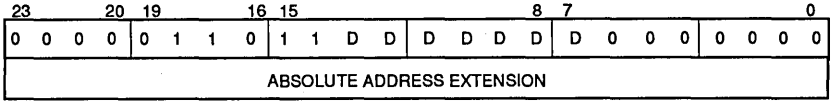
A

DO — Start Hardware Loop — DO

Instruction Format:

DO S, expr

Opcode:



Instruction Fields:

S=6-bit Source operand = DDDDDD,
expr=16-bit Absolute Address in 24-bit extension word

Source	D	D	D	D	D	D	S/L	Source	D	D	D	D	D	D
X0	0	0	0	1	0	0	no	SR	1	1	1	0	0	1
X1	0	0	0	1	0	1	no	OMR	1	1	1	0	1	0
Y0	0	0	0	1	1	0	no	SP*	1	1	1	0	1	1
Y1	0	0	0	1	1	1	no	SSL**	1	1	1	1	0	1
A0	0	0	1	0	0	0	no	LA	1	1	1	1	1	0
B0	0	0	1	0	0	1	no	LC	1	1	1	1	1	1
A2	0	0	1	0	1	0	no	R0-R7	0	1	0	r	r	r
B2	0	0	1	1	0	0	no	N0-N7	0	1	1	n	n	n
A1	0	0	1	1	0	1	no	M0-M7	1	0	0	m	m	m
A	0	0	1	1	1	0	yes [see Notes on page A-97]							
B	0	0	1	1	1	1	yes [see Notes on page A-97]							

where rrr=Rn register
where nnn=Nn register
where mmm=Mn register

*For DO SP, expr — The actual value that will be loaded into the loop counter (LC) is the value of the stack pointer (SP) before the execution of the DO instruction, incremented by 1.

Thus, if SP=3, the execution of the DO SP,expr instruction will load the loop counter (LC) with the value LC=4.

**For DO SSL, expr — The loop counter (LC) will be loaded with its previous value which was saved on the stack by the DO instruction itself.

A

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The LS 16 bits of the resulting 24 bit value is loaded into the loop counter (LC). The original contents of A or B are not changed.

Timing: 6+mv oscillator clock cycles

Memory: 2 program words

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 loop counter (LC) is needed, it must be read before the execution of the ENDDO instruction. Initially, the loop flag (LF) is restored from the system stack and the remaining portion of the status register (SR) and the program counter (PC) are purged from the system stack. The loop address (LA) and the loop counter (LC) registers are then restored from the system stack.

Restrictions: Due to pipelining and the fact that the ENDDO instruction accesses the program controller registers, the ENDDO instruction must not be immediately preceded by any of the following instructions:

Immediately before ENDDO	MOVEC to LA, LC, SR, SSH, SSL, or SP
	MOVEM to LA, LC, SR, SSH, SSL, or SP
	MOVEP to LA, LC, SR, SSH, SSL, or SP
	MOVEC from SSH
	MOVEM from SSH
	MOVEP from SSH
	ORI MR
	ANDI MR
	REP

Also, the ENDDO instruction cannot be the last (LA) instruction in a DO loop.

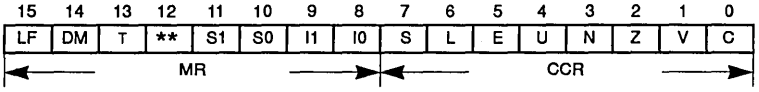
Example:

```
            :
            DO Y0,NEXT          ;exec. loop ending at NEXT (Y0) times
            :
            MOVEC LC,A          ;get current value of loop counter (LC)
            CMP Y1,A            ;compare loop counter with value in Y1
            JNE ONWARD          ;go to ONWARD if LC not equal to Y1
            ENDDO               ;LC equal to Y1, restore all DO registers
            JMP NEXT            ;go to NEXT
ONWARD      :                   ;LC not equal to Y1, continue DO loop
            :                   ;(last instruction in DO loop)
NEXT MOVE #$123456,X1           ;(first instruction AFTER DO loop)
```

A

Explanation of Example: This example illustrates the use of the ENDDO instruction to terminate the current DO loop. The value of the loop counter (LC) is compared with the value in the Y1 register to determine if execution of the DO loop should continue. Note that the ENDDO instruction updates certain program controller registers but does not automatically jump past the end of the DO loop. Thus, if this action is desired, a JMP instruction (i.e., JMP NEXT as previously shown) must be included after the ENDDO instruction to transfer program control to the first instruction past the end of the DO loop.

Condition Codes:

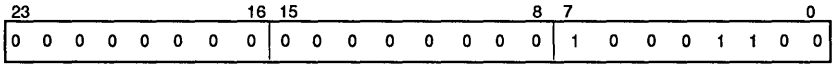


The condition codes are not affected by this instruction.

Instruction Format:

ENDDO

Opcode:



Instruction Fields:

None

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

EOR — Logical Exclusive OR — EOR

Operation:
S ⊕ D[47:24] →D[47:24] (parallel move)

Assembler Syntax:
EOR S,D (parallel move)

where ⊕ denotes the logical Exclusive OR operator

Description: Logically exclusive OR the source operand S with bits 47–24 of the destination operand D and store the result in bits 47–24 of the destination accumulator. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

Example:

```
        :
EOR Y1,B1    (R2)+        ;Exclusive OR Y1 with B1, update R2
        :
```

	Before Execution	After Execution
Y1	$000003	$000003
B	$00:000005:000000	$00:000006:000000

Explanation of Example: Prior to execution, the 24-bit Y1 register contains the value $000003, and the 56-bit B accumulator contains the value $00:000005:000000. The EOR Y1,B instruction logically exclusive ORs the 24-bit value in the Y1 register with bits 47–24 of the B accumulator (B1) and stores the result in the B accumulator with bits 55–48 and 23–0 unchanged.

Condition Codes:

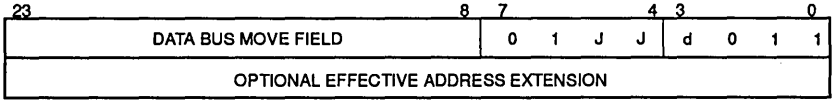
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
←				MR			→	←			CCR			→	

S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z— **Set if bits 47 - 24 of A or B result are zero**
V — Always cleared

Instruction Format:

EOR S,D

Opcode:



Instruction Fields:

S	J J	D	d
X0	0 0	A	0
X1	1 0	B	1
Y0	0 1		
Y1	1 1		

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

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. If a fast interrupt is used with the ILLEGAL instruction, an infinite loop will be formed (an illegal instruction interrupt normally returns to the illegal instruction) which can only be broken by a hardware reset. Therefore, only long interrupts should be used. Exiting an illegal instruction is a fatal error. The long exception routine should indicate this condition and cause the system to be restarted.

If the ILLEGAL instruction is in a DO loop at LA and the instruction at LA–1 is being interrupted, then LC will be decremented twice due to the same mechanism that causes LC to be decremented twice if JSR, REP, etc. are located at LA. This is why JSR, REP, etc. at LA are restricted. Clearly restrictions cannot be imposed on illegal instructions.

Since REP is uninterruptable, repeating an ILLEGAL instruction results in the interrupt not being initiated until after completion of the REP. After servicing the interrupt, program control will return to the address of the second word following the ILLEGAL instruction. Of course, the ILLEGAL interrupt service routine should abort further processing, and the processor should be reinitialized.

Example:

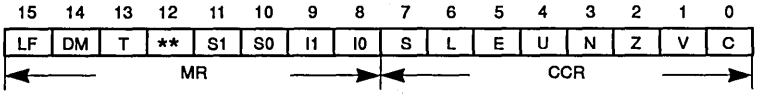
```
        :
ILLEGAL                 ;begin ILLEGAL exception processing
        :
```

Explanation of Example: The ILLEGAL instruction suspends normal instruction execution and initiates ILLEGAL exception processing.

A

ILLEGAL Illegal Instruction Interrupt ILLEGAL

Condition Codes:

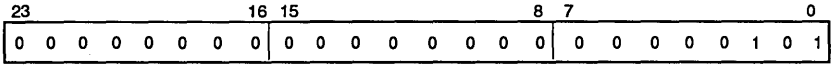


The condition codes are not affected by this instruction.

Instruction Format:

ILLEGAL

Opcode:



Instruction Fields:

None

Timing: 8 oscillator clock cycles

Memory: 1 program word

A

INC — Increment by One — INC

Operation:
D-1 → D

Assembler Syntax:
INC D

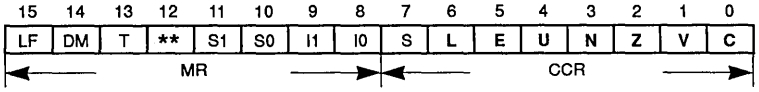
Description: Increment by one the specified operand and store the result in the destination accumulator. One is added from the LSB of D.

Example:

```
        :
INC     B       ;Increment the content of the B accumulator by one
        :
```

Explanation of Example: One is added to the content of the B accumulator.

Condition Codes:



L — Set if overflow has occurred in A or B result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry is generated from bit 55 of A or B result

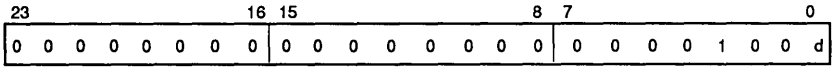
A

INC — Increment by One — INC

Instruction Format:

INC D

Opcode:



Instruction Fields:

D	d
A	0
B	1

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

Operation:

If cc, then 0xxx →PC
else PC+1 →PC

If cc, then ea →PC
else PC+1 →PC

Assembler Syntax:

Jcc xxx

Jcc xxx

Description: Jump to the location in program memory given by the instruction's effective address if the specified condition is true. If the specified condition is false, the program counter (PC) is incremented and the effective address is ignored. However, the address register specified in the effective address field is always updated independently of the specified condition. All memory alterable addressing modes may be used for the effective address. A Fast Short Jump addressing mode may also be used. The 12-bit data is zero extended to form the effective address. See Section A.9 for restrictions. The term "cc" may specify the following conditions:

	"cc" Mnemonic	Condition
CC (HS)	— carry clear (higher or same)	C=0
CS (LO)	— carry set (lower)	C=1
EC	— extension clear	E=0
EQ	— equal	Z=1
ES	— extension set	E=1
GE	— greater than or equal	N ⊕ V=0
GT	— greater than	Z+(N ⊕ V)=0
LC	— limit clear	L=0
LE	— less than or equal	Z+(N ⊕ V)=1
LS	— limit set	L=1
LT	— less than	N ⊕ V=1
MI	— minus	N=1
NE	— not equal	Z=0
NR	— normalized	$Z+(\overline{U}\bullet\overline{E})=1$
PL	— plus	N=0
NN	— not normalized	$Z+(\overline{U}\bullet\overline{E})=0$

where
$\overline{U}$ denotes the logical complement of U,
+ denotes the logical OR operator,
• denotes the logical AND operator, and
⊕ denotes the logical Exclusive OR operator

A

Restrictions: A Jcc instruction used **within a DO loop** cannot begin at the address LA within that DO loop.

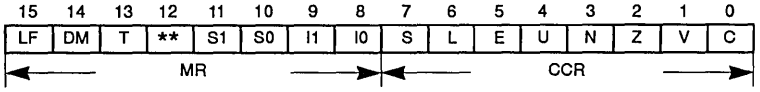
A Jcc instruction cannot be repeated using the REP instruction.

Example:

```
        :
JNN – (R4)                ;jump to P:(R4) –1 if not normalized
        :
```

Explanation of Example: In this example, program execution is transferred to the address P:(R4)–1 if the result is not normalized. Note that the contents of address register R4 are predecremented by 1, and the resulting address is then loaded into the program counter (PC) if the specified condition is true. If the specified condition is not true, no jump is taken, and the program counter is incremented by one.

Condition Codes:

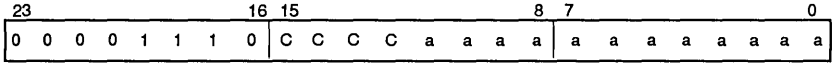


The condition codes are not affected by this instruction.

Instruction Format:

Jcc xxx

Opcode:



A

Instruction Fields:
cc=4-bit condition code=CCCC,
xxx=12-bit Short Jump Address=aaaaaaaaaaaa

Mnemonic	C C C C	Mnemonic	C C C C
CC (HS)	0 0 0 0	CS (LO)	1 0 0 0
GE	0 0 0 1	LT	1 0 0 1
NE	0 0 1 0	EQ	1 0 1 0
PL	0 0 1 1	MI	1 0 1 1
NN	0 1 0 0	NR	1 1 0 0
EC	0 1 0 1	ES	1 1 0 1
LC	0 1 1 0	LS	1 1 1 0
GT	0 1 1 1	LE	1 1 1 1

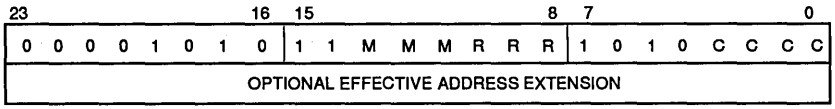
Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

Jcc ea

Opcode:



Instruction Fields:
cc=4-bit condition code=CCCC,
ea=6-bit Effective Address=MMMRRR

A

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r
Absolute Address	1	1	0	0	0	0

where "rrr" refers to an address register R0-R7

Mnemonic	C	C	C	C	Mnemonic	C	C	C	C
CC (HS)	0	0	0	0	CS (LO)	1	0	0	0
GE	0	0	0	1	LT	1	0	0	1
NE	0	0	1	0	EQ	1	0	1	0
PL	0	0	1	1	MI	1	0	1	1
NN	0	1	0	0	NR	1	1	0	0
EC	0	1	0	1	ES	1	1	0	1
LC	0	1	1	0	LS	1	1	1	0
GT	0	1	1	1	LE	1	1	1	1

Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

JCLR — Jump If Bit Clear — JCLR

Operation:

If S[n]=0, then xxxx→PC
else PC+1→PC

If S[n]=0, then xxxx →PC
else PC+1 →PC

If S[n]=0, then xxxx →PC
else PC+1 →PC

If S[n]=0, then xxxx →PC
else PC+1 →PC

If S[n]=0, then xxxx →PC
else PC+1 →PC

If S[n]=0, then xxxx →PC
else PC+1 →PC

If S[n]=0, then xxxx →PC
else PC+1 →PC

Assembler Syntax:

JCLR #n,X:ea,xxxx

JCLR #n,X:aa,xxxx

JCLR #n,X:pp,xxxx

JCLR #n,Y:ea,xxxx

JCLR #n,Y:aa,xxxx

JCLR #n,Y:pp,xxxx

JCLR #n,S,xxxx

Description: Jump to the 16-bit absolute address in program memory specified in the instruction's 24-bit extension word if the 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 may be used to reference the source operand S. Absolute Short and I/O Short addressing modes may also be used.

Restrictions: A JCLR instruction cannot be repeated using the REP instruction.

A JCLR located at LA, LA–1, or LA–2 of the DO loop cannot specify the program controller registers SR, SP, SSH, SSL, LA, or LC as its target.

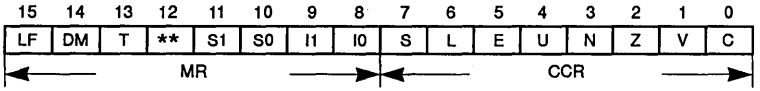
JCLR SSH or JCLR SSL cannot **follow** an instruction that changes the SP.

Example:

```
        :
JCLR #$5,X:<<$FFF1,$1234      ;go to P:$1234 if bit 5 in SCI SSR is clear
        :
```

Explanation of Example: In this example, program execution is transferred to the address P:$1234 if bit 5 (PE) of the 8-bit read-only X memory location X:$FFF1 (I/O SCI interface status register) is a zero. If the specified bit is not clear, no jump is taken, and the program counter (PC) is incremented by one.

Condition Codes:



For destination operand A or B:

S —Computed according to the definition. See Notes on page A-115.
L — Set if data limiting has occurred. See Notes on page A-115.
E — **Not affected**
U — **Not affected**
N — **Not affected**
Z — **Not affected**
V — **Not affected**
C — **Not affected**

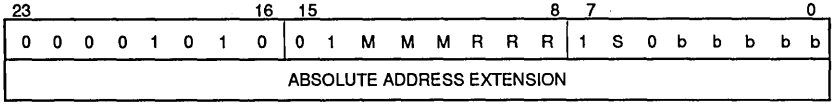
For other source operands:
The condition codes are not affected.

A

Instruction Format:

JCLR #n,X:ea,xxxx
JCLR #n,Y:ea,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR
xxxx=16-bit Absolute Address in extension word

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where "rrr" refers to an address register R0-R7

Timing: 6+jx oscillator clock cycles

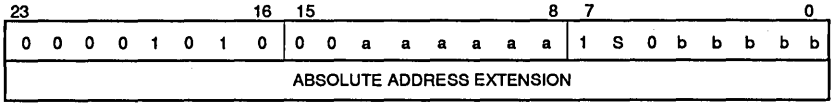
Memory: 2 program words

JCLR — Jump If Bit Clear — JCLR

Instruction Format:

JCLR #n,X:aa,xxxx
JCLR #n,Y:aa,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa
xxxx=16-bit Absolute Address in extension word

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

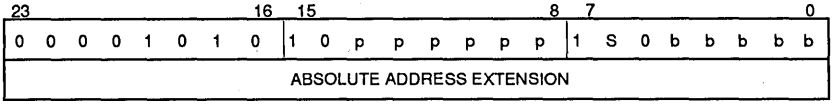
A

JCLR — Jump if Bit Clear — JCLR

Instruction Format:

JCLR #n,X:pp,xxxx
JCLR #n,Y:pp,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
pp=6-bit I/O Short Address=pppppp
xxxx=16-bit Absolute Address in extension word

I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

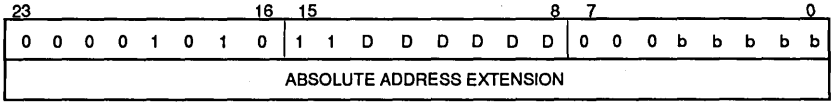
A

JCLR — Jump if Bit Clear — JCLR

Instruction Format:

JCLR #n,S,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
S=source register=DDDDDD
xxxx=16-bit Absolute Address in extension word

Source Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See Section A.10 and Table A-18 for specific register encodings.

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The bit test is performed on the resulting 24-bit value, and the jump is taken if the bit tested is clear. The original contents of A or B are not changed.

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Operation:
0xxx → PC
ea → PC

Assembler Syntax:
JMP xxx
JMP ea

Description: Jump to the location in program memory given by the instruction's effective address. All memory alterable addressing modes may be used for the effective address. A Fast Short Jump addressing mode may also be used. The 12-bit data is zero extended to form the effective address.

Restrictions: A JMP instruction used **within a DO loop** cannot begin at the address LA within that DO loop.

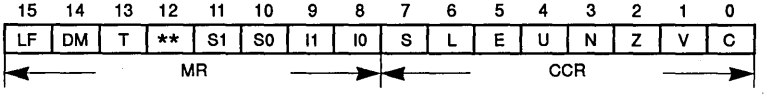
A JMP instruction cannot be repeated using the REP instruction.

Example:

```
        :
JMP (R1+N1)          ;jump to program address P:(R1+N1)
        :
```

Explanation of Example: In this example, program execution is transferred to the program address P:(R1+N1).

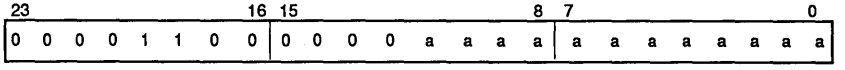
Condition Codes:



The condition codes are not affected by this instruction.

Instruction Format:
JMP xxx

Opcode:



A

Instruction Fields:

xxx=12-bit Short Jump Address=aaaaaaaaaaaa

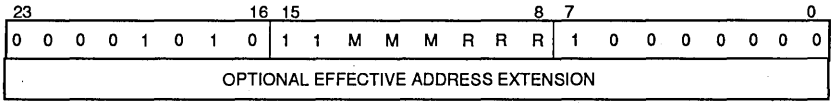
Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

JMP ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r
Absolute address	1 1 0 0 0 0

where "rrr" refers to an address register R0-R7

Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

A

JScc — Jump to Subroutine Conditionally — JScc

Operation:

If cc, then SP+1→SP; PC→SSH; SR→SSL; 0xxx→PC
else PC+1→PC

If cc, then SP+1→SP; PC→SSH; SR→SSL; ea→PC
else PC+1→PC

Assembler Syntax:

JScc xxx

JScc ea

Description: Jump to the subroutine whose location in program memory is given by the instruction's effective address if the specified condition is true. If the specified condition is true, the address of the instruction immediately following the JScc instruction (PC) and the system status register (SR) are pushed onto the system stack. Program execution then continues at the specified effective address in program memory. If the specified condition is false, the program counter (PC) is incremented, and any extension word is ignored. However, the address register specified in the effective address field is always updated independently of the specified condition. All memory alterable addressing modes may be used for the effective address. A fast short jump addressing mode may also be used. The 12-bit data is zero extended to form the effective address. The term "cc" may specify the following conditions:

	"cc" Mnemonic	Condition
CC (HS)	— carry clear (higher or same)	C=0
CS (LO)	— carry set (lower)	C=1
EC	— extension clear	E=0
EQ	— equal	Z=1
ES	— extension set	E=1
GE	— greater than or equal	$N \oplus V=0$
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	$Z+(\overline{U} \bullet \overline{E})=0$

A

where
$\overline{U}$ denotes the logical complement of U,
+ denotes the logical OR operator,
• denotes the logical AND operator, and
⊕ denotes the logical Exclusive OR operator

Restrictions: A JScc instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JScc instruction used **within in a DO loop** cannot begin at the address LA within that DO loop.

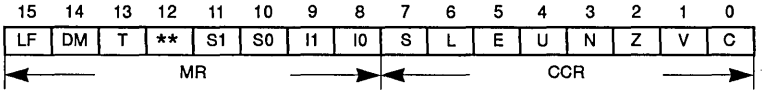
A JScc instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSLS (R3+N3)        ;jump to subroutine at P:(R3+N3) if limit set (L=1)
        :
```

Explanation of Example: In this example, program execution is transferred to the subroutine at address P:(R3+N3) in program memory if the limit bit is set (L=1). Both the return address (PC) and the status register (SR) are pushed onto the system stack prior to transferring program control to the subroutine if the specified condition is true. If the specified condition is not true, no jump is taken and the program counter is incremented by 1.

Condition Codes:

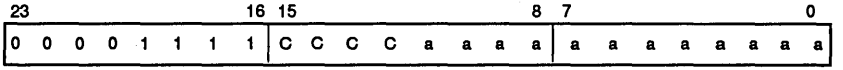


The condition codes are not affected by this instruction.

Instruction Format:

JScc xxx

Opcode:



Instruction Fields:

cc=4-bit condition code=CCCC,
xxx=12-bit Short Jump Address=aaaaaaaaaaaa

Mnemonic	C C C C	Mnemonic	C C C C
CC (HS)	0 0 0 0	CS (LO)	1 0 0 0
GE	0 0 0 1	LT	1 0 0 1
NE	0 0 1 0	EQ	1 0 1 0
PL	0 0 1 1	MI	1 0 1 1
NN	0 1 0 0	NR	1 1 0 0
EC	0 1 0 1	ES	1 1 0 1
LC	0 1 1 0	LS	1 1 1 0
GT	0 1 1 1	LE	1 1 1 1

Timing: 4+jx oscillator clock cycles

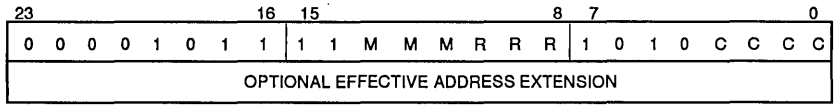
Memory: 1+ea program words

A

Instruction Format:

JScc ea

Opcode:



Instruction Fields:

cc=4-bit condition code=CCCC,
ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R	Mnemonic	C	C	C	C	Mnemonic	C	C	C	C
(Rn)-Nn	0	0	0	r	r	r	CC (HS)	0	0	0	0	CS (LO)	1	0	0	0
(Rn)+Nn0	0	0	1	r	r	r	GE	0	0	0	1	LT	1	0	0	1
(Rn)-	0	1	0	r	r	r	NE	0	0	1	0	EQ	1	0	1	0
(Rn)+	0	1	1	r	r	r	PL	0	0	1	1	MI	1	0	1	1
(Rn)	1	0	0	r	r	r	NN	0	1	0	0	NR	1	1	0	0
(Rn+Nn)	1	0	1	r	r	r	EC	0	1	0	1	ES	1	1	0	1
-(Rn)	1	1	1	r	r	r	LC	0	1	1	0	LS	1	1	1	0
Absolute address	1	1	0	0	0	0	GT	0	1	1	1	LE	1	1	1	1

where "rrr" refers to an address register R0-R7

Timing: 4+jx oscillator clock cycles

Memory: 1+ea program words

A

Operation:	Assembler Syntax
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx →PC else PC+1 →PC	JSCLR #n,X:ea,xxxx
f S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,X:aa,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,X:pp,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,Y:ea,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,Y:aa,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,Y:pp,xxxx
If S[n]=0, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSCLR #n,S,xxxx

Description: Jump to the subroutine at the 16-bit absolute address in program memory specified in the instruction's 24-bit extension word if the 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 system status register (SR) are pushed onto the system stack. Program execution then continues at the specified absolute address in the instruction's 24-bit extension word. If the specified memory bit is not clear, the program counter (PC) is incremented and the extension word is ignored. However, the address register

specified in the effective address field is always updated independently of the state of the nth bit. All address register indirect addressing modes may be used to reference the source operand S. Absolute short and I/O short addressing modes may also be used.

Restrictions: A JSCLR instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JSCLR located at LA, LA–1, or LA–2 of a DO loop, cannot specify the program controller registers SR, SP, SSH, SSL, LA, or LC as its target.

JSCLR SSH or JSCLR SSL cannot **follow** an instruction that changes the SP.

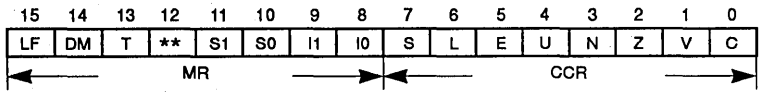
A JSCLR instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSCLR #$1,Y:<<$FFE3,$1357      ;go sub. at P:$1357 if bit 1 in Y:$FFE3 is clear
        :
```

Explanation of Example: In this example, program execution is transferred to the subroutine at absolute address P:$1357 in program memory if bit 1 of the external I/O location Y:<<$FFE3 is a zero. If the specified bit is not clear, no jump is taken and the program counter (PC) is incremented by 1.

Condition Codes:



For destination operand A or B:

S —Computed according to the definition. See Notes on page A-129.
L — Set if data limiting has occurred. See Notes on page A-129.
E — **Not affected**
U — **Not affected**
N — **Not affected**
Z — **Not affected**
V — **Not affected**
C — **Not affected**

For other source operands:
The condition codes are not affected.

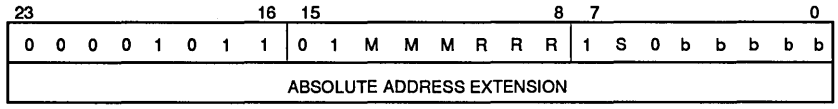
A

JSCLR Jump to Subroutine if Bit Clear JSCLR

Instruction Format:

JSCLR #n,X:ea,xxxx
JSCLR #n,Y:ea,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR,
xxxx=16-bit Absolute Address in extension word

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where "rrr" refers to an address register R0-R7

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

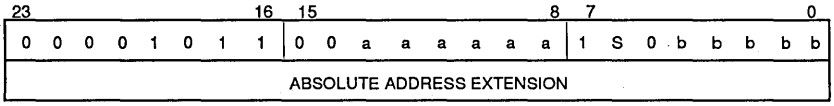
A

JSCLR Jump to Subroutine if Bit Clear JSCLR

Instruction Format:

JSCLR #n,X:aa,xxxx
JSCLR #n,Y:aa,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa,
xxxx=16-bit Absolute Address in extension word

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

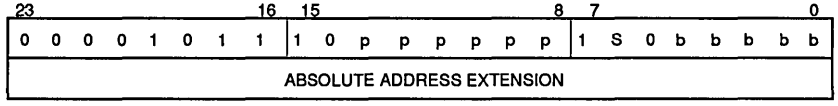
A

JSCLR Jump to Subroutine if Bit Clear JSCLR

Instruction Format:

JSCLR #n,X:pp,xxxx
JSCLR #n,Y:pp,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
pp=6-bit I/O Short Address=pppppp,
xxxx=16-bit Absolute Address in extension word

I/O Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

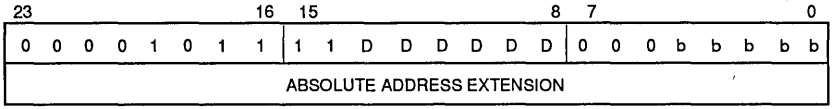
A

JSCLR Jump to Subroutine if Bit Clear JSCLR

Instruction Format:

JSCLR #n,S,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
S=source register=DDDDDD,
xxxx=16-bit Absolute Address in extension word

Source Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See Section A.10 and Table A-18 for specific register encodings.

A

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The bit test is performed on the resulting 24-bit value, and the jump to subroutine is taken if the bit tested is clear. The original contents of A or B are not changed.

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Operation:

If S[n]=0, then xxxx→PC
else PC+1→PC

If S[n]=1, then xxxx→PC
else PC+1→PC

If S[n]=1, then xxxx→PC
else PC+1→PC

If S[n]=1, then xxxx →PC
else PC+1→PC

If S[n]=1, then xxxx→PC
else PC+1→PC

If S[n]=1, then xxxx →PC
else PC+1→PC

If S[n]=1, then xxxx→PC
else PC+1→PC

If S[n]=1, then xxxx→PC
else PC+1→PC

Assembler Syntax:

JSET #n,X:ea,xxxx

JSET #n,X:ea,xxxx

JSET #n,X:aa,xxxx

JSET #n,X:pp,xxxx

JSET #n,Y:ea,xxxx

JSET #n,Y:aa,xxxx

JSET #n,Y:pp,xxxx

JSET #n,S,xxxx

Description: Jump to the 16-bit absolute address in program memory specified in the instruction's 24-bit extension word if the 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 may be used to reference the source operand S. Absolute short and I/O short addressing modes may also be used.

Restrictions: A JSET instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JSET located at LA, LA–1, or LA–2 of a DO loop cannot specify the program controller registers SR, SP, SSH, SSL, LA, or LC as its target.

JSET SSH or JSET SSL cannot follow an instruction that changes the SP.

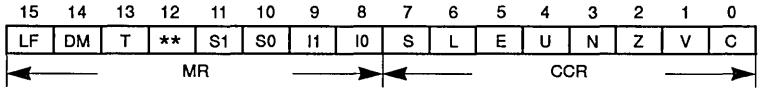
A JSET instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSET #12,X:<<$FFF2,$4321        ;$4321→(PC) if bit 12 (SCI COD) is set
        :
```

Explanation of Example: In this example, program execution is transferred to the address P:$4321 if bit 12 (SCI COD) of the 16-bit read/write I/O register X:$FFF2 is a one. If the specified bit is not set, no jump is taken and the program counter (PC) is incremented by 1.

Condition Codes:



For destination operand A or B:

S —Computed according to the definition. See Notes on page A-135.
L — Set if data limiting has occurred. See Notes on page A-135.
E — **Not affected**
U — **Not affected**
N — **Not affected**
Z — **Not affected**
V — **Not affected**
C — **Not affected**

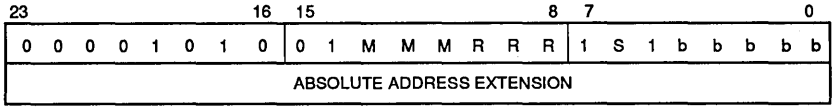
For other source operands:
The condition codes are not affected.

A

Instruction Format:

JSET #n,X:ea,xxxx
JSET #n,Y:ea,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR
xxxx=16-bit Absolute Address in extension word

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r

Memory Space	S
X Memory	0
Y Memory	1
	•
	•
	10111

Bit Number bbbbb
00000
•

where "rrr" refers to an address register R0-R7

Timing: 6+jx oscillator clock cycles

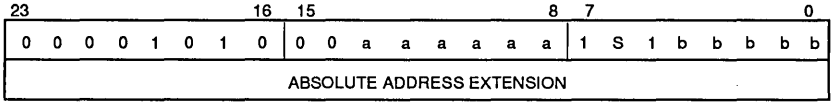
Memory: 2 program words

A

Instruction Format:

JSET #n,X:aa,xxxx
JSET #n,Y:aa,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa,
xxxx=16-bit Absolute Address in extension word

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 6+jx oscillator clock cycles

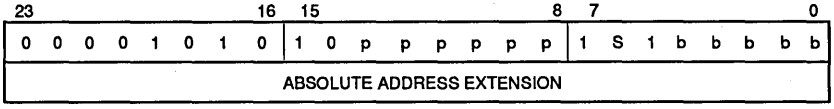
Memory: 2 program words

A

Instruction Format:

JSET #n,X:pp,xxxx
JSET #n,Y:pp,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
pp=6-bit I/O Short Address=pppppp,
xxxx=16-bit Absolute Address in extension word

I/O Short Address pppppp	Memory Space S		Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

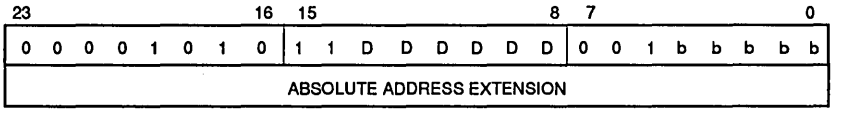
A

JSET — Jump if Bit Set — JSET

Instruction Format:

JSET #n,S,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
S=source register=DDDDDD,
xxxx=16-bit Absolute Address in extension word

Source Register	D D D D D D	Bit Number bbbbb
4 registers in Data ALU	0 0 0 1 D D	00000
8 accumulators in Data ALU	0 0 1 D D D	•
8 address registers in AGU	0 1 0 T T T	10111
8 address offset registers in AGU	0 1 1 N N N	
8 address modifier registers in AGU	1 0 0 F F F	
8 program controller registers	1 1 1 G G G	

See Section A.10 and Table A-18 for specific register encodings.

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The bit test is performed on the resulting 24-bit value, and the jump is taken if the bit tested is set. The original contents of A or B are not changed.

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

JSR — Jump to Subroutine — JSR

Operation: SP+1→SP; PC→SSH; SR→SSL; 0xxx→PC
Assembler Syntax: JSR xxx

SP+→SP; PC→SSH; SR→SSL; ea→PC
JSR ea

Description: Jump to the subroutine whose location in program memory is given by the instruction's effective address. The address of the instruction immediately following the JSR instruction (PC) and the system status register (SR) is pushed onto the system stack. Program execution then continues at the specified effective address in program memory. All memory alterable addressing modes may be used for the effective address. A fast short jump addressing mode may also be used. The 12-bit data is zero extended to form the effective address.

Restrictions: A JSR instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JSR instruction used **within a DO loop** cannot begin at the address LA within that DO loop.

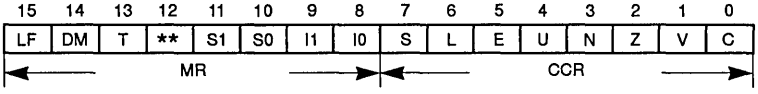
A JSR instruction cannot be repeated using the REP instruction.

Example:

```
        :
    JSR (R5)+          ;jump to subroutine at (R5), update R5
        :
```

Explanation of Example: In this example, program execution is transferred to the subroutine at address P:(R5) in program memory, and the contents of the R5 address register are then updated.

Condition Codes:



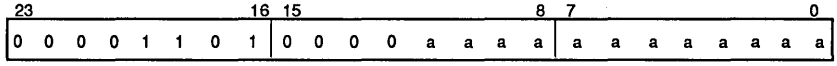
The condition codes are not affected by this instruction.

A

Instruction Format:

JSR xxx

Opcode:



Instruction Fields:

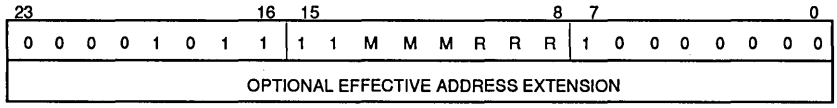
xxx=12-bit Short Jump Address=aaaaaaaaaaaa

Timing: 4+jx oscillator clock cycles
Memory: 1+ea program words

Instruction Format:

JSR ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r
Absolute address	1 1 0 0 0 0

where "rrr" refers to an address register R0-R7

Timing: 4+jx oscillator clock cycles
Memory: 1+ea program words

A

Operation:	Assembler Syntax
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET #n,X:ea,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET #n,X:aa,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET #n,X:pp,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET #n,Y:ea,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR →SSL; xxxx→PC else PC+1→PC	JSSET #n,Y:aa,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx →PC else PC+1→PC	JSSET #n,Y:pp,xxxx
If S[n]=1, then SP+1→SP; PC→SSH; SR→SSL; xxxx→PC else PC+1→PC	JSSET #n,S,xxxx

Description: Jump to the subroutine at the 16-bit absolute address in program memory specified in the instruction's 24-bit extension word if the 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 24-bit extension word. If the specified memory bit is not set, the program counter (PC) is incremented, and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the

n^{th} bit. All address register indirect addressing modes may be used to reference the source operand S. Absolute short and I/O short addressing modes may also be used.

Restrictions: A JSSET instruction used **within a DO loop** cannot specify the **loop address** (LA) as its target.

A JSSET located at LA, LA–1, or LA–2 of a DO loop, cannot specify the program controller registers SR, SP, SSH, SSL, LA, or LC as its target.

JSSET SSH or JSSET SSL cannot **follow** an instruction that changes the SP.

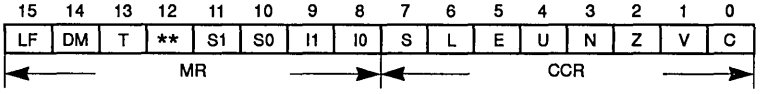
A JSSET instruction cannot be repeated using the REP instruction.

Example:

```
        :
JSSET #$17,Y:<$3F,$100      ;go to sub. at P:$0100 if bit 23 in Y:$3F is set
        :
```

Explanation of Example: In this example, program execution is transferred to the subroutine at absolute address P:$0100 in program memory if bit 23 of Y memory location Y:$003F is a one. If the specified bit is not set, no jump is taken and the program counter (PC) is incremented by 1.

Condition Codes:



For destination operand A or B:

S —Computed according to the definition. See Notes on page A-143.
L — Set if data limiting has occurred. See Notes on page A-143.
E — **Not affected**
U — **Not affected**
N — **Not affected**
Z — **Not affected**
V — **Not affected**
C — **Not affected**

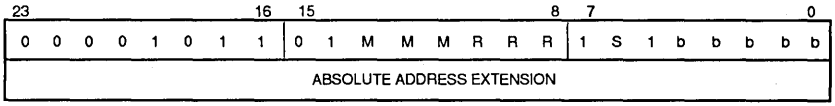
For other source operands:
The condition codes are not affected.

A

Instruction Format:

JSSET #n,X:ea,xxxx
JSSET #n,Y:ea,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
ea=6-bit Effective Address=MMMRRR,
xxxx=16-bit Absolute Address in extension word

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r

Memory Space	S
X Memory	0
Y Memory	1

Bit Number bbbbb
00000
•
•
•
10111

where "rrr" refers to an address register R0-R7

Timing: 6+jx oscillator clock cycles

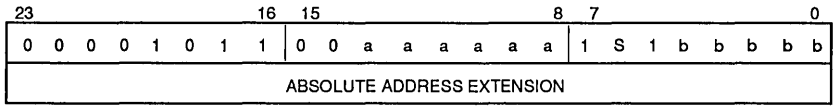
Memory: 2 program words

A

Instruction Format:

JSSET #n,X:aa,xxxx
JSSET #n,Y:aa,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
aa=6-bit Absolute Short Address=aaaaaa,
xxxx=16-bit Absolute Address in extension word

Absolute Short Address aaaaaa	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

Timing: 6+jx oscillator clock cycles

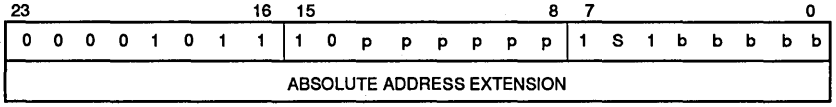
Memory: 2 program words

A

Instruction Format:

JSSET #n,X:pp,xxxx
JSSET #n,Y:pp,xxxx

Opcode:



Instruction Fields:

#n=bit number=bbbbb,
pp=6-bit I/O Short Address=pppppp,
xxxx=16-bit Absolute Address in extension word

I/O Short Address pppppp	Memory Space	S	Bit Number bbbbb
000000	X Memory	0	00000
•	Y Memory	1	•
•			10111
111111			

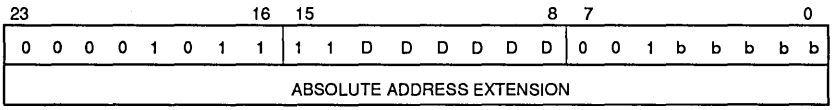
Timing: 6+jx oscillator clock cycles

Memory: 2 program words

A

Instruction Format:
JSSET #n,S,xxxx

Opcode:



Instruction Fields:
#n=bit number=bbbbb,
S=source register=DDDDDD,
xxxx=16-bit Absolute Address in extension word

Source Register	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
8 program controller registers	1 1 1 G G G

Bit Number bbbbb
00000
•
10111

See Section A.10 and Table A-18 for specific register encodings.

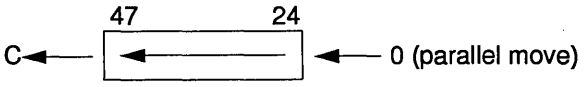
Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The bit test is performed on the resulting 24-bit value, and the jump to subroutine is taken if the bit tested is set. The original contents of A or B are not changed.

Timing: 6+jx oscillator clock cycles

Memory: 2 program words

Operation:



Assembler Syntax: LSL D (parallel move)

Description: Logically shift bits 47–24 of the destination operand D one bit to the left and store the result in the destination accumulator. Prior to instruction execution, bit 47 of D is shifted into the carry bit C, and a zero is shifted into bit 24 of the destination accumulator D. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected. If a zero shift count is specified, the carry bit is cleared. The difference between LSL and ASL is that LSL operates on only A1 or B1 and always clears the V bit.

Example:

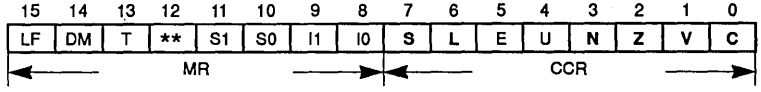
```
        :
LSL B1 #$7F,R0          ;shift B1 one bit to the left, set up R0
        :
```

	Before Execution		After Execution
B	$00:F01234:13579B	B	$00:E02468:13579B
SR	$0300	SR	$0309

Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $00:F01234:13579B. The execution of the LSL B instruction shifts the 24-bit value in the B1 register one bit to the left and stores the result back in the B1 register.

A

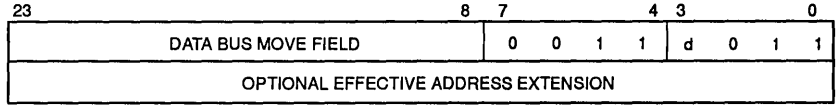
Condition Codes:



S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47–24 of A or B result are zero**
V — **Always cleared**
C — **Set if bit 47 of A or B was set prior to instruction execution**

Instruction Format:
LSL D

Opcode:



Instruction Fields:

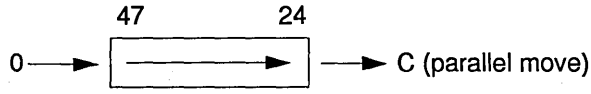
D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

Operation:

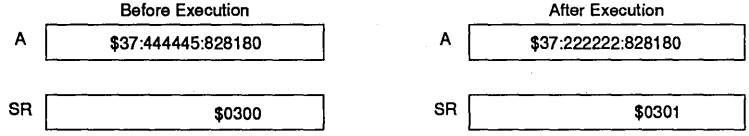


Assembler Syntax: LSR D (parallel move)

Description: Logically shift bits 47–24 of the destination operand D one bit to the right and store the result in the destination accumulator. Prior to instruction execution, bit 24 of D is shifted into the carry bit C, and a zero is shifted into bit 47 of the destination accumulator D. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

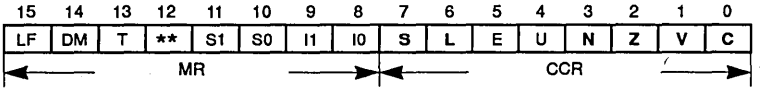
Example:

```
        :
LSR A1  A1,N4           ;shift A1 one bit to the right, set up N4
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $37:444445:828180. The execution of the LSR A instruction shifts the 24-bit value in the A1 register one bit to the right and stores the result back in the A1 register.

