

DSP56300

24-BIT DIGITAL SIGNAL PROCESSOR FAMILY MANUAL



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



MOTOROLA

TABLE OF CONTENTS

Paragraph Number	Title	Page Number
1	CORE DESCRIPTION	1-1
2	EXPANSION PORT	2-1
2.1	INTRODUCTION	2-1
2.2	EXPANSION PORT SIGNAL DESCRIPTION	2-1
2.2.1	Interrupt And Mode Control	2-1
2.2.2	Clock and Phase-Locked Loop (PLL)	2-3
2.2.3	On-Chip Emulator Interface (OnCE)/JTAG Interface	2-3
2.2.4	Expansion Port (Port A)	2-4
2.3	EXPANSION PORT OPERATION	2-7
2.3.1	Static RAM support	2-7
2.3.1.1	Synchronous Static RAM (SSRAM) Support	2-7
2.3.1.2	Asynchronous Static RAM (SRAM) Support	2-11
2.3.2	Dynamic Memories Support	2-12
2.3.3	Expansion Port Stalls	2-15
2.3.3.1	External Fetch From Synchronous SRAM	2-15
2.3.3.2	Non Synchronous SRAM Access Immediately Following Synchronous SRAM access	2-15
2.3.4	Expansion port Disable	2-16
2.4	BUS HANDSHAKE AND ARBITRATION	2-16
2.4.1	Bus Arbitration Signals	2-16
2.4.2	The Arbitration Protocol	2-17
2.4.3	Arbitration Scheme	2-18
2.4.4	Bus Arbitration Example Cases	2-19
2.4.4.1	Case 1 – Normal	2-19
2.4.4.2	Case 2 – Bus Busy	2-19
2.4.4.3	Case 3 – Low Priority	2-19
2.4.4.4	Case 4 – Default	2-19
2.4.4.5	Case 5 – Bus Lock during RMW	2-19
2.4.4.6	Case 6 – Bus Park	2-19
2.5	EXPANSION PORT CONTROL	2-20
2.5.1	AA control Registers (one for each AA pin)	2-20
2.5.1.1	BAT(1:0) - External Access Type and pin definition- bits 1-0	2-21
2.5.1.2	BAAP - AA pin Polarity - bit 2	2-22
2.5.1.3	BPEN - Program space Enable - bit 3	2-22

Table of Contents (Continued)

Paragraph Number	Title	Page Number
2.5.1.4	BXEN - X data space Enable - bit 4	2-22
2.5.1.5	BYEN - Y data space Enable - bit 5	2-22
2.5.1.6	BAM - Address Muxing - bit 6	2-22
2.5.1.7	BPAC- Packing Enable - bit 7	2-23
2.5.1.8	BNC(3:0) - Number of address bits to Compare - bits 11-8.....	2-23
2.5.1.9	BAC(11:0) - Address to compare - bits 23-12.....	2-23
2.5.2	Bus Control Register	2-24
2.5.2.1	BA0W(4:0) - Area 0 Wait control - bits 4-0	2-24
2.5.2.2	BA1W(4:0) - Area 1 Wait control - bits 9-5	2-25
2.5.2.3	BA2W(2:0) - Area 2 Wait control - bits 12-10	2-25
2.5.2.4	BA3W(2:0) - Area 3 Wait control - bits 15-13	2-25
2.5.2.5	BDFW(4:0)- Default Area Wait control - bits 20-16.....	2-25
2.5.2.6	BBS - Bus State - bit 21	2-26
2.5.2.7	BLH - Bus Lock Hold - bit 22	2-26
2.5.2.8	BRH - Bus Request Hold - bit 23	2-26
2.5.3	IDentification Register	2-26
2.6	DRAM CONTROLLER	2-26
2.6.1	DRAM Control Register.....	2-26
2.6.1.1	BCW(1:0) - In page Wait states - bits 1-0	2-27
2.6.1.2	BRW(1:0) - Out of page Wait states- bits 3-2	2-27
2.6.1.3	BPS(1:0) - DRAM Page Size - bits 9-8	2-28
2.6.1.4	BPLE - Page logic Enable - bit 11	2-28
2.6.1.5	BME - Mastership Enable - bit 12	2-29
2.6.1.6	BREN - Refresh Enable - bit 13.....	2-29
2.6.1.7	BSTR - Software Triggered Refresh - bit 14.....	2-29
2.6.1.8	BRF(7:0) Refresh rate - bits 22-15	2-30
2.6.1.9	BRP - Refresh Prescaler - bit 23	2-30
3	DATA ARITHMETIC LOGIC UNIT	3-1
3.1	DATA ALU ARCHITECTURE.....	3-1
3.1.1	Data ALU Input Registers (X1, X0, Y1, Y0)	3-1
3.1.2	MAC Unit.....	3-2
3.1.3	Data ALU Accumulator Registers (A2, A1, A0, B2, B1, B0).....	3-4
3.1.4	Accumulator Shifter.....	3-4
3.1.5	Bit Field Unit (BFU)	3-5
3.1.6	Data Shifter/Limiter	3-5
3.1.7	Scaling	3-5
3.1.8	Limiting.....	3-5
3.2	DATA ALU ARITHMETIC AND ROUNDING.....	3-6
3.2.1	Data Representation	3-6

Table of Contents (Continued)