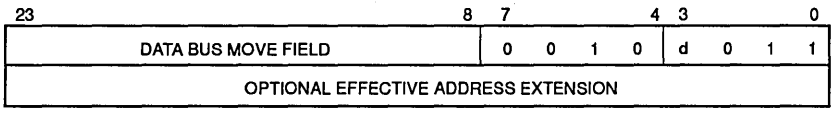
Condition Codes:



S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
N — **Always cleared**
Z— **Set if bits 47–24 of A or B result are zero**
V — **Always cleared**
C — **Set if bit 24 of A or B was set prior to instruction execution**

Instruction Format:
LSR D

Opcode:



Instruction Fields:

D d
A 0
B 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

LUA — Load Updated Address — LUA

Operation:
ea→d

Assembler Syntax:
LUA ea,D

Description: Load the updated address into the destination address register D. The source address register and the update mode used to compute the updated address are specified by the effective address (ea). **Note that the source address register specified in the effective address is not updated.** All update addressing modes may be used.

Note: This instruction is considered to be a move-type instruction. Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with this instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.

Example:

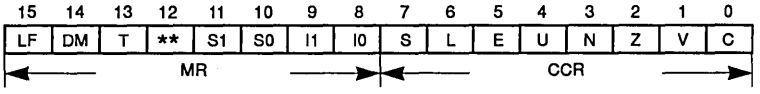
```
        :
LUA (R0)+N0,R1          ;update R1 using (R0)+N0
        :
```

	Before Execution	After Execution
R0	$0003	$0003
N0	$0005	$0005
M0	$FFFF	$FFFF
R1	$0004	$0008

Explanation of Example: Prior to execution, the 16-bit address register R0 contains the value $0003, the 16-bit address register N0 contains the value $0005, and the 16-bit address register R1 contains the value $0004. The execution of the LUA (R0)+N0,R1 instruction adds the contents of the R0 register to the contents of the N0 register and stores the resulting updated address in the R1 address register. Normally N0 would be added to R0 and deposited in R0. However, for an LUA instruction, the contents of both the R0 and N0 address registers are not affected.

A

Condition Codes:

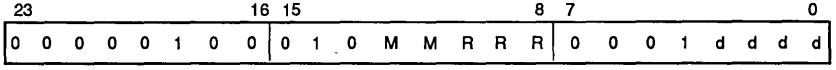


The condition codes are not affected by this instruction.

Instruction Format:

LUA ea,D

Opcode:



Instruction Fields:

ea=5-bit Effective Address=MMRRR,
D=4-bit destination address register=dddd

Effective Addressing Mode	M M M R R R	Dest. Addr. Reg. D	d d d d
(Rn)-Nn	0 0 0 r r r	R0–R7	0 n n n
(Rn)+Nn	0 0 1 r r r	N0–N7	1 n n n
(Rn)-	0 1 0 r r r		
(Rn)+	0 1 1 r r r		

where "rrr" refers to a **source** address register R0–R7
where "nnn" refers to a **destination** address register R0–R7 or N0–N7

Timing: 4 oscillator clock cycles

Memory: 1 program word

A

MAC — Signed Multiply-Accumulate — MAC

Operation:	**Assembler Syntax:**	
D±S1*S2→D (parallel move)	MAC	(±)S1,S2,D (parallel move)
D±S1*S2→D (parallel move)	MAC	(±)S2,S1,D (parallel move)
D±(S1*2^{-n})→D (**no** parallel move)	MAC	(±)S,#n,D (**no** parallel move)

Description: Multiply the two signed 24-bit source operands S1 and S2 (**or** the signed 24-bit source operand S by the positive 24-bit immediate operand 2^{-n}) and add/subtract the product to/from the specified 56-bit destination accumulator D. The "–" sign option is used to negate the specified product prior to accumulation. The default sign option is "+".

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 shown in Section 3.4 DOUBLE PRECISION MULTIPLY MODE:

MPY Y0, X0, A	MPY Y0, X0, B
MAC X1, Y0, A	MAC X1, Y0, B
MAC X0, Y1, A	MAC X0, Y1, B
MAC Y1, X1, A	MAC Y1, X1, B

All other Data ALU instructions are executed as NOP's when the processor is in the Double Precision Multiply Mode.

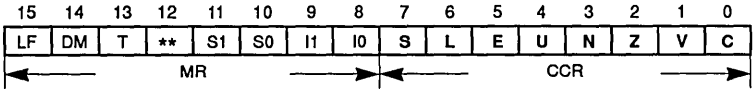
Example 1:

```
        :
MAC X0,X0,A     X:(R2)+N2,Y1     ;square X0 and store in A, update Y1 and R2
        :
```

	Before Execution		After Execution
X0	$123456	X0	$123456
A	$00:100000:00000	A	$00:1296CD:9619C8

Explanation of Example 1: Prior to execution, the 24-bit X0 register contains the value of $123456 (0.142222166), and the 56-bit A accumulator contains the value $00:100000:000000 (0.125). The execution of the MAC X0,X0,A instruction squares the 24-bit signed value in the X0 register and adds the resulting 48-bit product to the 56-bit A accumulator (X0*X0+lA=0.145227144519197 approximately= $00:1296CD:9619C8=A).

Condition Codes:



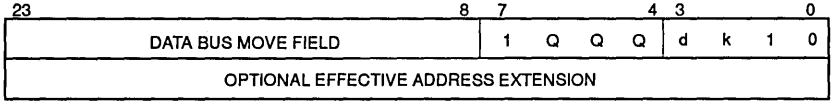
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION.
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format 1:

MAC (±)S1,S2,D
MAC (±)S2,S1,D

Opcode: 1



Instruction Fields:

S1*S2	Q Q Q	Sign	k	D d
X0 X0	0 0 0	+	0	A 0
Y0 Y0	0 0 1	–	1	B 1
X1 X0	0 1 0			
Y1 Y0	0 1 1			
X0 Y1	1 0 0			
Y0 X0	1 0 1			
X1 Y0	1 1 0			
Y1 X1	1 1 1			

Note: Only the indicated S1*S2 combinations are valid. X1*X1 and Y1*Y1 are **not** valid.

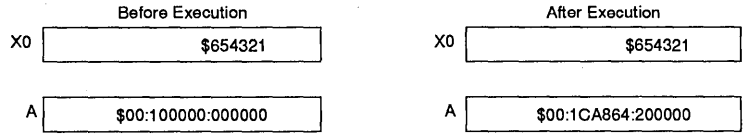
A

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Example 2:

```
        :
MAC   X0,   #3,  A          ;
        :
```

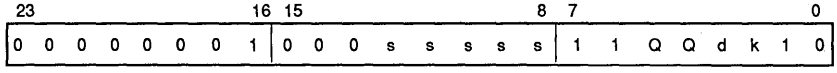


Explanation of Example 2: The content of X0 ($654321) is multiplied by 2^{-3} and then added to the content of the A accumulator ($00:100000:000000). The result is then placed in the A accumulator. The net effect of this operation is to **divide** the content of X0 **by 2^3** and add the result to the accumulator. An alternate interpretation is that X0 is **right shift**ed 3 places and filled with the sign bit (0 for a positive number and 1 for a negative number) and then the result is added to the accumulator.

Instruction Format 2:

MAC (±)S,#n,D

Opcode 2:



Instruction Fields:

S	Q	Q	Sign	k	D	d
Y1	0	0	+	0	A	0
X0	0	1	–	1	B	1
Y0	1	0				
X1	1	1				

n	sssss	constant
1	00001	010000000000000000000000
2	00010	001000000000000000000000
3	00011	000100000000000000000000
4	00100	000010000000000000000000
5	00101	000001000000000000000000
6	00110	000000100000000000000000
7	00111	000000010000000000000000
8	01000	000000001000000000000000
9	01001	000000000100000000000000
10	01010	000000000010000000000000
11	01011	000000000001000000000000
12	01100	000000000000100000000000
13	01101	000000000000010000000000
14	01110	000000000000001000000000
15	01111	000000000000000100000000
16	10000	000000000000000010000000
17	10001	000000000000000001000000
18	10010	000000000000000000100000
19	10011	000000000000000000010000
20	10100	000000000000000000001000
21	10101	000000000000000000000100
22	10110	000000000000000000000010
23	10111	000000000000000000000001

Timing: 2 oscillator clock cycles

Memory: 1 program word

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 24-bit source operands S1 and S2 (**or** the signed 24-bit source operand S by the positive 24-bit immediate operand 2^{-n}), add/subtract the product to/from the specified 56-bit destination accumulator D, and then round the result using convergent rounding. The rounded result is stored in the destination accumulator D.

The "–" sign option negates the specified product prior to accumulation. The default sign option is "+".

The contribution of the LS bits of the result is rounded into the upper portion of the destination accumulator (A1 or B1) by adding a constant to the LS bits of the lower portion of the accumulator (A0 or B0). The value of the constant added is determined by the scaling mode bits S0 and S1 in the status register. Once rounding has been completed, the LS bits of the destination accumulator D (A0 or B0) are loaded with zeros to maintain an unbiased accumulator value which may be reused by the next instruction. The upper portion of the accumulator (A1 or B1) contains the rounded result which may be read out to the data buses. Refer to the RND instruction for more complete information on the convergent rounding process.

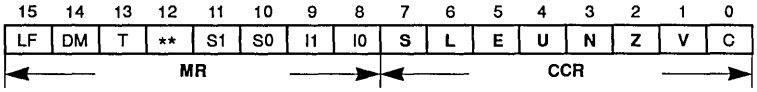
Example 1:

```
        :
MACR X0,Y0,B    B,X0   Y:(R4)+N4,Y0     ;X0*Y0+B→B, and B, update X0,Y0,R4
        :
```

	Before Execution	After Execution
X0	$123456	$100000
Y0	$123456	$987654
B	$00:100000:000000	$00:1296CE:000000

Explanation of Example 1: Prior to execution, the 24-bit X0 register contains the value $123456 (0.142222166), the 24-bit Y0 register contains the value $123456 (0.142222166), and the 56-bit B accumulator contains the value $00:100000:000000 (0.125). The execution of the MACR X0,Y0,B instruction multiples the 24-bit signed value in the X0 register by the 24-bit signed value in the Y0 register, adds the resulting product to the 56-bit B accumulator, rounds the result into the B1 portion of the accumulator, and then zeros the B0 portion of the accumulator (X0*Y0+B=0.145227144519197 approximately =$00:1296CD:9619C8, which is rounded to the value $00:1296CE:000000=0.145227193832397=B).

Condition Codes:



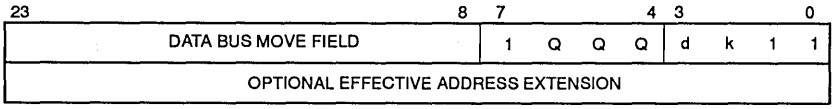
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 for complete details.

Instruction Format 1:

MACR (±)S1,S2,D
MACR (±)S2,S1,D

Opcode 1:



A

Instruction Fields 1:

S1*S2	Q	Q	Q	Sign	k	D	d
X0 X0	0	0	0	+	0	A	0
Y0 Y0	0	0	1	–	1	B	1
X1 X0	0	1	0				
Y1 Y0	0	1	1				
X0 Y1	1	0	0				
Y0 X0	1	0	1				
X1 Y0	1	1	0				
Y1 X1	1	1	1				

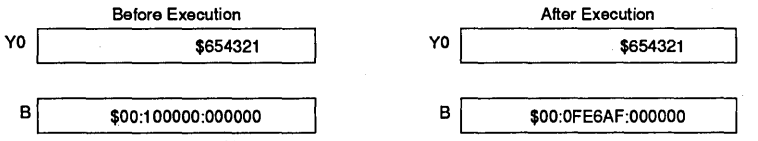
Note: Only the indicated S1*S2 combinations are valid. X1*X1 and Y1*Y1 are not valid.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Example 2:

```
        :
MACR  -Y0,   #10, B          ;
        :
```



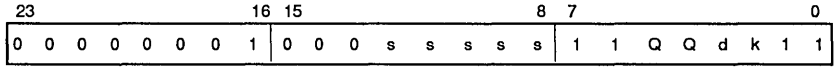
Explanation of Example 2: The content of Y0 ($654321) is negated, multiplied by 2^{-10}, added to the content of the B accumulator ($00:100000:000000), placed in the B accumulator and then rounded to a single precision number (24 bits in B1). The net effect of this operation is to negate the content of Y0, **divide** the result **by 2^{10}** and add the result to the accumulator. An alternate interpretation is that Y0 is negated, **right shift**ed 10 places, filled with the sign bit (0 for a positive number and 1 for a negative number), the result is added to the accumulator and then rounded to a single precision number.

A

Instruction Format 2:

MACR (±)S,#n,D

Opcode 2:



Instruction Fields 2:

S	Q Q	Sign	k	D	d
Y1	0 0	+	0	A	0
X0	0 1	–	1	B	1
Y0	1 0				
X1	1 1				

n	sssss	constant
1	00001	01000000000000000000000
2	00010	00100000000000000000000
3	00011	00010000000000000000000
4	00100	00001000000000000000000
5	00101	00000100000000000000000
6	00110	00000010000000000000000
7	00111	00000001000000000000000
8	01000	00000000100000000000000
9	01001	00000000010000000000000
10	01010	00000000001000000000000
11	01011	00000000000100000000000
12	01100	00000000000010000000000
13	01101	00000000000001000000000
14	01110	00000000000000100000000
15	01111	00000000000000010000000
16	10000	00000000000000001000000
17	10001	00000000000000000100000
18	10010	00000000000000000010000
19	10011	00000000000000000001000
20	10100	00000000000000000000100
21	10101	00000000000000000000010
22	10110	00000000000000000000001
23	10111	00000000000000000000000

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

Operation:
S→D

Assembler Syntax:
MOVE S,D

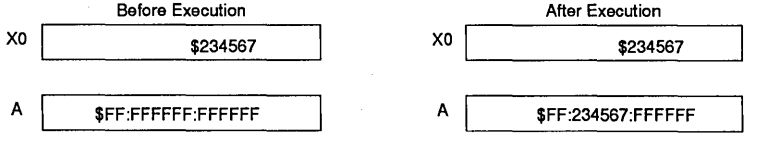
Description: Move the contents of the specified data source S to the specified destination D. This instruction is equivalent to a data ALU NOP with a parallel data move.

When a 56-bit accumulator (A or B) is specified as a **source** operand S, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). If the data out of the shifter indicates that the accumulator extension register is in use and the data is to be moved into a 24- or 48-bit destination, the value stored in the destination D is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the condition code register is latched.

When a 56-bit accumulator (A or B) is specified as a **destination** operand D, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 23) and appending the source operand with 24 LS zeros. Similarly, any 48-bit source data to be loaded into a 56-bit accumulator is automatically sign extended to 56 bits. Note that for 24-bit source operands both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1). Similarly, for 48-bit source operands, the automatic sign-extension feature may be disabled by using the long memory move addressing mode and specifying A10 or B10 as the destination operand.

Example:

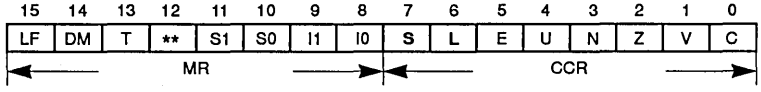
```
        :
MOVE X0,A1              ;move X0 to A1 without sign ext. or zeroing
        :
```



A

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $FF:FFFFFF:FFFFFF, and the 24-bit X0 register contains the value $234567. The execution of the MOVE X0,A1 instruction moves the 24-bit value in the X0 register into the 24-bit A1 register without automatic sign extension and without automatic zeroing.

Condition Codes:

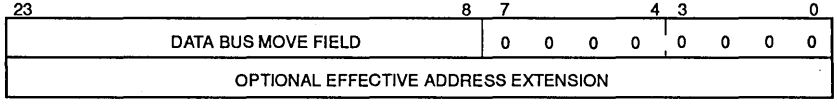


S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move.

Instruction Format:

MOVE S,D

Opcode:



Instruction Fields:

See **Parallel Move Descriptions** for data bus move field encoding.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

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. Addressing mode restrictions which apply to specific types of moves are noted in the individual move operation descriptions. The following section contains detailed descriptions about each type of parallel move operation.

When a 56-bit accumulator (A or B) is specified as a **source** operand S, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). If the data out of the shifter indicates that the accumulator extension register is in use and the data is to be moved into a 24- or 48-bit destination, the value stored in the destination D is limited to a maximum positive or negative saturation constant to minimize truncation error. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the condition code register is latched.

Note: Whenever an instruction uses an accumulator as both a destination operand for a data ALU operation and as a source for a parallel move operation, the parallel move operation occurs **first** and will use the data that exists in the accumulator **before the execution of the data ALU operation has occurred.**

When a 56-bit accumulator (A or B) is specified as a **destination** operand D, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 23) and appending the source operand with 24 LS zeros. Similarly, any 48-bit source data to be loaded into a 56-bit accumulator is automatically sign extended to 56 bits. Note that for 24-bit source operands both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1). Similarly, for 48-bit source operands, the automatic sign-extension feature may be disabled by using the long memory move addressing mode and specifying A10 or B10 as the destination operand.

Note that the symbols used in decoding the various opcode fields of an instruction or parallel move are **completely arbitrary**. Furthermore, the opcode symbols used in one instruction or parallel move are **completely independent** of the opcode symbols used in a different instruction or parallel move.

No Parallel Data Move

Operation:
(. . . . .)

Assembler Syntax:
(. . . . .)

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

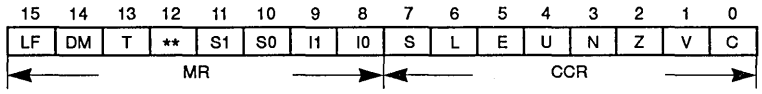
Description: Many (30 of the total 66) instructions in the DSP56K instruction set allow parallel moves. The parallel moves have been divided into 10 opcode categories. This category is a parallel move NOP and does not involve data bus move activity.

Example:

```
        :
ADD X0,A        ;add X0 to A (no parallel move)
        :
```

Explanation of Example: This is an example of an instruction which allows parallel moves but does not have one.

Condition Codes:



The condition codes are affected by the instruction, not the move.

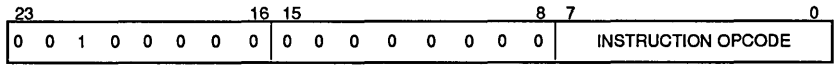
A

No Parallel Data Move

Instruction Format:

(. . . . .)

Opcode:



Instruction Format:

(defined by instruction)

Timing: mv oscillator clock cycles

Memory: mv program words

A

Immediate Short Data Move

Operation:
(. . . . .), #xx→D

Assembler Syntax:
(. . . . .) #xx,D

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move the 8-bit immediate data value (#xx) into the destination operand D.

If the destination register D is A0, A1, A2, B0, B1, B2, R0–R7, or N0–N7, the 8-bit immediate short operand is interpreted as an **unsigned integer** and is stored in the specified destination register. That is, the 8-bit data is stored in the eight LS bits of the destination operand, and the remaining bits of the destination operand D are zeroed.

If the destination register D is X0, X1, Y0, Y1, A, or B, the 8-bit immediate short operand is interpreted as a **signed fraction** and is stored in the specified destination register. That is, the 8-bit data is stored in the eight MS bits of the destination operand, and the remaining bits of the destination operand D are zeroed.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D. That is, **duplicate destinations are NOT allowed within the same instruction.**

Note: Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with this instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.

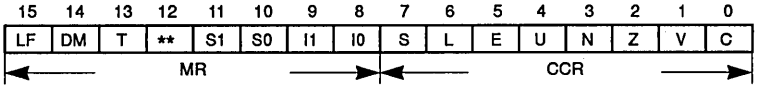
Example:

```
        :
ABS B #$18,R1          ;take absolute value of B, #$18→R1
        :
```

Explanation of Example: Prior to execution, the 16-bit address register R1 contains the value $0000. The execution of the parallel move portion of the instruction, #$18,R1, moves the 8-bit immediate short operand into the eight LS bits of the R1 register and zeros the remaining eight MS bits of that register. The 8-bit value is interpreted as an unsigned integer since its destination is the R1 address register.

Immediate Short Data Move

Condition Codes:

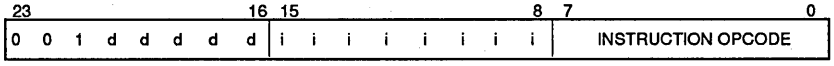


The condition codes are not affected by this type of parallel move.

Instruction Format:

(. . . . .) #xx,D

Opcode:



Instruction Fields:

#xx=8-bit Immediate Short Data=iiiiiiii

A

Immediate Short Data Move

D	d	d	d	d	d	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no
X1	0	0	1	0	1	no	no
Y0	0	0	1	1	0	no	no
Y1	0	0	1	1	1	no	no
A0	0	1	0	0	0	no	no
B0	0	1	0	0	1	no	no
A2	0	1	0	1	0	no	no
B2	0	1	0	1	1	no	no
A1	0	1	1	0	0	no	no
B1	0	1	1	0	1	no	no
A	0	1	1	1	0	A2	A0
B	0	1	1	1	1	B2	B0
R0-R7	1	0	r	r	r		
N0-N7	1	1	n	n	n		

where “rrr”=Rn number
where “nnn”=Nn number

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:
(. . . . .); S→D

Assembler Syntax:
(. . . . .) S,D

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move the source register S to the destination register D.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D. That is, **duplicate destinations are NOT allowed within the same instruction**.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S in the parallel data bus move operation. This allows data to be moved in the same instruction in which it is being used as a source operand by a data ALU operation. That is, **duplicate sources are allowed within the same instruction**.

When a 24-bit source operand is moved into a 16-bit destination register, the 16 LS bits of the 24-bit source operand are stored in the 16-bit destination register. When a 16-bit source operand is moved into a 24-bit destination register, the 16 LS bits of the destination register are loaded with the contents of the 16-bit source operand, and the eight MS bits of the 24-bit destination register are zeroed.

Note: The MOVE A,B operation will result in a 24-bit positive or negative saturation constant being stored in the B1 portion of the B accumulator if the signed integer portion of the A accumulator is in use.

A

Note: Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with this instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.

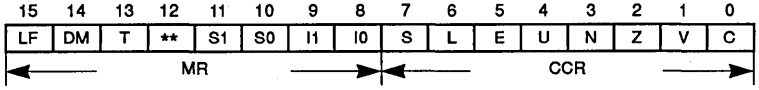
Example:

```
        :
MACR-X0,Y0,A   Y1,N5        ;-X0*Y0+A→A, move Y1→N5
        :
```

	Before Execution		After Execution
Y1	$001234	Y1	$001234
N5	$0000	N5	$1234

Explanation of Example: Prior to execution, the 24-bit Y1 register contains the value $001234 and the 16-bit address offset register N5 contains the value $0000. The execution of the parallel move portion of the instruction, Y1,N5, moves the 16 LS bits of the 24-bit value in the Y1 register into the 16-bit N5 register.

Condition Codes:

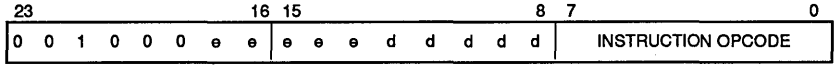


S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move

Instruction Format:

(. . . . .) S,D

Opcode:



A

Instruction Fields:

S or D	ed	ed	ed	ed	ed	S S/L	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where "rrr"=Rn number
where "nnn"=Nn number

Timing: mv oscillator clock cycles

Memory: mv program words

A

Operation:
(. . . . .); ea→Rn

Assembler Syntax:
(. . . . .) ea

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Update the specified address register according to the specified effective addressing mode. All update addressing modes may be used.

Example:

```
        :
RND B  (R3)+N3          ;round value in B into B1, R3+N3→R3
        :
```

	Before Execution	After Execution
R3	$0007	$000B
N3	$0004	$0004

Explanation of Example: Prior to execution, the 16-bit address register R3 contains the value $0007, and the 16-bit address offset register N3 contains the value $0004. The execution of the parallel move portion of the instruction, (R3)+N3, updates the R3 address register according to the specified effective addressing mode by adding the value in the R3 register to the value in the N3 register and storing the 16-bit result back in the R3 address register.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
←			MR			→		←			CCR			→	

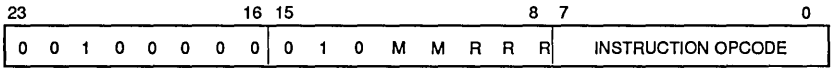
The condition codes are not affected by this type of parallel move.

A

Instruction Format:

(. . . . .) ea

Opcode:



Instruction Fields:

ea=5-bit Effective Address=MMRRR

Effective Addressing Mode	M M R R R
(Rn)-Nn	0 0 r r r
(Rn)+Nn	0 1 r r r
(Rn)-	1 0 r r r
(Rn)+	1 1 r r r

where "rrr" refers to an address register R0-R7

Timing: mv oscillator clock cycles

Memory: mv program words

A

Operation:	Assembler Syntax:
(. . . . .); X:ea→D	(. . . . .) X:ea,D
(. . . . .); X:aa→D	(. . . . .) X:aa,D
(. . . . .); S→X:ea	(. . . . .) S,X:ea
(. . . . .); S→X:aa	(. . . . .) S,X:aa
(. . . . .); #xxxxxx→D	(. . . . .) #xxxxxx,D

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move the specified word operand from/to X memory. All memory addressing modes, including absolute addressing and 24-bit immediate data, may be used. Absolute short addressing may also be used.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D. That is, **duplicate destinations are NOT allowed within the same instruction**.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S in the parallel data bus move operation. This allows data to be moved in the same instruction in which it is being used as a source operand by a data ALU operation. That is, **duplicate sources are allowed within the same instruction**.

A

When a 24-bit source operand is moved into a 16-bit destination register, the 16 LS bits of the 24-bit source operand are stored in the 16-bit destination register. When a 16-bit source operand is moved into a 24-bit destination register, the 16 LS bits of the destination register are loaded with the contents of the 16-bit source operand, and the eight MS bits of the 24-bit destination register are zeroed.

Note:Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with this instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.

Example:

```
        :
ASL A   R2,X:-(R2)          ;A*2→A, save updated R2 in X:(R2)
        :
```

	Before Execution		After Execution
R2	$1001	R2	$1000
X:$1000	$000000	X:$1000	$001000

Explanation of Example: Prior to execution, the 16-bit R2 address register contains the value $1001, and the 24-bit X memory location X:$1000 contains the value $000000. The execution of the parallel move portion of the instruction, R2,X:-(R2), predecrements the R2 address register and then uses the R2 address register to move the updated contents of the R2 address register into the 24-bit X memory location X:$1000.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

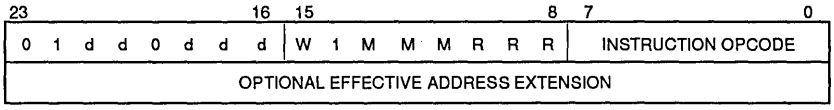
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION.
L — Set if data limiting has occurred during parallel move.

Note: The MOVE A,X:ea operation will result in a 24-bit positive or negative saturation constant being stored in the specified 24-bit X memory location if the signed integer portion of the A accumulator is in use.

Instruction Format:

(. . . . .) X:ea,D
(. . . . .) S,X:ea
(. . . . .) #xxxxxx,D

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate data	1	1	0	1	0	0

A

X Memory Data Move

S,D	d	d	d	d	d	S S/L	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where "rrr"=Rn number
where "nnn"=Nn number

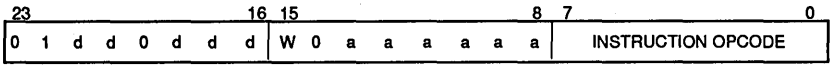
Timing: mv oscillator clock cycles

Memory: mv program words

Instruction Format:

(. . . . .) X:aa,D
(. . . . .) S,X:aa

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Register W		**Absolute Short Address a a a a a a**
Read S	0	0 0 0 0 0 0
Write D	1	•
		1 1 1 1 1 1

A

S,D	d	d	d	d	d	S S/L	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where "rrr"=Rn number
where "nnn"=Nn number

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:	Assembler Syntax:
Class I	**Class I**
(. . . . .); X:ea→D1; S2→D2	(. . . . .) X:ea,D1 S2,D2
(. . . . .); S1→X:ea; S2→D2	(. . . . .) S1,X:ea S2,D2
(. . . . .); #xxxxxx→D1; S2→D2	(. . . . .) #xxxxxx,D1 S2,D2
Class II	**Class II**
(. . . . .); A→X:ea; X0→A	(. . . . .) A,X:ea X0,A
(. . . . .); B→X:ea; X0→B	(. . . . .) B,X:ea X0,B

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Class I: Move a one-word operand from/to X memory and move another word operand from an accumulator (S2) to an input register (D2). All memory addressing modes, including absolute addressing and 24-bit immediate data, may be used. The register to register move (S2,D2) allows a data ALU accumulator to be moved to a data ALU input register for use as a data ALU operand in the following instruction.

Class II: Move one-word operand from a data ALU accumulator to X memory and one-word operand from data ALU register X0 to a data ALU accumulator. One effective address is specified. All memory addressing modes, excluding long absolute addressing and long immediate data, may be used.

For both Class I and Class II X:R parallel data moves, if the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D1 in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D1. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D1. That is, **duplicate destinations are NOT allowed within the same instruction.**

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S1 and/or S2 in the parallel data bus move operation. This allows data to be moved in the same instruction in which it is being used as a source operand by a data ALU operation. That is, **duplicate sources are allowed within the same instruction**. Note that S1 and S2 may specify the same register.

Class I Example:

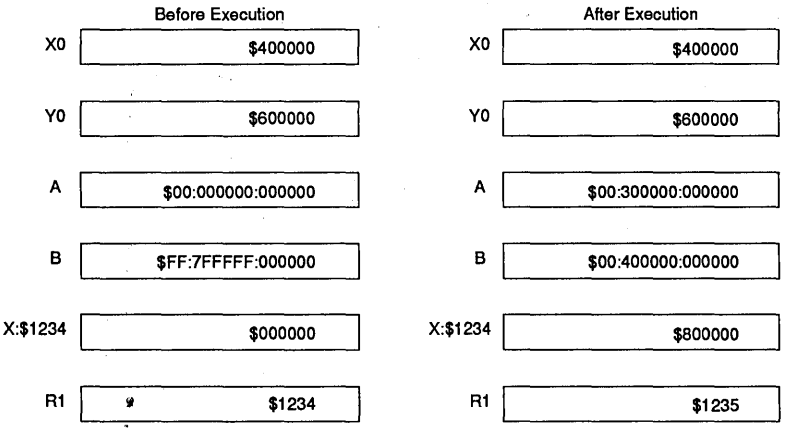
```
        :
CMPM Y0,A A,X:$1234 A,Y0      ;compare A,Y0 mag., save A, update Y0
        :
```

	Before Execution		After Execution
A	$00:800000:000000	A	$00:800000:000000
X:$1234	$000000	X:$1234	$7FFFFF
Y0	$000000	Y0	$7FFFFF

Explanation of the Class I Example: Prior to execution, the 56-bit A accumulator contains the value $00:800000:000000, the 24-bit X memory location X:$1234 contains the value $000000, and the 24-bit Y0 register contains the value $000000. The execution of the parallel move portion of the instruction, A,X:$1234 A,Y0, moves the 24-bit limited positive saturation constant $7FFFFF into both the X:$1234 memory location and the Y0 register since the signed portion of the A accumulator was in use.

Class II Example:

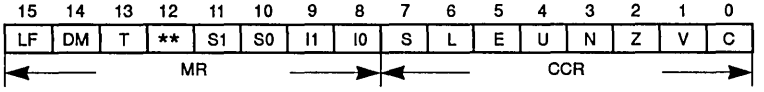
```
        :
MAC X0,Y0,A   B,X:(R1)+   X0,B   ;multiply X0 and Y0 and accumulate in A
        :                        ;move B to X memory location pointed to
                                 ;by R1 and postincrement R1
                                 ;move X0 to B
```



Explanation of the Class II Example: Prior to execution, the 24-bit registers X0 and Y0 contain $400000 and $600000, respectively. The 56-bit accumulators A and B contain the values $00:000000:000000 and $FF:7FFFFF:000000, respectively. The 24-bit X memory location X:$1234 contains the value $000000, and the 16-bit R1 register contains the value $1234. Execution of the parallel move portion of the instruction (B,X:(R1)+X0,B) moves the 24-bit limited value of B ($800000) into the X:$1234 memory location and the X0 register ($400000) into accumulator B1 ($400000), sign extends B1 into B2 ($00), and zero fills B0 ($000000). It also increments R1 to $1235.

A

Condition Codes:

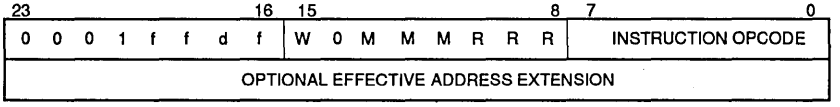


S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move.

Class I Instruction Format:

(. . . . .) X:ea,D1 S2,D2
(. . . . .) S1,X:ea S2, D2
(. . . . .) #xxxxxx, S2,D2

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate data	1	1	0	1	0	0

where "rrr" refers to an address register R0–R7

A

S1,D1	f	f	S1 S/L	D1 Sign Ext	D1 Zero	S2	d	S2 S/L	D2	f	D2 Sign Ext	D2 Zero
X0	0	0	no	no	no	A	0	yes	Y0	0	no	no
X1	0	1	no	no	no	B	1	yes	Y1	1	no	no
A	1	0	yes	A2	A0							
B	1	1	yes	B2	B0							

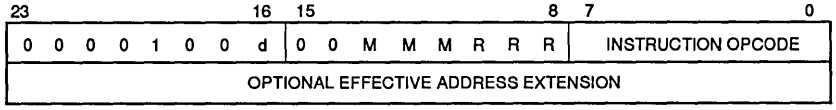
Timing: mv oscillator clock cycles

Memory: mv program words

Class II Instruction Format:

(. . . . .) A→X:ea X0→A
(. . . . .) B→X:ea X0→B

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r

where "rrr" refers to an address register R0–R7

S D	S S/L	D Sign Ext	D Zero	d	MOVE Opcode	
X0	no	N/A	N/A	0	A→X:ea	X0→A
Y0	no	N/A	N/A	1	B→X:ea	X0→B
A	yes	A2	A0			
B	yes	B2	B0			

Timing: mv oscillator clock cycles

Memory: mv program words

A

Operation:	Assembler Syntax:
(. . . . .); Y:ea→D	(. . . . .) Y:ea,D
(. . . . .); Y:aa→D	(. . . . .) Y:aa,D
(. . . . .); S→Y:ea	(. . . . .) S,Y:ea
(. . . . .); S→Y:aa	(. . . . .) S,Y:aa
(. . . . .); #xxxxxx→D	(. . . . .) #xxxxxx,D

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move the specified word operand from/to Y memory. All memory addressing modes, including absolute addressing and 24-bit immediate data, may be used. Absolute short addressing may also be used.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D. That is, **duplicate destinations are NOT allowed within the same instruction**.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S in the parallel data bus move operation. This allows data to be moved in the same instruction in which it is being used as a source operand by a data ALU operation. That is, **duplicate sources are allowed within the same instruction**.