Paragraph Number	Title	Page Number
3.2.2	Rounding Modes	3-8
3.2.2.1	Convergent Rounding	3-8
3.2.2.2	Two's Complement Rounding	3-8
3.2.3	Arithmetic Saturation Mode	3-11
3.2.4	Multiprecision Arithmetic Support	3-12
3.2.4.1	Double Precision Multiply Mode	3-13
3.2.5	Block Floating Point FFT Support	3-14
3.3	DATA ALU PROGRAMMING MODEL	3-15
3.4	SIXTEEN BIT ARITHMETIC MODE	3-15
3.4.1	Moves in sixteen bit arithmetic mode	3-16
3.4.1.1	Moves into registers or accumulators	3-16
3.4.1.2	Moves from registers or accumulators	3-17
3.4.1.3	Short Immediate moves	3-18
3.4.1.4	Scaling and Limiting	3-18
3.4.2	Sixteen bit arithmetic	3-18
3.5	PIPELINE CONFLICTS	3-20
4	ADDRESS GENERATION UNIT	4-1
4.1	AGU ARCHITECTURE	4-1
4.2	SIXTEEN-BIT COMPATIBILITY MODE	4-2
4.3	PROGRAMMING MODEL	4-3
4.3.1	Address Register Files (R0 - R3, EP and R4 - R7)	4-3
4.3.1.1	Stack Extension Pointer (EP)	4-4
4.3.2	Offset Register Files (N0 - N3 and N4 - N7)	4-4
4.3.3	Modifier Register Files (M0 - M3 and M4 - M7)	4-4
4.4	ADDRESSING MODES	4-4
4.4.1	Register Direct Mode	4-4
4.4.1.1	Data or Control Register Direct	4-5
4.4.1.2	Address Register Direct	4-5
4.4.2	Address Register Indirect Modes	4-5
4.4.2.1	No Update (Rn)	4-5
4.4.2.2	Postincrement By 1 (Rn)+	4-5
4.4.2.3	Postdecrement By 1 (Rn)-	4-5
4.4.2.4	Postincrement By Offset Nn (Rn)+Nn	4-5
4.4.2.5	Postdecrement By Offset Nn (Rn)-Nn	4-5
4.4.2.6	Indexed By Offset Nn (Rn+Nn)	4-6
4.4.2.7	Predecrement By 1 -(Rn)	4-6
4.4.2.8	Short displacement (Rn+short displacement)	4-6
4.4.2.9	Long displacement (Rn+long displacement)	4-6
4.4.3	PC Relative Modes	4-6

Table of Contents (Continued)

Paragraph Number	Title	Page Number
4.4.3.1	Short Displacement PC Relative	4-6
4.4.3.2	Long Displacement PC Relative	4-6
4.4.3.3	Address Register PC Relative	4-7
4.4.4	Special Address Modes	4-7
4.4.4.1	Immediate Data	4-7
4.4.4.2	Immediate Short Data.....	4-7
4.4.4.3	Absolute Address.....	4-7
4.4.4.4	Absolute Short Address	4-7
4.4.4.5	Short Jump Address	4-7
4.4.4.6	I/O Short Address	4-7
4.4.4.7	Implicit Reference	4-7
4.5	ADDRESS MODIFIER TYPES.....	4-8
4.5.1	Linear Modifier (Mn=\$XXFFFF)	4-8
4.5.2	Reverse-Carry Modifier (Mn=\$000000)	4-8
4.5.3	Modulo Modifier (Mn=MODULUS-1)	4-8
4.5.4	Multiple Wrap-Around Modulo Modifier	4-11
4.5.5	Address-Modifier-Type Encoding Summary	4-11
5	INSTRUCTION CACHE CONTROLLER.....	5-1
5.1	INTRODUCTION	5-1
5.2	INSTRUCTION CACHE ARCHITECTURE	5-1
5.2.1	Instruction Cache Structure.....	5-1
5.2.2	Cache Programmer's Model	5-2
5.2.3	Cache Operation	5-3
5.2.3.1	program fetch.....	5-3
5.2.3.2	hit.....	5-3
5.2.3.3	word miss when burst mode is disabled	5-3
5.2.3.4	word miss when burst mode is enabled.....	5-3
5.2.3.5	sector miss.....	5-4
5.2.4	Default Mode On Hardware Reset	5-4
5.2.5	Cache Locking	5-5
5.2.6	Cache Unlocking	5-5
5.2.7	Cache Flush	5-6
5.2.8	Sector Replacement Unit	5-7
5.2.9	Data Transfers to/from ICACHE Space	5-7
5.2.9.1	DMA transfers.....	5-7
5.2.9.2	Software-Controlled transfers	5-7
5.2.10	Cache Observability Via OnCE	5-8

Table of Contents (Continued)

Paragraph Number	Title	Page Number
6	PROGRAM CONTROL UNIT	6-1
6.1	OVERVIEW	6-1
6.2	PROGRAM CONTROL UNIT ARCHITECTURE	6-2
6.2.1	Instruction Pipeline	6-2
6.2.2	Clock Oscillator	6-3
6.3	PROGRAMMING MODEL	6-4
6.3.1	Program Counter (PC)	6-4
6.3.2	Vector Base Address Register (VBA)	6-4
6.3.3	Loop Counter Register (LC)	6-5
6.3.4	Loop Address Register (LA)	6-5
6.3.5	System Stack (SS)	6-5
6.3.6	Stack Extension Pointer (EP)	6-6
6.3.7	Stack Size Register (SZ)	6-6
6.3.8	Stack Counter Register (SC)	6-6
6.3.9	Stack Pointer Register (SP)	6-7
6.3.9.1	Stack Pointer (Bits 0–3)	6-7
6.3.9.2	Stack Error Flag / P4 bit (Bit 4)	6-7
6.3.9.3	Underflow Flag / P5 bit (Bit 5)	6-8
6.3.10	Status Register (SR)	6-9
6.3.10.1	Carry (Bit 0)	6-10
6.3.10.2	Overflow (Bit 1)	6-10
6.3.10.3	Zero (Bit 2)	6-10
6.3.10.4	Negative (Bit 3)	6-11
6.3.10.5	Unnormalized (Bit 4)	6-11
6.3.10.6	Extension (Bit 5)	6-11
6.3.10.7	Limit (Bit 6)	6-11
6.3.10.8	Scaling (Bit 7)	6-12
6.3.10.9	Interrupt Masks (Bits 8 and 9)	6-12
6.3.10.10	Scaling Mode (Bits 10 and 11)	6-12
6.3.10.11	Reserved SR Bit (Bit 12)	6-13
6.3.10.12	Sixteen-Bit Compatibility Mode (Bit 13)	6-13
6.3.10.13	Double Precision Multiply Mode (Bit 14)	6-13
6.3.10.14	DO-Loop Flag (Bit 15)	6-14
6.3.10.15	DO-Forever flag (Bit 16)	6-14
6.3.10.16	Sixteen-Bit Arithmetic Mode (Bit 17)	6-14
6.3.10.17	Reserved SR Bit (Bit 18)	6-14
6.3.10.18	Cache Enable (Bit 19)	6-15
6.3.10.19	Arithmetic Saturation Mode (Bit 20)	6-15
6.3.10.20	Rounding Mode (Bit 21)	6-15

Table of Contents (Continued)

Paragraph Number	Title	Page Number
6.3.10.21	Core Priority (Bits 22 and 23)	6-15
6.3.11	Operating Mode Register	6-16
6.3.11.1	Chip Operating Mode (Bits 0, 1,2 and 3)	6-17
6.3.11.2	External Bus Disable (Bit 4)	6-17
6.3.11.3	Reserved COM Bit (Bit 5)	6-17
6.3.11.4	Stop Delay (Bit 6)	6-17
6.3.11.5	Memory Switch (Bit 7)	6-18
6.3.11.6	Core-Dma Priority Bits (Bits 9 and 8)	6-18
6.3.11.7	Burst Mode Enable (Bit 10)	6-18
6.3.11.8	TA Synchronize Select (Bit 11)	6-18
6.3.11.9	Bus Release Timing (Bit 12)	6-19
6.3.11.10	Reserved EOM Bits (Bits 15, 14 and 13)	6-19
6.3.11.11	XY Select for Stack extension (Bit 16)	6-19
6.3.11.12	Extended Stack Underflow Flag (Bit 17)	6-19
6.3.11.13	Extended Stack Overflow Flag (Bit 18)	6-19
6.3.11.14	Extended Stack Wrap Flag (Bit 19)	6-20
6.3.11.15	Extended Stack Enable (Bit 20)	6-20
6.3.11.16	Reserved SCS Bits (Bits 21-23)	6-20
6.4	SIXTEEN-BIT COMPATIBILITY MODE	6-20
7	PROCESSING STATES	7-1
7.1	NORMAL PROCESSING STATE	7-1
7.1.1	Instruction Pipeline	7-1
7.2	EXCEPTION PROCESSING STATE (INTERRUPT PROCESSING)	7-2
7.2.1	Interrupt Sources	7-3
7.2.1.1	Hardware Interrupt Source	7-4
7.2.1.2	Software Interrupt Source	7-5
7.2.2	Interrupt Priority Structure	7-5
7.2.2.1	Interrupt Priority Levels	7-6
7.2.2.2	Exception Priorities within an IPL	7-7
7.2.3	Instructions Preceding the Interrupt Instruction Fetch	7-7
7.2.4	Interrupt Types	7-8
7.2.5	Interrupt Arbitration	7-8
7.2.6	Interrupt Instruction Fetch	7-10
7.2.7	Interrupt Instruction Execution	7-10

Table of Contents (Continued)