A

When a 24-bit source operand is moved into a 16-bit destination register, the 16 LS bits of the 12-bit source operand are stored in the 16-bit destination register. When a 16-bit source operand is moved into a 24-bit destination register, the 16 LS bits of the destination register are loaded with the contents of the 16-bit source operand, and the eight MS bits of the 24-bit destination register are zeroed.

Note: This parallel data move is considered to be a move-type instruction. Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with this instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.

Example:

```
        :
EOR X0,B   #$123456,A      ;exclusive OR X0 and B, update A accumulator
        :
```

	Before Execution		After Execution
A	$FF:FFFFFF:FFFFFF	A	$00:123456:000000

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $FF:FFFFFF:FFFFFF. The execution of the parallel move portion of the instruction, #$123456,A, moves the 24-bit immediate value $123456 into the 24-bit A1 register, then sign extends that value into the A2 portion of the accumulator, and zeros the lower 24-bit A0 portion of the accumulator.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

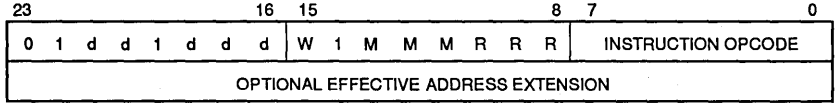
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move.

Note: The MOVE A,Y:ea operation will result in a 24-bit positive or negative saturation constant being stored in the specified 24-bit Y memory location if the signed integer portion of the A accumulator is in use.

Instruction Format:

(. . . . .) Y:ea,D
(. . . . .) S,Y:ea
(. . . . .) #xxxxxx,D

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate data	1	1	0	1	0	0

where "rrr" refers to an address register R0–R7

S,D	d	d	d	d	d	S S/L	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where "rrr"=Rn number
where "nnn"=Nn number

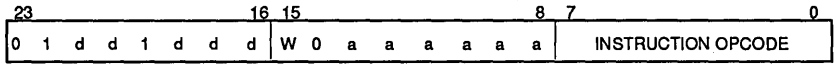
Timing: mv oscillator clock cycles

Memory: mv program words

Instruction Format:

(. . . . .) Y:aa,D
(. . . . .) S,Y:aa

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Register W		Absolute Short Address aaaaaa
Read S	0	000000
Write D	1	•
		111111

A

S,D	d	d	d	d	d	S S/L	D Sign Ext	D Zero
X0	0	0	1	0	0	no	no	no
X1	0	0	1	0	1	no	no	no
Y0	0	0	1	1	0	no	no	no
Y1	0	0	1	1	1	no	no	no
A0	0	1	0	0	0	no	no	no
B0	0	1	0	0	1	no	no	no
A2	0	1	0	1	0	no	no	no
B2	0	1	0	1	1	no	no	no
A1	0	1	1	0	0	no	no	no
B1	0	1	1	0	1	no	no	no
A	0	1	1	1	0	yes	A2	A0
B	0	1	1	1	1	yes	B2	B0
R0-R7	1	0	r	r	r			
N0-N7	1	1	n	n	n			

where "rrr"=Rn number
where "nnn"=Nn number

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:

Class I

(. . . . .); S1→D1; Y:ea→D2

(. . . . .); S1→D1; S2→Y:ea

(. . . . .); S1→D1; #xxxxxx→D2

Class II

(. . . . .); Y0 →A; A→Y:ea

(. . . . .); Y0→B; B→Y:ea

Assembler Syntax:

Class I

(. . . . .) S1,D1 Y:ea,D2

(. . . . .) S1,D1 S2,Y:ea

(. . . . .) S1,D1 #xxxxxx,D2

Class II

(. . . . .) Y0,A A,Y:ea

(. . . . .) Y0,B B,Y:ea

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Class I: Move a one-word operand from an accumulator (S1) to an input register (D1) and move another word operand from/to Y memory. All memory addressing modes, including absolute addressing and 24-bit immediate data, may be used. The register to register move (S1,D1) allows a data ALU accumulator to be moved to a data ALU input register for use as a data ALU operand in the following instruction.

Class II: Move one-word operand from a data ALU accumulator to Y memory and one-word operand from data ALU register Y0 to a data ALU accumulator. One effective address is specified. All memory addressing modes, excluding long absolute addressing and long immediate data, may be used. Class II move operations have been added to the R:Y parallel move (and a similar feature has been added to the X:R parallel move) as an added feature available in the first quarter of 1989.

For both Class I and Class II R:Y parallel data moves, if the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D2 in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A0, A1, A2, or A as its destination D2. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B0, B1, B2, or B as its destination D2. That is, duplicate destinations are NOT allowed within the same instruction.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S1 and/or S2 in the parallel data bus move operation. This allows data to be moved in the same instruction in which it is being used as a source operand by a data ALU operation. That is, **duplicate sources are allowed within the same instruction**. Note that S1 and S2 may specify the same register.

Class I Example:

```
        :
ADDL  B,A   B,X1   Y:(R6)-N6,B      ;2*A+B → A, update X1,B and R6
        :
```

	Before Execution		After Execution
B	$80:123456:789ABC	B	$00:654321:000000
X1	$000000	X1	$800000
R6	$2020	R6	$2000
N6	$0020	N6	$0020
Y:$2020	$654321	Y:$2020	$654321

Explanation of the Class I Example: Prior to execution, the 56-bit B accumulator contains the value $80:123456:789ABC, the 24-bit X1 register contains the value $000000, the 16-bit R6 address register contains the value $2020, the 16-bit N6 address offset register contains the value $0020 and the 24-bit Y memory location Y:$2020 contains the value $654321. The execution of the parallel move portion of the instruction, B,X1 Y:(R6)-N6,B, moves the 24-bit limited negative saturation constant $800000 into the X1 register since the signed integer portion of the B accumulator was in use, uses the value in the 16-bit R6 address register to move the 24-bit value in the Y memory location Y:$2020 into the 56-bit B accumulator with automatic sign extension of the upper portion of the accumulator (B2) and automatic zeroing of the lower portion of the accumulator (B0), and finally uses the contents of the 16-bit N6 address offset register to update the value in the 16-bit R6 address register. The contents of the N6 address offset register are not affected.

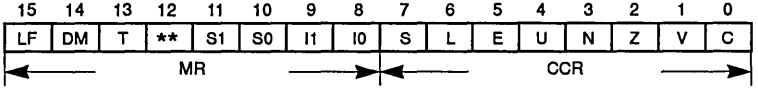
Class II Example:

```
        :
MAC X0,Y0,A   Y0,B    B,Y:(R1)+    ;multiply X0 and Y0 and accumulate in A
        :                          ;move B to Y memory location pointed to
                                   ;by R1 and postincrement R1
                                   ;move Y0 to B
```

	Before Execution	After Execution
X0	$400000	$400000
Y0	$600000	$600000
A	$00:000000:000000	$00:300000:000000
B	$00:800000:000000	$00:600000:000000
Y:$1234	$000000	$7FFFFF
R1	$1234	$1235

Explanation of the Class II Example: Prior to execution, the 24-bit registers, X0 and Y0, contain $400000 and $600000, respectively. The 56-bit accumulators A and B contain the values $00:000000:000000 and $00:800000:000000 (+1.0000), respectively. The 24-bit Y memory location Y:$1234 contains the value $000000, and the 16-bit R1 register contains the value $1234. Execution of the parallel move portion of the instruction (Y0,B B,Y:(R1)+) moves the Y0 register ($600000) into accumulator B1 ($600000), sign extends B1 into B2 ($00), and zero fills B0 ($000000). It also moves the 24-bit limited value of B ($7FFFFF) into the Y:$1234 memory location and increments R1 to $1235.

Condition Codes:

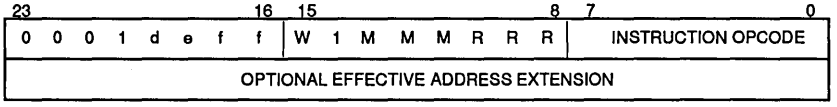


S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move.

Class I Instruction Format:

(. . . . .) S1,D1 Y:ea,D2
(. . . . .) S1,D1 S2,Y:ea
(. . . . .) S1,D1 #xxxxxx,D2

Opcode:



A

Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register W	Effective Addressing Mode	M	M	M	R	R	R
Read S2 0	(Rn)-Nn	0	0	0	r	r	r
Write D2 1	(Rn)+Nn	0	0	1	r	r	r
	(Rn)-	0	1	0	r	r	r
	(Rn)+	0	1	1	r	r	r
	(Rn)	1	0	0	r	r	r
	(Rn+Nn)	1	0	1	r	r	r
	-(Rn)	1	1	1	r	r	r
	Absolute address	1	1	0	0	0	0
	Immediate data	1	1	0	1	0	0

where "rrr" refers to an address register R0–R7

S1	d	S1 S/L	D1	e	D1 Sign Ext	D1 Zero
A	0	yes	X0	0	no	no
B	1	yes	X1	1	no	no

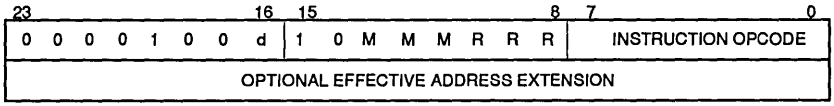
S2,D2	f	f	S2 S/L	D2 Sign Ext	D2 Zero
Y0	0	0	no	no	no
Y1	0	1	no	no	no
A	1	0	yes	A2	A0
B	1	1	yes	B2	B0

Timing: mv oscillator clock cycles

Memory: mv program words

Class II Instruction Format:
(. . . . .) Y0 → A A → Y:ea
(. . . . .) Y0 → B B → Y:ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Effective Addressing Mode	M	M	M	R	R	R
(Rn)-Nn	0	0	0	r	r	r
(Rn)+Nn	0	0	1	r	r	r
(Rn)-	0	1	0	r	r	r
(Rn)+	0	1	1	r	r	r
(Rn)	1	0	0	r	r	r
(Rn+Nn)	1	0	1	r	r	r
-(Rn)	1	1	1	r	r	r

where "rrr" refers to an address register R0-R7

S, D	SRC S/L	DEST Sign Ext	DEST Zero
X0	no	N/A	N/A
Y0	no	N/A	N/A
A	yes	A2	A0
B	yes	B2	B0

d	MOVE Opcode
0	Y0 → A A → Y:ea
1	Y0 → B B → Y:ea

Timing: mv oscillator clock cycles

Memory: mv program words

A

Operation:	Assembler Syntax:
(. . . . .); X:ea → D1; Y:ea → D2	(. . . . .) L:ea,D
(. . . . .); X:aa → D1; Y:aa → D2	(. . . . .) L:aa,D
(. . . . .); S1 → X:ea; S2 → Y:ea	(. . . . .) S,L:ea
(. . . . .); S1 → X:aa; S2 → Y:aa	(. . . . .) S,L:aa

where (. . . . .) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Move one 48-bit long-word operand from/to X and Y memory. Two data ALU registers are concatenated to form the 48-bit long-word operand. This allows efficient moving of both double-precision (high:low) and complex (real:imaginary) data from/to one effective address in L (X:Y) memory. The same effective address is used for both the X and Y memory spaces; thus, only one effective address is required. Note that the A, B, A10, and B10 operands reference a single 48-bit signed (double-precision) quantity while the X, Y, AB, and BA operands reference two separate (i.e., real and imaginary) 24-bit signed quantities. All memory alterable addressing modes may be used. Absolute short addressing may also be used.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A, A10, AB, or BA as destination D. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B, B10, AB, or BA as its destination D. That is, duplicate destinations are NOT allowed within the same instruction.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S in the parallel data bus move operation. This allows data to be moved in the same instruction in which it is being used as a source operand by a data ALU operation. That is, duplicate sources are allowed within the same instruction.

Note: The operands A10, B10, X, Y, AB, and BA may be used only for a 48-bit long memory move as previously described. These operands may not be used in any other type of instruction or parallel move.

Example:

```
        :
CMP Y0,B    A,L:$1234      ;compare Y0 and B, save 48-bit A1:A0 value
        :
```

	Before Execution	After Execution
A	$01:234567:89ABCD	$01:234567:89ABCD
X:$1234	$000000	$7FFFFF
Y:$1234	$:000000	$FFFFFF

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $01:234567:89ABCD, the 24-bit X memory location X:$1234 contains the value $000000, and the 24-bit Y memory location Y:$1234 contains the value $000000. The execution of the parallel move portion of the instruction, A,L:$1234, moves the 48-bit limited positive saturation constant $7FFFFF:FFFFFF into the specified long memory location by moving the MS 24 bits of the 48-bit limited positive saturation constant ($7FFFFF) into the 24-bit X memory location X:$1234 and by moving the LS 24 bits of the 48-bit limited positive saturation constant ($FFFFFF) into the 24-bit Y memory location Y:$1234 since the signed integer portion of the A accumulator was in use.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

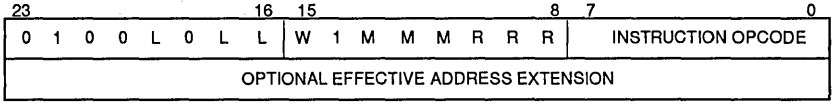
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move.

Note: The MOVE A,L:ea operation will result in a 48-bit positive or negative saturation constant being stored in the specified 24-bit X and Y memory locations if the signed integer portion of the A accumulator is in use. The MOVE AB,L:ea operation will result in either one or two 24-bit positive and/or negative saturation constant(s) being stored in the specified 24-bit X and/or Y memory location(s) if the signed integer portion of the A and/or B accumulator(s) is in use.

Instruction Format:

(. . . . .) L:ea,D
(. . . . .) S,L:ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	0	r	r	r
		Absolute address	1	1	0	0	0	0

where "rrr" refers to an address register R0-R7

S	S1	S2	S S/L	D	D1	D2	D Sign Ext	D Zero	L	L	L
A10	A1	A0	no	A10	A1	A0	no	no	0	0	0
B10	B1	B0	no	B10	B1	B0	no	no	0	0	1
X	X1	X0	no	X	X1	X0	no	no	0	1	0
Y	Y1	Y0	no	Y	Y1	Y0	no	no	0	1	1
A	A1	A0	yes	A	A1	A0	A2	no	1	0	0
B	B1	B0	yes	B	B1	B0	B2	no	1	0	1
AB	A	B	yes	AB	A	B	A2,B2	A0,B0	1	1	0
BA	B	A	yes	BA	B	A	B2,A2	B0,A0	1	1	1

Timing: mv oscillator clock cycles

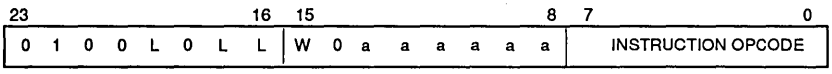
Memory: mv program words

L: Long Memory Data Move L:

Instruction Format:

(. . . . .) L:aa,D
(. . . . .) S,L:aa

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Register W		Absolute Short Address aaaaaa
Read S	0	000000
Write D	1	•
		•
		111111

S	S1	S2	S S/L	D	D1	D2	D Sign Ext	D Zero	L	L	L
A10	A1	A0	no	A10	A1	A0	no	no	0	0	0
B10	B1	B0	no	B10	B1	B0	no	no	0	0	1
X	X1	X0	no	X	X1	X0	no	no	0	1	0
Y	Y1	Y0	no	Y	Y1	Y0	no	no	0	1	1
A	A1	A0	yes	A	A1	A0	A2	no	1	0	0
B	B1	B0	yes	B	B1	B0	B2	no	1	0	1
AB	A	B	yes	AB	A	B	A2,B2	A0,B0	1	1	0
BA	B	A	yes	BA	B	A	B2,A2	B0,A0	1	1	1

Timing: mv oscillator clock cycles

Memory: mv program words

A

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 which allows parallel moves.

Description: Move a one-word operand from/to X memory and move another word operand from/to Y memory. Note that two independent effective addresses are specified (<eax> and <eay>) where one of the effective addresses uses the lower bank of address registers (R0–R3) while the other effective address uses the upper bank of address registers (R4–R7). All parallel addressing modes may be used.

If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D1 or D2 in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction specifies the 56-bit A accumulator as its destination, the parallel data bus move portion of the instruction may not specify A as its destination D1 or D2. Similarly, if the opcode-operand portion of the instruction specifies the 56-bit B accumulator as its destination, the parallel data bus move portion of the instruction may not specify B as its destination D1 or D2. That is, **duplicate destinations are NOT allowed within the same instruction**. D1 and D2 may not specify the same register.

If the opcode-operand portion of the instruction specifies a given source or destination register, that same register or portion of that register may be used as a source S1 and/or S2 in the parallel data bus move operation. This allows data to be moved in the same instruction in which it is being used as a source operand by a data ALU operation. That is, **duplicate sources are allowed within the same instruction**. Note that S1 and S2 may specify the same register.

A

Example:

```
        :
MPYR X1,Y0,A   X1,X:(R0)+    Y0,Y:(R4)+N4    ;X1*Y0 → A,save X1 and Y0
        :
```

	Before Execution		After Execution
X1	$123123	X1	$123123
Y0	$456456	Y0	$456456
R0	$1000	R0	$1001
R4	$0100	R4	$0123
N4	$0023	N4	$0023
X:$1000	$000000	X:$1000	$123123
Y:$0100	$000000	Y:$0100	$456456

Explanation of Example: Prior to execution, the 24-bit X1 register contains the value $123123, the 24-bit Y0 register contains the value $456456, the 16-bit R0 address register contains the value $1000, the 16-bit R4 address register contains the value $0100, the 16-bit N4 address offset register contains the value $0023, the 24-bit X memory location X:$1000 contains the value $000000, and the 24-bit Y memory location Y:$0100 contains the value $000000. The execution of the parallel move portion of the instruction, X1,X:(R0)+ Y0,Y:(R4)+N4, moves the 24-bit value in the X1 register into the 24-bit X memory location X:$1000 using the 16-bit R0 address register, moves the 24-bit value in the Y0 register into the 24-bit Y memory location Y:$0100 using the 16-bit R4 address register, updates the 16-bit value in the R0 address register, and updates the 16-bit R4 address register using the 16-bit N4 address offset register. The contents of the N4 address offset register are not affected.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

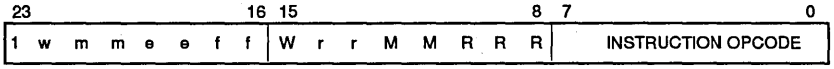
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move.

Note: The MOVE A,X:<eax> B,Y:<eay> operation will result in one or two 24-bit positive and/or negative saturation constant(s) being stored in the specified 24-bit X and/or Y memory location(s) if the signed integer portion of the A and/or B accumulator(s) is in use.

Instruction Format:

(. . . . .) X:<eax>,D1 Y:<eay>,D2
(. . . . .) X:<eax>,D1 S2,Y:<eay>
(. . . . .) S1,X:<eax> Y:<eay>,D2
(. . . . .) S1,X:<eax> S2,Y:<eay>

Opcode:



Instruction Fields:

X:<eax>=6-bit X Effective Address=WMMRRR (R0–R3 or R4–R7)
Y:<eay>=5-bit Y Effective Address=wmmrr (R4–R7 or R0–R3)

X Effective Addressing Mode	M	M	R	R	R
(Rn)+Nn	0	1	s	s	s
(Rn)-	1	0	s	s	s
(Rn)+	1	1	s	s	s
(Rn)	0	0	s	s	s

where "sss" refers to an address register R0–R7

A

Register	w	S1, D1	e	e	S1 S/L	D1 Sign Ext	D1 Zero	Y Effective Addressing Mode	m	m	r	r
Read S1	0	X0	0	0	no	no	no	(Rn) +Nn	0	1	t	t
Write D1	1	X1	0	1	no	no	no	(Rn) -	1	0	t	t
		A	1	0	yes	A2	A0	(Rn) +	1	1	t	t
		B	1	1	yes	B2	B0	(Rn)	0	0	t	t

where "tt" refers to an address register R4 - R7 or R0 - R3 which is in the **opposite** address register bank from the one used in the X effective address, previously described

Register	W	S2, D2	f	f	S2 S/L	D2 Sign Ext	D2 Zero
Read S2	0	Y0	0	0	no	no	no
Write D2	1	Y1	0	1	no	no	no
		A	1	0	yes	A2	A0
		B	1	1	yes	B2	B0

Timing: mv oscillator clock cycles

Memory: mv program words

Operation:	Assembler Syntax:	
X:ea→D1	MOVE(C)	X:ea,D1
X:aa→D1	MOVE(C)	X:aa,D1
S1→X:ea	MOVE(C)	S1,X:ea
S1→X:aa	MOVE(C)	S1,X:aa
Y:ea→D1	MOVE(C)	Y:ea,D1
Y:aa→D1	MOVE(C)	Y:aa,D1
S1→Y:ea	MOVE(C)	S1,Y:ea
S1→Y:aa	MOVE(C)	S1,Y:aa
S1→D2	MOVE(C)	S1,D2
S2→D1	MOVE(C)	S2,D1
#xxxx→D1	MOVE(C)	#xxxx,D1
#xx→D1	MOVE(C)	#xx,D1

Description: Move the contents of the specified source **control register** S1 or S2 to the specified destination 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 may be moved to or from any other register or memory space. Al memory addressing modes, as well as an immediate short addressing mode, may be used.

If the system stack register SSH is specified as a source operand, the system stack pointer (SP) is postdecremented by 1 after SSH has been read. If the system stack register SSH is specified as a destination operand, the system stack pointer (SP) is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

When a 56-bit accumulator (A or B) is specified as a **source** operand, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system

status register (SR). If the data out of the shifter indicates that the accumulator extension register is in use, and the data is to be moved into a 24-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. If the data is to be moved into a 16-bit destination and the accumulator extension register is in use, the value is limited to a maximum positive or negative saturation constant whose LS 16 bits are then stored in the 16-bit destination register. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the condition code register is latched.

When a 56-bit accumulator (A or B) is specified as a **destination** operand, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 23) and appending the source operand with 24 LS zeros. Whenever a 16-bit source operand is to be moved into a 24-bit destination, the 16-bit value is stored in the LS 16 bits of the 24-bit destination, and the MS 8 bits of that destination are zeroed. Similarly, whenever a 16-bit source operand is to be moved into a 56-bit accumulator, the 16-bit value is moved into the LS 16 bits of the MSP portion of the accumulator (A1 or B1), the MS 8 bits of the MSP portion of that accumulator are zeroed, and the resulting 24-bit value is extended to 56 bits by sign extending the MS bit and appending the result with 24 LS zeros. Note that for 24-bit source operands both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1).

Note: Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with this instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.

Restrictions: The following restrictions represent very unusual operations which probably would never be used but are listed only for completeness.

A MOVEC instruction used **within a DO loop** which specifies SSH as the **source** operand or **LA, LC, SR, SP, SSH, or SSL** as the **destination** operand cannot begin at the address LA – 2, LA – 1, or LA within that DO loop.

A MOVEC instruction which specifies **SSH** as the **source** operand or **LA, LC, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** a DO instruction.

A MOVEC instruction which specifies SSH as the **source** operand or **LA, LC, SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an ENDDO instruction.

A MOVEC instruction which specifies SSH as the **source** operand or **SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTI instruction.

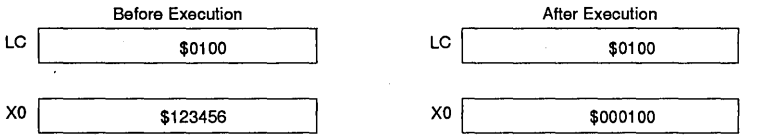
A MOVEC instruction which specifies SSH as the **source** operand or **SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTS instruction.

A MOVEC instruction which specified **SP** as the **destination** operand cannot be used **immediately before** a MOVEC, MOVEM, or MOVEP instruction which specifies **SSH or SSL** as the **source** operand.

A MOVEC SSH, SSH instruction is **illegal** and cannot be used.

Example:

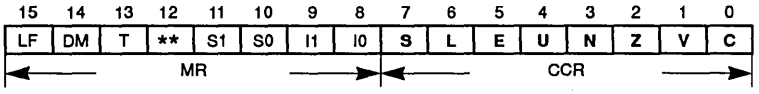
```
        :
MOVEC LC,X0          ;move LC into X0
        :
```



Explanation of Example: Prior to execution, the 16-bit loop counter (LC) register contains the value $0100, and the 24-bit X0 register contains the value $123456. The execution of the MOVEC LC,X0 instruction moves the contents of the 16-bit LC register into the 16 LS bits of the 24-bit X0 register and zeros the 8 MS bits of the X0 register.

A

Condition Codes:



For D1 or D2=SR operand:

S — **Set according to bit 7 of the source operand**
L — **Set according to bit 6 of the source operand**
E — **Set according to bit 5 of the source operand**
U — **Set according to bit 4 of the source operand**
N — **Set according to bit 3 of the source operand**
Z — **Set according to bit 2 of the source operand**
V — **Set according to bit 1 of the source operand**
C — **Set according to bit 0 of the source operand**

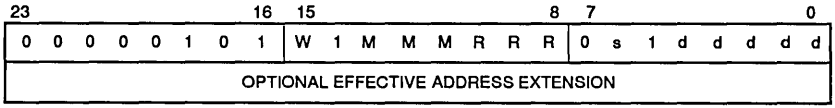
For D1 and D2≠SR operand:

S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during the move

Instruction Format:

MOVE(C) X:ea,D1
MOVE(C) S1,X:ea
MOVE(C) Y:ea,D1
MOVE(C) S1,Y:ea
MOVE(C) #xxxx,D1

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

A

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0
		Immediate Data	1	1	0	1	0	0

where "rrr" refers to an address register R0–R7

Memory Space	s	S1, D1	d	d	d	d	d
X Memory	0	M0–M7	0	0	n	n	n
Y Memory	1	SR	1	1	0	0	1
		OMR	1	1	0	1	0
		SP	1	1	0	1	1
		SSH	1	1	1	0	0
		SSL	1	1	1	0	1
		LA	1	1	1	1	0
		LC	1	1	1	1	1

where "nnn" = Mn number (M0–M7)

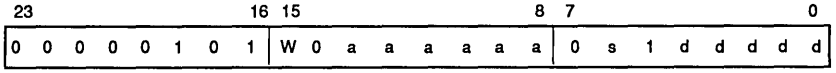
Timing: 2+mvc oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

MOVE(C) X:aa,D1
MOVE(C) S1,X:aa
MOVE(C) Y:aa,D1
MOVE(C) S1,Y:aa

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Register	W	Absolute Short Address aaaaaa
Read S	0	000000
Write D	1	•
		•
		111111

Memory Space	s	S1, D1	d d d d d
X Memory	0	M0–M7	0 0 n n n
Y Memory	1	SR	1 1 0 0 1
		OMR	1 1 0 1 0
		SP	1 1 0 1 1
		SSH	1 1 1 0 0
		SSL	1 1 1 0 1
		LA	1 1 1 1 0
		LC	1 1 1 1 1

where "nnn" = Mn number (M0–M7)

Timing: 2+mvc oscillator clock cycles

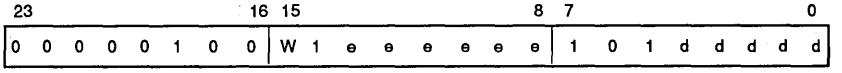
Memory: 1+ea program words

A

Instruction Format:

MOVE(C) S1,D2
MOVE(C) S2,D1

Opcode:



Instruction Fields:

Register	W	S1, D1	d d d d d
Read S1	0	M0–M7	0 0 n n n
Write D1	1	SR	1 1 0 0 1
		OMR	1 1 0 1 0
		SP	1 1 0 1 1
Memory Space	**s**	SSH	1 1 1 0 0
X Memory	0	SSL	1 1 1 0 1
Y Memory	1	LA	1 1 1 1 0
		LC	1 1 1 1 1

where "nnn" = Mn number (M0–M7)

S2, D2	e e e e e e	S2 S/L	D2 Sign Ext	D2 Zero
X0	0 0 0 1 0 0	no	no	no
X1	0 0 0 1 0 1	no	no	no
Y0	0 0 0 1 1 0	no	no	no
Y1	0 0 0 1 1 1	no	no	no
A0	0 0 1 0 0 0	no	no	no
B0	0 0 1 0 0 1	no	no	no
A2	0 0 1 0 1 0	no	no	no
B2	0 0 1 0 1 1	no	no	no
A1	0 0 1 1 0 0	no	no	no
B1	0 0 1 1 0 1	no	no	no
A	0 0 1 1 1 0	yes	A2	A0
B	0 0 1 1 1 1	yes	B2	B0

S2, D2	e e e e e e
R0 - R7	0 1 0 n n n
N0 - N7	0 1 1 n n n
M0 - M7	1 0 0 n n n
SR	1 1 1 0 0 1
OMR	1 1 1 0 1 0
SP	1 1 1 0 1 1
SSH	1 1 1 1 0 0
SSL	1 1 1 1 0 1
LA	1 1 1 1 1 0
LC	1 1 1 1 1 1

where "nnn" = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

A

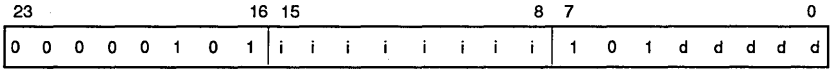
Timing: 2+mvc oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

MOVE(C) #xx,D1

Opcode:



Instruction Fields:

#xx=8-bit Immediate Short Data=iiiiiiii

D1	**d d d d d**
M0–M7	0 0 n n n
SR	1 1 0 0 1
OMR	1 1 0 1 0
SP	1 1 0 1 1
SSH	1 1 1 0 0
SSL	1 1 1 0 1
LA	1 1 1 1 0
LC	1 1 1 1 1

where "nnn" = Mn number (M0–M7)

Timing: 2+mvc oscillator clock cycles

Memory: 1+ea program words

A

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 may be **any** register. All memory alterable addressing modes may be used as well as the absolute short addressing mode.

If the system stack register SSH is specified as a source operand, the system stack pointer (SP) is postdecremented by 1 after SSH has been read. If the system stack register SSH is specified as a destination operand, the system stack pointer (SP) is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

When a 56-bit accumulator (A or B) is specified as a **source** operand S, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). If the data out of the shifter indicates that the accumulator extension register is in use and the data is to be moved into a 24-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. If a 24-bit source operand is to be moved into a 16-bit destination register D, the 8 MS bits of the 24-bit source operand are discarded, and the 16 LS bits are stored in the 16-bit destination register. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the condition code register is latched.

When a 56-bit accumulator (A) is specified as a **destination** operand D, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 24) and appending the source operand with 24 LS zeros. Whenever a 16-bit source operand S is to be moved into a 24-bit destination, the 16-bit source is loaded into the LS 16 bits of the destination operand, and the remaining 8 MS bits of the destination are zeroed. Note that for 24-bit source

operands, both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1).

Note: Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with this instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.

Restrictions: The following restrictions represent very unusual operations, which probably would never be used but are listed only for completeness.

A MOVEM instruction **used within a DO loop** which specifies **SSH** as the **source** operand or **LA, LC, SR, SP, SSH, or SSL** as the **destination** operand cannot begin at the address LA–2, LA–1, or LA within that DO loop.

A MOVEM instruction which specifies **SSH** as the **source** operand or **LA, LC, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** a DO instruction.

A MOVEM instruction which specifies **SSH** as the **source** operand or **LA, LC, SR, SSH, SL, or SP** as the **destination** operand cannot be used **immediately before** an ENDDO instruction.

A MOVEM instruction which specifies **SSH** as the **source** operand or **SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTI instruction.

A MOVEM instruction which specifies **SSH** as the **source** operand or **SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTS instruction.

A MOVEM instruction which specifies **SP** as the **destination** operand cannot be used **immediately before** a MOVEC, MOVEM, or MOVEP instruction which specifies **SSH or SSL** as the **source** operand.

Example:

```
        :
MOVEM P:(R5+N5), LC        :move P:(R5+N5) into the loop counter (LC)
        :
```

	Before Execution		After Execution
P:(R5 + N5)	$000116	P:(R5 + N5)	$000116
LC	$0000	LC	$0116

Explanation of Example: Prior to execution, the 16-bit loop counter (LC) register contains the value $0000, and the 24-bit program (P) memory location P:(R5+N5) contains the value $000116. The execution of the MOVEM P:(R5+N5), LC instruction moves the 16 LS bits of the 24-bit program (P) memory location P:(R5+N5) into the 16-bit LC register.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

For D=SR operand:

S — **Set according to bit 7 of the source operand**
L — **Set according to bit 6 of the source operand**
E — **Set according to bit 5 of the source operand**
U — **Set according to bit 4 of the source operand**
N — **Set according to bit 3 of the source operand**
Z — **Set according to bit 2 of the source operand**
V — **Set according to bit 1 of the source operand**
C — **Set according to bit 0 of the source operand**

For D≠SR operand:

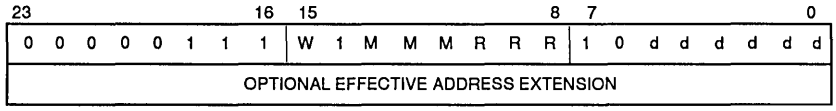
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during the move

A

Instruction Format:

MOVE(M) S,P:ea
MOVE(M) P:ea,D

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR

Register	W	Effective Addressing Mode	M	M	M	R	R	R
Read S	0	(Rn)-Nn	0	0	0	r	r	r
Write D	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
		(Rn)+	0	1	1	r	r	r
		(Rn)	1	0	0	r	r	r
		(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0

where "rrr" refers to an address register R0–R7

A

S,D	d d d d d d	S S/L	D Sign Ext	D Zero
X0	0 0 0 1 0 0	no	no	no
X1	0 0 0 1 0 1	no	no	no
Y0	0 0 0 1 1 0	no	no	no
Y1	0 0 0 1 1 1	no	no	no
A0	0 0 1 0 0 0	no	no	no
B0	0 0 1 0 0 1	no	no	no
A2	0 0 1 0 1 0	no	no	no
B2	0 0 1 0 1 1	no	no	no
A1	0 0 1 1 0 0	no	no	no
B1	0 0 1 1 0 1	no	no	no
A	0 0 1 1 1 0	yes	A2	A0
B	0 0 1 1 1 1	yes	B2	B0

S,D	d d d d d d
R0 - R7	0 1 0 n n n
N0 - N7	0 1 1 n n n
M0 - M7	1 0 0 n n n
SR	1 1 1 0 0 1
OMR	1 1 1 0 1 0
SP	1 1 1 0 1 1
SSH	1 1 1 1 0 0
SSL	1 1 1 1 0 1
LA	1 1 1 1 1 0
LC	1 1 1 1 1 1

where "nnn" = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

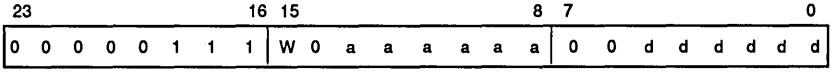
Timing: 2+mvm oscillator clock cycles

Memory: 1+ea program words

Instruction Format:

MOVE(M) S,P:aa
MOVE(M) P:aa,D

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaa

Register	W	Absolute Short Address aaaaaa
Read S	0	000000
Write D	1	•
		•
		111111

S,D	d d d d d d	S S/L	D Sign Ext	D Zero
X0	0 0 0 1 0 0	no	no	no
X1	0 0 0 1 0 1	no	no	no
Y0	0 0 0 1 1 0	no	no	no
Y1	0 0 0 1 1 1	no	no	no
A0	0 0 1 0 0 0	no	no	no
B0	0 0 1 0 0 1	no	no	no
A2	0 0 1 0 1 0	no	no	no
B2	0 0 1 0 1 1	no	no	no
A1	0 0 1 1 0 0	no	no	no
B1	0 0 1 1 0 1	no	no	no
A	0 0 1 1 1 0	yes	A2	A0
B	0 0 1 1 1 1	yes	B2	B0

S,D	d d d d d d
R0 - R7	0 1 0 n n n
N0 - N7	0 1 1 n n n
M0 - M7	1 0 0 n n n
SR	1 1 1 0 0 1
OMR	1 1 1 0 1 0
SP	1 1 1 0 1 1
SSH	1 1 1 1 0 0
SSL	1 1 1 1 0 1
LA	1 1 1 1 1 0
LC	1 1 1 1 1 1

where "nnn" = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

Timing: 2+mvm oscillator clock cycles

Memory: 1+ea program words

Operation:	Assembler Syntax:	
X:pp → D	MOVEP	X:pp,D
X:pp → X:ea	MOVEP	X:pp,X:ea
X:pp → Y:ea	MOVEP	X:pp,Y:ea
X:pp → P:ea	MOVEP	X:pp,P:ea
S → X:pp	MOVEP	S,X:pp
#xxxxxx → X:pp	MOVEP	#xxxxxx,X:pp
X:ea → X:pp	MOVEP	X:ea,X:pp
Y:ea → X:pp	MOVEP	Y:ea,X:pp
P:ea → X:pp	MOVEP	P:ea,X:pp
Y:pp → D	MOVEP	Y:pp,D
Y:pp → X:ea	MOVEP	Y:pp,X:ea
Y:pp → Y:ea	MOVEP	Y:pp,Y:ea
Y:pp → P:ea	MOVEP	Y:pp,P:ea
S → Y:pp	MOVEP	S,Y:pp
#xxxxxx → Y:pp	MOVEP	#xxxxxx,Y:pp
X:ea → Y:pp	MOVEP	X:ea,Y:pp
Y:ea → Y:pp	MOVEP	Y:ea,Y:pp
P:ea → Y:pp	MOVEP	P:ea,Y:pp

Description: Move the specified operand from/to the specified **X or Y I/O peripheral**. The I/O short addressing mode is used for the I/O peripheral address. All memory addressing modes may be used for the X or Y memory effective address; all memory alterable addressing modes may be used for the P memory effective address.

If the system stack register SSH is specified as a source operand, the system stack pointer (SP) is postdecremented by 1 after SSH has been read. If the system stack reg-

ister SSH is specified as a destination operand, the system stack pointer (SP) is preincremented by 1 before SSH is written. This allows the system stack to be efficiently extended using software stack pointer operations.

When a 56-bit accumulator (A or B) is specified as a **source** operand S, the accumulator value is optionally shifted according to the scaling mode bits S0 and S1 in the system status register (SR). If the data out of the shifter indicates that the accumulator extension register is in use and the data is to be moved into a 24-bit destination, the value stored in the destination is limited to a maximum positive or negative saturation constant to minimize truncation error. If a 24-bit source operand is to be moved into a 16-bit destination register D, the 8 MS bits of the 24-bit source operand are discarded, and the 16 LS bits are stored in the 16-bit destination register. Limiting does not occur if an individual 24-bit accumulator register (A1, A0, B1, or B0) is specified as a source operand instead of the full 56-bit accumulator (A or B). This limiting feature allows block floating-point operations to be performed with error detection since the L bit in the condition code register is latched.

When a 56-bit accumulator (A or B) is specified as a **destination** operand D, any 24-bit source data to be moved into that accumulator is automatically extended to 56 bits by sign extending the MS bit of the source operand (bit 23) and appending the source operand with 24 LS zeros. Whenever a 16-bit source operand S is to be moved into a 24-bit destination, the 16-bit source is loaded into the LS 16 bits of the destination operand, and the remaining 8 MS bits of the destination are zeroed. Note that for 24-bit source operands both the automatic sign-extension and zeroing features may be disabled by specifying the destination register to be one of the individual 24-bit accumulator registers (A1 or B1).

Note: Unlike other MOVE-type instructions, if an AGU register (Mn, Nn, or Rn) is directly changed with MOVEP, the new contents **will** be available for use during the immediately following instruction. There is no instruction cycle pipeline delay associated with MOVEP.

Restrictions: The following restrictions represent very unusual operations, which probably would never be used but are listed only for completeness.