Paragraph Number	Title	Page Number
7.3	RESET PROCESSING STATE	7-12
7.4	WAIT PROCESSING STATE	7-12
7.5	STOP PROCESSING STATE	7-13
8	DMA CONTROLLER	8-1
8.1	DMA CONTROLLER PROGRAMMING MODEL	8-2
8.1.1	DMA Source Address Register (DSR)	8-3
8.1.2	DMA Destination Address Register (DDR).....	8-3
8.1.3	DMA Offset Register (DOR)	8-3
8.1.4	DMA Counter (DCO)	8-4
8.1.4.1	DMA counter mode A - single counter	8-4
8.1.4.2	DMA counter mode B - dual counter.....	8-5
8.1.4.3	DMA counter modes C, D and E- triple counter.....	8-5
8.1.5	DMA Control Register (DCR)	8-8
8.1.5.1	DCR DMA Channel Enable Bit (DE) Bit 23.....	8-8
8.1.5.2	DCR DMA Interrupt Enable Bit (DIE) Bit 22.....	8-8
8.1.5.3	DCR DMA Transfer Mode (DTM2-DTM0)- bits 21:19.....	8-8
8.1.5.4	DCR DMA Channel priority(DPR1-DPR0) - bits 18:17	8-10
8.1.5.5	DCR DMA Continuous Mode (DCON) - bit 16	8-11
8.1.5.6	DCR DMA Request Source (DRS0-DRS4) Bits 15-11	8-11
8.1.5.7	DCR DMA three Dimensional mode (D3D)- bit 10.....	8-12
8.1.5.8	DCR DMA Address Mode (DAM5-DAM0)- bits 9:4.....	8-12
8.1.5.9	DCR DMA Destination Space (DDS0-DDS1) Bits 3, 2	8-16
8.1.5.10	DCR DMA Source Space (DSS0-DSS1) Bits 1, 0	8-16
8.1.6	DMA Status Register (DSTR).....	8-16
8.1.6.1	DSTR DMA Channel Transfer Done Status (DTD)- bits 5-0.....	8-17
8.1.6.2	DSTR reserved bits - bits 23:12 and 7:6.....	8-17
8.1.6.3	DSTR DMA Active state (DACT) - bit 8	8-17
8.1.6.4	DSTR DMA Active Channel (DCH2-DCH0) - bits 11:9	8-17
8.2	DMA Restrictions	8-18
9	PLL and CLOCK GENERATOR.....	9-1
9.1	INTRODUCTION	9-1
9.1.1	Clock Input Division	9-2
9.1.2	Frequency Multiplication.....	9-2
9.1.3	Skew Elimination	9-2
9.1.4	Low Power Divide and Output Stage	9-3
9.2	PLL BLOCK DIAGRAM	9-3
9.2.1	Frequency Pre-Divider	9-4
9.2.2	Phase Frequency Detector and Charge Pump Loop Filter	9-4

Table of Contents (Continued)

Paragraph Number	Title	Page Number
9.2.3	PLL Control Register (PCTL)	9-4
9.2.3.1	Multiplication Factor Bits (MF0-MF11) - Bits 0-11	9-5
9.2.3.2	Division Factor Bits (DF2-DF0) - Bits 12-14	9-6
9.2.3.3	Crystal Range Bit (XTLR) - Bit 15.....	9-7
9.2.3.4	XTAL Disable Bit (XTLD) - Bit 16.....	9-7
9.2.3.5	STOP Processing State Bit (PSTP) - Bit 17	9-7
9.2.3.6	PLL Enable Bit (PEN) - Bit 18.....	9-8
9.2.3.7	Clock Output Disable Bit (COD) - Bit 19	9-8
9.2.3.8	PreDivider Factor Bits (PD0-PD3) - Bits 20-23.....	9-9
9.2.4	Voltage Controlled Oscillator (VCO)	9-9
9.2.5	Divide by 2	9-10
9.2.6	Frequency Divider	9-10
9.3	CLKGEN BLOCK DIAGRAM.....	9-10
9.3.1	Low Power Divider (LPD).....	9-10
9.3.2	Divide by 2	9-11
9.3.3	Operating Frequency	9-11
9.3.4	Synchronization among EXTAL, CLKOUT, and the Internal Clock.....	9-11
9.4	PLL PINS.....	9-11
10	ON-CHIP EMULATOR (OnCE™)	10-1
10.1	INTRODUCTION	10-1
10.2	ON-CHIP EMULATION (OnCE™) PINS.....	10-1
10.2.1	Debug Event (DE)	10-2
10.3	OnCE™ CONTROLLER	10-2
10.3.1	OnCE™ Command Register (OCR).....	10-3
10.3.1.1	Register Select (RS4-RS0) Bits 0-4.....	10-3
10.3.1.2	Exit Command (EX) Bit 5.....	10-3
10.3.1.3	Go Command (GO) Bit 6	10-5
10.3.1.4	Read/Write Command (R/W) Bit 7.....	10-5
10.3.2	OnCE™ Decoder (ODEC).....	10-5
10.3.3	OnCE™ Status and Control Register (OSCR)	10-5
10.3.3.1	Trace Mode Enable (TME) Bit 0	10-6
10.3.3.2	Interrupt Mode Enable (IME) Bit 1	10-6
10.3.3.3	Software Debug Occurrence (SWO) Bit 2	10-6
10.3.3.4	Memory Breakpoint Occurrence (MBO) Bit 3	10-6
10.3.3.5	Trace Occurrence (TO) Bit 4	10-6
10.3.3.6	Cache Hit (HIT) Bit 5.....	10-6
10.3.3.7	Core Status (OS0,OS1) Bits 6-7	10-6
10.3.3.8	Reserved Bits 8-23	10-7

Table of Contents (Continued)