A MOVEP instruction used **within a DO loop** which specifies **SSH** as the **source** operand or **LA, LC, SR, SP, SSH, or SSL** as the **destination** operand cannot begin at the address LA–2, LA–1, or LA within that DO loop.

A MOVEP instruction which specifies **SSH** as the **source** operand or **LA, LC, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** a DO instruction.

A MOVEP instruction which specifies **SSH** as the **source** operand or **LA, LC, SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an ENDDO instruction.

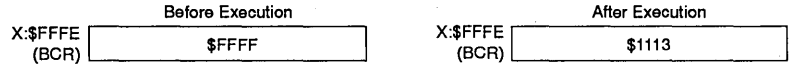
A MOVEP instruction which specifies **SSH** as the **source** operand or **SR, SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTI instruction.

A MOVEP instruction which specifies **SSH** as the **source** operand or **SSH, SSL, or SP** as the **destination** operand cannot be used **immediately before** an RTS instruction.

A MOVEP instruction which specifies **SP** as the **destination** operand cannot be used **immediately before** a MOVEC, MOVEM, or MOVEP instruction which specifies **SSH or SSL** as the **source** operand.

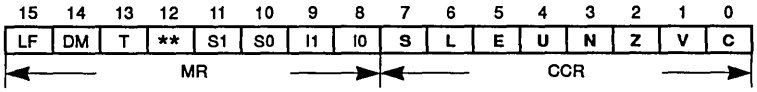
Example:

```
        :
MOVEP #1113,X:<<$FFFE        :initialize Bus Control Register wait states
        :
```



Explanation of Example: Prior to execution, the 16-bit, X memory-mapped, I/O bus control register (BCR) contains the value $FFFF. The execution of the MOVEP #$1113,X:<<$FFFE instruction moves the value $1113 into the 16-bit bus control register X:$FFFE, resulting in one wait state for all external X, external Y, and external program memory accesses and three wait states for all external I/O accesses.

Condition Codes:



For D=SR operand:

S — **Set according to bit 7 of the source operand**
L — **Set according to bit 6 of the source operand**
E — **Set according to bit 5 of the source operand**
U — **Set according to bit 4 of the source operand**
N — **Set according to bit 3 of the source operand**
Z — **Set according to bit 2 of the source operand**
V — **Set according to bit 1 of the source operand**
C — **Set according to bit 0 of the source operand**

For D≠SR operand:

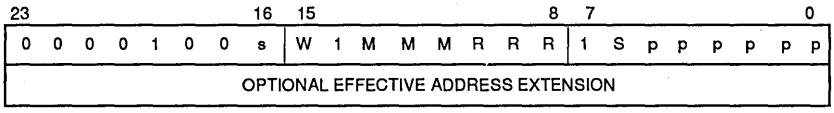
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during the move

A

Instruction Format (X: or Y: Reference):

MOVEP X:ea,X:pp
MOVEP Y:ea,X:pp
MOVEP #xxxxxx,X:pp
MOVEP X:pp,X:ea
MOVEP X:pp,Y:ea
MOVEP X:ea,Y:pp
MOVEP Y:ea,Y:pp
MOVEP #xxxxxx,Y:pp
MOVEP Y:pp,Y:ea
MOVEP Y:pp,Y:ea

Opcode:



A

Instruction Fields:

ea=6-bit Effective Address=MMMRRR,
pp=6-bit I/O Short Address=pppppp

Memory Space	S	Effective Addressing Mode	M	M	M	R	R	R
X Memory	0	(Rn)-Nn	0	0	0	r	r	r
Y Memory	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
Peripheral Space	**s**	(Rn)+	0	1	1	r	r	r
X Memory	0	(Rn)	1	0	0	r	r	r
Y Memory	1	(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
Peripheral	**W**	Absolute address	1	1	0	0	0	0
Read	0	Immediate data	1	1	0	1	0	0
Write	1							

where "rrr" refers to an address register R0–R7

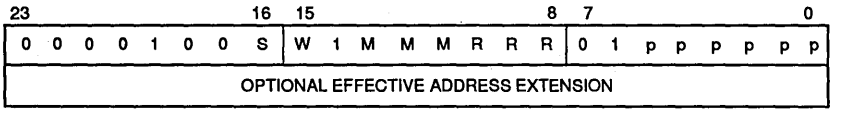
Timing: 4+mvp oscillator clock cycles

Memory: 1+ea program words

Instruction Format (P: Reference):

MOVEP P:ea,X:pp
MOVEP X:pp,P:ea
MOVEP P:ea,Y:pp
MOVEP Y:pp,P:ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR
pp=6-bit I/O Short Address=pppppp

Peripheral Space	S	Effective Addressing Mode	M	M	M	R	R	R
X Memory	0	(Rn)-Nn	0	0	0	r	r	r
Y Memory	1	(Rn)+Nn	0	0	1	r	r	r
		(Rn)-	0	1	0	r	r	r
Peripheral	**W**	(Rn)+	0	1	1	r	r	r
Read	0	(Rn)	1	0	0	r	r	r
Write	1	(Rn+Nn)	1	0	1	r	r	r
		-(Rn)	1	1	1	r	r	r
		Absolute address	1	1	0	0	0	0

where "rrr" refers to an address register R0-R7

Timing: 4+mvp oscillator clock cycles

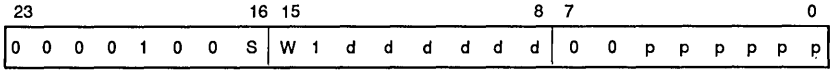
Memory: 1+ea program words

A

Instruction Format (Register Reference):

```
MOVEP   S,X:pp
MOVEP   X:pp,D
MOVEP   S,Y:pp
MOVEP   Y:pp,D
```

Opcode:



Instruction Fields:

pp=6-bit I/O Short Address=pppppp

Peripheral Space	S	Peripheral	W
X Memory	0	Read	0
Y Memory	1	Write	1

S,D	d d d d d d	S S/L	D Sign Ext	D Zero
X0	0 0 0 1 0 0	no	no	no
X1	0 0 0 1 0 1	no	no	no
Y0	0 0 0 1 1 0	no	no	no
Y1	0 0 0 1 1 1	no	no	no
A0	0 0 1 0 0 0	no	no	no
B0	0 0 1 0 0 1	no	no	no
A2	0 0 1 0 1 0	no	no	no
B2	0 0 1 0 1 1	no	no	no
A1	0 0 1 1 0 0	no	no	no
B1	0 0 1 1 0 1	no	no	no
A	0 0 1 1 1 0	yes	A2	A0
B	0 0 1 1 1 1	yes	B2	B0

S,D	d d d d d d
R0 - R7	0 1 0 n n n
N0 - N7	0 1 1 n n n
M0 - M7	1 0 0 n n n
SR	1 1 1 0 0 1
OMR	1 1 1 0 1 0
SP	1 1 1 0 1 1
SSH	1 1 1 1 0 0
SSL	1 1 1 1 0 1
LA	1 1 1 1 1 0
LC	1 1 1 1 1 1

where "nnn" = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

Timing: 4+mvp oscillator clock cycles

Memory: 1+ea program words

A

Operation:

±S1*S2 → D (parallel move)

±S1*S2 → D (parallel move)

±(S1*2^{-n})→D (**no** parallel move)

Assembler Syntax:

MPY (±)S1,S2,D (parallel move)

MPY (±)S2,S1,D (parallel move)

MPY (±)S,#n,D (**no** parallel move)

Description: Multiply the two signed 24-bit source operands S1 and S2 and store the resulting product in the specified 56-bit destination accumulator D. Or, multiply the signed 24-bit source operand S by the positive 24-bit immediate operand 2^{-n} and add/subtract to/from the specified 56-bit destination accumulator D. The "–" sign option is used to negate the specified product prior to accumulation. The default sign option is "+".

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 shown in Section 3.4 DOUBLE PRECISION MULTIPLY MODE:

MPY Y0, X0, A	MPY Y0, X0, B
MAC X1, Y0, A	MAC X1, Y0, B
MAC X0, Y1, A	MAC X0, Y1, B
MAC Y1, X1, A	MAC Y1, X1, B

All other Data ALU instructions are executed as NOP's when the processor is in the Double Precision Multiply Mode.

Example 1:

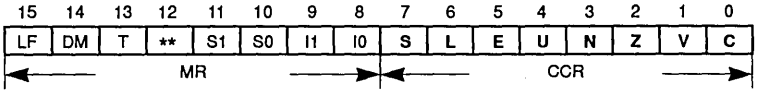
```
        :
MPY -X1,Y1,A #$543210,Y0        ;-(X1*Y1) → A, update Y0
        :
```

	Before Execution		After Execution
X1	$800000	X1	$800000
Y1	$C00000	Y1	$C00000
A	$00:000000:000000	A	$FF:C00000:000000

A

Explanation of Example 1: Prior to execution, the 24-bit X1 register contains the value $800000 (−1.0), the 24-bit Y1 register contains the value $C00000, (−0.5), and the 56-bit A accumulator contains the value $00:000000:000000 (0.0). The execution of the MPY −X1,Y1,A instruction multiples the 24-bit signed value in the X1 register by the 24-bit signed value in the Y1 register, negates the 48-bit product, and stores the result in the 56-bit A accumulator (−X1*Y1=−0.5=$FF:C00000:000000=A).

Condition Codes:



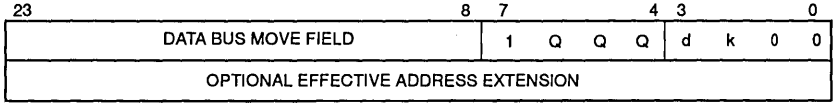
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting occurred during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format 1:

MPY (±)S1,S2,D
MPY (±)S2,S1,D

Opcode 1:



A

Instruction Fields 1:

S1*S2	Q	Q	Q	Sign	k	D	d
X0 X0	0	0	0	+	0	A	0
Y0 Y0	0	0	1	–	1	B	1
X1 X0	0	1	0				
Y1 Y0	0	1	1				
X0 Y1	1	0	0				
Y0 X0	1	0	1				
X1 Y0	1	1	0				
Y1 X1	1	1	1				

Note: Only the indicated S1*S2 combinations are valid. X1*X1 and Y1*Y1 are **not** valid.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Example 2:

```
        :
MPY   X1,   #9,  A          ;
        :
```

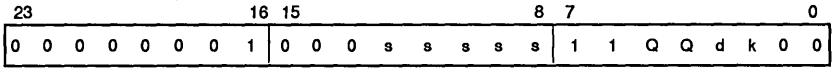
	Before Execution	After Execution
X1	$654321	$654321
A	$00:000000:000000	$00:0032A1:908000

Explanation of Example 2: The content of X1 is multiplied by 2^{-9} and the result is placed in the A accumulator. The net effect of this operation is to **divide** the content of X1 **by** 2^9 and place the result in the accumulator. An alternate interpretation is that X1 is **right shift**ed 9 places and filled with the sign bit (0 for a positive number and 1 for a negative number) and then the result is placed in the accumulator.

Instruction Format 2:

MPY (±)S,#n,D

Opcode 2:



Instruction Fields:

S	Q Q	Sign	k	D	d
Y1	0 0	+	0	A	0
X0	0 1	–	1	B	1
Y0	1 0				
X1	1 1				

n	sssss	constant
1	00001	010000000000000000000000
2	00010	001000000000000000000000
3	00011	000100000000000000000000
4	00100	000010000000000000000000
5	00101	000001000000000000000000
6	00110	000000100000000000000000
7	00111	000000010000000000000000
8	01000	000000001000000000000000
9	01001	000000000100000000000000
10	01010	000000000010000000000000
11	01011	000000000001000000000000
12	01100	000000000000100000000000
13	01101	000000000000010000000000
14	01110	000000000000001000000000
15	01111	000000000000000100000000
16	10000	000000000000000010000000
17	10001	000000000000000001000000
18	10010	000000000000000000100000
19	10011	000000000000000000010000
20	10100	000000000000000000001000
21	10101	000000000000000000000100
22	10110	000000000000000000000010
23	10111	000000000000000000000001

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

MPYR Signed Multiply and Round MPYR

Operation:

±S1*S2+r → D (parallel move)

±S1*S2+r → D (parallel move)

±(S1*2^{-n})+r → D (**no** parallel move)

Assembler Syntax:

MPYR (±)S1,S2,D (parallel move)

MPYR (±)S2,S1,D (parallel move)

MPYR (±)S,#n,D (**no** parallel move)

Description: Multiply the two signed 24-bit source operands S1 and S2 (**or** the signed 24-bit source operand S by the positive 24-bit immediate operand 2^{-n}), round the result using convergent rounding, and store it in the specified 56-bit destination accumulator D. The "–" sign option is used to negate the product prior to rounding. The default sign option is "+". The contribution of the LS bits of the result is rounded into the upper portion of the destination accumulator (A1 or B1) by adding a constant to the LS bits of the lower portion of the accumulator (A0 or B0). The value of the constant added is determined by the scaling mode bits S0 and S1 in the status register. Once the rounding has been completed, the LS bits of the destination accumulator D (A0 or B0) are loaded with zeros to maintain an unbiased accumulator value which may be reused by the next instruction. The upper portion of the accumulator (A1 or B1) contains the rounded result which may be read out to the data buses. Refer to the RND instruction for more complete information on the convergent rounding process.

Example 1:

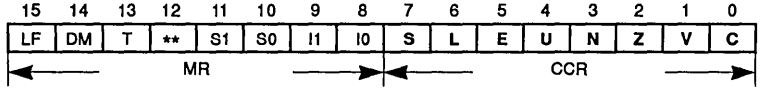
```
        :
MPYR -Y0,Y0,B (R3)-N3        ;square and negate Y0, update R3
        :
```

	Before Execution	After Execution
Y0	$654321	$654321
B	$00:000000:000000	$FF:AFE3ED:000000

Explanation of Example 1: Prior to execution, the 24-bit Y0 register contains the value $654321 (0.791111112), and the 56-bit B accumulator contains the value $00:000000:000000 (0.0). The execution of the MPYR -Y0,Y0,B instruction squares the 24-bit signed value in the Y0 register, negates the resulting 48-bit product, rounds the result into B1, and zeros B0 (-Y0*Y0=-0.625856790961748 approximately= $FF:AFE3EC:B76B7E, which is rounded to the value $FF:AFE3ED:000000= -0.625856757164002=B).

A

Condition Codes:



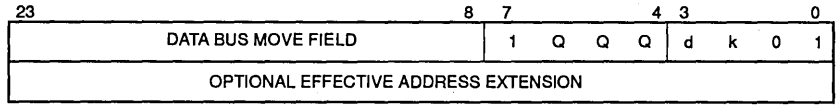
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting occurred during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z— Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format 1:

MPYR (±)S1,S2,D
MPYR (±)S2,S1,D

Opcode 1:



A

Instruction Fields 1:

S1*S2	Q	Q	Q	Sign	k	D	d
X0 X0	0	0	0	+	0	A	0
Y0 Y0	0	0	1	–	1	B	1
X1 X0	0	1	0				
Y1 Y0	0	1	1				
X0 Y1	1	0	0				
Y0 X0	1	0	1				
X1 Y0	1	1	0				
Y1 X1	1	1	1				

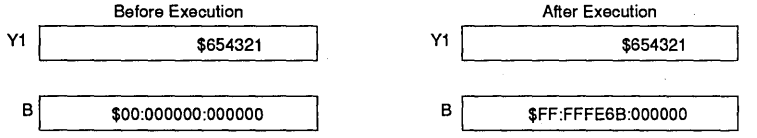
Note: Only the indicated S1*S2 combinations are valid. X1*X1 and Y1*Y1 are **not** valid.

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Example 2:

```
        :
MPYR  -Y1,   #14, B          ;
        :
```



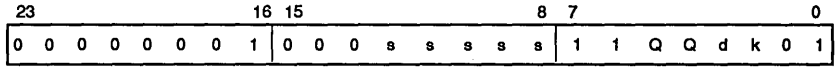
Explanation of Example 2: The content of Y1 is negated, multiplied by 2^{-14}, rounded to a single precision number (24 bits in B1) and placed in the B accumulator. The net effect of this operation is negate the content of Y1 and **divide** the result **by 2^{14}**, place the result in the accumulator and then round to a single precision number. An alternate interpretation is that X1 is negated and placed in the accumulator, **right shift**ed 14 places, filled with the sign bit (0 for a positive number and 1 for a negative number) and then rounded to a single precision number.

A

Instruction Format 2:

MPYR (±)S,#n,D

Opcode 2:



Instruction Fields 2:

S	Q	Q	Sign	k	D	d
Y1	0	0	+	0	A	0
X0	0	1	–	1	B	1
Y0	1	0				
X1	1	1				

n	sssss	constant
1	00001	010000000000000000000000
2	00010	001000000000000000000000
3	00011	000100000000000000000000
4	00100	000010000000000000000000
5	00101	000001000000000000000000
6	00110	000000100000000000000000
7	00111	000000010000000000000000
8	01000	000000001000000000000000
9	01001	000000000100000000000000
10	01010	000000000010000000000000
11	01011	000000000001000000000000
12	01100	000000000000100000000000
13	01101	000000000000010000000000
14	01110	000000000000001000000000
15	01111	000000000000000100000000
16	10000	000000000000000010000000
17	10001	000000000000000001000000
18	10010	000000000000000000100000
19	10011	000000000000000000010000
20	10100	000000000000000000001000
21	10101	000000000000000000000100
22	10110	000000000000000000000010
23	10111	000000000000000000000001

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

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 56-bit, twos-complement operation.

Example:

```
        :
NEG B X1,X:(R3)+ Y:(R6)−,A      ;0−B → B, update A,X1,R3,R6
        :
```

	Before Execution	After Execution
B	$00:123456:789ABC	$FF:EDCBA9:876544

Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $00:123456:789ABC. The NEG B instruction takes the twos complement of the value in the B accumulator and stores the 56-bit result back in the B accumulator.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	**S**	**L**	**E**	**U**	**N**	**Z**	**V**	C
← MR →								← CCR →							

S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

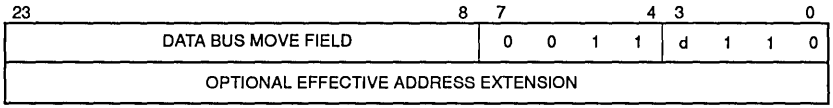
Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

A

Instruction Format:

NEG D

Opcode:



Instruction Fields:

D	d
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

NOP — No Operation — NOP

Operation:
PC+1→PC

Assembler Syntax:
NOP

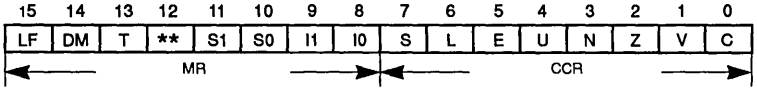
Description: Increment the program counter (PC). Pending pipeline actions, if any, are completed. Execution continues with the instruction following the NOP.

Example:

```
        :
NOP                     ;increment the program counter
        :
```

Explanation of Example: The NOP instruction increments the program counter and completes any pending pipeline actions.

Condition Codes:



The condition codes are not affected by this instruction.

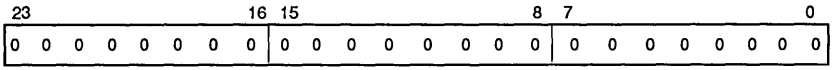
A

NOP — No Operation — NOP

Instruction Format:

NOP

Opcode:



Instruction Fields:

None

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

Operation:

If $\overline{E} \bullet U \bullet \overline{Z}=1$, then ASL D and Rn−1→Rn
else if E=1, then ASR D and Rn+1→Rn
else NOP

where $\overline{E}$ denotes the logical complement of E, and
where • denotes the logical AND operator

Assembler Syntax:

NORM Rn,D

Description: Perform one normalization iteration on the specified destination operand D, update the specified address register Rn based upon the results of that iteration, and store the result back in the destination accumulator. This is a 56-bit operation. If the accumulator extension is not in use, the accumulator is unnormalized, and the accumulator is not zero, the destination operand is arithmetically shifted one bit to the left, and the specified address register is decremented by 1. If the accumulator extension register is in use, the destination operand is arithmetically shifted one bit to the right, and the specified address register is incremented by 1. If the accumulator is normalized or zero, a NOP is executed and the specified address register is not affected. Since the operation of the NORM instruction depends on the E, U, and Z condition code register bits, these bits must correctly reflect the current state of the destination accumulator prior to executing the NORM instruction. Note that the L and V bits in the condition code register will be cleared unless they have been improperly set up prior to executing the NORM instruction.

Example:

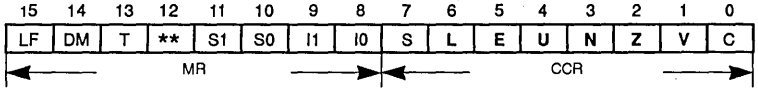
```
    :
REP #$2F            ;maximum number of iterations needed
NORM R3,A           ;perform 1 normalization iteration
    :
```

	Before Execution	After Execution
A	$00:000000:000001	$00:400000:000000
R3	$0000	$FFD2

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:000000:000001, and the 16-bit R3 address register contains the value $0000. The repetition of the NORM R3,A instruction normalizes the value in the 56-bit accumulator and stores the resulting number of shifts performed during that normalization pro-

cess in the R3 address register. A negative value reflects the number of left shifts performed; a positive value reflects the number of right shifts performed during the normalization process.

Condition Codes:



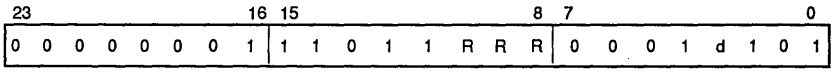
L — Set if overflow has occurred in A or B result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — **Set if bit 55 is changed as a result of a left shift**

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

NORM Rn,D

Opcode:



Instruction Fields:

D	d	Rn	R R R
A	0	Rn	n n n
B	1		

where "nnn" = Rn number

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

NOT — Logical Complement — NOT

Operation:
$\overline{D[47:24]} \rightarrow D[47:24]$ (parallel move)
where "—" denotes the logical NOT operator

Assembler Syntax:
NOT D (parallel move)

Description: Take the ones complement of bits 47–24 of the destination operand D and store the result back in bits 47–24 of the destination accumulator. This is a 24-bit operation. The remaining bits of D are not affected.

Example:

```
NOT A1  AB,L:(R2)+        ;save A1,B1, take the ones complement of A1
```

	Before Execution		After Execution
A	$00:123456:789ABC	A	$00:EDCBA9:789AB

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:123456:789ABC. The NOT A instruction takes the ones complement of bits 47–24 of the A accumulator (A1) and stores the result back in the A1 register. The remaining bits of the A accumulator are not affected.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	**S**	L	E	U	**N**	**Z**	V	C
← MR →								← CCR →							

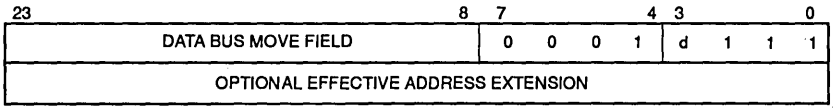
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47-24 of A or B result are zero**
V — **Always cleared**

A

Instruction Format:

NOT D

Opcode:



Instruction Fields:

D	d
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
S+D[47:24] → D[47:24] (parallel move)
where + denotes the logical inclusive OR operator

Assembler Syntax:
OR S,D (parallel move)

Description: Logically inclusive OR the source operand S with bits 47–24 of the destination operand D and store the result in bits 47–24 of the destination accumulator. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

Example:

```
        :
OR Y1,B1        BA,L:$1234          ;save A1,B1, OR Y1 with B
        :
```

	Before Execution	After Execution
Y1	$FF0000	$FF0000
B	$00:123456:789ABC	$00:FF3456:789ABC

Explanation of Example: Prior to execution, the 24-bit Y1 register contains the value $FF0000, and the 56-bit B accumulator contains the value $00:123456:789ABC. The OR Y1,B instruction logically ORs the 24-bit value in the Y1 register with bits 47–24 of the B accumulator (B1) and stores the result in the B accumulator with bits 55–48 and 23–0 unchanged.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	**S**	**L**	E	U	**N**	**Z**	**V**	C
← MR →								← CCR →							

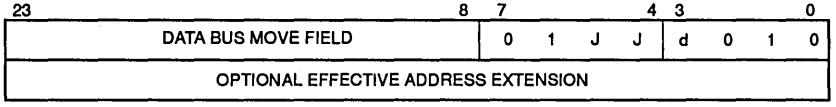
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47-24 of A or B result are zero**
V — **Always cleared**

A

Instruction Format:

OR S,D

Opcode:



Instruction Fields:

S	J J	D d
X0	0 0	A 0
X1	1 0	B 1
Y0	0 1	
Y1	1 1	

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
#xx+D → D
where + denotes the logical inclusive OR operator

Assembler Syntax:
OR(I) #xx,D

Description: Logically OR the 8-bit immediate operand (#xx) with the contents of the destination control register D and store the result in the destination control register. The condition codes are affected only when the condition code register is specified as the destination operand.

Restrictions: The ORI #xx,MR instruction cannot be used **immediately before** an ENDDO or RTI instruction and cannot be one of the **last three** instructions in a DO loop (at LA–2, LA–1, or LA).

Example:

```
        :
OR  #$8,MR              ;set scaling mode bit S1 to scale up
        :
```

	Before Execution		After Execution
MR	$03	MR	$0B

Explanation of Example: Prior to execution, the 8-bit mode register (MR) contains the value $03. The OR #$8,MR instruction logically ORs the immediate 8-bit value $8 with the contents of the mode register and stores the result in the mode register.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

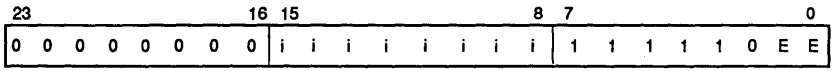
For CCR operand:
S — **Set if bit 7 of the immediate operand is set**
L — **Set if bit 6 of the immediate operand is set**
E — **Set if bit 5 of the immediate operand is set**
U — **Set if bit 4 of the immediate operand is set**
N — **Set if bit 3 of the immediate operand is set**
Z — **Set if bit 2 of the immediate operand is set**
V — **Set if bit 1 of the immediate operand is set**
C — **Set if bit 0 of the immediate operand is set**

A

For MR and OMR operands:
The condition codes are not affected using these operands.

Instruction Format:
OR(I) #xx,D

Opcode:



Instruction Fields:
#xx=8-bit Immediate Short Data = i i i i i i i i

D	E E
MR	0 0
CCR	0 1
OMR	1 0

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

Operation:	Assembler Syntax:	
LC → TEMP; X:ea → LC Repeat next instruction until LC=1 TEMP → LC	REP	X:ea
LC → TEMP; X:aa → LC Repeat next instruction until LC=1 TEMP → LC	REP	X:aa
LC → TEMP; Y:ea → LC Repeat next instruction until LC=1 TEMP → LC	REP	Y:ea
LC → TEMP; Y:aa → LC Repeat next instruction until LC=1 TEMP → LC	REP	Y:aa
LC → TEMP; S → LC Repeat next instruction until LC=1 TEMP → LC	REP	S
LC → TEMP; #xxx → LC Repeat next instruction until LC=1 TEMP → LC	REP	#xxx

Description: Repeat the **single-word instruction** immediately following the REP instruction the specified number of times. The value specifying the number of times the given instruction is to be repeated is loaded into the 16-bit loop counter (LC) register. The single-word instruction is then executed the specified number of times, decrementing the loop counter (LC) after each execution until LC=1. When the REP instruction is in effect, the repeated instruction is fetched only one time, and it remains in the instruction register for the duration of the loop count. Thus, **the REP instruction is not interruptible** (sequential repeats are also not interruptible). The current loop counter (LC) value is stored in an internal temporary register. If LC is set equal to zero, the instruction is repeated 65,536 times. The instruction's effective address specifies the address of the value which is to be loaded into the loop counter (LC). All address register indirect addressing modes may be used. The absolute short and the immediate short addressing modes may also be used. The four MS bits of the 12-bit immediate value are zeroed to form the 16-bit value that is to be loaded into the loop counter (LC).

Restrictions: The REP instruction can repeat any single-word instruction except the REP instruction itself and any instruction that changes program flow. The following instructions are not allowed to follow an REP instruction:

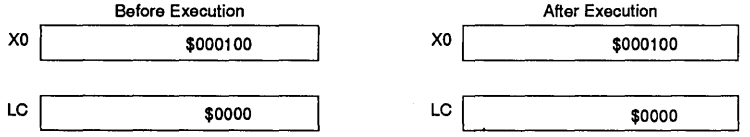
Immediately after REP

DO	JSSET
Jcc	REP
JCLR	RTI
JMP	RTS
JSET	STOP
JScc	SWI
JSCLR	WAIT
JSR	ENDDO

Also, a REP instruction cannot be the **last** instruction in a DO loop (at LA). The assembler will generate an error if any of the previous instructions are found immediately following an REP instruction.

Example:

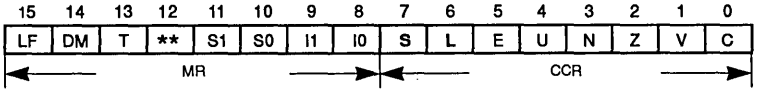
```
        :
REP X0                                      ;repeat (X0) times
MAC X1,Y1,A   X:(R1)+,X1   Y:(R4)+,Y1       ;X1*Y1+A → A, update X1,Y1
        :
```



A

Explanation of Example: Prior to execution, the 24-bit X0 register contains the value $000100, and the 16-bit loop counter (LC) register contains the value $0000. The execution of the REP X0 instruction takes the 24-bit value in the X0 register, truncates the MS 8 bits, and stores the 16 LS bits in the 16-bit loop counter (LC) register. Thus, the single-word MAC instruction immediately following the REP instruction is repeated $100 times.

Condition Codes:



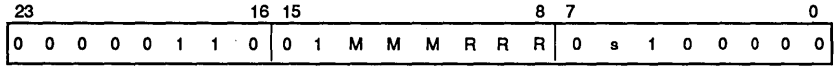
For source operand A or B:
S — Computed according to the definition. See Notes on page A-255.
L — Set if data limiting occurred. See Notes on page A-255.

For other source operands:
The condition code bits are not affected.

Instruction Format:

REP X:ea
REP Y:ea

Opcode:



Instruction Fields:

ea=6-bit Effective Address=MMMRRR,

Effective Addressing Mode	M M M R R R
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r

Memory Space	s
X Memory	0
Y Memory	1

where "rrr" refers to an address register R0-R7

Timing: 4+mv oscillator clock cycles

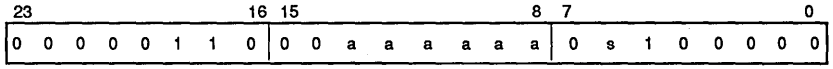
Memory: 1 program word

A

Instruction Format:

REP X:aa
REP Y:aa

Opcode:



Instruction Fields:

aa=6-bit Absolute Short Address=aaaaaa

Absolute Short Address aaaaaa	Memory Space s	
000000	X Memory	0
•	Y Memory	1
•		
111111		

Timing: 4+mv oscillator clock cycles

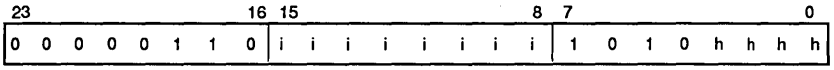
Memory: 1 program word

A

Instruction Format:

REP #xxx

Opcode:



Instruction Fields:

#xxx=12-bit Immediate Short Data = hhhh iiiiiiii

Immediate Short Data hhhh iiiiiiiii

000000000000
•
•
111111111111

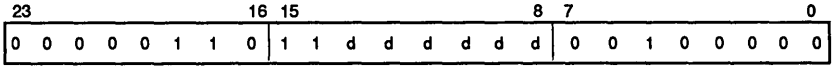
Timing: 4+mv oscillator clock cycles

Memory: 1 program word

Instruction Format:

REP S

Opcode:



Instruction Fields:

S	d d d d d d	S S/L	S	d d d d d d
X0	0 0 0 1 0 0	no	R0 - R7	0 1 0 n n n
X1	0 0 0 1 0 1	no	N0 - N7	0 1 1 n n n
Y0	0 0 0 1 1 0	no	M0 - M7	1 0 0 n n n
Y1	0 0 0 1 1 1	no	SR	1 1 1 0 0 1
A0	0 0 1 0 0 0	no	OMR	1 1 1 0 1 0
B0	0 0 1 0 0 1	no	SP	1 1 1 0 1 1
A2	0 0 1 0 1 0	no	SSH	1 1 1 1 0 0
B2	0 0 1 0 1 1	no	SSL	1 1 1 1 0 1
A1	0 0 1 1 0 0	no	LA	1 1 1 1 1 0
B1	0 0 1 1 0 1	no	LC	1 1 1 1 1 1
A	0 0 1 1 1 0	yes (See Notes on page A-255)		
B	0 0 1 1 1 1	yes (See Notes on page A-255)		

where "nnn" = Rn number (R0 - R7)
Nn number (N0 - N7)
Mn number (M0 - M7)

A

Notes: If A or B is specified as the destination operand, the following sequence of events takes place:

1. The S bit is computed according to its definition (See Section A.5 CONDITION CODE COMPUTATION)
2. The accumulator value is scaled according to the scaling mode bits S0 and S1 in the status register (SR).
3. If the accumulator extension is in use, the output of the shifter is limited to the maximum positive or negative saturation constant, and the L bit is set.
4. The LS 16 bits of the resulting 24 bit value is loaded into the loop counter (LC). The original contents of A or B are not changed.

If the system stack register SSH is specified as a source operand, the system stack pointer (SP) is postdecremented by 1 after SSH has been read.

Timing: 4 oscillator clock cycles

Memory: 1 program word

Operation:
Reset the interrupt priority register and all on-chip peripherals

Assembler Syntax:
RESET

Description: Reset the interrupt priority register and all on-chip peripherals. This is a **software reset** which is **NOT** equivalent to a hardware reset since only on-chip peripherals and the interrupt structure are affected. The processor state is not affected, and execution continues with the next instruction. All interrupt sources are disabled except for the trace, stack error, NMI, illegal instruction, and hardware reset interrupts.

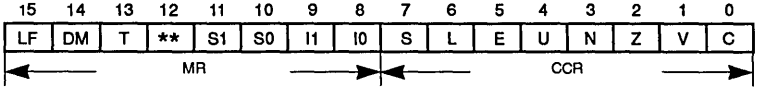
Restrictions: A RESET instruction cannot be the **last** instruction in a DO loop (at LA).

Example:

```
        :
RESET                   ;reset all on-chip peripherals and IPR
        :
```

Explanation of Example: The execution of the RESET instruction resets all on-chip peripherals and the interrupt priority register (IPR).

Condition Codes:

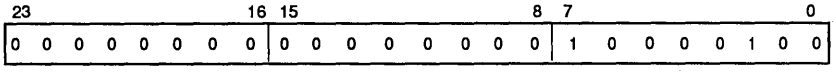


The condition codes are not affected by this instruction

A

Instruction Format:
RESET

Opcode:



Instruction Fields:
None

Timing: 4 oscillator clock cycles

Memory: 1 program word

A

RND — Round Accumulator — RND

Operation:
D+r → D (parallel move)

Assembler Syntax:
RND D (parallel move)

Description: Round the 56-bit value in the specified destination operand D and store the result in the MSP portion of the destination accumulator (A1 or B1). This instruction uses a convergent rounding technique. The contribution of the LS bits of the result (A0 and B0) is rounded into the upper portion of the result (A1 or B1) by adding a rounding constant to the LS bits of the result. The MSP portion of the destination accumulator contains the rounded result which may be read out to the data buses.

The value of the rounding constant added is determined by the scaling mode bits S0 and S1 in the system status register (SR). A "1" is added in the rounding position as shown below:

S1	S0	Scaling Mode	Rounding Position	Rounding Constant 55 - 25	24	23	22	21 - 0
0	0	No Scaling	23	0. . . .0	0	1	0	0. . . .0
0	1	Scale Down	24	0. . . .0	1	0	0	0. . . .0
1	0	Scale Up	22	0. . . .0	0	0	1	0. . . .0

Normal or "standard" rounding consists of adding a rounding constant to a given number of LS bits of a value to produce a rounded result. The rounding constant depends on the scaling mode being used as previously shown. Unfortunately, when using a twos-complement data representation, this process introduces a positive bias in the statistical distribution of the roundoff error.

A

Convergent rounding differs from "standard" rounding in that convergent rounding attempts to remove the aforementioned positive bias by equally distributing the round-off error. The convergent rounding technique initially performs "standard" rounding as previously described. Again, the rounding constant depends on the scaling mode being used. Once "standard" rounding has been done, the convergent rounding method tests the result to determine if **all** bits **including and to the right** of the rounding position are **zero. If, and only if**, this **special condition** is true, the convergent rounding method will clear the bit immediately to the **left** of the rounding position. When this special condition is true, numbers which have a "1" in the bit immediately to the left of the rounding position are rounded **up**; numbers with a "0" in the bit immediately to the left of the rounding position are rounded **down**. Thus, these numbers are rounded **up** half the time and rounded **down** the rest of the time. Therefore, the **roundoff error averages out to zero.** The LS bits of the convergently rounded result are then cleared so that the rounded result may be immediately used by the next instruction.

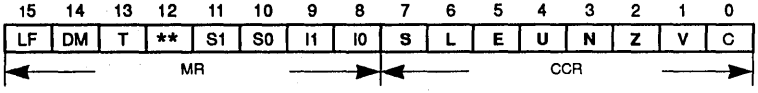
Example:

```
        :
RND A #$123456,X1 B,Y1      ;round A accumulator into A1, zero A0
        :
```

	Before Execution		After Execution
Case I: A	$00:123456:789ABC	A	$00:123456:000000
Case II: A	$00:123456:800000	A	$00:123456:000000
Case III: A	$00:123456:800000	A	$00:123456:000000

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:123456:789ABC for Case I, the value $00:123456:800000 for Case II, and the value $00:123455:800000 for Case III. The execution of the RND A instruction rounds the value in the A accumulator into the MSP portion of the A accumulator (A1), using convergent rounding, and then zeros the LSP portion of the A accumulator (A0). Note that Case II is the special case that distinguishes convergent rounding from standard or biased rounding.

Condition Codes:



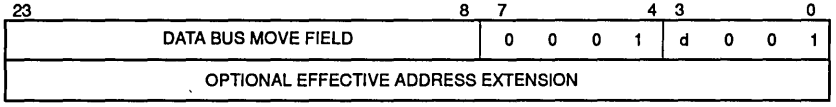
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

RND D

Opcode:



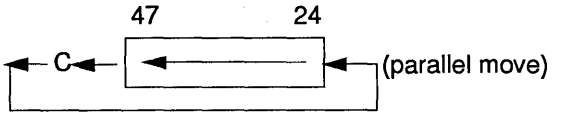
Instruction Fields:

D	D
A	0
B	1

Timing: 4 oscillator clock cycles

Memory: 1 program word

Operation:



Assembler Syntax: ROL D (parallel move)

Description: Rotate bits 47–24 of the destination operand D one bit to the left and store the result in the destination accumulator. Prior to instruction execution, bit 47 of D is shifted into the carry bit C, and, prior to instruction execution, the value in the carry bit C is shifted into bit 24 of the destination accumulator D. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

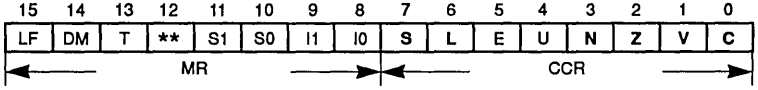
Example:

```
        :
ROL A1 #314,N2          ;rotate A1 one left bit, update N2
        :
```

	Before Execution	After Execution
A	$00:000000:000000	$00:000001:000000
SR	$0301	$0300

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:000000:000000. The execution of the ROL A instruction shifts the 24-bit value in the A1 register one bit to the left, shifting bit 47 into the carry bit C, rotating the carry bit C into bit 24, and storing the result back in the A1 register.