Paragraph Number	Title	Page Number
10.4	OnCE™ MEMORY BREAKPOINT LOGIC	10-7
10.4.1	Memory Address Latch (OMAL)	10-7
10.4.2	Memory Limit Register 0 (OMLR0).....	10-7
10.4.3	Memory Address Comparator 0 (OMAC0)	10-8
10.4.4	Memory Limit Register 1 (OMLR1).....	10-8
10.4.5	Memory Address Comparator 1 (OMAC1)	10-8
10.4.6	Breakpoint Control Register (OBCR)	10-8
10.4.6.1	Memory Breakpoint Select (MBS0-MBS1) Bits 0-1	10-9
10.4.6.2	Breakpoint 0 Read/Write Select (RW00-RW01) Bits 2-3	10-9
10.4.6.3	Breakpoint 0 Condition Code Select (CC00-CC01) Bits4-5	10-10
10.4.6.4	Breakpoint1 Read/Write Select (RW10-RW11) Bits 6-7	10-10
10.4.6.5	Breakpoint1 Condition Code Select (CC10-CC11) Bits8-9	10-10
10.4.6.6	Breakpoint 0 and 1 Event Select (BT1-BT0) Bits10-11	10-11
10.4.7	Memory Breakpoint Counter (OMBC)	10-11
10.5	CACHE SUPPORT	10-12
10.6	OnCE™ TRACE LOGIC	10-13
10.6.1	Trace Counter (OTC)	10-14
10.7	METHODS OF ENTERING THE DEBUG MODE	10-14
10.7.1	External Debug Request During RESET	10-15
10.7.2	External Debug Request During Normal Activity.....	10-15
10.7.3	Executing the JTAG DEBUG_REQUEST Instruction.....	10-15
10.7.4	External Debug Request During STOP	10-15
10.7.5	External Debug Request During WAIT	10-15
10.7.6	Software Request During Normal Activity	10-15
10.7.7	Enabling Trace Mode	10-16
10.7.8	Enabling Memory Breakpoints	10-16
10.8	PIPELINE INFORMATION AND GDB REGISTER.....	10-16
10.8.1	PDB Register (OPDBR)	10-16
10.8.2	PIL Register (OPILR)	10-17
10.8.3	GDB Register (OGDBR).....	10-17
10.9	TRACE BUFFER	10-17
10.9.1	PAB Register for Fetch (OPABFR)	10-17
10.9.2	PAB Register for Decode (OPABDR).....	10-17
10.9.3	PAB Register for Execute (OPABEX)	10-18
10.9.4	Trace Buffer.....	10-18
10.10	SERIAL PROTOCOL DESCRIPTION	10-19
10.10.1	OnCE™ Commands	10-20

Table of Contents (Continued)

Paragraph Number	Title	Page Number
10.11	TARGET SITE DEBUG SYSTEM REQUIREMENTS	10-20
10.12	EXAMPLES OF USING THE OnCE™	10-20
10.12.1	Checking whether the chip has entered the Debug Mode	10-21
10.12.2	Polling the JTAG instruction shift register	10-21
10.12.3	Saving Pipeline Information	10-21
10.12.4	Reading the Trace Buffer	10-22
10.12.5	Displaying a specified register	10-22
10.12.6	Displaying X memory area starting at address \$xxxxxx.....	10-23
10.12.7	Returning from Debug Mode to Normal Mode to current program....	10-24
10.12.8	Returning from Debug Mode to Normal Mode to a new program	10-24
10.13	EXAMPLES OF JTAG-OnCE INTERACTION.....	10-25
11	JTAG (IEEE 1149.1) Test Access Port	11-1
11.1	INTRODUCTION	11-1
11.2	OVERVIEW	11-1
11.2.1	JTAG PINS.....	11-3
11.2.1.1	Test Clock (TCK)	11-3
11.2.1.2	Test Mode Select (TMS).....	11-3
11.2.1.3	Test Data Input (TDI)	11-3
11.2.1.4	Test Data Output (TDO)	11-3
11.2.1.5	Test Reset (TRST~)	11-3
11.2.2	TAP CONTROLLER.....	11-3
11.2.3	BOUNDARY SCAN REGISTER	11-4
11.2.4	INSTRUCTION REGISTER	11-5
11.2.4.1	EXTEST.....	11-6
11.2.4.2	SAMPLE/PRELOAD	11-6
11.2.4.3	BYPASS	11-6
11.2.4.4	IDCODE	11-7
11.2.4.5	HI-Z.....	11-7
11.2.4.6	CLAMP	11-8
11.2.4.7	ENABLE_ONCE	11-8
11.2.4.8	DEBUG_REQUEST.....	11-8
11.3	DSP56300 RESTRICTIONS	11-9
12	OPERATING MODES AND MEMORY SPACES.....	12-1
12.1	CHIP OPERATING MODES.....	12-1
12.1.1	Expanded Modes (Modes 0 and 8)	12-2
12.1.2	System Configuration Modes 1-15 (Mode 1-7 and 9-F).....	12-2
12.2	DSP56300 CORE MEMORY MAP	12-2
12.2.1	X Data Memory Space	12-2

Table of Contents (Continued)

Paragraph Number	Title	Page Number
12.2.2	Y Data Memory Space	12-5
12.2.3	Program Memory	12-6
12.3	SIXTEEN-BIT COMPATIBILITY MODE	12-7
12.4	MEMORY SWITCH MODE	12-8