Condition Codes:

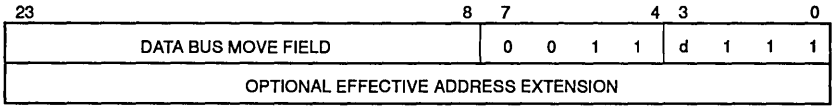


S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47–24 of A or B result are zero**
V — **Always cleared**
C — **Set if bit 47 of A or B was set prior to instruction execution**

Instruction Format:

ROL D

Opcode:



Instruction Fields:

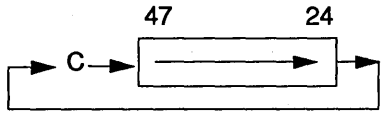
D	**d**
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

Operation:



Assembler Syntax: ROR D (parallel move)

Description: Rotate bits 47–24 of the destination operand D one bit to the right and store the result in the destination accumulator. Prior to instruction execution, bit 24 of D is shifted into the carry bit C, and, prior to instruction execution, the value in the carry bit C is shifted into bit 47 of the destination accumulator D. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

Example:

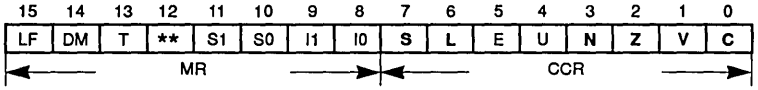
```
        :
ROR B1#$1234,R2      ;rotate B1 right one bit, update R2
        :
```

	Before Execution		After Execution
B	$00:000001:222222	B	$00:000000:222222
SR	$0300	SR	$0305

Explanation of Example: Prior to execution, the 56-bit B accumulator contains the value $00:000001:222222. The execution of the ROR B instruction shifts the 24-bit value in the B1 register one bit to the right, shifting bit 24 into the carry bit C, rotating the carry bit C into bit 47, and storing the result back in the B1 register.

A

Condition Codes:

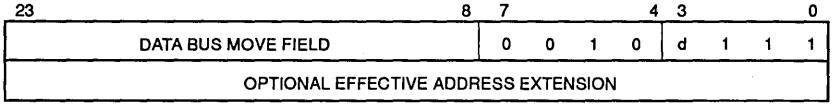


S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
N — **Set if bit 47 of A or B result is set**
Z — **Set if bits 47–24 of A or B result are zero**
V — **Always cleared**
C — **Set if bit 24 of A or B was set prior to instruction execution.**

Instruction Format:

ROR D

Opcode:



Instruction Fields:

D	d
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

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 program counter and status register are lost.

Restrictions: Due to pipelining in the program controller and the fact that the RTI instruction accesses certain program controller registers, the RTI instruction must not be immediately preceded by any of the following instructions:

Immediately before RTI
MOVEC to SR, SSH, SSL, or SP
MOVEM to SR, SSH, SSL, or SP
MOVEP to SR, SSH, SSL, or SP
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH
ANDI MR or ANDI CCR
ORI MR or ORI CCR

An RTI instruction cannot be the **last** instruction in a DO loop (at LA).

An RTI instruction cannot be repeated using the REP instruction.

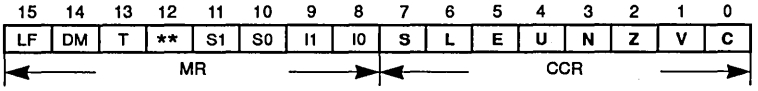
Example:

```
        :
RTI             ;pull PC and SR from system stack
        :
```

Explanation of Example: The RTI instruction pulls the 16-bit program counter (PC) and the 16-bit status register (SR) from the system stack and updates the system stack pointer (SP).

A

Condition Codes:

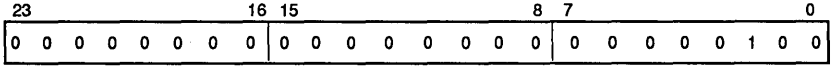


S — **Set according to the value pulled from the stack**
L — **Set according to the value pulled from the stack**
E — **Set according to the value pulled from the stack**
U — **Set according to the value pulled from the stack**
N — **Set according to the value pulled from the stack**
Z — **Set according to the value pulled from the stack**
V — **Set according to the value pulled from the stack**
C — **Set according to the value pulled from the stack**

Instruction Format:

RTI

Opcode:



Instruction Fields:

None

Timing: 4+rx oscillator clock cycles

Memory: 1 program word

A

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 program counter is lost. The status register (SR) is not affected.

Restrictions: Due to pipelining in the program controller and the fact that the RTS instruction accesses certain controller registers, the RTS instruction must not be immediately preceded by any of the following instructions:

Immediately before RTS
MOVEC to SSH, SSL, or SP
MOVEM to SSH, SSL, or SP
MOVEP to SSH, SSL, or SP
MOVEC from SSH
MOVEM from SSH
MOVEP from SSH

An RTS instruction cannot be the **last** instruction in a DO loop (at LA).

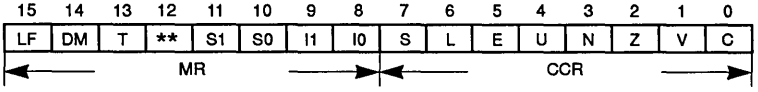
An RTS instruction cannot be repeated using the REP instruction.

Example:

```
        :
RTS             ;pull PC from system stack
        :
```

Explanation of Example: The RTS instruction pulls the 16-bit program counter (PC) from the system stack and updates the system stack pointer (SP).

Condition Codes:



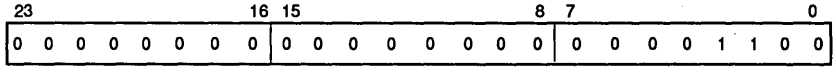
The condition codes are not affected by this instruction.

RTS — Return from Subroutine — RTS

Instruction Format:

RTI

Opcode:



Instruction Fields:

None

Timing: 4+rx oscillator clock cycles

Memory: 1 program word

A

Operation:
D–S–C → D (parallel move)

Assembler Syntax:
SBC S,D (parallel move)

Description: Subtract the source operand S and the carry bit C of the condition code register from the destination operand D and store the result in the destination accumulator. Long words (48 bits) may be subtracted from the (56-bit) destination accumulator.

Note: The carry bit is set correctly for multiple-precision arithmetic using long-word operands if the extension register of the destination accumulator (A2 or B2) is the sign extension of bit 47 of the destination accumulator (A or B).

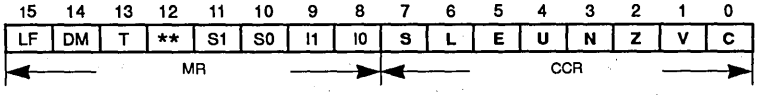
Example:

```
        :
MOVE L:<$0,X        ;get a 48-bit LS long-word operand in X
MOVE L:<$1,A        ;get other LS long word in A (sign ext.)
MOVE L:<$2,Y        ;get a 48-bit MS long-word operand in Y
SUB X,A L:<$3,B     ;sub. LS words; get other MS word in B
SBC YB A10,L:<$4    ;sub. MS words with carry; save LS dif.
MOVE B10,L:<$5      ;save MS difference
        :
```

	Before Execution	After Execution
A	$00:000000:000000	$00:800000:000000
X	$800000:000000	$800000:000000
B	$00:000000:000003	$00:000000:000001
Y	$000000:000001	$000000:000001

Explanation of Example: This example illustrates long-word double-precision (96-bit) subtraction using the SBC instruction. Prior to execution of the SUB and SBC instructions, the 96-bit value $000000:000001:800000:000000 is loaded into the Y and X registers (X:Y), respectively. The other double-precision 96-bit value $000000:000003:000000:000000 is loaded into the B and A accumulators (B:A), respectively. Since the 48-bit value loaded into the A accumulator is automatically sign extended to 56 bits and the other 48-bit long-word operand is internally sign extended to 56 bits during instruction execution, the carry bit will be set correctly after the execution of the SUB X,A instruction. The SBC Y,B instruction then produces the correct MS 56-bit result. The actual 96-bit result is stored in memory using the A10 and B10 operands (instead of A and B) because shifting and limiting is not desired.

Condition Codes:



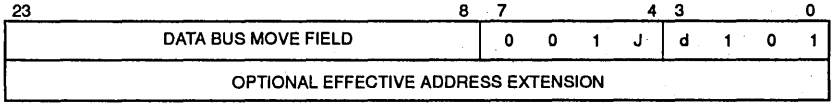
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

SBC S,D

Opcode:



Instruction Fields:

S,D	J d
X,A	0 0
X,B	0 1
Y,A	1 0
Y,B	1 1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

STOP — Stop Instruction Processing — STOP

Operation:
Enter the stop processing state and stop the clock oscillator

Assembler Syntax:
STOP

Description: Enter the STOP processing state. All activity in the processor is suspended until the $\overline{\text{RESET}}$ or $\overline{\text{IRQA}}$ pin is asserted. The clock oscillator is gated off internally. The STOP processing state is a low-power standby state.

During the STOP state, port A is in an idle state with the control signals held inactive (i.e., $\overline{\text{RD}}$=$\overline{\text{WR}}$=V_{CC} etc.), the data pins (D0–D23) are high impedance, and the address pins (A1–A15) are unchanged from the previous instruction. If the bus grant was asserted when the STOP instruction was executed, port A will remain three-stated until the DSP exits the STOP state.

If the exit from the STOP state was caused by a low level on the $\overline{\text{RESET}}$ pin, then the processor will enter the reset processing state. The time to recover from the STOP state using $\overline{\text{RESET}}$ will depend on the oscillator used. Consult the *DSP56001 Advance Information Data Sheet* (ADI1290) for details.

If the exit from the STOP state was caused by a low level on the $\overline{\text{IRQA}}$ pin, then the processor will service the highest priority pending interrupt and will not service the $\overline{\text{IRQA}}$ interrupt unless it is highest priority. The interrupt will be serviced after an internal delay counter counts 65,536 clock cycles (or a three clock cycle delay if the stop delay bit in the OMR is set to one) plus 17T (see the *DSP56001 Technical Data Sheet* (ADI1290) for details). During this clock stabilization count delay, all peripherals and external interrupts are cleared and re-enabled/arbitrated at the start of the 17T period following the count interval. The processor will resume program execution at the instruction following the STOP instruction that caused the entry into the STOP state after the interrupt has been serviced or, if no interrupt was pending, immediately after the delay count plus 17T. If the $\overline{\text{IRQA}}$ pin is asserted when the STOP instruction is executed, the clock will not be gated off, and the internal delay counter will be started.

A

STOP Stop Instruction Processing STOP

Restrictions:
A STOP instruction cannot be used in a fast interrupt routine.

A STOP instruction cannot be the **last** instruction in a DO loop (i.e., at LA).

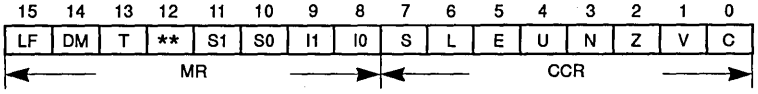
A STOP instruction cannot be repeated using the REP instruction.

Example:

```
        :
STOP                        ;enter low-power standby mode
        :
```

Explanation of Example: The STOP instruction suspends all processor activity until the processor is reset or interrupted as previously described. The STOP instruction puts the processor in a low-power standby state.

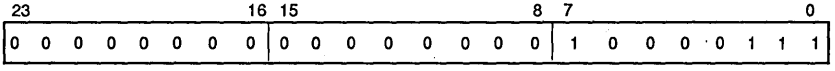
Condition Codes:



The condition codes are not affected by this instruction.

Instruction Format:
STOP

Opcode:



Instruction Fields:
None

Timing: The STOP instruction disables the internal clock oscillator and internal distribution of the external clock.

Memory: 1 program word

A

SUB — Subtract — SUB

Operation:
D–S → D (parallel move)

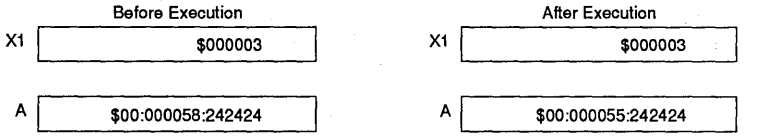
Assembler Syntax:
SUB S,D (parallel move)

Description: Subtract the source operand S from the destination operand D and store the result in the destination operand D. Words (24 bits), long words (48 bits), and accumulators (56 bits) may be subtracted from the destination accumulator.

Note: The carry bit is set correctly using word or long-word source operands if the extension register of the destination accumulator (A2 or B2) is the sign extension of bit 47 of the destination accumulator (A or B). The carry bit is always set correctly using accumulator source operands.

Example:

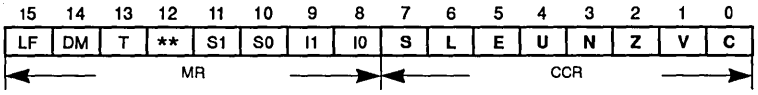
```
        :
SUB X1,A X:(R2)+N2,R0        ;24-bit subtract, load R0, update R2
        :
```



Explanation of Example: Prior to execution, the 24-bit X1 register contains the value $000003, and the 56-bit A accumulator contains the value $00:000058:242424. The SUB instruction automatically appends the 24-bit value in the X1 register with 24 LS zeros, sign extends the resulting 48-bit long word to 56 bits, and subtracts the result from the 56-bit A accumulator. Thus, 24-bit operands are subtracted from the MSP portion of A or B (A1 or B1) because all arithmetic instructions assume a fractional, twos complement data representation. Note that 24-bit operands can be subtracted from the LSP portion of A or B (A0 or B0) by loading the 24-bit operand into X0 or Y0, forming a 48-bit word by loading X1 or Y1 with the sign extension of X0 or Y0, and executing a SUB X,A or SUB Y,A instruction.

SUB — Subtract — SUB

Condition Codes:



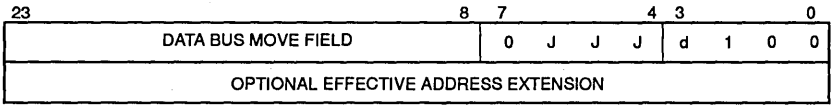
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

SUB S,D

Opcode:



Instruction Fields:

S,D	J J J d	S,D	J J J d	S,D	J J J d
B,A	0 0 1 0	X0,A	1 0 0 0	Y1,A	1 1 1 0
A,B	0 0 1 1	X0,B	1 0 0 1	Y1,B	1 1 1 1
X,A	0 1 0 0	Y0,A	1 0 1 0		
X,B	0 1 0 1	Y0,B	1 0 1 1		
Y,A	0 1 1 0	X1,A	1 1 0 0		
Y,B	0 1 1 1	X1,B	1 1 0 1		

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

SUBL Shift Left and Subtract Accumulators SUBL

Operation:
2*D–S → D (parallel move)

Assembler Syntax:
SUBL SD (parallel move)

Description: Subtract the source operand S from two times the destination operand D and store the result in the destination accumulator. The destination operand D is arithmetically shifted one bit to the left, and a zero is shifted into the LS bit of D prior to the subtraction operation. The carry bit is set correctly if the source operand does not overflow as a result of the left shift operation. The overflow bit may be set as a result of either the shifting or subtraction operation (or both). This instruction is useful for efficient divide and decimation in time (DIT) FFT algorithms.

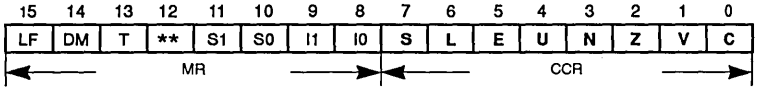
Example:

```
        :
SUBL  A,B     Y:(R5+N5),R7   ;2*B–A → B, load R7, no R5 update
        :
```

	Before Execution		After Execution
A	$00:004000:000000	A	$00:004000:000000
B	$00:005000:000000	B	$00:006000:000000

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $00:004000:000000, and the 56-bit B accumulator contains the value $00:005000:000000. The SUBL A,B instruction subtracts the value in the A accumulator from two times the value in the B accumulator and stores the 56-bit result in the B accumulator.

Condition Codes:



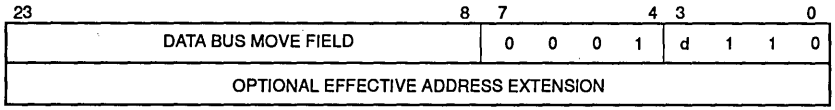
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — **Set if overflow has occurred in A or B result or if the MS bit of the destination operand is changed as a result of the instruction's left shift**
C — Set if a carry (or borrow) occurs from bit 55 of A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

SUBL S,D

Opcode:



Instruction Fields:

S,D	d
B,A	0
A,B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

Operation:
D/2–S → D (parallel move)

Assembler Syntax:
SUBR S,D (parallel move)

Description: Subtract the source operand S from one-half the destination operand D and store the result in the destination accumulator. The destination operand D is arithmetically shifted one bit to the right while the MS bit of D is held constant prior to the subtraction operation. In contrast to the SUBL instruction, the carry bit is always set correctly, and the overflow bit can only be set by the subtraction operation, and not by an overflow due to the initial shifting operation. This instruction is useful for efficient divide and decimation in time (DIT) FFT algorithms.

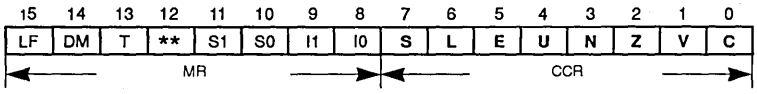
Example:

```
        :
SUBR B,A N5,Y:-(R5)          ;A/2-B → A, update R5, save N5
        :
```

	Before Execution	After Execution
A	$80:000000:2468AC	$C0:000000:000000
B	$00:000000:123456	$00:000000:123456

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $80:000000:2468AC, and the 56-bit B accumulator contains the value $00:000000:123456. The SUBR B,A instruction subtracts the value in the B accumulator from one-half the value in the A accumulator and stores the 56-bit result in the A accumulator.

Condition Codes:



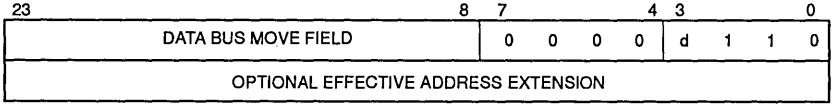
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if limiting (parallel move) or overflow has occurred in result
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — Set if overflow has occurred in A or B result
C — Set if a carry (or borrow) occurs from bit 55 of A or B result

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

SUBR S,D

Opcode:



Instruction Fields:

S,D	d
B,A	0
A,B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

SWI — Software Interrupt — SWI

Operation:
Begin SWI exception processing

Assembler Syntax:
SWI

Description: Suspend normal instruction execution and begin SWI exception processing. The interrupt priority level (I1,I0) is set to 3 in the status register (SR) if a long interrupt service routine is used.

Restrictions:
An SWI instruction cannot be used in a fast interrupt routine.

An SWI instruction cannot be repeated using the REP instruction.

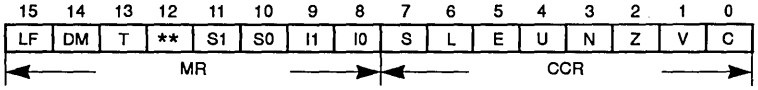
Example:

```
        :
SWI             ;begin SWI exception processing
        :
```

Explanation of Example: The SWI instruction suspends normal instruction execution and initiates SWI exception processing.

A

Condition Codes:

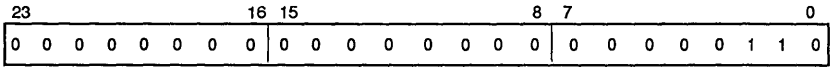


The condition codes are not affected by this instruction.

Instruction Format:

SWI

Opcode:



Instruction Fields:

None

Timing: 8 oscillator clock cycles

Memory: 1 program word

A

Operation:

If cc, then S1 → D1

If cc, then S1 → D1 and S2 → D2

Assembler Syntax:

Tcc S1,D1

Tcc S1,D1 S2,D2

Description: Transfer data from the specified source register S1 to the specified destination accumulator D1 if the specified condition is true. If a second source register S2 and a second destination register D2 are also specified, transfer data from address register S2 to address register D2 if the specified condition is false, a NOP is executed. The term "cc" may specify the following conditions:

	"cc" Mnemonic	Condition
CC (HS)	— carry clear (higher or same)	C=0
CS (LO)	— carry set (lower)	C=1
EC	— extension clear	E=0
EQ	— equal	Z=1
ES	— extension set	E=1
GE	— greater than or equal	$N \oplus V=0$
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
$\overline{U}$ denotes the logical complement of U,
+ denotes the logical OR operator,
• denotes the logical AND operator, and
$\oplus$ denotes the logical Exclusive OR operator

A

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 destina-

tion accumulator D1. If address register S2 is used as an address pointer into an array of data, the address of the desired value is stored in the address register D2. The Tcc instruction may be used after any instruction and allows efficient searching and sorting algorithms.

The Tcc instruction uses the internal data ALU paths and internal address ALU paths. The Tcc instruction does not affect the condition code bits.

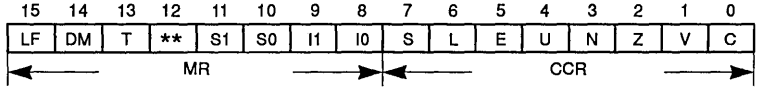
Note: This instruction is considered to be a move-type instruction. Due to instruction pipelining, if an AGU register (Mn, Nn, or Rn) is directly changed with this instruction, the new contents may not be available for use until the second following instruction. See the restrictions discussed in A.9.6 - R, N, and M Register Restrictions on page A-310.

Example:

```
        :
CMP X0,A                ;compare X0 and A (sort for minimum)
TGT X0,A R0,R1          ;transfer X0 → A and R0 → R1 if X0<A
        :
```

Explanation of Example: In this example, the contents of the 24-bit X0 register are transferred to the 56-bit A accumulator, and the contents of the 16-bit R0 address register are transferred to the 16-bit R1 address register if the specified condition is true. If the specified condition is not true, a NOP is executed.

Condition Codes:



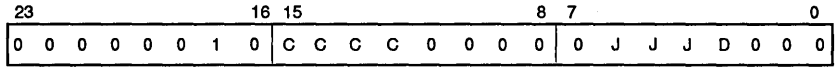
The condition codes are not affected by this instruction.

A

Instruction Format:

Tcc S1,D1

Opcode:



Instruction Fields:

cc=4=bit condition code=CCCC

S1,D1	J	J	J	D
B,A	0	0	0	0
A,B	0	0	0	1
X0,A	1	0	0	0
X0,B	1	0	0	1
X1,A	1	1	0	0
X1,B	1	1	0	1
Y0,A	1	0	1	0
Y0,B	1	0	1	1
Y1,A	1	1	1	0
Y1,B	1	1	1	1

Mnemonic	C	C	C	C
CC (HS)	0	0	0	0
GE	0	0	0	1
NE	0	0	1	0
PL	0	0	1	1
NN	0	1	0	0
EC	0	1	0	1
LC	0	1	1	0
GT	0	1	1	1

Mnemonic	C	C	C	C
CS (LO)	1	0	0	0
LT	1	0	0	1
EQ	1	0	1	0
MI	1	0	1	1
NR	1	1	0	0
ES	1	1	0	1
LS	1	1	1	0
LE	1	1	1	1

Timing: 2 oscillator clock cycles

Memory: 1 program word

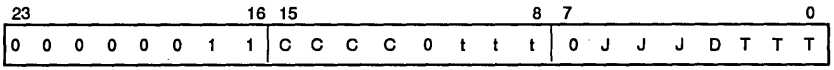
A

Tcc — Transfer Conditionally — Tcc

Instruction Format:

Tcc S1,D1 S2,D2

Opcode:



Instruction Fields:

cc=4=bit condition code=CCCC

S1,D1	J	J	J	D
B,A	0	0	0	0
A,B	0	0	0	1
X0,A	1	0	0	0
X0,B	1	0	0	1
X1,A	1	1	0	0
X1,B	1	1	0	1
Y0,A	1	0	1	0
Y0,B	1	0	1	1
Y1,A	1	1	1	0
Y1,B	1	1	1	1

S2	t	t	t
Rn	n	n	n

D2	T	T	T
Rn	n	n	n

Mnemonic	C	C	C	C
CC (HS)	0	0	0	0
GE	0	0	0	1
NE	0	0	1	0
PL	0	0	1	1
NN	0	1	0	0
EC	0	1	0	1
LC	0	1	1	0
GT	0	1	1	1

Mnemonic	C	C	C	C
CS (LO)	1	0	0	0
LT	1	0	0	1
EQ	1	0	1	0
MI	1	0	1	1
NR	1	1	0	0
ES	1	1	0	1
LS	1	1	1	0
LE	1	1	1	1

where "nnn"=Rn number (R0–R7)

Timing: 2 oscillator clock cycles

Memory: 1 program word

A

TFR Transfer Data ALU Register TFR

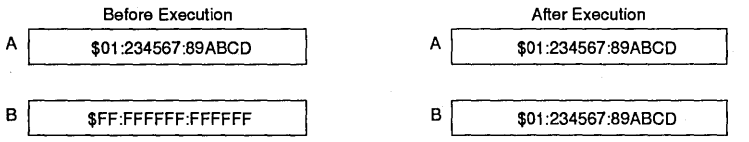
Operation:
S→D (parallel move)

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 56-bit contents of one of the accumulators to be transferred into the other accumulator **without** data shifting and/or limiting. Moreover, since TFR uses the internal data ALU data paths, parallel moves are possible. The TFR instruction only affects the L condition code bit which can be set by data limiting associated with the instruction's **parallel move** operations.

Example:

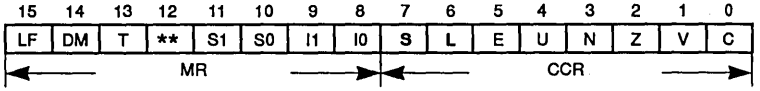
```
        :
TFR A,B  A,X1     Y:(R4+N4),Y0      ;move A to B and X1, update Y0
        :
```



Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $01:234567:89ABCD, and the 56-bit B accumulator contains the value $ff:FFFFFF:FFFFFF. The execution of the TFR A,B instruction moves the 56-bit value in the A accumulator into the 56-bit B accumulator using the internal data ALU data paths without any data shifting and/or limiting. The value in the B accumulator **would** have been limited if a MOVE A,B instruction had been used. Note, however, that the **parallel move** portion of the TFR instruction **does** use the data shifter/limiters. Thus, the value stored in the 24-bit X1 register (not shown) **would** have been limited in this example. This example illustrates a **triple** move instruction.

A

Condition Codes:

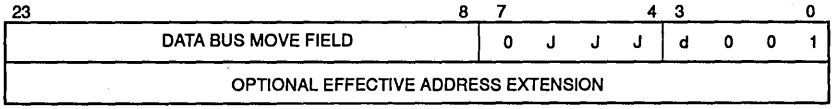


S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move

Instruction Format:

TFR S,D

Opcode:



Instruction Fields:

S,D	J	J	J	D
B,A	0	0	0	0
A,B	0	0	0	1
X0,A	1	0	0	0
X0,B	1	0	0	1
X1,A	1	1	0	0
X1,B	1	1	0	1
Y0,A	1	0	1	0
Y0,B	1	0	1	1
Y1,A	1	1	1	0
Y1,B	1	1	1	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

Operation:
S–0 (parallel move)

Assembler Syntax:
TST S (parallel move)

Description: Compare the specified source accumulator S with zero and set the condition codes accordingly. No result is stored although the condition codes are updated.

Example:

```
        :
TST A   #$345678,B      ;set CCR bits for value in A, update B
        :
```

	Before Execution	After Execution
A	$01:020304:000000	$01:020304:000000
CCR	$0300	$0330

Explanation of Example: Prior to execution, the 56-bit A accumulator contains the value $01:020304:000000, and the 16-bit condition code register contains the value $0300. The execution of the TST A instruction compares the value in the A register with zero and updates the condition code register accordingly. The contents of the A accumulator are not affected.

Condition Codes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	**	S1	S0	I1	I0	S	L	E	U	N	Z	V	C
← MR →								← CCR →							

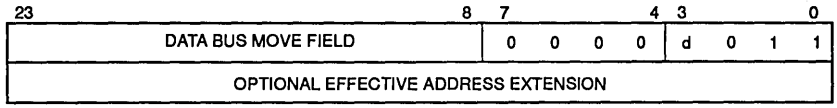
S — Computed according to the definition in A.5 CONDITION CODE COMPUTATION
L — Set if data limiting has occurred during parallel move
E — Set if the signed integer portion of A or B result is in use
U — Set if A or B result is unnormalized
N — Set if bit 55 of A or B result is set
Z — Set if A or B result equals zero
V — **Always cleared**

Note: The definitions of the E and U bits vary according to the scaling mode being used. Refer to Section A.5 CONDITION CODE COMPUTATION for complete details.

Instruction Format:

TST S

Opcode:



Instruction Fields:

S	d
A	0
B	1

Timing: 2+mv oscillator clock cycles

Memory: 1+mv program words

A

WAIT — Wait for Interrupt — WAIT

Operation:
Disable clocks to the processor core and enter the WAIT processing state.

Assembler Syntax:
WAIT

Description: Enter the WAIT processing state. The internal clocks to the processor core and memories are gated off, and all activity in the processor is suspended until an unmasked interrupt occurs. The clock oscillator and the internal I/O peripheral clocks remain active. If WAIT is executed when an interrupt is pending, the interrupt will be processed; the effect will be the same as if the processor never entered the WAIT state and three NOPs followed the WAIT instruction. When an unmasked interrupt or external (hardware) processor RESET occurs, the processor leaves the WAIT state and begins exception processing of the unmasked interrupt or RESET condition. The BR/BG circuits remain active during the WAIT state. The WAIT state is a low-power standby state. The processor always leaves the WAIT state in the T2 clock phase (see the DSP56001 Advance Information Data Sheet (ADI1290)). Therefore, multiple processors may be synchronized by having them all enter the WAIT state and then interrupting them with a common interrupt.

Restrictions: A WAIT instruction cannot be used in a fast interrupt routine.

A WAIT instruction cannot be the **last** instruction in a DO loop (at LA).

A WAIT instruction cannot be repeated using the REP instruction.

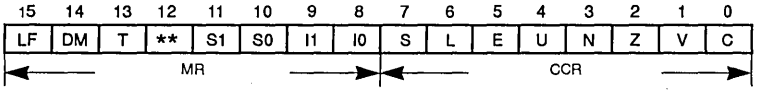
Example:

```
        :
WAIT                    ;enter low power mode, wait for interrupt
        :
```

Explanation of Example: The WAIT instruction suspends normal instruction execution and waits for an unmasked interrupt or external RESET to occur.

A

Condition Codes:

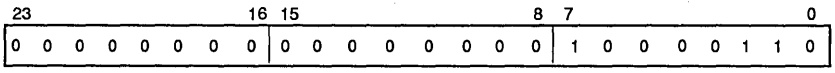


The condition codes are not affected by this instruction.

Instruction Format:

WAIT

Opcode:



Instruction Fields:

None

Timing: The WAIT instruction takes a minimum of 16 cycles to execute when an internal interrupt is pending during the execution of the WAIT instruction

Memory: 1 program word

A

A.8 INSTRUCTION TIMING

This section describes how to calculate DSP56K instruction timing manually using the tables provided. Three complete examples illustrate the "layered" nature of the tables.

Alternatively, the user can determine the number of instruction program words and the number of oscillator clock cycles required for a given instruction by using the DSP56K simulator. *This method of determining instruction timing information is much faster and much simpler than using the tables.* This powerful software package is available for the IBM*™ PC and SUN workstation.

- Table A-6 gives the number of instruction program words and the number of oscillator clock cycles for each instruction mnemonic.
- Table A-7 gives the number of additional (if any) instruction words and additional (if any) clock cycles for each type of parallel move operation.
- Table A-8 gives the number of additional (if any) clock cycles for each type of MOVEC operation.
- Table A-9 gives the number of additional (if any) clock cycles for each type of MOVEP operation.
- Table A-10 gives the number of additional (if any) clock cycles for each type of bit manipulation (BCHG, BCLR, BSET, and BTST) operation.
- Table A-11 gives the number of additional (if any) clock cycles for each type of jump (Jcc, JCLR, JMP, JScc, JSCLR, JSET, JSR, and JSSET) operation.
- Table A-12 gives the number of additional (if any) clock cycles for the RTI and RTS instructions.
- Table A-13 gives the number of additional (if any) instruction words and additional (if any) clock cycles for each effective addressing mode.
- Table A-14 gives the number of additional (if any) clock cycles for external data, external program, and external I/O memory accesses.

The number of words per instruction is dependent on the addressing mode and the type of parallel data bus move operation specified. The symbols used reference subsequent tables to complete the instruction word count.

The number of oscillator clock cycles per instruction is dependent on many factors,

*IBM is a trademark of International Business Machines.
SUN is a trademark of Sun Microsystems, Inc.

including the number of words per instruction, the addressing mode, whether the instruction fetch pipe is full or not, the number of external bus accesses, and the number of wait states inserted in each external access. The symbols used reference subsequent tables to complete the execution clock cycle count.

All tables are based on the following assumptions:

1. All instruction cycles are counted in **oscillator clock cycles**.
2. The instruction fetch pipeline is **full**.
3. There is no contention for **instruction** fetches. Thus, external program instruction fetches are assumed not to have to contend with external data memory accesses.
4. There are no wait states for **instruction** fetches done sequentially (as for non-change-of-flow instructions), but they are taken into account for change-of-flow instructions which flush the pipeline such as JMP, Jcc, RTI, etc.

To help the user better understand and use the timing tables, the following three examples illustrate the tables' "layered" nature. (Remember that it is faster and simpler to use the DSP56K simulator to calculate instruction timing.)

Example 16: Arithmetic Instruction with Two Parallel Moves

Problem: Calculate the number of 24-bit instruction program words and the number of oscillator clock cycles required for the instruction

```
MACR -X0,X0,A    X1,X:(R6)-    Y0,Y:(R0)+
```

where
Operating Mode Register (OMR) = $02 (normal expanded memory map),
Bus Control Register (BCR) = $1135,
R6 Address Register = $0052 (internal X memory), and
R0 Address Register = $0523 (external Y memory).

Solution: To determine the number of instruction program words and the number of oscillator clock cycles required for the given instruction, the user should perform the following operations:

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in Table A-6.

A

According to Table A-6, the MACR instruction will require (1+mv) instruction program words and will execute in (2+mv) oscillator clock cycles. The term "mv" represents the additional (if any) instruction program words and the additional (if any) oscillator clock

cycles that may be required over and above those needed for the basic MACR instruction due to the parallel move portion of the instruction.

2. Evaluate the "mv" term using Table A-7.

The parallel move portion of the MACR instruction consists of an XY memory move. According to Table A-7, the parallel move portion of the instruction will require mv=0 additional instruction program words and mv=(ea+axy) additional oscillator clock cycles. The term "ea" represents the number of additional (if any) oscillator clock cycles that are required for the effective addressing move specified in the parallel move portion of the instruction. The term "axy" represents the number of additional (if any) oscillator clock cycles that are required to access an **XY** memory operand.

3. Evaluate the "ea" term using Table A-13.

The parallel move portion of the MACR instruction consists of an XY memory move which uses both address register banks (R0–R3 and R4–R7) in generating the effective addresses of the XY memory operands. Thus, the two effective address operations occur in parallel, and the larger of the two "ea" terms should be used. The X memory move operation uses the "postdecrement by 1" effective addressing mode. According to Table A-13, this operation will require ea=0 additional oscillator clock cycles. The Y memory move operation uses the "postincrement by 1" effective addressing mode. According to Table A-13, this operation will also require ea=0 additional oscillator clock cycles. Thus, using the maximum value of "ea", the effective addressing modes used in the parallel move portion of the MACR instruction will require ea=0 additional oscillator clock cycles.

4. Evaluate the "axy" term using Table A-14.

The parallel move portion of the MACR instruction consists of an XY memory move. According to Table A-14, the term "axy" depends upon where the referenced X and Y memory locations are located in the DSP56K memory space. **External** memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the DSP56K bus control register (BCR). Thus, assuming that the 16-bit bus control register contains the value $1135, external **X** memory accesses require wx=1 **wait** state of additional oscillator clock cycle while external **Y** memory accesses require wy=1 **wait** state or additional oscillator clock cycle. For this example, the X memory reference is assumed to be an **internal** reference; the Y memory reference is assumed to be an **external** reference. Thus, according to Table A-14, the XY memory reference in the parallel move portion of the MACR instruction will require axy=wy=1 additional oscillator clock cycle.

5. Compute final results.

Thus, based upon the assumptions given for Table A-6 and those listed in the problem statement for Example 1, the instruction

MACR -X0,X0,A X1,X:(R6)- Y0,Y:(R0)+

will require

(1+mv)
= (1+0)
= 1 instruction program word

and will execute in

= (2+mv)
= (2+ea+axy)
= (2+ea+wy)
= (2+0+1) oscillator clock cycles.
= 3

Note that if a similar calculation were to be made for a MOVEC, MOVEM, MOVEP, or one of the bit manipulation (BCHG, BCLR, BSET, or BTST) instructions, the use of Table A-7 would no longer be appropriate. For one of these cases, the user would refer to Table A-8, Table A-9, or Table A-10, respectively.

Example 17: Jump Instruction

Problem: Calculate the number of 24-bit instruction program words and the number of oscillator clock cycles required for the instruction

JLC (R2+N2)

where Operating Mode Register (OMR) = $02 (normal expanded memory map),
Bus Control Register (BCR) = $2246,
R2 Address Register = $1000 (external P memory), and
N2 Address Register = $0037.

Solution: To determine the number of instruction program words and the number of oscillator clock cycles required for the given instruction, the user should perform the following operations:

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in Table A-6.

According to Table A-6, the Jcc instruction will require (1+ea) instruction program words and will execute in (4+jx) oscillator clock cycles. The term "ea" represents the number of

A

additional (if any) instruction program words that are required for the effective address of the Jcc instruction. The term "jx" represents the number of additional (if any) oscillator clock cycles required for a jump-type instruction.

2. Evaluate the "jx" term using Table A-11.

According to Table A-11, the Jcc instruction will require jx=ea+(2 * ap) additional oscillator clock cycles. The term "ea" represents the number of additional (if any) oscillator clock cycles that are required for the **e**ffective **a**ddressing mode specified in the Jcc instruction. The term "ap" represents the number of additional (if any) oscillator clock cycles that are required to **a**ccess a **P** memory operand. Note that the "+(2 * ap)" term represents the two program memory instruction fetches executed at the end of a one-word jump instruction to refill the instruction pipeline.

3. Evaluate the "ea" term using Table A-13.

The JLC (R2+N2) instruction uses the "indexed by offset Nn" effective addressing mode. According to Table A-13, this operation will require ea=0 additional instruction program words and ea=2 additional oscillator clock cycles.

4. Evaluate the "ap" term using Table A-14.

According to Table A-14, the term "ap" depends upon where the referenced P memory location is located in the DSP56K memory space. **External** memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the DSP56K bus control register (BCR). Thus, assuming that the 16-bit bus control register contains the value $2246, external **P** memory accesses require wp=4 **w**ait states or additional oscillator clock cycles. For this example, the P memory reference is assumed to be an **external** reference. Thus, according to Table A-14, the Jcc instruction will use the value ap=wp=4 oscillator clock cycles.

5. Compute final results.

Thus, based upon the assumptions given for Table A-6 and those listed in the problem statement for Example 2, the instruction

JLC (R2+N2)

will require

= (1+ea)
= (1+0)
= 1 instruction program word

and will execute in

= (4+jx)
= (4+ea+(2 * ap))
= (4+ea+(2 * wp))
= (4+2+(2 * 4)) oscillator clock cycles.
= 14

Example 18: RTI Instruction

Problem: Calculate the number of 24-bit instruction program words and the number of oscillator clock cycles required for the instruction

RTI

where Operating Mode Register (OMR) = 02 (normal expanded memory map),
Bus Control Register (BCR) = $0012, and,
Return Address (on the stack) = $0100 (internal P memory).

Solution: To determine the number of instruction program words and the number of oscillator clock cycles required for the given instruction, the user should perform the following operations:

1. Look up the number of instruction program words and the number of oscillator clock cycles required for the opcode-operand portion of the instruction in Table A-6.

According to Table A-6, the RTI instruction will require one instruction program word and will execute in (4+rx) oscillator clock cycles. The term "rx" represents the number of additional (if any) oscillator clock cycles required for an RTI or RTS instruction.

2. Evaluate the "rx" term using Table A-12.

According to Table A-12, the RTI instruction will require rx=(2 * ap) additional oscillator clock cycles. The term "ap" represents the number of additional (if any) oscillator clock cycles that are required to access a P memory operand. Note that the term "(2 * ap)" represents the two program memory instruction fetches executed at the end of an RTI or RTS instruction to refill the instruction pipeline.

3. Evaluate the "ap" term using Table A-14.

According to Table A-14, the term "ap" depends upon where the referenced P memory location is located in the DSP56K memory space. **External** memory accesses require additional oscillator clock cycles according to the number of wait states programmed into the DSP56K bus control register (BCR). Thus, assuming that the 16-bit bus control register contains the value $0012, external **P** memory accesses require wp=1 **w**ait state or additional oscillator clock cycles. For this example, the P memory reference is assumed to be an **internal** reference. This means that the return address ($0100) pulled from the system stack by the RTI instruction is in internal P memory. Thus, according to Table A-14, the RTI instruction will use the value ap=0 additional oscillator clock cycles.

4. Compute final results.

Thus, based upon the assumptions given for Table A-6 and those listed in the problem statement for Example 3, the instruction

RTI

will require

1 instruction program word

and will execute in

(4+rx)
= (4+(2 * ap))
= (4+(2 * 0))
= 4 oscillator clock cycles

A

Table A-6 Instruction Timing Summary (see Note 3)

Mnemonic	Instruction Program Words	Osc. Clock Cycles	Notes	Mnemonic	Instruction Program Words	Osc. Clock Cycles	Notes
ABS	1 + mv	2 + mv		LSR	1 + mv	2 + mv	
ADC	1 + mv	2 + mv		LUA	1	4	
ADD	1 + mv	2 + mv		MAC	1 + mv	2 + mv	
ADDL	1 + mv	2 + mv		MACR	1 + mv	2 + mv	
ADDR	1 + mv	2 + mv		MOVE	1 + mv	2 + mv	
AND	1 + mv	2 + mv		MOVEC	1 + ea	2 + mvc	
ANDI	1	2		MOVEM	1 + ea	6 + ea + ap	
ASL	1 + mv	2 + mv		MOVEP	1 + ea	2 + mvp	
ASR	1 + mv	2 + mv		MPY	1 + mv	2 + mv	
BCHG	1 + ea	4 + mvb		MPYR	1 + mv	2 + mv	
BCLR	1 + ea	4 + mvb		NEG	1 + mv	2 + mv	
BSET	1 + ea	4 + mvb		NOP	1	2	
BTST	1 + ea	4 + mvb		NORM	1	2	
CLR	1 + mv	2 + mv		NOT	1 + mv	2 + mv	
CMP	1 + mv	2 + mv		OR	1 + mv	2 + mv	
CMPM	1 + mv	2 + mv		ORI	1	2	
DEBUG	1	4		REP	1	4 + mv	
DEBUGcc	1	4		RESET	1	4	
DEC	1	2		RND	1 + mv	2 + mv	
DIV	1	2		ROL	1 + mv	2 + mv	
DO	2	6 + mv		ROR	1 + mv	2 + mv	
ENDDO	1	2		RTI	1	4 + rx	
EOR	1 + mv	2 + mv		RTS	1	4 + rx	
INC	1	2		SBC	1 + mv	2 + mv	
Jcc	1 + ea	4 + jx		STOP	1	n/a	1
JCLR	2	6 + jx		SUB	1 + mv	2 + mv	
JMP	1 + ea	4 + jx		SUBL	1 + mv	2 + mv	
JScc	1 + ea	4 + jx		SUBR	1 + mv	2 + mv	
JSCLR	2	6 + jx		SWI	1	8	
JSET	2	6 + jx		Tcc	1	2	
JSR	1 + ea	4 + jx		TFR	1 + mv	2 + mv	
JSSET	2	6 + jx		TST	1 + mv	2 + mv	
LSL	1 + mv	2 + mv		WAIT	1	n/a	2

Note 1: The STOP instruction disables the internal clock oscillator. After clock turn on, an internal counter counts 65,536 clock cycles (if bit 6 in the OMR is clear) before enabling the clock to the internal DSP circuits. If bit 6 in the OMR is set, only six clock cycles are counted before enabling the clock to the external DSP circuits.

Note 2: The WAIT instruction takes a minimum of 16 cycles to execute when an internal interrupt is pending during the execution of the WAIT instruction.

Note 3: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ap" term should be added, and, to each two-word instruction, a "+(2*ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

A

Table A-7 Parallel Data Move Timing

Parallel Move Operation		+ mv Words	+ mv Cycles	Comments
	No Parallel Data Move	0	0	
I	Immediate Short Data	0	0	
R	Register to Register	0	0	
U	Address Register Update	0	0	
X:	X Memory Move	ea	ea + ax	See Note 1
X:R	X Memory and Register	ea	ea + ax	See Note 1
Y:	Y Memory Move	ea	ea + ay	See Note 1
R:Y	Y Memory and Register	ea	ea + ay	See Note 1
L:	Long Memory Move	ea	ea + axy	
X:Y:	XY Memory Move	0	ea + axy	
LMS(X)	LMS X Memory Moves	0	ea + ax	See Notes 1,2
LMS(Y)	LMS Y Memory Moves	0	ea + ay	See Notes 1,2

Note 1: The ax or ay term does not apply to MOVE IMMEDIATE DATA.
Note 2: The ea term does not apply to ABSOLUTE ADDRESS and IMMEDIATE DATA.

Table A-8 MOVEC Timing Summary (see Note 2)

MOVEC Operation	+ mvc Cycles	Comments
Immediate Short → Register	0	
Register↔ Register	0	
X Memory↔ Register	ea + ax	See Note 1
Y Memory↔ Register	ea + ay	See Note 1
P Memory↔ Register	4 + ea + ap	

Note 1: The ax or ay term does not apply to MOVE IMMEDIATE DATA.
Note 2: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ ap" term should be added, and to each two-word instruction, a "+ (2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

Table A-9 MOVEP Timing Summary (see Note 2)

MOVEP Operation	+ mvp Cycles	Comments
Register↔ Peripheral	aio	See Note 3
Register↔ Peripheral	2+aio	See Note 4
X Memory↔ Peripheral	2 + ea + ax + aio	See Note 1
Y Memory↔ Peripheral	2 + ea + ay + aio	See Note 1
P Memory↔ Peripheral	4 + ea + ap + aio	

Note 1: The" 2+ax" or "2+ay" terms do not apply to MOVE IMMEDIATE DATA.
Note 2: If assumption 4 is not applicable, then to each one-word instruction timing,a "+ ap" term should be added, and to each two-word instruction, a "+ (2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.
Note 3: "Register" refers to DATA_ALU register
Note 4: "Register" refers to non DATA_ALU register

A

Note that the "ap" term in Table A-8 and Table A-9 for the P memory move represents the wait states spent when accessing the program memory during DATA read or write operations and does not refer to instruction fetches.

Table A-10 Bit Manipulation Timing Summary (see Note 2)

Bit Manipulation Operation	+ mvb Cycles	Comments
Bxxx Peripheral	2 * aio	See Note 1
Bxxx X Memory	ea + (2 * ax)	See Note 1
Bxxx Y Memory	ea + (2 * ay)	See Note 1
Bxxx Register Direct	0	See Note 1
BTST Peripheral	aio	
BTST X Memory	ea + ax	
BTST Y Memory	ea + ay	

Note 1: Bxxx = BCHG, BCLR, or BSET.
Note 2: If assumption 4 is not applicable, then to each one-word instruction timing, a "+ ap" term should be added, and to each two-word instruction, a "+ (2 * ap)" term should be added to account for the program memory wait states spent to fetch an instruction word to fill the pipeline.

Table A-11 Jump Instruction Timing Summary

Jump Instruction Operation	+ jx Cycles	Comments
Jbit Register Direct	2 * ap	See Note 1
Jbit Peripheral	aio + (2 * ap)	See Note 1
Jbit X Memory	ea + ax + (2 * ap)	See Note 1
Jbit Y Memory	ea + ay + (2 * ap)	See Note 1
Jxxx	ea + (2 * ap)	See Note 2

Note 1: Jbit = JCLR, JSCLR, JSET, and JSSET
Note 2: Jxxx = Jcc, JMP, JScc, and JSR

All one-word jump instructions execute TWO program memory fetches to refill the pipeline, which is represented by the "+(2 * ap)" term.

All two-word jumps execute THREE program memory fetches to refill the pipeline, but one of those fetches is sequential (the instruction word located at the jump instruction 2nd word address+1), so it is not counted as per assumption 4. If the jump instruction was fetched from a program memory segment with wait states, another "ap" should be added to account for that third fetch.

Table A-12 RTI/RTS Timing Summary

Operation	+ rx Cycles
RTI	2 * ap
RTS	2 * ap

The term "2 * ap" comes from the two instruction fetches done by the RTI/RTS instruction to refill the pipeline.

Table A-13 Addressing Mode Timing Summary

Effective Addressing Mode	+ ea Words	+ ea Cycles
Address Register Indirect		
No Update	0	0
Postincrement by 1	0	0
Postdecrement by 1	0	0
Postincrement by Offset Nn	0	0
Postdecrement by Offset Nn	0	0
Indexed by Offset Nn	0	2
Predecrement by 1	0	2
Special		
Immediate Data	1	2
Absolute Address	1	2
Immediate Short Data	0	0
Short Jump Address	0	0
Absolute Sort Address	0	0
I/O Short Address	0	0
Implicit	0	0

Table A-14 Memory Access Timing Summary

Access Type	X Mem Access	Y Mem Access	P Mem Access	I/O Access	+ ax Cycle	+ ay Cycle	+ ap Cycle	+ aio Cycle	+ axy Cycle
X:	Int	—	—	—	0	—	—	—	—
X:	Ext	—	—	—	wx	—	—	—	—
Y:	—	Int	—	—	—	0	—	—	—
Y:	—	Ext	—	—	—	wy	—	—	—
P:	—	—	Int	—	—	—	0	—	—
P:	—	—	Ext	—	—	—	wp	—	—
I/O:	—	—	—	Int	—	—	—	0	—
I/O:	—	—	—	Ext	—	—	—	wio	—
L: XY:	Int	Int	—	—	—	—	—	—	0
L: XY:	Int	Ext	—	—	—	—	—	—	wy
L: XY:	Ext	Int	—	—	—	—	—	—	wx
L: XY:	Ext	Ext	—	—	—	—	—	—	2 + wx + wy

Note 1: wx = external X memory access wait states
wy = external Y memory access wait states
wp = external P memory access wait states
wio = external I/O memory access wait states

Note 2: wx, wy, wp, and wio are programmable from 0 - 15 wait states in the port A bus control register (BCR).

A.9 INSTRUCTION SEQUENCE RESTRICTIONS

Due to the pipelined nature of the DSP56K central processor, there are certain instruction sequences that are forbidden and will cause undefined operation. Most of these restricted sequences would cause contention for an internal resource, such as the stack register. The DSP assembler will flag these as assembly errors.

Most of the following restrictions represent very unusual operations which probably would never be used but are listed only for completeness.

Note: The DSP56K macro assembler is designed to recognize all restrictions and flag them as errors at the source code level. Since many of these are instruction sequence restrictions, they cannot be flagged as errors at the object code level such as when using the DSP56K simulator's single-line assembler. Therefore, if any changes are made at the object code level using the simulator, the user should always re-assemble his program at the source code level using the DSP56K macro assembler to verify that no restricted instruction sequences have been generated.

A.9.1 Restrictions Near the End of DO Loops

Proper DO loop operation is not guaranteed if an instruction **starting** at address **LA–2, LA–1, or LA** specifies one of the **program controller registers** SR, SP, SSL, LA, LC, or (implicitly) PC as a **destination** register. Similarly, the SSH register may not be specified as a **source or destination** register in an instruction starting at address **LA–2, LA–1, or LA**. Additionally, the SSH register cannot be specified as a **source** register in the **DO** instruction itself, and **LA** cannot be used as a **target** for **jumps to subroutine** (i.e., JSR, JScc, JSSET, or JSCLR to LA). The following instructions cannot **begin** at the indicated position(s) near the end of a DO loop:

At LA–2, LA–1, and LA	DO BCHG LA, LC, SR, SP, SSH, or SSL BCLR LA, LC, SR, SP, SSH, or SSL BSET LA, LC, SR, SP, SSH, or SSL BTST SSH JCLR/JSET/JSCLR/JSSET SSH MOVEC from SSH MOVEM from SSH MOVEP from SSH MOVEC to LA, LC, SR, SP, SSH, or SSL MOVEM to LA, LC, SR, SP, SSH, or SSL MOVEP to LA, LC, SR, SP, SSH, or SSL ANDI MR ORI MR
At LA	**any** two-word instruction* Jcc JMP JScc JSR REP RESET RTI RTS STOP WAIT

*This restriction applies to the situation in which the DSP56K simulator's single-line assembler is used to change the **last** instruction in a DO loop from a one-word instruction to a two-word instruction. All changes made using the simulator should be reassembled at the **source code** level using the DSP56K macro assembler to verify that no restricted instruction sequences have been generated.

A

Other Restrictions

DO SSH,xxxx
JSR to (LA) whenever the loop flag (LF) is set
JScc to (LA) whenever the loop flag (LF) is set
JSCLR to (LA) whenever the loop flag (LF) is set
JSSET to (LA) whenever the loop flag (LF) is set

Note: Due to pipelining, if an address register (R0–R7, N0–N7, or M0–M7) is changed using a move-type instruction (LUA, Tcc, MOVE, MOVEC, MOVEM, MOVEP, or parallel move), the new contents of the destination address register will not be available for use during the **following** instruction (i.e., there is a single instruction cycle pipeline delay). This restriction also applies to the situation in which the **last** instruction in a **DO** loop changes an address register **and** the **first** instruction at the **top** of the DO loop uses that same address register. The **top** instruction becomes the **following** instruction because of the loop construct. The assembler will generate a warning if this condition is detected.

A.9.2 Other DO Restrictions

Due to pipelining, the DO instruction must not be **immediately preceded** by any of the following instructions:

Immediately before DO

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

A.9.3 ENDDO Restrictions

Due to pipelining, the ENDDO instruction must not be **immediately preceded** by any of the following instructions:

Immediately before ENDDO

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

A.9.4 RTI and RTS Restrictions

Due to pipelining, the RTI and RTS instructions must not be **immediately preceded** by any of the following instructions:

Immediately before RTI

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

Immediately before RTS

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

A

A.9.5 SP and SSH/SSL Manipulation Restrictions

In addition to all the above restrictions concerning MOVEC, MOVEM, MOVEP, SP, SSH, and SSL, the following MOVEC, MOVEM, and MOVEP restrictions apply:

Immediately before MOVEC from SSH or SSL	BCHG to SP BCLR to SP BSET to SP
Immediately before MOVEM from SSH or SSL	BCHG to SP BCLR to SP BSET to SP
Immediately before MOVEP from SSH or SSL	BCHG to SP BCLR to SP BSET to SP
Immediately before MOVEC from SSH or SSL	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before MOVEM from SSH or SSL	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before MOVEP from SSH or SSL	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JCLR #n,SSH or SSL,xxxx	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JSET #n,SSH or SSL,xxxx	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JSCLR #n,SSH or SSL,xxxx	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JSSET #n,SSH or SSL,xxxx	MOVEC to SP MOVEM to SP MOVEP to SP
Immediately before JCLR #n,SSH or SSL,xxxx	BCHG to SP BCLR to SP BSET to SP
Immediately before JSET #n,SSH or SSL,xxxx	BCHG to SP BCLR to SP BSET to SP

Immediately before JSCLR from SSH or SSL,xxxx	BCHG to SP BCLR to SP BSET to SP
Immediately before JSSET from SSH or SSL,xxxx	BCHG to SP BCLR to SP BSET to SP

Also, the instruction MOVEC SSH,SSH is illegal.

A.9.6 R, N, and M Register Restrictions

Due to pipelining, if an address register Rn is the destination of a MOVE-type instruction except MOVEP (MOVE, MOVEC, MOVEM, LUA, Tcc), the new contents will not be available for use as an address pointer until the **second following instruction cycle**.

Likewise, if an offset register Nn or a modifier register Mn is the destination of a MOVE-type instruction except MOVEP, the new contents will not be available for use in address calculations until the **second following instruction cycle**.

However, if the processor is in the No Update addressing mode (where Mn and Nn are ignored) and register Mn or Nn is the destination of a MOVE instruction, the next instruction may use the corresponding Rn register as an address pointer. Also, if the processor is in the Postincrement by 1, Postdecrement by 1, or Predecrement by 1 addressing mode (where Nn is ignored), a MOVE to Nn may be immediately followed by an instruction that uses Rn as an address pointer.

Note: This restriction also applies to the situation in which the **last** instruction in a **DO** loop changes an address register using a move-type instruction **and** the **first** instruction at the **top** of the DO loop uses that same address register. The **top** instruction becomes the **following** instruction because of the loop construct. The DSP assembler will generate a warning if this condition is detected.

A.9.7 Fast Interrupt Routines

The following instructions may not be used in a fast interrupt routine:

In a fast interrupt routine

DO	MOVEM from SSH	STOP
ENDDO	MOVEP from SSH	SWI
RTI	ORI MR or ORI CCR	WAIT
RTS	ANDI MR or ANDI CCR	

MOVEC to LA, LC, SSH, SSL, SP, or SR
MOVEM to LA, LC, SSH, SSL, SP, or SR
MOVEP to LA, LC, SSH, SSL, SP, or SR
MOVEC from SSH

A

A.9.8 REP Restrictions

The REP instruction can repeat any single-word instruction except the REP instruction itself and any instruction that changes program flow. The following instructions are not allowed to follow an REP instruction:

Immediately after REP	DO
	Jcc
	JCLR
	JMP
	JSET
	JScc
	JSCLR
	JSR
	JSSET
	REP
	RTI
	RTS
	STOP
	SWI
	WAIT
	ENDDO

Also, an REP instruction cannot be the **last** instruction in a DO loop (at LA).

A.10 INSTRUCTION ENCODING

This section summarizes instruction encoding for the DSP56K instruction set. The instruction codes are listed in nominally descending order. The symbols used in decoding the various fields of an instruction are identical to those used in the Opcode section of the individual instruction descriptions. The user should always refer to the actual instruction description for complete information on the encoding of the various fields of that instruction.

Section A.10.1 gives the encodings for (1) various groupings of registers used in the instruction encodings, (2) condition code combinations, (3) addressing, and (4) addressing modes.

Section A.10.2 gives the encoding for the parallel move portion of an instruction. These 16-bit partial instruction codes may be combined with the 8-bit data ALU opcodes listed in Section A.10.3 to form a complete 24-bit instruction word.

Section A.10.3 gives the complete 24-bit instruction encoding for those instructions which do not allow parallel moves.

Section A.10.4 gives the encoding for the data ALU portion of those instructions which allow parallel data moves. These 8-bit partial instruction codes may be combined with the 16-bit parallel move opcodes listed in Section A.10.1 to form a complete 24-bit instruction word.

A.10.1 Partial Encodings for Use in Instruction Encoding

Table A-15 Single-Bit Register Encodings

Code	d*	e	f	Where:
0	A	X0	Y0	d = 2 Accumulators in Data ALU
1	B	X1	Y1	e = 2 Registers in Data ALU
				f = 2 Registers in Data ALU

* For class II encodings for R:Y and X:R, see Table A-16

Table A-16 Single-Bit Special Register Encodings

d	X:R Class II Opcode	R:Y Class II Opcode
0	A → X:<ea> X0 → A	Y0 → A A → Y:<ea>
1	B → X:<ea> X0 → B	Y0 → B B → Y:<ea>

Table A-17 Double-Bit Register Encodings

Code	DD	ee	ff
00	X0	X0	Y0
01	X1	X1	Y1
10	Y0	A	A
11	Y1	B	B

Where: DD = 4 registers in data ALU
ee = 4 XDB registers in data ALU
ff = 4 YDB registers in data ALU

A

Table A-18 Triple-Bit Register Encodings

Code	DDD	LLL	FFF	NNN	TTT	GGG
000	A0	A10	M0	N0	R0	*
001	B0	B10	M1	N1	R1	SR
010	A2	X	M2	N2	R2	OMR
011	B2	Y	M3	N3	R3	SP
100	A1	A	M4	N4	R4	SSH
101	B1	B	M5	N5	R5	SSL
110	A	AB	M6	N6	R6	LA
111	B	BA	M7	N7	R7	LC

* Reserved

Where: DDD: 8 accumulators in data ALU
LLL: 8 extended-precision registers in data ALU; LLL field is encoded as L0LL
FFF: 8 address modifier registers in address ALU
NNN: 8 address offset registers in address ALU
TTT: 8 address registers in address
FFF: 8 program controller registers

Table A-19(a) Four-Bit Register Encodings for 12 Registers in Data ALU

D	D	D	D	Description
0	0	X	X	Reserved
0	1	D	D	Data ALU Register
1	D	D	D	Data ALU Register

Table A-19(b) Four-Bit Register Encodings for 16 Condition Codes

Mnemonic	C	C	C	C	Mnemonic	C	C	C	C
CC(HS)	0	0	0	0	CS(LO)	1	0	0	0
GE	0	0	0	1	LT	1	0	0	1
NE	0	0	1	0	EQ	1	0	1	0
PL	0	0	1	1	MI	1	0	1	1
NN	0	1	0	0	NR	1	1	0	0
EC	0	1	0	1	ES	1	1	0	1
LC	0	1	1	0	LS	1	1	1	0
GT	0	1	1	1	LE	1	1	1	1

A

Table A-20 Five-Bit Register Encodings for 28 Registers in Data ALU and Address ALU

e d	e d	e or d	e d	e d	Description
0	0	0	0	X	Reserved
0	0	0	1	X	Reserved
0	0	1	D	D	Data ALU Register
0	1	D	D	D	Data ALU Register
1	0	T	T	T	Address ALU Register
1	1	N	N	N	Address Offset Register

Where: eeeee = source
ddddd = destination

Table A-21 Six-Bit Register Encodings for 43 Registers On-Chip

d	d	d	d	d	d	Description
0	0	0	0	X	X	Reserved
0	0	0	1	D	D	Data ALU Register
0	0	1	D	D	D	Data ALU Register
0	1	0	T	T	T	Address ALU Register
0	1	1	N	N	N	Address Offset Register
1	0	0	F	F	F	Address Modifier Register
1	0	1	X	X	X	Reserved
1	1	0	X	X	X	Reserved
1	1	1	G	G	G	Program Controller Register

Table A-22 Write Control Encoding

W	Operation
0	Read Register or Peripheral
1	Write Register or Peripheral

Table A-23 Memory Space Bit Encoding

S	Operation
0	X Memory
1	Y Memory

Table A-24 Program Control Unit Register Encoding

E	E	Register	
0	0	MR	Mode Register
0	1	CCR	Condition Code Register
1	0	OMR	Operating Mode Register
1	1	—	Reserved

Table A-25 Condition Code and Address Encoding

Code				Code Definition
			cccc	16 Condition Code Combinations
		b	bbbb	5-Bit Immediate Data
		iiii	iiii	8-Bit Immediate Data (int, frac, mask)
iiii	iiii	xxxx	hhhh	12-Bit Immediate Data (iiii iiii hhhh)
		aa	aaaa	6-Bit Absolute Short (Low) Address
		pp	pppp	6-Bit Absolute I/O (High) Address
	aaaa	aaaa	aaaa	12-Bit Fast Absolute Short (Low) Address

Table A-26 Effective Addressing Mode Encoding

M2	M1	M0	R2	R1	R0	Code Definition
0	0	0	r	r	r	Post - N
0	0	1	r	r	r	Post + N
0	1	0	r	r	r	Post - 1
0	1	1	r	r	r	Post + 1
1	0	0	r	r	r	No Update
1	0	1	r	r	r	Indexed + N
1	1	1	r	r	r	Pre - 1
1	1	0	0	0	0	Absolute Address
1	1	0	1	0	0	Immediate Data

MMM = three bits M2, M1, M0 determine mode

RRR = three bits R2, R1, R0 determine which address register number where rrr refers to the binary representation of the number

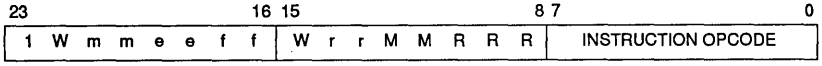
Notes:

(1) R2 is 0 for low register bank and 1 for the high register bank.
(2) M2 is 0 for all post update modes and 1 otherwise.
(3) M1 is 0 for update by register offset and no update and 1 otherwise.
(4) M0 is 0 for minus and 1 for plus, except for predecrement which is also 1.
(5) For X:Y: parallel data moves, bits 14 and 13 of the opcode are a subset of the above RRR and are labelled rr. See the XY parallel data move description for a detailed explanation.
(6) For X:Y: parallel data moves, bits 21 and 20 of the opcode are a subset of the above MMM and are labelled mm. See the XY parallel data move description for a detailed explanation

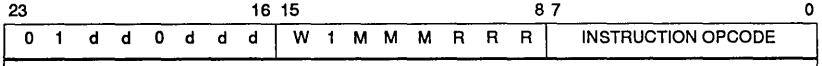
A

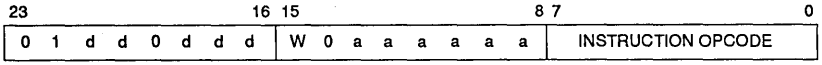
A.10.2 Instruction Encoding for the Parallel Move Portion of an Instruction

X: Y: Parallel Data Move

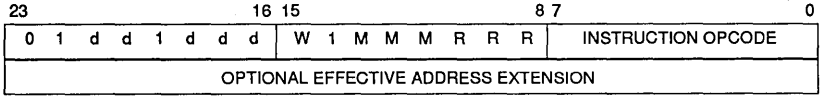


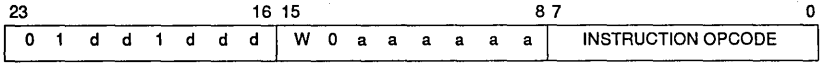
X: Parallel Data Move



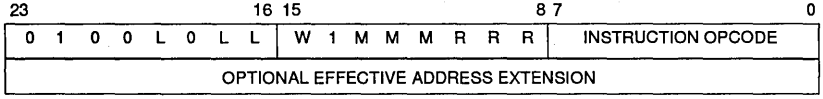


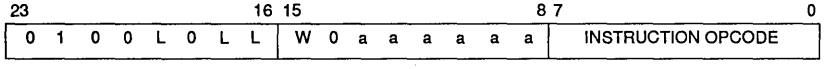
Y: Parallel Data Move





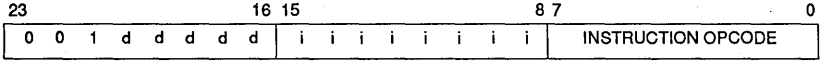
L: Parallel Data Move



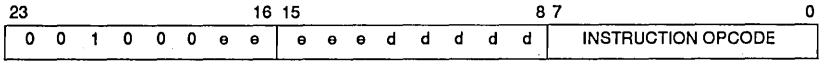


A

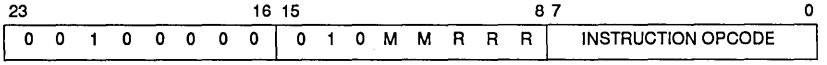
I: Immediate Short Parallel Data Move



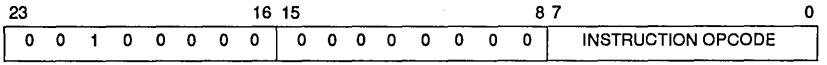
R: Register to Register Parallel Data Move



U: Address Register Update Parallel Data Move

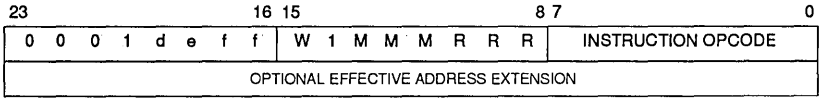


Parallel Data Move NOP

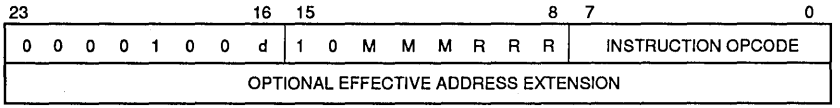


R:Y Parallel Data Move

(Class I)

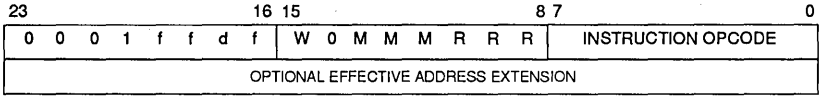


(Class II)

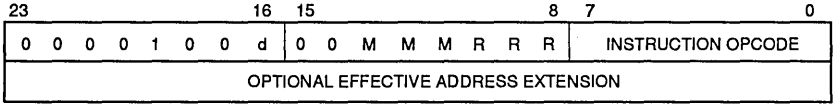


X:R Parallel Data Move

(Class I)



(Class II)

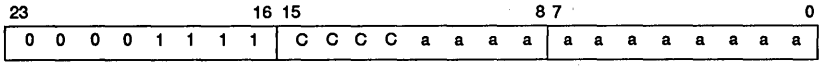


A

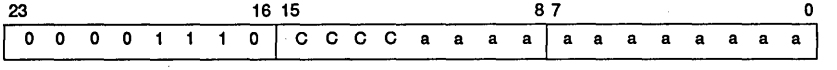
A.10.3 Instruction Encoding for Instructions Which Do Not Allow Parallel Moves

Note: For following bit class instructions bbbbb = 11bbb is reserved:
JSSET, JSCLR, JSET, JCLR, BTST, BCHG, BSET, and BCLR.

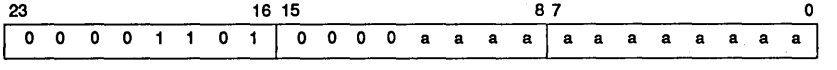
JScc xxx



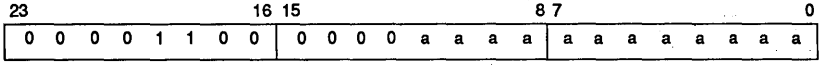
Jcc xxx



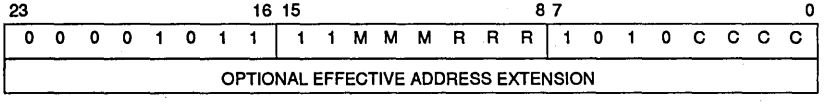
JSR xxx



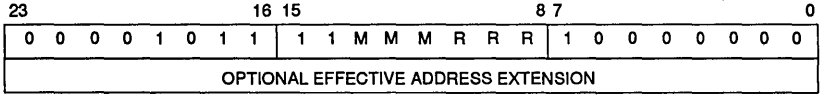
JMP xxx



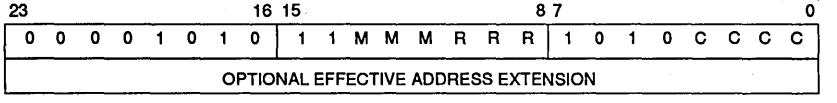
JScc ea



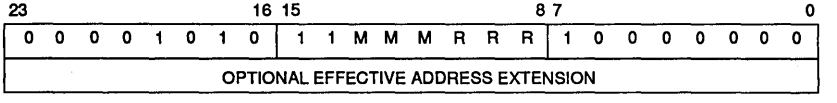
JSR ea



Jcc ea

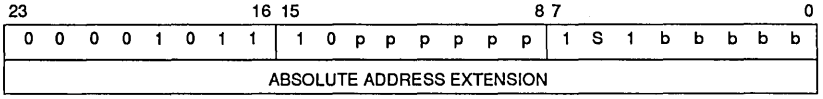


JMP ea

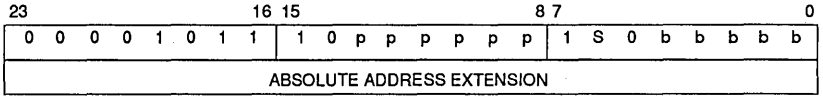


A

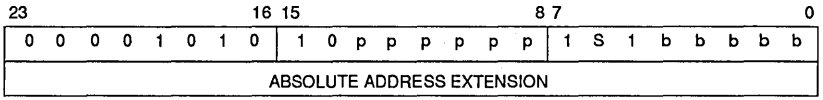
JSSET #n,X:pp,xxxx
JSSET #n,Y:pp,xxxx



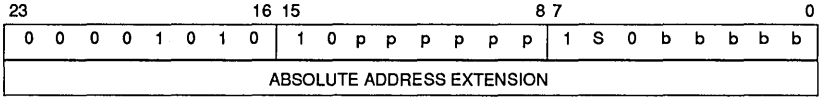
JSCLR #n,X:pp,xxxx
JSCLR #n,Y:pp,xxxx



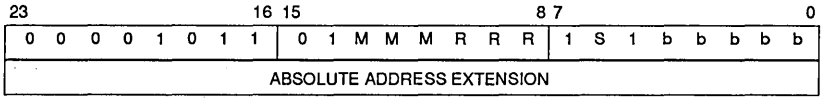
JSET #n,X:pp,xxxx
JSET #n,Y:pp,xxxx



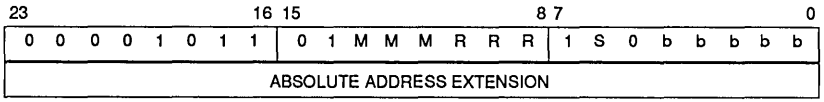
JCLR #n,X:pp,xxxx
JCLR #n,Y:pp,xxxx



JSSET #n,X:ea,xxxx
JSSET #n,Y:ea,xxxx

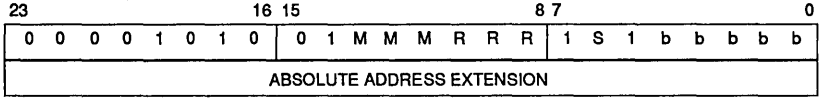


JSCLR #n,X:ea,xxxx
JSCLR #n,Y:ea,xxxx

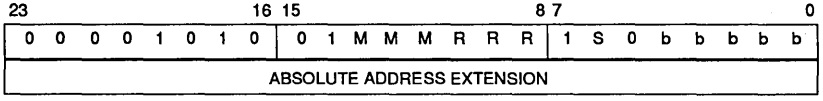


A

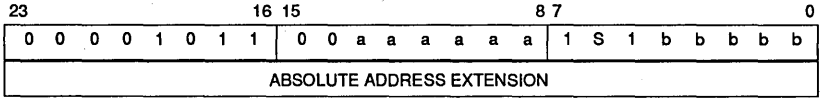
JSET #n,X:ea,xxxx
JSET #n,Y:ea,xxxx



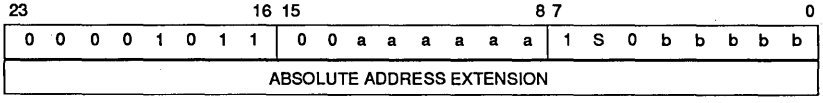
JCLR #n,X:ea,xxxx
JCLR #n,Y:ea,xxxx



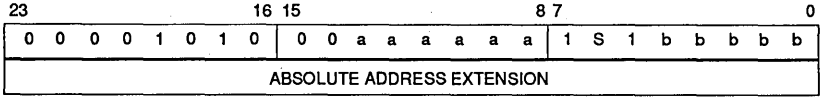
JSSET #n,X:aa,xxxx
JSSET #n,Y:aa,xxxx



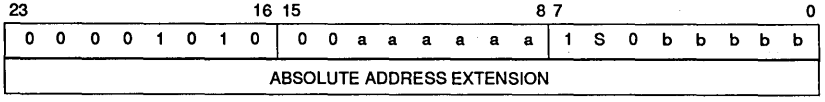
JSCLR #n,X:aa,xxxx
JSCLR #n,Y:aa,xxxx



JSET #n,X:aa,xxxx
JSET #n,Y:aa,xxxx

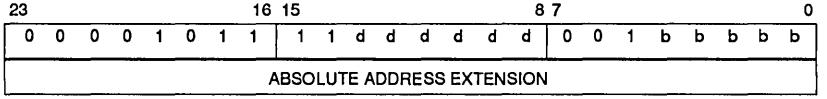


JCLR #n,X:aa,xxxx
JCLR #n,Y:aa,xxxx

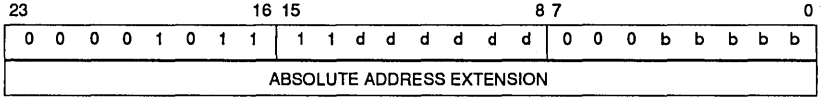


A

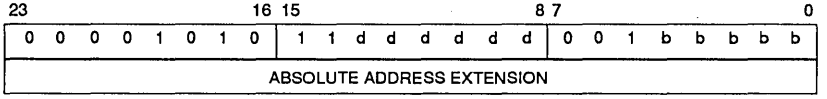
JSSET #n,S,xxxx



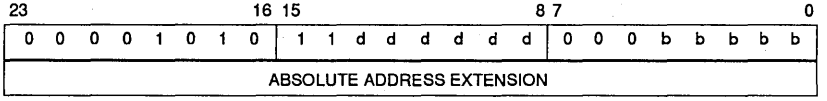
JSCLR #n,S,xxxx



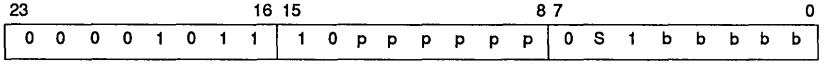
JSET #n,S,xxxx



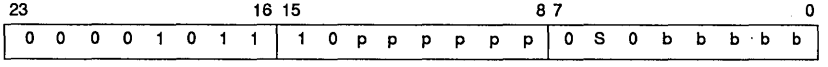
JCLR #n,S,xxxx



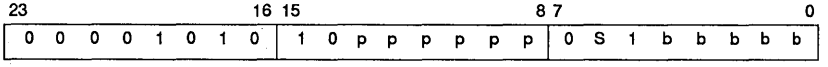
BTST #n,X:pp
BTST #n,Y:pp



BCHG #n,X:pp
BCHG #n,Y:pp

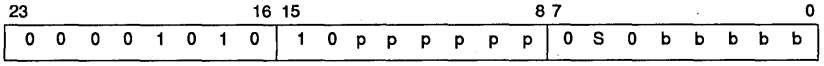


BSET #n,X:pp
BSET #n,Y:pp

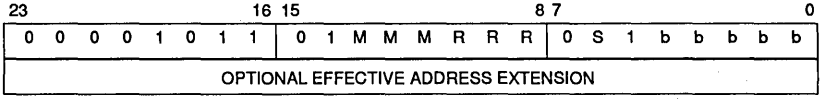


A

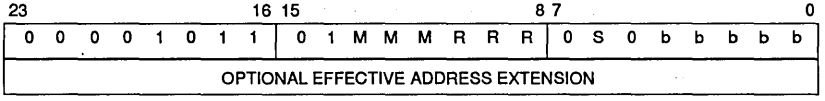
BCLR #n,X:pp
BCLR #n,Y:pp



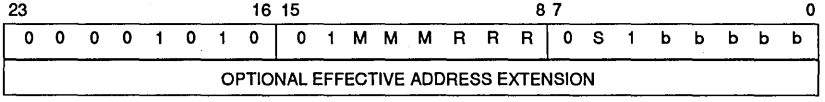
BTST #n,X:ea
BTST #n,Y:ea



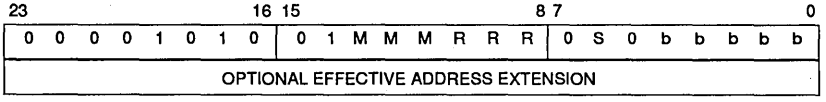
BCHG #n,X:ea
BCHG #n,Y:ea



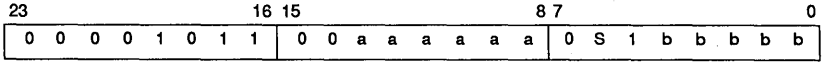
BSET #n,X:ea
BSET #n,Y:ea



BCLR #n,X:ea
BCLR #n,Y:ea

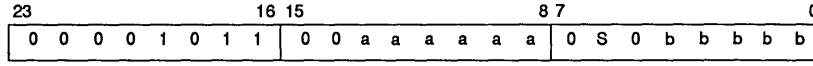


BTST #n,X:aa
BTST #n,Y:aa

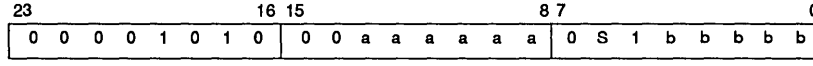


A

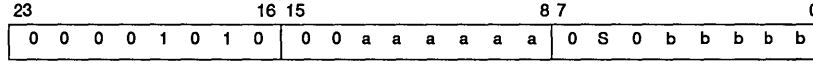
BCHG #n,X:aa
BCHG #n,Y:aa



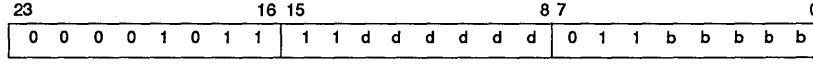
BSET #n,X:aa
BSET #n,Y:aa



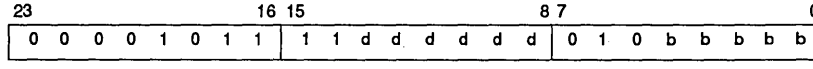
BCLR #n,X:aa
BCLR #n,Y:aa



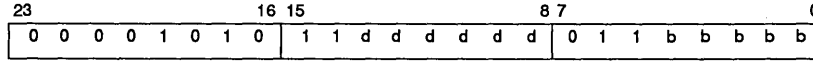
BTST #n,D



BCHG #n,D

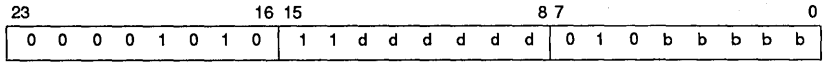


BSET #n,D

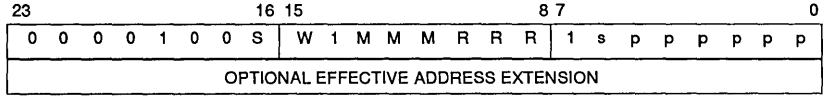


A

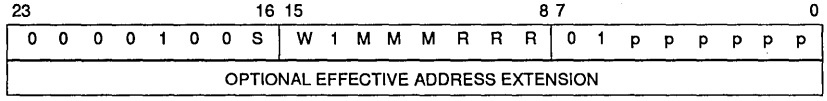
BCLR #n,D



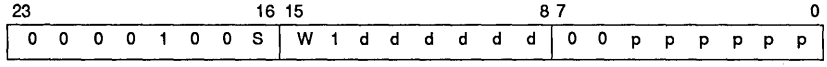
MOVEP X:ea,X:pp
MOVEP Y:ea,X:pp
MOVEP #xxxxxx,X:pp
MOVEP X:pp,X:ea
MOVEP X:pp,Y:ea
MOVEP X:ea,Y:pp
MOVEP Y:ea,Y:pp
MOVEP #xxxxxx,Y:pp
MOVEP Y:pp,X:ea
MOVEP Y:pp,Y:ea