INSTRUCTION SET

Appendix A	INSTRUCTION SET	A-3
A-1	INTRODUCTION	A-3
A-2	INSTRUCTION FORMATS AND SYNTAX	A-3
A-2.1	Operand Sizes	A-5
A-2.2	Data Organization in Registers	A-6
A-2.3	Data ALU Registers	A-6
A-2.4	AGU Registers	A-7
A-2.5	Program Control Registers	A-7
A-2.6	Data Organization in Memory	A-8
A-3	INSTRUCTION GROUPS	A-8
A-3.1	Arithmetic Instructions	A-9
A-3.2	Logical Instructions	A-11
A-3.3	Bit Manipulation Instructions	A-12
A-3.4	Loop Instructions	A-13
A-3.5	Move Instructions	A-13
A-3.6	Program Control Instructions	A-15
A-4	INSTRUCTION GUIDE	A-17
A-4.1	NOTATION	A-17
A-5	CONDITION CODE COMPUTATION	A-22
A-6	INSTRUCTIONS DESCRIPTIONS	A-26
A-6.1	Absolute Value (ABS)	A-27
A-6.2	Add Long with Carry (ADC)	A-28
A-6.3	Add (ADD)	A-29
A-6.4	Shift Left and Add Accumulators (ADDL)	A-31
A-6.5	Shift Right and Add Accumulators (ADDR)	A-32
A-6.6	Logical AND (AND)	A-33
A-6.7	AND Immediate with Control Register (ANDI)	A-35
A-6.8	Arithmetic Shift Accumulator Left (ASL)	A-37
A-6.9	Arithmetic Shift Accumulator Right (ASR)	A-40

Table of Contents (Continued)

Paragraph Number	Title	Page Number
A-6.10	Branch Conditionally (Bcc)	A-43
A-6.11	Bit Test and Change (BCHG)	A-45
A-6.12	Bit Test and Clear (BCLR)	A-48
A-6.13	Branch Always (BRA)	A-51
A-6.14	Branch if Bit Clear (BRCLR)	A-53
A-6.15	Exit Current Do Loop Conditionally (BRKcc)	A-56
A-6.16	Branch if Bit Set (BRSET)	A-57
A-6.17	Branch to Subroutine Conditionally (BScc)	A-60
A-6.18	Branch to Subroutine if Bit Clear (BSCLR)	A-62
A-6.19	Bit Test and Set (BSET)	A-65
A-6.20	Branch to Subroutine (BSR)	A-68
A-6.21	Branch to Subroutine if Bit Set (BSSET)	A-70
A-6.22	Bit Test (BTST)	A-73
A-6.23	Count Leading Bits (CLB)	A-75
A-6.24	Clear accumulator (CLR)	A-77
A-6.25	Compare (CMP)	A-78
A-6.26	Compare Magnitude (CMPM)	A-80
A-6.27	Compare Unsigned (CMPU)	A-82
A-6.28	Enter Debug Mode (DEBUG)	A-83
A-6.29	Enter Debug Mode Conditionally (DEBUGcc)	A-84
A-6.30	Decrement by One (DEC)	A-85
A-6.31	Divide Iteration (DIV)	A-86
A-6.32	Double Precision MAC with 24 bit Right Shift (DMAC)	A-89
A-6.33	Start Hardware Loop (DO)	A-91
A-6.34	Start Infinite Loop (DO FOREVER)	A-95
A-6.35	Start PC Relative Hardware Loop (DOR)	A-97
A-6.36	Start PC Relative Infinite Loop (DOR FOREVER)	A-100
A-6.37	End Current DO Loop (ENDDO)	A-102
A-6.38	Logical Exclusive OR (EOR)	A-103
A-6.39	Bit Field (EXTRACT)	A-105
A-6.40	Extract Unsigned Bit Field (EXTRACTU)	A-108
A-6.41	Execute Conditionally without CCR Update (IFcc)	A-111
A-6.42	Execute Conditionally with CCR Update (IFcc.U)	A-112
A-6.43	Illegal Instruction Interrupt (ILLEGAL)	A-113
A-6.44	Increment by One (INC)	A-115
A-6.45	Insert Bit field (INSERT)	A-116
A-6.46	Jump Conditionally (JCC)	A-119
A-6.47	Jump if Bit Clear (JCLR)	A-121
A-6.48	Jump (JMP)	A-123

Table of Contents (Continued)

Paragraph Number	Title	Page Number
A-6.49	Jump to Subroutine Conditionally (JScC)	A-124
A-6.50	Jump to Subroutine if Bit Clear (JSCLR)	A-126
A-6.51	Jump if Bit Set (JSET)	A-129
A-6.52	Jump to Subroutine (JSR)	A-131
A-6.53	Jump to Subroutine if Bit Set (JSSET)	A-133
A-6.54	Load PC Relative Address (LRA)	A-136
A-6.55	Logical Shift Left (LSL)	A-138
A-6.56	Logical Shift Right (LSR)	A-141
A-6.57	Load Updated Address (LUA)	A-144
A-6.58	Signed Multiply-Accumulate (MAC)	A-146
A-6.59	Signed MAC with Immediate Operand (MACI)	A-148
A-6.60	Mixed Multiply-Accumulate (MAC su/uu)	A-149
A-6.61	Signed MAC and Round (MACR)	A-150
A-6.62	Signed MAC and Round with Immediate Operand (MACRI)	A-152
A-6.63	Transfer by Signed Value (MAX)	A-154
A-6.64	Transfer by Magnitude (MAXM)	A-155
A-6.65	Merge Two Half Words (MERGE)	A-156
A-6.66	Move Data (MOVE)	A-158
A-6.67	NO Parallel Data Move	A-159
A-6.68	Immediate Short Data Move (I)	A-160
A-6.69	Register to Register Data Move (R)	A-162
A-6.70	Address Register Update (U)	A-164
A-6.71	X Memory Data Move (X:)	A-165
A-6.72	X Memory and Register Data Move (X:R)	A-168
A-6.73	Y Memory Data Move (Y:)	A-170
A-6.74	Register and Y Memory Data Move (R:Y)	A-173
A-6.75	Long Memory Data Move (L:)	A-176
A-6.76	XY Memory Data Move (X: Y:)	A-178
A-6.77	Move Control Register (MOVEC)	A-180
A-6.78	Move Program Memory (MOVEM)	A-183
A-6.79	Move Peripheral Data (MOVEP)	A-185
A-6.80	Signed Multiply (MPY)	A-188
A-6.81	Mixed Multiply (MPY su/uu)	A-190
A-6.82	Signed Multiply with Immediate Operand (MPYI)	A-191
A-6.83	Signed Multiply and Round (MPYR)	A-192
A-6.84	Signed Multiply and Round with Immediate Operand (MPYRI)	A-194
A-6.85	Negate Accumulator (NEG)	A-196
A-6.86	No Operation (NOP)	A-197
A-6.87	Norm Accumulator Iteration (NORM)	A-198

Table of Contents (Continued)

Paragraph Number	Title	Page Number
A-6.88	Fast Accumulator Normalization (NORMF)	A-200
A-6.89	Logical Complement (NOT)	A-202
A-6.90	Logical Inclusive OR (OR)	A-203
A-6.91	OR Immediate with Control Register (ORI)	A-205
A-6.92	Program-Cache Flush (PFLUSH)	A-207
A-6.93	Program-Cache Flush Unlock Sectors(PFLUSHUN)	A-208
A-6.94	Program-Cache Global Unlock (PFREE)	A-209
A-6.95	Lock Instruction Cache Relative Sector (PLOCKR)	A-210
A-6.96	Unlock instruction Cache Sector (PUNLOCK).....	A-211
A-6.97	Unlock instruction Cache Relative Sector (PUNLOCKR)	A-212
A-6.98	Repeat Next Instruction (REP)	A-213
A-6.99	Reset On-Chip Peripheral Devices (RESET)	A-215
A-6.100	Round Accumulator (RND).....	A-216
A-6.101	Rotate Left (ROL)	A-218
A-6.102	Rotate Right (ROR)	A-220
A-6.103	Return from Interrupt (RTI)	A-222
A-6.104	Return from Subroutine (RTS).....	A-223
A-6.105	Subtract Long with Carry (SBC)	A-224
A-6.106	Stop Instruction Processing (STOP).....	A-225
A-6.107	Subtract (SUB)	A-227
A-6.108	Shift Left and Subtract Accumulators (SUBL)	A-229
A-6.109	Shift Right and Subtract Accumulators (SUBR)	A-231
A-6.110	Transfer Conditionally (Tcc)	A-232
A-6.111	Transfer Data ALU Register (TFR).....	A-234
A-6.112	Software Interrupt (TRAP)	A-235
A-6.113	Conditional Software Interrupt (TRAPcc)	A-236
A-6.114	Test Accumulator (TST)	A-237
A-6.115	Wait for interrupt (WAIT).....	A-238
A-7	INSTRUCTION PARTIAL ENCODING	A-239
A-7.1	Partial Encodings for Use in Instruction Encoding.....	A-239
A-7.2	Parallel Instruction Encoding of the Operation Code.....	A-251
A-7.2.1	Multiply Instruction Encoding	A-251
A-7.2.2	NonMultiply Instruction Encoding.....	A-252

INSTRUCTION EXECUTION TIMING

Appendix B INSTRUCTION EXECUTION TIMING	B-3
---	-----

Table of Contents (Continued)

Paragraph Number	Title	Page Number
B-1	INTRODUCTION.....	B-3
B-2	INSTRUCTION TIMING	B-3
B-3	INSTRUCTION SEQUENCE DELAYS	B-13
B-3.1	External Bus Wait States	B-13
B-3.2	External Bus Contention	B-13
B-3.3	Instruction Fetch delays.....	B-14
B-3.4	Data ALU Interlock.....	B-15
B-3.4.1	Arithmetic Stall	B-15
B-3.4.2	Transfer Stall	B-15
B-3.4.3	Status Stall	B-15
B-3.5	Address Registers Interlocks	B-15
B-3.5.1	Conditional Transfer Interlock	B-15
B-3.5.2	Address Generation Interlock.....	B-15
B-3.6	Stack Extension Delays	B-17
B-3.7	Program Flow-Control delays	B-18
B-3.7.1	MOVE to CR.....	B-19
B-3.7.2	MOVE from CR	B-19
B-3.7.3	MOVE to SP/SC	B-19
B-3.7.4	MOVE to LA register	B-19
B-3.7.5	MOVE to SR.....	B-19
B-3.7.6	MOVE to SSH/SSL.....	B-19
B-3.7.7	JMP to (LA) or to (LA-1)	B-20
B-3.7.8	RTI to (LA) or to (LA-1).....	B-20
B-3.7.9	MOVE from SSH	B-20
B-3.7.10	Conditional Instructions	B-20
B-3.7.11	Interrupt Abort	B-20
B-3.7.12	Degenerated DO loop	B-20
B-3.7.13	Annulled REP and DO.....	B-20
B-4	INSTRUCTION SEQUENCE RESTRICTIONS.....	B-21
B-4.1	Restrictions Near the End of DO Loops.....	B-21
B-4.1.1	At LA-3	B-21
B-4.1.2	At LA-2	B-21
B-4.1.3	At LA-1	B-22
B-4.1.4	At LA.....	B-22
B-4.2	General DO Restrictions	B-22
B-4.3	ENDDO Restrictions	B-23
B-4.4	BRKcc Restrictions	B-23
B-4.5	RTI and RTS Restrictions	B-23
B-4.6	SP, SC and SSH/SSL Manipulation Restrictions.....	B-23