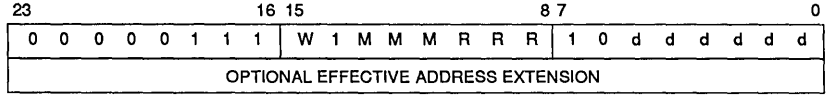
MOVEP P:ea,X:pp
MOVEP X:pp,P:ea
MOVEP P:ea,Y:pp
MOVEP Y:pp,P:ea



MOVEP S,X:pp
MOVEP X:pp,D
MOVEP S,Y:pp
MOVEP Y:pp,D

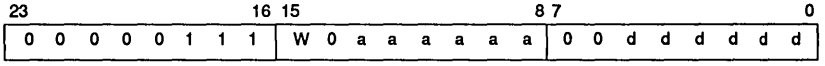


MOVE(M) S,P:ea
MOVE(M) P:ea,D

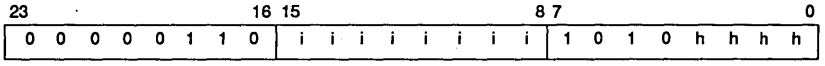


A

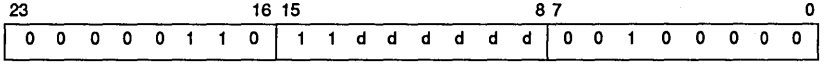
MOVE(M) S,P:aa
MOVE(M) P:aa,D



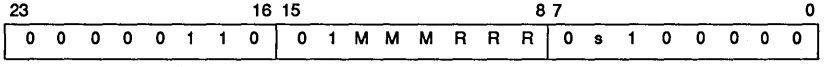
REP #xxx



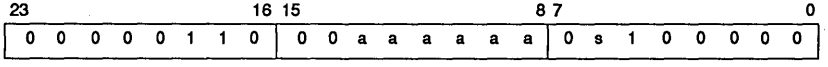
REP S



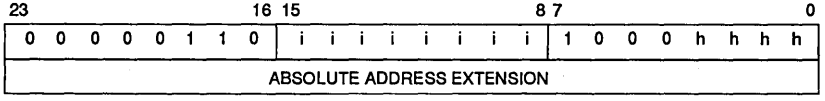
REP X:ea
REP Y:ea



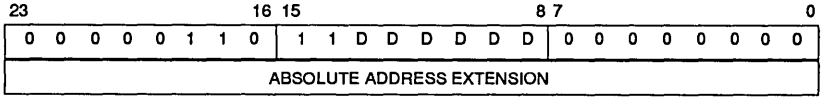
REP X:aa
REP Y:aa



DO #xxx,expr

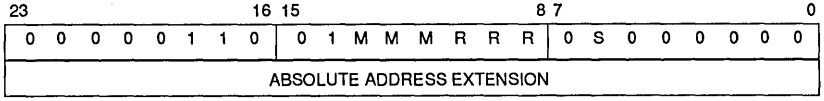


DO S,expr

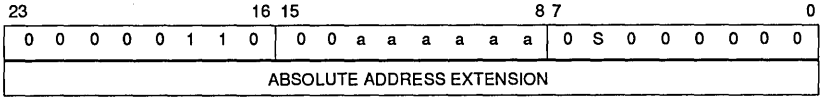


A

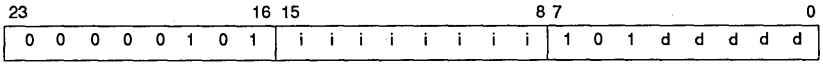
DO X:ea,expr
DO Y:ea,expr



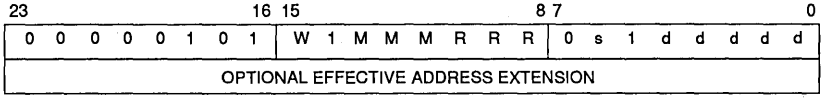
DO X:aa,expr
DO Y:aa,expr



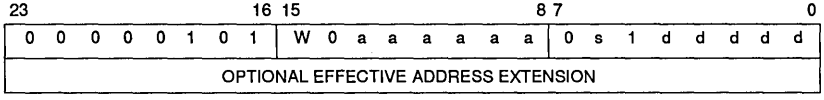
MOVE(C) #xx,D1



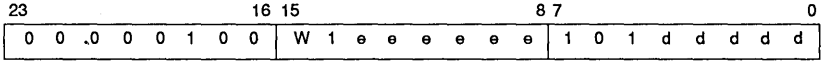
MOVE(C) X:ea,D1
MOVE(C) S1,X:ea
MOVE(C) Y:ea,D1
MOVE(C) S1,Y:ea
MOVE(C) #xxxx,D1



MOVE(C) X:aa,D1
MOVE(C) S1,X:aa
MOVE(C) Y:aa,D1
MOVE(C) S1,Y:aa

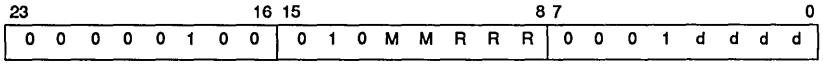


MOVE(C) S1,D2
MOVE(C) S2,D1

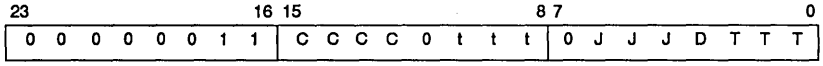


A

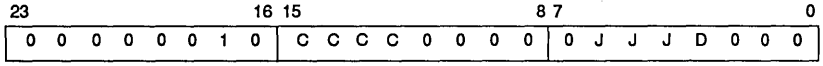
LUA ea,D



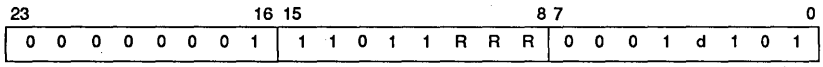
Tcc S1,D1 S2,D2



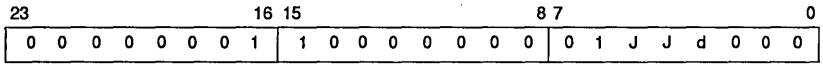
Tcc S1,D1



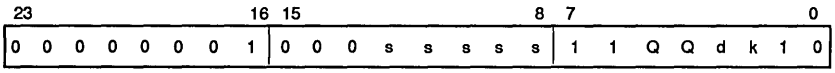
NORM Rn,D



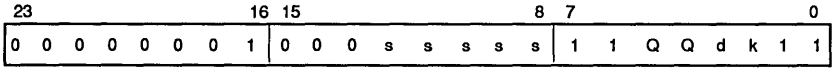
DIV S,D



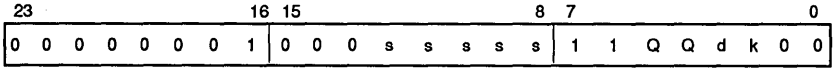
MAC (±)S,#n,D



MACR (±)S,#n,D

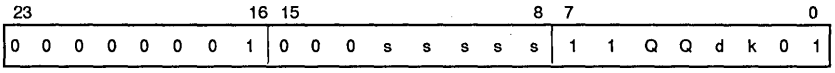


MPY (±)S,#n,D

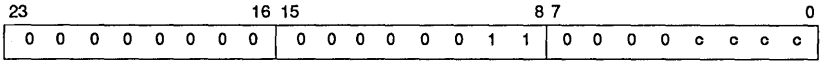


A

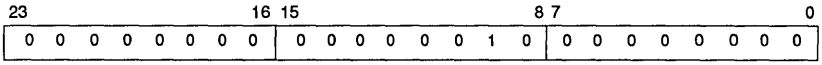
MPYR (±)S,#n,D



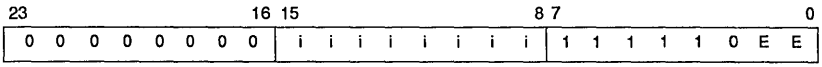
DEBUGcc



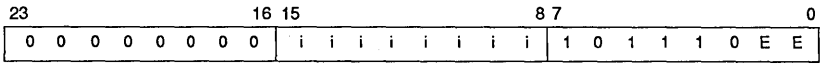
DEBUG



OR(I) #xx,D

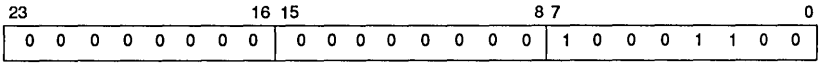


AND(I) #xx,D

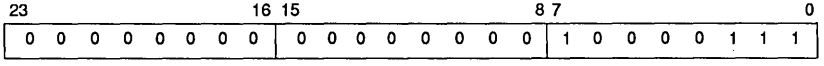


A

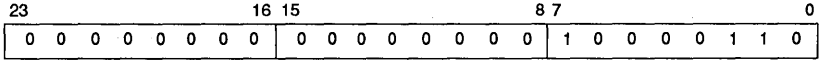
ENDDO



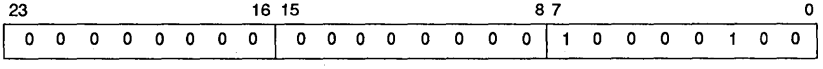
STOP



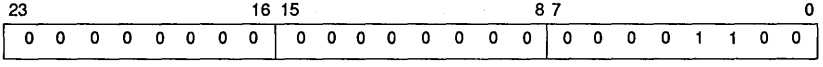
WAIT



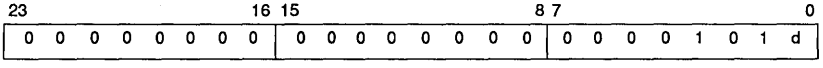
RESET



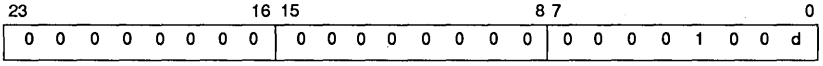
RTS



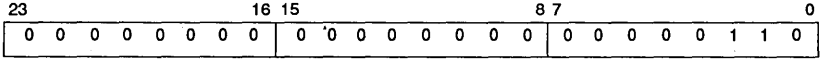
DEC



INC

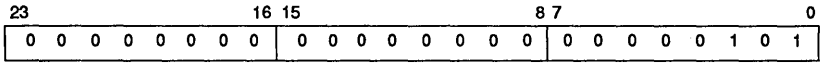


SWI

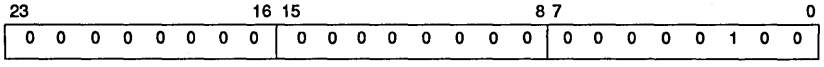


A

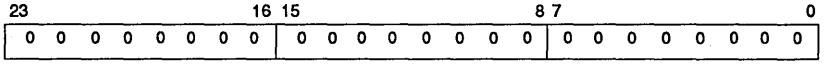
ILLEGAL



RTI



NOP



A.10.4 Parallel Instruction Encoding of the Operation Code

The operation code encoding for the instructions which allow parallel moves is divided into the multiply and nonmultiply instruction encodings shown in the following subsection.

Multiply Instruction Encoding

The 8-bit operation code for multiply instructions allowing parallel moves has different fields than the nonmultiply instruction's operation code.

The 8-bit operation code=**1QQQ dkkk** where QQQ=selects the inputs to the multiplier
kkk = three unencoded bits k2, k1, k0
d = destination accumulator
d = 0 → A
d = 1 → B

Table A-27 Operation Code K0-2 Decode

Code	k2	k1	k0
0	positive	mpy only	don't round
1	negative	mpy and acc	round

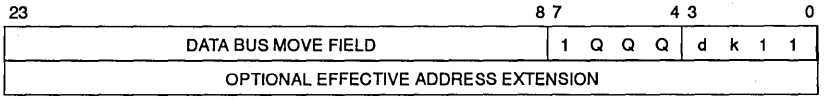
A

Table A-28 Operation Code QQQ Decode

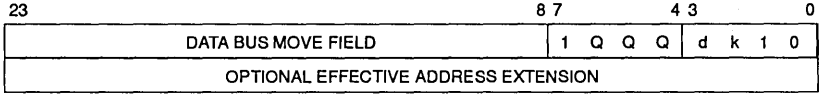
Q	Q	Q	S1	S2
0	0	0	X0	X0
0	0	1	Y0	Y0
0	1	0	X1	X0
0	1	1	Y1	Y0
1	0	0	X0	Y1
1	0	1	Y0	X0
1	1	0	X1	Y0
1	1	1	Y1	X1

NOTE: S1 and S2 are the inputs to the multiplier.

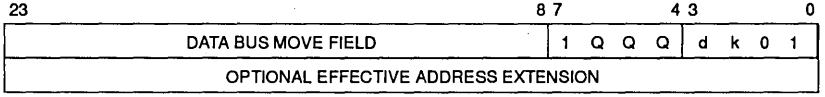
MACR (-)S1,S2,D
MACR (-)S2,S1,D



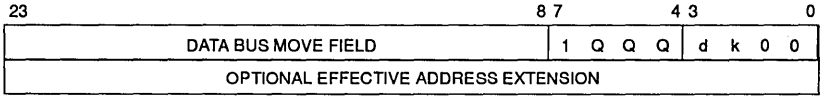
MAC (-)S1,S2,D
MAC (-)S2,S1,D



MPYR (-)S1,S2,D
MPYR (-)S2,S1,D



MPY (-)S1,S2,D
MPY (-)S2,S1,D



A

Nonmultiply Instruction Encoding

The 8-bit operation code for instructions allowing parallel moves contains two 3-bit fields defining which instruction the operation code represents and one bit defining the destination accumulator register.

The 8-bit operation code = **0JJJ Dkkk** where JJJ=1/2 instruction number
kkk=1/2 instruction number
D=0 → A
D=1 → B

Table A-29 Nonmultiply Instruction Encoding

JJJ	D = 0 Src Oper	D = 1 Src Oper	kkk							
			000	001	010	011	100	101	110	111
000	B	A	MOVE[1]	TFR	ADDR	TST	*	CMP	SUBR	CMPM
001	B	A	ADD	RND	ADDL	CLR	SUB	*	SUBL	NOT
010^2	B	A	—	—	ASR	LSR	—	—	ABS	ROR
011^2	B	A	—	—	ASL	LSL	—	—	NEG	ROL
010^2	X1X0	X1X0	ADD	ADC	—	—	SUB	SBC		
011^2	Y1Y0	Y1Y0	ADD	ADC	—	—	SUB	SBC		
100	X0_0	X0_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
101	Y0_0	Y0_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
110	X1_0	X1_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM
111	Y1_0	Y1_0	ADD	TFR	OR	EOR	SUB	CMP	AND	CMPM

NOTES:
* = Reserved
1 = Special Case #1 (See Table A-30)
2 = Special Case #2 (See Table A-31)

A

Table A-30 Special Case #1

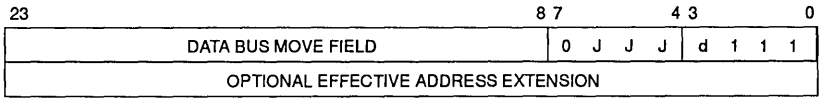
O P E R C O D E	Operation
0 0 0 0 0 0 0 0	MOVE
0 0 0 0 1 0 0 0	Reserved

For JJJ=010 and 011, k1 qualifies source register selection:

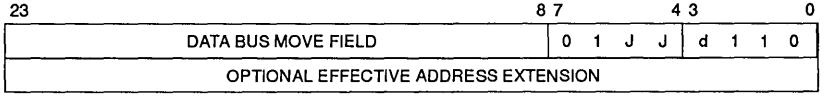
Table A-31 Special Case #2

0 J J J d k k k	Operation
0 0 1 0 x x 0 x	Selects X1X0
0 0 1 1 x x 0 x	Selects Y1Y0
0 0 1 x x x 1 x	Selects A/B

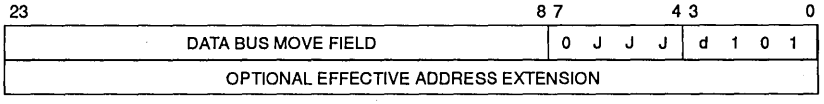
CMPM S1,S2



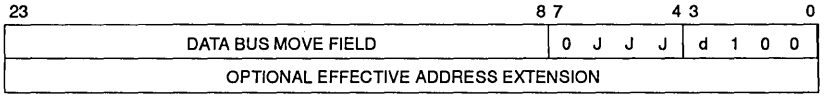
AND S,D



CMP S1,S2

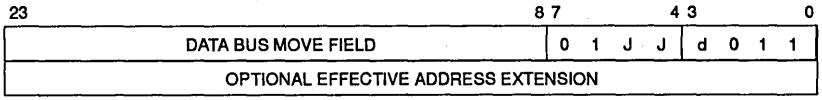


SUB S,D

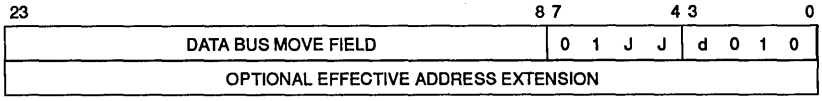


A

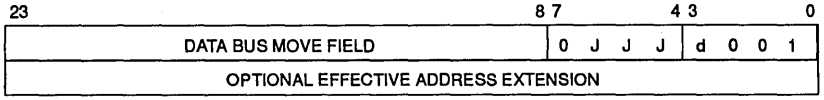
EOR S,D



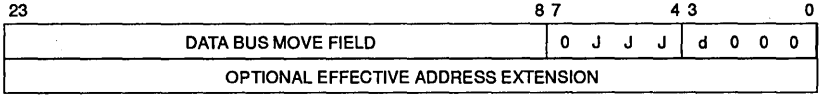
OR S,D



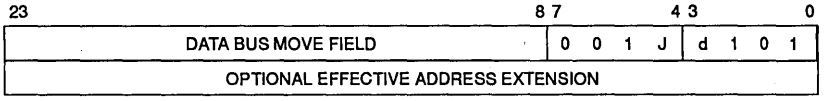
TFR S,D



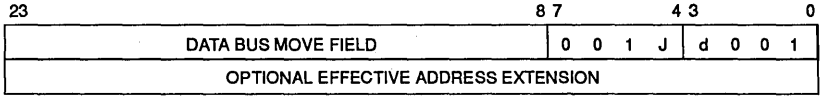
ADD S,D



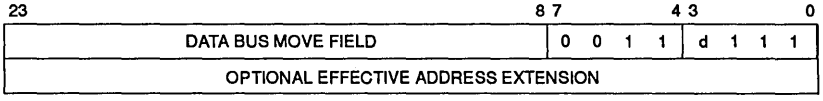
SBC S,D



ADC S,D

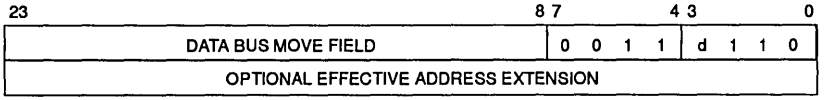


ROL D

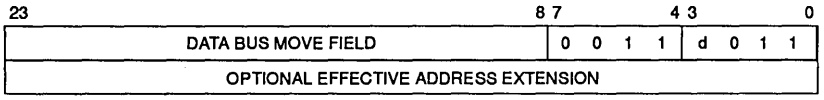


A

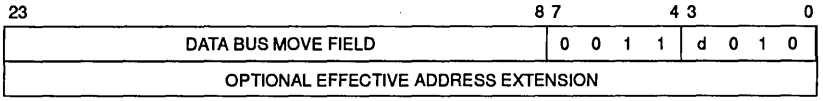
NEG D



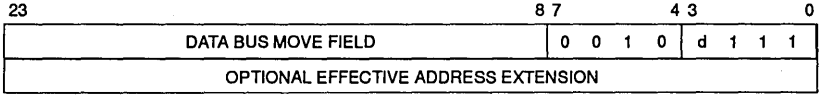
LSL D



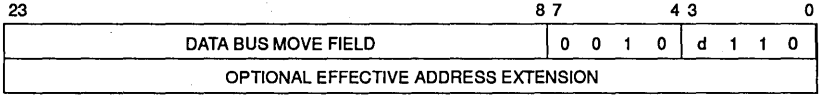
ASL D



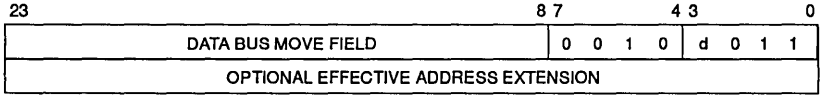
ROR D



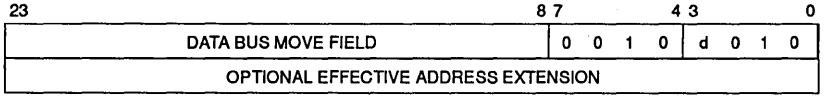
ABS D



LSR D

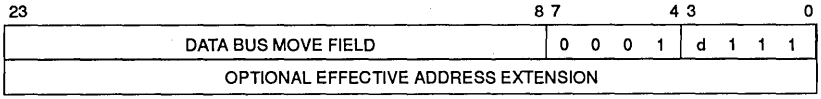


ASR D

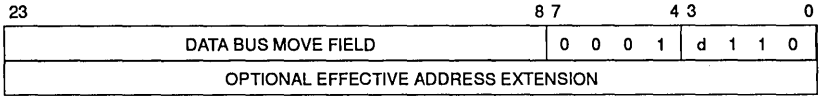


A

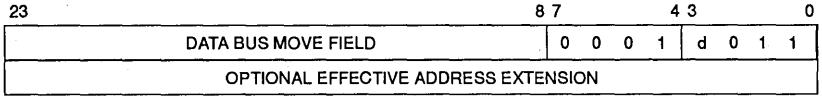
NOT D



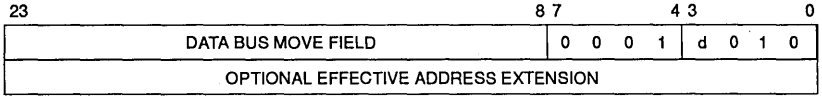
SUBL S,D



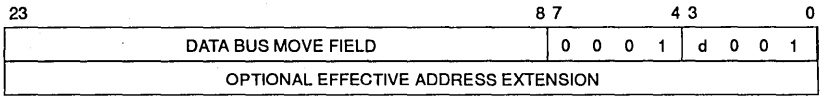
CLR D



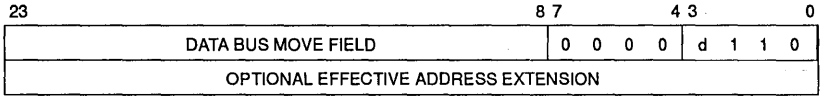
ADDL S,D



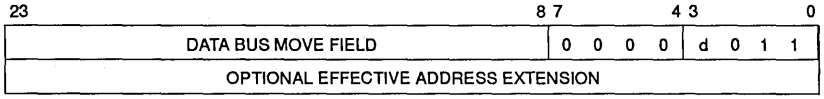
RND D



SUBR S,D

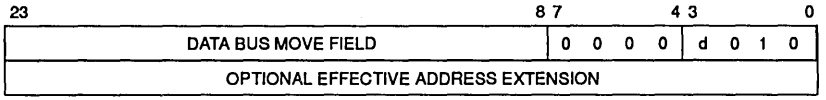


TST D

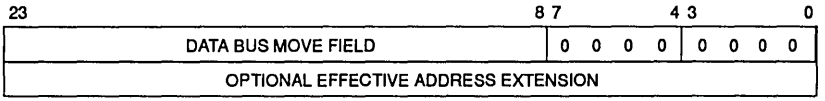


A

ADDR S,D

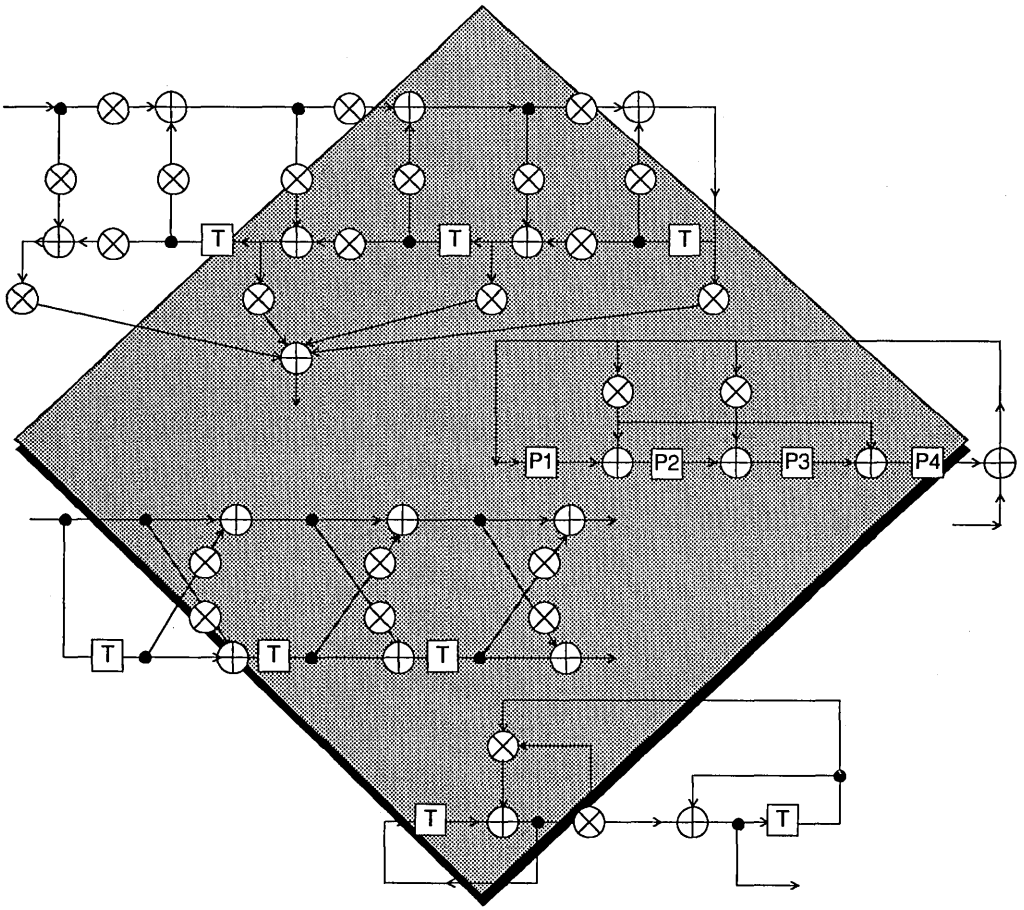


MOVE S,D



A

APPENDIX B
BENCHMARK PROGRAMS



B

SECTION CONTENTS

SECTION B.1 INTRODUCTION 3

SECTION B.2 BENCHMARK PROGRAMS 3

B

B.1 INTRODUCTION

Table B-1 provides benchmark numbers for 18 common DSP programs implemented on the 27-MHz DSP56001.

The four code examples shown in Figures B-1 to B-4 represent the benchmark programs shown in Table B-1.

B.2 BENCHMARK PROGRAMS

Figure B-1 is the code for the 20-tap FIR filter shown in Table B-1. Figure B-2 is the code for an FFT using a triple nested DO LOOP. Although this code is easier to understand and very compact, it is not as fast as the code used for the benchmarks shown in Table B-1, which are highly optimized using the symmetry of the FFT and the parallelism of the DSP. Figure B-3 is the code for the 8-pole cascaded canonic biquad IIR filter, which uses four coefficients (see Table B-1). Figure B-4 is the code for a 2N delayed least mean square (LMS) FIR adaptive filter, which is useful for echo cancelation and other adaptive filtering applications.The code example shown in Figure B-5 represents the Real FFT code for the DSP56002, based on the Glenn Bergland algorithm.

The code for these and other programs is free and available through the Dr. BuB electronic bulletin board.

Table B-1 27-MHz Benchmark Results for the DSP56001R27

Benchmark Program	Sample Rate (Hz) or Execution Time	Memory Size (Words)	Number of Clock Cycles
20 - Tap FIR Filter	500.0 kHz	50	54
64 - Tap FIR Filter	190.1 kHz	138	142
67 - Tap FIR Filter	182.4 kHz	144	148
8 - Pole Cascaded Canonic Biquad IIR Filter (4x)	540.0 kHz	40	50
8 - Pole Cascaded Canonic Biquad IIR Filter (5x)	465.5 kHz	45	58
8 - Pole Cascaded Transpose Biquad IIR Filter	385.7 kHz	48	70
Dot Product	444.4 ns	10	12
Matrix Multiply 2x2 times 2x2	1.556 µs	33	42
Matrix Multiply 3x3 times 3x1	1.259 µs	29	34
M-to-M FFT 64 Point	98.33 µs	489	2655
M-to-M FFT 256 Point	489.8 µs	1641	13255
M-to-M FFT 1024 Point	2.453 ms	6793	66240
P-to-M FFT 64 Point	92.56 µs	704	2499
P-to-M FFT 256 Point	347.9 µs	2048	9394
P-to-M FFT 1024 Point	1.489 ms	7424	40144

B

```
    page 132,66,0,6
    opt   rc
;*****************************************************
;Motorola Austin DSP Operation     June 30, 1988
;*****************************************************
;
;DSP56000/1
;20 - tap FIR filter
;File name: 1-56.asm
;*****************************************************************************************************************
;
;   Maximum sample rate: 379.6 kHz at 20.5 MHz/500.0 kHz at 27.0 MHz
;   Memory Size: Prog: 4+6 words; Data: 2x20 words
;   Number of clock cycles: 54 (27 instruction cycles)
;   Clock Frequency: 20.5 MHz/27.0 MHz
;   Instruction cycle time: 97.6 ns/74.1 ns
;*****************************************************************************************************************
;
;   This FIR filter reads the input sample
;   from the memory location Y:input
;   and writes the filtered output sample
;   to the memory location Y:output
;
;   The samples are stored in the X memory
;   The coefficients are stored in the Y memory
;*****************************************************************************************************************
```

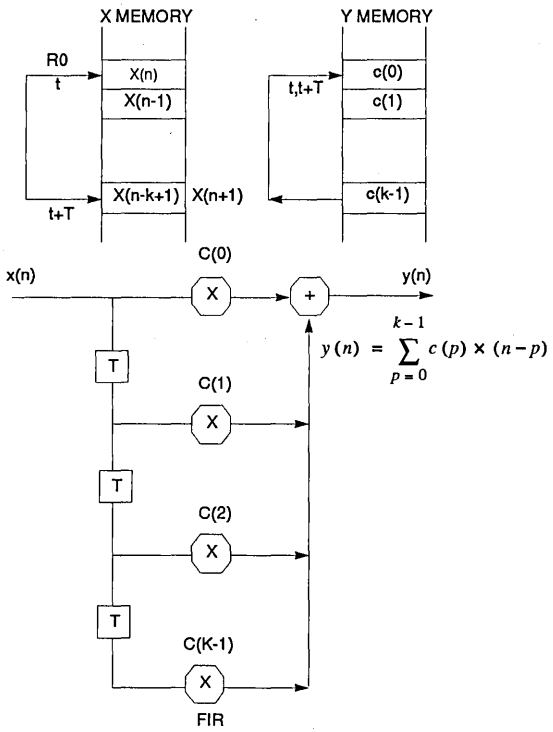


Figure B-1 20-Tap FIR Filter Example (Sheet 1 of 2)

B

```
;
;
;
;*****************************************************************************************************
;
;               initialization
;*****************************************
n               equ             20
start           equ             $40
wddr            equ             $0
cddr            equ             $0
input           equ             $ffe0
output          equ             $ffe1
;
                org             p:start
                move            #wddr,r0                        ;r0 ➡ samples
                move            #cddr,r4                        ;r1 ➡ coefficients
                move            #n-1,m0                         ;set modulo arithmetic
                move            m0,m4                           ;for the 2 circular buffers
;
                opt             cc
;               filter loop :8+(n-1) cycles
;*****************************************************************************************************
                movep           y:input,x: (r0)                 ;input sample in memory
                clr             a               x:(r0)+,x0      y: (r4)+,y0

                rep             #n-1
                mac             x0,y0,a         x:(r0)+,x0      y:(r4)+,y0
                macr            x0,x0,a         (r0)-

                movep           a,y:output                      ;output filtered sample
;*****************************************************************************************************
            end
```

B

Figure B-1 20-Tap FIR Filter Example (Sheet 2 of 2)

```
;This program originally available on the Motorola DSP bulletin board.
;It is provided under a DISCLAIMER OF WARRANTY available from
;Motorola DSP Operation, 6501 William Cannon Drive, Austin, TX, 78735
;
;Radix-2, In-Place, Decimation-In-Time FFT (smallest code size).
;
;Last Update 30 Sep 86        Version 1.1
;
fftr2a      macro       points,data,coef
fftr2a      ident       1,1
;
;Radix-2 Decimation-In-Time In-Place FFT Routine
;
;    Complex input and output data
;        Real data in X memory
;        Imaginary data in Y memory
;     Normally ordered input data
;     Bit reversed output data
;        Coefficient lookup table
;        -Cosine values in X memory
;        -Sine values in Y memory
;
;Macro Call — ffr2a      points,data,coef
;
;           points                  number of points (2-32768, power of 2)
;           data                    start of data buffer
;           coef                    start of sine/cosine table
;
;Alters Data ALU Registers
;           x1          x0          y1          y0
;           a2          a1          a0          a
;           b2          b1          b0          b
;
;Alters Address Registers
;           r0          n0          m0
;           r1          n1          m1
;                       n2
;
;           r4          n4          m4
;           r5          n5          m5
;           r6          n6          m6
;
'Alters Program Control Registers
;           pc          sr
;
;Uses 6 locations on System Stack
;
```

Figure B-2 Radix 2, In-Place, Decimation-In-Time FFT (Sheet 1 of 2)

B

```
;Latest Revision — September 30, 1986
;
        move    #points/2,n0        ;initialize butterflies per group
        move    #1,n2               ;initialize groups per pass
        move    #points/4,n6        ;initialize C pointer offset
        move    #-1,m0              ;initialize A and B address modifiers
        move    m0,m1               ;for linear addressing
        move    m0,m4
        move    m0,m5
        move    #0,m6               ;initialize C address modifier for
                                    ;reverse carry (bit-reversed) addressing
;
;Perform all FFT passes with triple nested DO loop
;
        do      #@cvi (@log(points)/@log(2)+0.5),_end_pass
        move    #data,r0            ;initialize A input pointer
        move    r0,r4               ;initialize A output pointer
        lua     (r0)+n0,r1          ;initialize B input pointer
        move    #coef,r6            ;initialize C input pointer
        lua     (r1)-,r5            ;initialize B output pointer
        move    n0,n1               ;initialize pointer offsets
        move    n0,n4
        move    n0,n5

        do      n2,_end_grp
        move    x:(r1),X1   y:(r6),y0       ;lookup -sine and
                                            ; -cosine values
        move    x:(r5),a    y:(r0),b        ;preload data
        move    x:(r6)+n6,x0                ;update C pointer

        do      n0,_end_bfy
        mac     x1,y0,b     y:(r1)+,y1      ;Radx 2 DIT
                                            ;butterfly kernel
        macr    -x0,y1,b    a,x:(r5)+   y:(r0),a
        subl    b,a         x:(r0),b    b,y:(r4)
        mac     -x1,x0,b    x:(r0)+,a   a,y:(r5)
        macr    -y1,y0,b    x:(r1),x1
        subl    b,a         b,x:(r4)+   y:(r0),b
_end_bfy
        move    a,x:(r5)+n5     y:(r1)+n1,y1    ;update A and B pointers
        move    x:(r0)+n0,x1    y:(r4)+n4,y1
_end_grp
        move    n0,b1               ;divide butterflies per group by two
        lsr     b   n2,a1           ;multiply groups per pass by two
        lsl     a   b1,n0
        move    a1,n2
_end_pass
        endm
```

Figure B-2 Radix 2, In-Place, Decimation-In-Time FFT (Sheet 2 of 2)

```
    page 132,66,0,6
    opt   rc
;*********************************************************
;Motorola Austin DSP Operation       June 30, 1988
;*********************************************************

;DSP56000/1
;8-pole 4-multiply cascaded canonic IIR filter
;File name: 4-56.asm
;*********************************************************************************************************************
;   Maximum sample rate: 410.0 kHz at 20.5 MHz/540.0 kHz at 27.0 MHz
;   Memory Size: Prog: 6+10 words; Data: 4(2+4) words
;   Number of clock cycles: 50 (25 instruction cycles)
;   Clock Frequency: 20.5 MHz/27.0 MHz
;   Instruction cycle time: 97.5 ns/74.1 ns
;*********************************************************************************************************************
;
;   This IIR filter reads the input sample
;   from the memory location Y:input
;   and writes the filtered output sample
;   to the memory location Y:output
;
;   The samples are stored in the X memory
;   The coefficients are stored in the Y memory
;
;
;   The equations of the filter are:
;       w(n)=       x(n)-ai1*w(n-1)-ai2*w(n-2)
;       y(n)=       w(n)+bi1*w(n-1)+bi2*w(n-2)
```

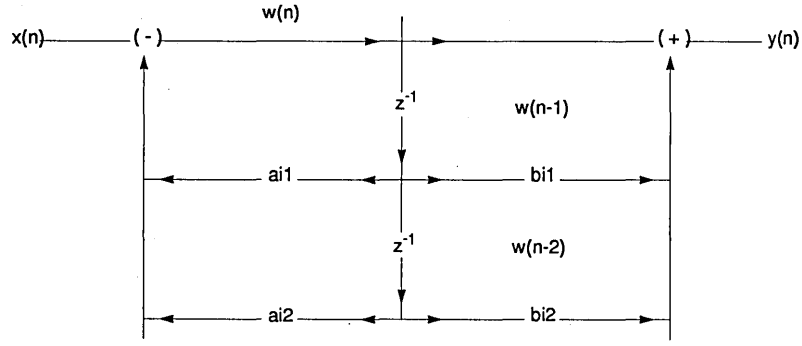


Figure B-3 8-Pole 4-Multiply Cascaded Canonic IIR Filter (Sheet 1 of 2)

B

```
;   All coefficients are divided by 2:
;   w(n)/2=x(n)/2-ai1/2*w(n-1)-ai2/2*w(n-2)
;   y(n)/2=w(n)/2+bi1/2*w(n-1)+bi2/2*w(n-2)
;
;       X Memory Organization                   Y Memory Organization
;                |              |                       | b1N/2  |  Coef. + 4*nsec - 1
;                |              |                       | b2N/2  |
;                |              |                       | a1N/2  |
;                |              |                       | a2N/2  |
;                | wN(n-1)      |  Data + 2*nsec - 1    |   •    |
;                | wN(n-2)      |                       |   •    |
;                |   •          |                       | b11/2  |
;                |   •          |                       | b21/2  |
;                | w1(n-1)      |                       | a11/2  |
;     R0 ➡       | w1(n-2)      |  Data          R4 ➡   | a21/2  |  Coef.
;
;*****************************************************************************************************
;   initialization
;*********************************
nsec        equ         4
start       equ         $40
data        equ         0
coef        equ         0
input       equ         $ffe0
output      equ         $ffe1
igain       equ         0.5
            ori         #$08,mr             ;set scaling mode
            move        #data,r0            ;point to filter states
            move        #coef,r4            ;point to filter coefficients
            move        #2*nsec - 1,m0
            move        #4*nsec - 1,m4
            move        #igain,y1           ;y1=initial gain

            opt         cc
;                   filter loop: 4*nsec + 9
;*********************************************************************

            movep         y:input,y0                          ;get sample
            mpy         y0,y1,a     x:(r0) +,x0     y:(r4)+,y0  ;x0=1st section w(n-2),y0=ai2/2
;
            do            #nsec,end_cell                      ;do each section
            mac         -x0,y0,a    x:(r0) -,x1     y:(r4) +,y0 ;x1=w(n-1),y0=ai1/2
            macr        -x1,y0,a    x1,x:(r0) +     y:(r4) +,y0 ;push w(n-1) to w(n-2),y0=bi2/2
            mac         x0,y0,a     a,x:(r0)+       y:(r4) +,y0 ;push w(n) to w(n-1),y0=bi1/2
            mac         x1,y0,a     x:(r0) +,x0     y:(r4) +,y0 ;next iter:x0=w(n-2),y0=ai2/2
end_cell
            rnd         a                                       ;round result
            movep       a,y:output                              ;output sample
;*****************************************************************************
            end
```

Figure B-3 8-Pole 4-Multiply Cascaded Canonic IIR Filter (Sheet 2 of 2)

```
  page 132,60,1,1
;newlms2n.asm
;    New Implementation of the delayed LMS on the DSP56000 Revision C
;Memory map:
;    Initial X                                          H
;    x(n)    x(n-1)    x(n-2)    x(n-3)    x(n-4)       hx    h0    h1    h2    h3
;    ]                                                  ]     ]
;    r0                                                 r5    r4
;hx is an unused value to make the calculations faster.
;
  opt     cc
ntaps     equ     4
input     equ     $FFC0
output    equ     $FFC1
  org     x:$0
state     ds      5
  org     y:$0
coef      ds      5
;
  org     p:$40
  move    #state,r0               ;start of X
  move    #2,n0
  move    #ntaps,m0               ;mod 5
  move    #coef +1,r4             ;coefficients
  move    #ntaps,m4               ;mod 5
  move    #coef,r5                ;coefficients
  move    m4,m5                   ;mod 5
_smploop                          ;                                  Prog    Icyc
  movep                 y:input,a ;get input sample                  word
  move          a,x:(r0)          ;save input sample                   1       1
;error signal is in y1
;FIR sum in a=a+h(k) old*x(n-k)
;h(k)new in b=h(k)old + error*x(n-k-1)

  cir     a       x:(r0)+,x0      ;x0=x(n)                             1       1
  move            x:(r0)+,x1  y:(r4)+,y0  ;x1=x(n-1),y0=h(0)           1       1
  do      #taps/2,_lms            ;                                    2       3
  mac     x0,y0,a     y0,b        b,y:(r5)+   ;a=h(0)*x(n),b=h(0)      1       1
  macr    x1,y1,b     x:(r0)+,x0  y:(r4)+,y0  ;b=h(0)+e*x(n-1)=h(0)new 1       1
                                              ;x0=x(n-2) y0=h(1)
  mac     x1,y0,a     y0,b        b,y:(r5)+   ;a=a+h(1)*x(n-1) b=h(1)  1       1
  macr    x0,y1,b     x:(r0)+,x1  y:(r4)+,y0  ;b=h(1)+e*x(n-2)         1       1
;                                             ;x1=x(n-3) y0=H(2)
_lms
  move                        b,y:(r5)+   ;save last new c( )          1       1
  move          (r0) -n0                  ;pointer update              1       1

;(Get d(n), subtract fir output (reg a), multiply by "u", put
;the result in y1. This section is application dependent.)
  movep         a,y:output                ;output fir if desired
  jmp           _smploop
  end                                     ;
;                                                   Totals:  11    2N+8
```

Figure B-4 LMS FIR Adaptive Filter

```
Real input FFT based on Glenn Bergland algorithm
;
; Normal order input and normal order output.
;
; Since 56001 does not support bergland addressing, extra instruction cycles are needed
; for converting bergland order to normal order. It has been done in the last pass by
; looking at the bergtable.
; The micro 'bergsincos' generates SIN and COS table with size of points/4, COS in Y, SIN in X
; The micro 'bergorder' generates table for address conversion, the size of twiddle factors is half
; of FFT output's.
; The micro 'norm2berg' converts normal order data to bergland order.
; The micro 'rfft-56b' does FFT.
;
; Real input data are split into two parts, the first part is put in X, the second in Y.
; Real output data are in X, imaginary output data are in Y.
; The bergland table for converting berglang order to normal order is stored in output buffer.
; In the last pass the FFT output overwrites this table.
; The first real output plus the first imaginary output is DC value of the spectrum.
; Note that only DC to Nyquist frequency range is calculated by this algorithm.
; After twiddle factors and bergtable are generated, you may overwrite 'bergorder',
; 'norm2berg' by 'rfft-56b' for saving P memory.
;
;                          Performance
;-----------------------------------------------------------
;  Real input data points                    Clock cycle
;              64                              1686
;             128                              3846
;             256                              8656
;             512                             19296
;            1024                             49776
;-----------------------------------------------------------
;
;                          Memory (word)
;-----------------------------------------------------------
;        P memory        X memory                  Y memory
;            87        points/2 (real input)  +  points/2 (imaginary input)
;                      points/4 (SIN table)   +  points/4 (COS table)
;                      points/2 (real output) +  points/2 (imaginary output)
;                      points/2 (bergtable)
;-----------------------------------------------------------
;
;
rfft56bt          ident  1,3
                  page   132,60
                  opt    nomd,nomex,loc,nocex,mu
                  include 'bergsincos'
                  include 'bergorder'
                  include 'norm2berg'
                  include 'rfft-56b'
```

B

Figure B-5 Real Input FFT Based on Glenn Bergland Algorithm (Sheet 1 of 8)

```
;
; Main program to call the rfft-56b macro
;     Argument list
;
; Latest modifying date - 4-March-92

reset               equ  0
start               equ  $40
points              equ  512
binlogsz            equ  9
idata               equ  $000
odata               equ  $400
bergtable           equ  $600
twiddle             equ  $800

bergsincos          points,odata ;generate normal order twiddle factors with size of points/4

                    opt     mex
                    org     p:reset
                    jmp     start

                    org     p:start
                    movep #0,x:$fffe            ;0 wait states
bergorder           points/4,bergtable,odata    ;generates bergland table for twiddle factor
norm2berg           points/4,bergtable,twiddle  ;converting twiddle factor from normal order to bergland
order
bergorder           points/2,bergtable,odata    ;table for final output
rifft               points,binlogsz,idata,odata,twiddle,bergtable
                    end



;
bergsincos          macro  points,coef
bergsincos          ident  1,2
;
;    sincos -    macro to generate sine and cosine coefficient
;                lookup tables for Decimation in Time FFT
;                twiddle factors.
;
;    points -    number of points (2 - 32768, power of 2)
;    coef   -    base address of sine/cosine table
;                negative cosine value in X memory
;                negative sine value in Y memory
;
;

pi                  equ    3.141592654
freq                equ    2.0*pi/@cvf(points)
```

Figure B-5 Real Input FFT Based on Glenn Bergland Algorithm (Sheet 2 of 8)

B

```
                org   y:coef
count           set   0
                dup   points/4
                dc    @cos(@cvf(count)*freq)
ount            set   count+1
                endm

                org   x:coef
count           set   0
                dup   points/4
                dc    @sin(@cvf(count)*freq)
count           set   count+1
                endm

                endm  ;end of bergsincos macro



bergorder  macro  points,bergtable,offset
bergorder  ident  1,3
;bergorder generates bergland order table

                move  #>4,a
                move  #points,r4              ;points=number of points of bergtable to be
generated
                move  #>points/4,b            ;nitial pointer
                move  #bergtable,r0           ;table resides in
                move  b,n0                    ;init offset
                move  #>0,x0
                move  x0,x:(r0)+n0            ;seeds
                move  #>2,x0
                move  x0,x:(r0)+n0
                move  #>1,x0
                move  x0,x:(r0)+n0
                move  #>3,x0
                move  x0,x:(r0)
                move  #bergtable,n0           ;location of bergtable
                do    #@cvi(@log(points/4)/@log(2)),_endl
                move  b,x0                    ;x0=i+i
                lsr   b                       ;b=i
                move  b,r0                    ;r0=i
                nop
                move  a,x:(r0+n0)             ;k-> bergtable
                lsl   a                        ;k=k*2
                move  a,y1                    ;save A content
_star           move  r4,a                    ;r4=# of points
                cmp   x0,a                     ;x0=j, if j< points, cont
```

Figure B-5 Real Input FFT Based on Glenn Bergland Algorithm (Sheet 3 of 8)

```
                jle     _loop
                move    x0,r0                   ;r0=i+i=j,b=i
                move    y1,a                    ;recover A=k
                move    x:(r0+n0),y0            ;y0=bergtabl[j]
                sub     y0,a                    ;k-bergtabl[j]
                move    b,x1                    ;save b, x1=i
                move    r0,y0                   ;y0=j=i+i
                add     y0,b                    ;b=j+i
                move    b,r0                    ;r0=j+i
                nop
                move    a,x:(r0+n0)             ;store bergtabl[j+i]
                add     x1,b                    ;b=j+i+i
                move    b,x0                    ;save b
                move    x1,b                    ;recover b=i
                jmp     _star
_loop           move    y1,a                    ;recover a
_endl
                move    #>offset,a              ;offset is the location of output data or twiddle
                move    #bergtable,r0
                do #points,_add_offset
                move    x:(r0),B
                add     A,B
                move    B,x:(r0)+
_add_offset
                endm    ;end of sincos macro


;convert normal order to berglang order
norm2berg       macro   points,bergtable,twiddle
;points is actual size of table to be converting
                move    #bergtable,r0           ;r0=pointer of bergland table
                move    #twiddle,r2             ;r2=twiddle pointer for X
                move    r2,r6                   ;r6=twiddle pointer for Y
                do      #points,data_temp
                move    x:(r0)+,r3              ;get index
                move    r3,r7
                move    x:(r3),a
                move    y:(r7),b                ;get value
                move    a,x:(r2)+ b,y:(r6)+     ;write back
data_temp
                endm
```

Figure B-5 Real Input FFT Based on Glenn Bergland Algorithm (Sheet 4 of 8)

B

```
; Real-Valued FFT for MOTOROLA DSP56000/1/2
;
;   based on Glenn Bergland's algorithm
;
; ____________________________________
rifft macro points,binlogsz,idata,odata,twiddle,bergtable

                    move   #idata,r0                    ;r0 = ptr to a
                    move   #points/4,n0                 ;bflys in ea group, half at ea pass
                    move   #twiddle+1,r7                ;r7 always points to start location of twiddle
                    lua    (r0)+n0,r1                   ;r1 = ptr to b
                    move   r0,r4                        ;r4 points to c
                    move   r1,r5                        ;r5 points to d,with predecrement
                    move   #1,r3                        ;group per pass, double at ea pass
                    move   x:(r0),A      y:(r4),y0      ;A=a, y0=c

                    do     n0,pass1                     ;first pass is trivial, no multiplications
; ------------------------------------------------
;   First Pass -- W(n) = 1
;
;   A---\    /---A'= Re[ A + jB + (C + jD) ] =  A + C
;   B----\_|_/----B'= Im[ A + jB + (C + jD) ] =j(D + B)
;   C----/ | \----C'= Re[ A + jB - (C + jD) ] =  A - C
;   D---/    \---D'= Im[-A - jB + (C + jD) ] =j(D - B)
;------------------------------------------------
;
                    sub    y0,A  x:(r1),x0 y:(r5),B     ;A=a-c=c',B=d,x0=b,
                    add    x0,B  A,x:(r1)+ y:(r5),A     ;B=d+b=b', A=d,PUT c' to x:b
                    sub    x0,A  x:(r0)+,B B,y:(r4)+    ;A=d-b=d',B=a,PUT b' to y:c
                    add    y0,B  x:(r0)-, A A,y:(r5)+   ;B=a+c=a', A=next a,PUT d'
                    move   B,x:(r0)+ y:(r4),y0          ;y0=next c, PUT a'
pass1


                    move   #idata,r0                    ;r0 = ptr to a

                    do     #binlogsz-3,end_pass         ;do all passes except first and last
                    move   r7,r2                        ;r2 points to real twiddle
                    move   r2,r6                        ;r6 points to imag twiddle
                    move   n0,A                         ;half bflys per group
                    lsr    A r3,B                       ;double group per pass
                    lsl    A,n0
                    move   B,r3                         ;r3 is temp reg.
                    lua    (r0)+n0,r1                   ;r1 = ptr to b
                    move   r0,r4                        ;r4 points to c
                    move   r1,r5                        ;r5 points to d
                    lua    (r3)-,n2                     ;n2=group per pass -1
                    move   x:(r0),A      y:(r4),y0      ;A=a, y0=c
```

B

Figure B-5 Real Input FFT Based on Glenn Bergland Algorithm (Sheet 5 of 8)

```
                do   n0,FirstGroupInPass          ;first group in a pass
                sub     y0,A  x:(r1),x0  y:(r5),B ;A=a-c=c',B=d,x0=b,
                add     x0,B  A,x:(r1)+ y:(r5),A  ;B=d+b=b', A=d,PUT c' to x:b
                sub     x0,A  x:(r0)+,B B,y:(r4)+ ;A=d-b=d',B=a,PUT b' to y:c
                add     y0,B  x:(r0)-,A A,y:(r5)+ ;B=a+c=a', A=next a,PUT d'
                move    B,x:(r0)+  y:(r4),y0      ;y0=next c, PUT a'
FirstGroupInPass

                do      n2,end_group              ;rest  groups in this pass
                move    r5,r0                     ;r0 ptr to next group a
                move    r0,r4                     ;r4 ptr to next group c
                lua     (r0)+n0,r1                ;r1 ptr to next group b
                move    r1,r5                     ;r5 ptr to next group d

;   Intermediate Passes -- W(n) < 1
;
;   A---\    /---A'=  Re[ A + jC + (B - jD)W(k) ] = A+BWr+DWi=A+T1
;   B----\_|_/----B'=  Im[ A + jC - (B - jD)W(k) ] = C+DWr-BWi=T2+C
;   C----/ | \----C'=  Re[ A + jC - (B - jD)W(k) ] = A-(BWr+DWi)=A-T1
;   D---/    \---D'=  Im[-A - jC - (B - jD)W(k) ] = -C+DWr-BWi=T2-C
;   ____________________________________________
                move    x:(r2)+,x0   y:(r6)+,y0   ;x0=Wi, y0=Wr
                move    x:(r1)-,x1   y:(r5),y1    ;x1=b,y1=d
                move    x:(r1),B                  ;for pointer reason

                do      n0,end_bfly               ;n0 bfly in this group
                mpy     -x1,x0,B  B,x:(r1)        ;B=-bWi, PUT c' to x:b
                mac     y0,y1,B y:(r4),A          ;B=dWr-bWi=T2, A=c
                sub     A,B                       ;B=T2-c=d'
                addl    B,A x:(r1)+,B   B,x:(r5)+     ;A=T2+c=b', PUT d'
                mpy     -x1,y0,B x:(r0),A A,y:(r4)+   ;B=-bWr,  A=a, PUT b' to y:c
                mac     -x0,y1,B x:(r1)-,x1       ;B=-bWr-dWi=-T1, x1=next b
                sub     B,A                       ;A=a+T1=a'
                addl    A,B A,x:(r0)+  y:(r5),y1  ;B=a-T1=c', y1=next d, PUT a'
end_bfly
                move    B,x:(r1)+                 ;PUT last b'
end_group
                move    #idata,r0                 ;r0 = ptr to a
end_pass

;the last pass converts bergland order to normal order by calling bergtable
                move    r7,r2                     ;r2 points to real twiddle
                move    r2,r6                     ;r6 points to imag twiddle
                move    r0,r4                     r4 points to c
                move    #bergtable,r3             ;r3=pointer of bergland table
                move    #(points/4)-1,n2          ;n2=group per pass -1
                move    x:(r3)+,r7                ;get first index
                move    x:(r3)+,r1                ;get second index
                move    #2,n4
```

B

Figure B-5 Real Input FFT Based on Glenn Bergland Algorithm (Sheet 6 of 8)

```
   ; first group in the last pass
                    move   x:(r0)+,A   y:(r4)+,B          ;A=a, B=c
                    sub    B,A   x:(r0)+,x0  y:(r6)+,y0   ;A=a-c=c',x0=b, y0=Wr for next bfly
                    addl   A,B   A,x:(r1) y:(r4),A        ;B=a+c=a', A=d,PUT c' to x:b
                    sub    x0,A  B,x:(r7)                 ;A=d-b=d',PUT a' to x
                    move   y:(r4)+,B                      ;B=d
                    add    x0,B  A,y:(r1)                 ;B=d+b=b', A=next a,PUT d'
                    move   x:(r0)+,A B,y:(r7)             ;A=next a, PUT b'

                    move   x:(r2)+,x0 y:(r4)+,B           ;x0=Wi,B=next c

                    do     n2,end_lastg                   ;rest  groups in the last pass

;   Intermediate Passes -- W(n) < 1
;
;   A---\    /---A'=  Re[ A + jC + (B - jD)W(k) ] = A+BWr+DWi=A+T1
;   B----\_|_/----B'=  Im[ A + jC - (B - jD)W(k) ] = C+DWr-BWi=T2+C
;   C----/ | \----C'=  Re[ A + jC - (B - jD)W(k) ] = A-(BWr+DWi)=A-T1
;   D---/    \---D'=  Im[-A - jC - (B - jD)W(k) ] = -C+DWr-BWi=T2-C
;
                    move   x:(r0)+,x1 y:(r4)-,y1          ;x1=b, y1=d, r4 ptr back to c
                    mpy    x1,y0,B x:(r3)+,r7             ;A=bWr,
                    mac    x0,y1,B  x:(r3)+,r1            ;B=bWr+dWi=T1, get first index
                    sub    B,A                            ;A=a-T1=c', get second index
                    addl   A,B     A,x:(r1)               ;B=a+T1=a', PUT c' to x:b
                    mpy    y1,y0,A B,x:(r7)               ;B=dWr, B=c PUT a'
                    mac    -x1,x0,A y:(r4)+n4,B           ;A=dWi-bWr=T2, B=c, r4 ptr to next c
                    sub    B,A   x:(r2)+,x0 y:(r6)+,y0    ;A=T2-c=d',x0=next Wi, y0=next Wr
                    addl   A,B   A,y:(r1)                 ;B=T2+c=b', update r4, A=next a, PUT d'
                    move   x:(r0)+,A  B,y:(r7)            ;PUT b', A=next a
                    move   y:(r4)+,B                      ;B=next c
end_lastg
                    endm

; Real input FFT based on Glenn Bergland algorithm
;
; Normal order input and normal order output.
; Since 56001 does not support Bergland addressing, extra instruction cycles are needed
; for converting Bergland order to normal order.It has been done in the last pass by
; looking at the bergtable.
; 'bergsincos' generates sin and cos table with size of points/4,COS in Y, SIN in X
; 'bergorder' generates table for address conversion, the size of twiddle factors is half
; of FFT output's
; 'rfft-56b' does FFT
;
; Normal order input and normal order output.
```

B

Figure B-5 Real Input FFT Based on Glenn Bergland Algorithm (Sheet 7 of 8)

```
;
; Real input data are split into two parts, the first part is put in X, the second in Y.
; Real output data are in X, imaginary output data are in Y.
; The first real output is DC
; The first imaginary output is the Nyquist frequency.
; Note that only DC to Nyquist frequency range is calculated by this algorithm
; After twiddle factors and bergtable are generated, you may overwrite 'bergorder',
; 'norm2berg' by 'rfft-56b' for saving P memory.
;
;
Performance
;-----------------------------------------------------------
;  Real input data points                    Clock cycle
;              64                            1686
;              128                           3846
;              256                           8656
;              512                           19296
;              1024                          49776
;-----------------------------------------------------------
;
;
Memory (word)
;-----------------------------------------------------------
;              P memory                      X memory                  Y memory
;              87                            points/2+ (real input)    points/2+ (imaginary input )
;                                            points/4+ (SIN table)     points/4+ (COS table)
;                                            points/2+ (real output)   points/2 (imaginary output)
;                                            points/2  (bergtable)
;-----------------------------------------------------------
```

Figure B-5 Real Input FFT Based on Glenn Bergland Algorithm (Sheet 8 of 8)

B

—A—

A Accumulator 3-7
Aborted Instructions 7-25
ABS A-22
Absolute Address 6-14
Absolute Short 6-14
Accumulator 6-5
Accumulator Shifter 3-9
Accumulators, A and B 3-7
ADC A-24
ADD A-26
ADDL A-28
ADDR A-30
Address ALU 4-5
Address Bus Signals (A0-A15) 8-3, 8-5
Address Buses 2-3, 2-4
Address Generation Unit (see AGU) 4-3
Address Modifier Arithmetic Types 4-14
  linear modifier 4-16
  modulo modifier 4-18
  reverse-carry modifier 4-22
  summary 4-25
Address Operands 6-10
  table A-6
Address Register Files 4-7
  R, N, and M register restrictions A-310
Addressing Modes 4-3, 4-8, 6-12, A-10
  address register direct 6-13
  address register indirect 4-9
  operators table A-8
  register direct 6-13
  special 6-14
  timing summary A-304
AGU
  address ALU 4-5
  address output multiplexers 4-6
  address register 4-3, 4-7
  address register restrictions 7-10
  architecture 4-3
  modifier register 4-5, 4-8
  modifier register restrictions 7-10
  offset register 4-4, 4-7
  offset register restrictions 7-10
  register restrictions 7-10
  registers 6-7
  registers operands table A-5
AND A-32
ANDI A-34
Application Development System 11-6
Applications 1-7
Arithmetic Instructions 6-22
ASL A-36
ASR A-38
Assembler/Simulator 11-4
Assistance 11-16

—B—

B Accumulator 3-7
BCHG A-40
BCLR A-48
Benchmark Programs B-3
Binary Operators A-7
Bit Manipulation Instructions 6-24
Bit Reverse 4-22
Bit Weighing 3-12
BSET A-56
BTST A-64
Bus Control Signals 8-3, 8-5
Buses
  address 2-4
  data 2-3
  transfers between 2-5
Byte, length of 6-5

—C—

Carry Bit 5-10, A-18
C-Compiler Features 11-5
CCR 5-9
CKOUT 9-10
  considerations 9-13
  synch with EXTAL 9-14
CKP 9-10
CLGND 9-10
Clock Stabilization Delay 7-38
CLR A-70
CLVCC 9-10
CMP A-72
CMPM A-74
Condition Code Computations (table) A-19
Condition Code Register (CCR) 5-9, A-15
  carry (bit 0) 5-10, A-18
  extension (bit 5) 5-11, A-16
  limit (bit 6) 5-11, A-16
  negative (bit 3) 5-10, A-17
  overflow (bit 1) 5-10, A-17
  scaling (bit 7) 5-11, A-16
  symbols table A-8
  unnormalized (bit 4) 5-10, A-17

zero (bit 2) 5-10, A-17
Condition Codes A-3
Convergent Rounding 3-6

—D—

Data ALU
double precision multiply mode 3-16
MAC 3-13
MAC and logic unit 3-6
programming model 3-19
summary 3-19
Data ALU Accumulator Registers 3-7
Data ALU components 3-3
Data ALU Registers 3-3, 6-6
input registers 3-5
operands table A-5
Data Arithmetic Logic Unit (see Data ALU) . . .3-3
Data Bus Move Field 6-5
Data Bus Signals (D0-D15) 8-3, 8-5
Data Buses 2-3
Data Conversion 3-11
Data Organization 6-6, 6-9
Data Shifter/Limiter 3-9
DEBUG A-76
Debug Mode
entering 10-14
Debug Request Input (DR) 10-6
Debug Serial Output (DSO) 10-5
DEBUGcc A-78
DEC A-80
Design Verification Support 11-3
DFO-DF3 9-12
DIV A-82
DO A-88
DO Instruction Restrictions 7-8
DO loop control 2-5
Double Precision Multiply Mode 3-16
algorithm examples 3-16
Double Precision Multiply Mode Bit 5-13
Dr. BuB 11-7
DSP Applications 1-7
DSP Functions 1-7
DSP News 11-16
DSP56K Central Architecture
central components 2-3
address buses 2-4
address generation unit 2-5
data ALU 2-5
data buses 2-3
memory expansion port (port A) 2-6
on-chip emulator (OnCE) 2-6
phase-locked loop (PLL)
based clocking 2-6
program control unit 2-5

—E—

Edge Sensitive 7-16
Edge Triggered 5-6
Electronic Bulletin Board 11-7
Encodings A-311
condition code and address A-315
double-bit register A-312
effective addressing mode A-315
five-bit register A-314
four-bit register A-313
memory space bit A-314
no parallel move A-318
nonmultiply instruction A-332
parallel instruction opcode A-330
parallel move A-316
program control unit registers A-315
single-bit register A-312
six-bit register A-314
triple bit register A-313
write control A-314
ENDDO A-98
ENDDO Instruction Restrictions 7-9
EOR A-100
Exception (Interrupt) Priorities 7-12
Exception Processing State 7-10
EXTAL
synch w/CKOUT 9-14
Extension Bit 5-11, A-16
External Interrupt Request Pins 5-6

—F—

Fast Interrupt 7-10, 7-12
Fast Interrupt Execution 7-26
FFT Code B-3
FIR Filter B-3
Frequency Multiplication 9-3
Frequency Multiplier 9-5

—G—

Global Data Bus (GDB) 2-3

—H—

Hardware DO Loop 6-24, A-88
Hardware Interrupt 7-11
Hardware Interrupt Sources 7-16
  IRQA 7-16
  IRQB 7-16
  NMI 7-16
  RESET 7-16
Hardware Reset
  OnCE pins and 10-5
Help Line 11-16

—I—

IIR Filter B-3
ILLEGAL A-102
Illegal Instruction Interrupt (III) 7-17
Immediate Data 6-14
Immediate Short 6-14
INC A-104
Instruction Descriptions A-21
Instruction Encoding A-311
Instruction Format 6-3, A-3
Instruction Groups 6-20
Instruction Guide A-3
Instruction Pipeline 5-6, 7-3
  restrictions 7-8
Instruction Sequence Restrictions A-305
Instruction Syntax 6-3
Instruction Timing A-294
Instruction Timing Summary A-301
Instruction Timing Symbols A-9
Instructions
  arithmetic 6-22
  bit manipulation 6-24
  logical 6-23
  loop 6-24
  move 6-26
  program control 6-27
Interrupt
  fast 7-12
  hardware 7-11
  long 7-12
  restrictions 7-10
  sources 7-11
Interrupt Arbitration 7-24
Interrupt Control Pins 2-6
Interrupt Controller 7-24
Interrupt Delay Possibilities 7-25
Interrupt Execution 7-26
  fast 7-26
  long 7-29
Interrupt Instruction Fetch 7-24
  instructions preceding 7-25
Interrupt Masks 5-12
Interrupt Priority Levels (IPL) 5-6, 7-14
Interrupt Priority Register 7-14
Interrupt Priority Structure 7-12
Interrupt Processing State 7-10
Interrupt Sources 7-16
  hardware 7-16
  other 7-22
  software 7-17
  trace 7-22
Interrupt Types 7-12
IPL 7-14
IRQA 5-6
IRQB 5-6

—J—

Jcc A-106
JCLR A-110
JMP A-116
JScc A-118
JSCLR A-122
JSET A-130
JSR A-136
JSSET A-138

—L—

LA 5-5, 5-17
LC 5-5, 5-17
Level Sensitive 5-6, 7-16
Limit Bit 5-11, A-16
Limiting (Saturation Arithmetic) 3-9
Linear Arithmetic 4-14
Linear Modifier 4-16
Lock, PLL, loss of 9-13
Logic Unit 3-6
Logical Instructions 6-23
Long Interrupt 7-12
Long Interrupt Execution 7-29
Long Word 6-5
Loop Address (LA) Register 5-5, 5-17
Loop Counter (LC) Register 5-5, 5-17
Loop Flag Bit 5-13
Loop Instructions 6-24
Low Power Divider 9-3

Low Power Divider (LPD) 9-5
LSL A-144
LSR A-146
LUA A-148

—M—

MAC 3-6, 3-13
MAC Instruction A-150
MACR A-154
Memory Breakpoint Control Bits 10-9
Memory Breakpoint Occurrence Bit 10-11
Memory Upper Limit Register 10-12
MFO-MF11 9-12
MODA/IRQA 5-6
MODB/IRQB 5-6
MODC/NMI 5-6
Mode Control Pins 2-6
Mode Register (MR) 5-9
  double precision multiply mode (bit 14) . . 5-13
  interrupt masks (bits 8 and 9) 5-12
  loop flag (bit 15) 5-13
  scaling mode (bits 10 and 11) 5-12
  symbols table A-8
  trace mode (bit 13) 5-13, 7-22
Modulo Arithmetic 4-14
Modulo Modifier 4-18
  linear addressing 4-18
  multiple wrap-around addressing 4-21
MOVE A-158
Move Instructions 6-26
MOVE(C) A-206
MOVE(M) A-214
MOVEP A-220
MPY A-228
MPYR A-232

—N—

NEG A-236
Negative Bit 5-10, A-17
NMI 5-6, 7-17
Nonmaskable Interrupt (NMI) 7-17
NOP A-238
NORM A-240
Normal Processing State 7-3
NOT A-242

—O—

Offset Registers 4-4
OnCE 2-6, 10-3
  using the OnCE 10-20
OnCE Bit Counter 10-8
OnCE Commands 10-19
OnCE Controller 10-6
OnCE Decoder 10-9
OnCE Memory Breakpoint 10-11
OnCE Pins 10-3
OnCE Serial Interface 10-6
OnCE Status and Control Register 10-9
On-Chip Emulator (OnCE) 2-6
Opcode 6-3
Opcode Field 6-5
Operands 6-3
  accumulator 6-5
  byte 6-5
  long word 6-5
  miscellaneous A-7
  short word 6-5
  symbols for 6-9
  word 6-5
Operating Mode Register (OMR) 5-5, 5-14
  stop delay (SD) bit 7-38
Operation Word 6-3
Operators
  table, binary A-7
  table, unary A-7
Optional Effective Address Extension Word . . . 6-3
OR A-244
OR(I) A-246
Overflow Bit 5-10, A-17
Overflow Protection 3-8

—P—

Parallel Move Descriptions A-20, A-160
  address register update A-172
  immediate short data move A-164
  long memory data move A-198
  no parallel data move A-162
  register and Y memory data move A-192
  register to register data move A-168
  X memory and register data move A-180
  X memory data move A-174
  XY memory data move A-202
  Y memory data move A-186
PC 5-5
PCAP 9-10

PGND 9-9
Phase Detector 9-4
Phase-Locked Loop (PLL) 2-6, 9-3
PINIT 9-10
PLL 2-6, 9-3
  frequency multiplier 9-5
  hardware reset and 9-11
  introduction 9-3
  loss of lock 9-13
  low power divider 9-5
  operating frequency 9-11
  operation while disabled 9-12
  phase detector 9-4
  PLL control register 9-5
  stop processing state and 9-13
  voltage controlled oscillator (VCO) 9-5
PLL Control Register 9-5
  division factor bits 9-12
  multiplication factor bits 9-12
PLL Pins 9-9
  ckout 9-10
  ckp 9-10
  clgnd 9-10
  clvcc 9-10
  pcap 9-10
  pgnd 9-9
  pinit 9-10
  plock 9-10
  pvcc 9-9
PLOCK 9-10
Port A 2-6, 8-3
Port A Interface 8-3
Port A Signals 8-3
  bus control 8-5
  data bus 8-5
  Port A address 8-5
Port A Wait States 8-6
Power Consumption 7-37
Processing States 7-3
  interrupt (exception) 7-10
  normal 7-3
  stop 7-37
  wait 7-36
Program Address Bus (PAB) 2-4
Program Address Generator (PAG) 5-5
Program Control Instructions 6-27
Program Control Registers
  OMR and SR 6-8
Program Control Unit 5-3
  loop address (LA) 2-6
  loop counter (LC) 2-6
  operating mode register (OMR) 2-6
  program address generator 2-5, 5-5
  program counter (PC) 2-6
  program decode controller 2-5, 5-5
  program interrupt controller 2-5, 5-6
  registers operands table A-6
  stack pointer (SP) 2-6
  status register (SR) 2-6
  system stack 2-5, 5-3
Program Counter (PC) 5-5, 5-8
Program Data Bus (PDB) 2-3
Program Decode Controller 5-5
Program Interrupt Controller 5-6
Programming Model
  AGU 4-6
  data ALU 3-19
  program control unit 5-8
  summary 5-17
PVCC 9-9

—R—

Read/Write Controls 8-5
References
  memory 6-11
  operand 6-11
  program 6-11
  register 6-11
  stack 6-11
Register Direct 6-13
Register Indirect 4-8
Register References 6-11
REP Instruction 5-5, A-248
RESET Instruction A-256
RESET Pin 5-6
Reset Processing State
  entering 7-33
  leaving 7-33
  PLL and 9-11
Reverse-Carry Arithmetic 4-14
Reverse-Carry Modifier 4-22
RND A-258
ROL A-262
ROR A-264
Rounding 3-10
RTI A-266
RTI and RTS Instruction Restrictions 7-9
RTS A-268

—S—

Saturation Arithmetic 3-9
SBC A-270
Scaling 3-10
Scaling Bit 5-11, A-16
Scaling Mode Bits 5-12
SD Bit 7-38
Short Jump 6-14
Short Word 6-5
Sign Extension 3-8
Simulator Features 11-5
Software Debug Occurrence Bit 10-11
Software Interrupt Sources 7-17
  illegal instruction (III) 7-18
  SWI 7-17
SP 5-5, 5-15
SS 5-5
Stack Pointer (SP) Register 5-15
  restrictions 7-10
Stack Pointer Register (SP) 5-5
Status Register (SR) 5-5, 5-9
  condition code register 5-9
  mode register 5-9
Stop Cycles 7-38
Stop Delay Bit 7-38
STOP Instruction 7-37, A-274
Stop Processing State 7-37
  debug request during 10-15
  PLL and 7-41, 9-13
SUB A-276
SUBL A-278
SUBR A-280
Support 11-3
SWI Instruction A-282
Syntax 6-3
System Stack (SS) 5-3, 5-5, 5-14
  system stack high (SSH) 5-14
  system stack high (SSH) restrictions ... 7-10
  system stack low (SSL) 5-14
  system stack low (SSL) restrictions 7-10

—T—

Tcc A-284
Technical Assistance 11-16
TFR A-288
Timing Calculations A-294
Timing Skew 9-3
Trace Mode Bit 5-13, 10-10
Trace Occurrence Bit 10-11
Tracing
  OnCE trace logic 10-13
Tracing (DSP56000/56001 only) 7-22
Training 11-17
TST A-290

—U—

Unary Operators A-7
Unnormalized Bit 5-10, A-17
User Support 11-3

—V—

V-bit A-17
Voltage Controlled Oscillator (VCO) 9-5

—W—

WAIT Instruction 7-36, A-292
Wait Processing State 7-36
  debug request during 10-15
  PLL and 9-14
Word
  length of 6-5
  operation 6-3
  optional effective address extension 6-3

—X—

X Address Bus (XAB) 2-4
X Data Bus (XDB) 2-3

—Y—

Y Address Bus (YAB) 2-4
Y Data Bus (YDB) 2-3

—Z—

Zero Bit 5-10, A-17

DSP56K FAMILY INTRODUCTION 1

DSP56K CENTRAL ARCHITECTURE OVERVIEW 2

DATA ARITHMETIC LOGIC UNIT 3

ADDRESS GENERATION UNIT 4

PROGRAM CONTROL UNIT 5

INSTRUCTION SET INTRODUCTION 6

PROCESSING STATES 7

PORT A 8

PLL CLOCK OSCILLATOR 9

ON-CHIP EMULATION (OnCE) 10

ADDITIONAL SUPPORT 11

INSTRUCTION SET DETAILS A

BENCHMARK PROGRAMS B

INDEX I

1 DSP56K FAMILY INTRODUCTION

2 DSP56K CENTRAL ARCHITECTURE OVERVIEW

3 DATA ARITHMETIC LOGIC UNIT

4 ADDRESS GENERATION UNIT

5 PROGRAM CONTROL UNIT

6 INSTRUCTION SET INTRODUCTION

7 PROCESSING STATES

8 PORT A

9 PLL CLOCK OSCILLATOR

10 ON-CHIP EMULATION (OnCE)

11 ADDITIONAL SUPPORT

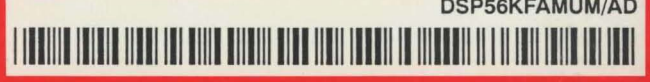
A INSTRUCTION SET DETAILS

B BENCHMARK PROGRAMS

I INDEX

1ATX31154-0 Printed in USA 10/1/92 BANTA CO. MOTO #13 20,000 DSP YGAVAA

Literature Distribution Centers:
USA: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.
EUROPE: Motorola Ltd.; European Literature Centre; 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.
JAPAN: Nippon Motorola Ltd.; 4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141, Japan.
ASIA PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Center, No. 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong.

MOTOROLA

DSP56KFAMUM/AD