Table of Contents (Continued)

Paragraph Number	Title	Page Number
B-4.7	Fast Interrupt Routines	B-24
B-4.8	REP Restrictions	B-24
B-4.9	Stack Extension Restrictions	B-24
B-4.10	Instruction Cache General Restrictions	B-25
B-5	Peripheral pipeline restrictions	B-25
B-5.1	Polling a peripheral device for write	B-25
B-5.2	Writing to a read-only register	B-26

BENCHMARK PROGRAMS

Appendix C	BENCHMARK PROGRAMS	C-3
C-1	INTRODUCTION	C-3
C-2	SET OF BENCHMARKS	C-3
C-2.1	Real Multiply	C-3
C-2.2	N Real Multiplies	C-3
C-2.3	Real Update	C-4
C-2.4	N Real Updates	C-4
C-2.5	Real Correlation Or Convolution (FIR Filter)	C-5
C-2.6	Real * Complex Correlation Or Convolution (FIR Filter)	C-7
C-2.7	Complex Multiply	C-7
C-2.8	N Complex Multiplies	C-8
C-2.9	Complex Update	C-9
C-2.10	N Complex Updates	C-10
C-2.11	Complex Correlation Or Convolution (FIR Filter)	C-11
C-2.12	Nth Order Power Series (Real)	C-12
C-2.13	2nd Order Real Biquad IIR Filter	C-13
C-2.14	N Cascaded Real Biquad IIR Filter	C-14
C-2.15	N Radix-2 FFT Butterflies (DIT, in-place algorithm)	C-15
C-2.16	True (Exact) LMS Adaptive Filter	C-16
C-2.17	Delayed LMS Adaptive Filter	C-19
C-2.18	FIR Lattice Filter	C-20
C-2.19	All Pole IIR Lattice Filter	C-21
C-2.20	General Lattice Filter	C-23
C-2.21	Normalized Lattice Filter	C-24
C-2.22	[1x3][3x3] Matrix Multiplication	C-25
C-2.23	N Point 3x3 2-D FIR Convolution	C-26

Table of Contents (Continued)		
Paragraph Number	Title	Page Number
C-2.24	Parsing data stream	C-28
C-2.25	Creating data stream	C-30
C-2.26	Parsing Hoffman code data stream	C-32
C-3	BENCHMARK OVERVIEW	C-36

LIST of FIGURES

Figure Number	Title	Page Number
1	CORE DESCRIPTION	
Figure 1-1.	DSP56300 Core Block Diagram	1-3
2	EXPANSION PORT	
Figure 2-1.	Bus operation - zero wait states Sync. SRAM access	2-9
Figure 2-2.	Bus operation - one wait state Sync. SRAM access.....	2-10
Figure 2-3.	Synchronous Static RAM connection diagram.....	2-10
Figure 2-4.	Bus operation one wait state - SRAM access.....	2-11
Figure 2-5.	Static RAM connection diagram.....	2-12
Figure 2-6.	Dynamic RAM connection diagram.....	2-14
Figure 2-7.	Bus operation two wait states - DRAM read access (in-page).....	2-14
Figure 2-8.	Bus operation two wait states - DRAM write access (in-page)	2-15
Figure 2-9.	Bus Arbitration scheme.....	2-18
Figure 2-10.	Address Attribute Registers (AAR3-0)	2-20
Figure 2-11.	Bus Control Register (BCR).....	2-24
Figure 2-12.	DRAM Control Register (DCR)	2-27
3	DATA ARITHMETIC LOGIC UNIT	
Figure 3-1.	Data ALU Block Diagram	3-3
Figure 3-2.	Bit Weighting and Alignment of Operands	3-7
Figure 3-3.	Integer/Fractional Multiplication	3-7
Figure 3-4.	Convergent Rounding (no scaling)	3-9
Figure 3-5.	Two's Complement Rounding (no scaling)	3-10
Figure 3-6.	DMAC Implementation.....	3-12
Figure 3-7.	Double Precision Multiplication Using DMAC	3-13
Figure 3-8.	Double precision algorithm	3-14
Figure 3-9.	DSP56300 Core Programming Model	3-15
Figure 3-10.	Sixteen Bit Arithmetic Mode Data Organization.....	3-16
Figure 3-11.	Pipeline Conflicts - Arithmetic stall.....	3-20
Figure 3-12.	Pipeline Conflicts - Status stall.....	3-21
Figure 3-13.	Pipeline Conflicts - Transfer stall	3-22

List of Figures (Continued)

Figure Number	Title	Page Number
------------------	-------	----------------

4 ADDRESS GENERATION UNIT

Figure 4-1.	AGU Block Diagram.....	4-1
Figure 4-2.	AGU Programming Model.....	4-3

5 INSTRUCTION CACHE CONTROLLER

Figure 5-1.	Instruction Cache Block Diagram.....	5-2
-------------	--------------------------------------	-----

6 PROGRAM CONTROL UNIT

Figure 6-1.	Program Control Unit Architecture	6-2
Figure 6-2.	Seven Stage Pipeline.....	6-3
Figure 6-3.	Program Control Unit Programming Model.....	6-4
Figure 6-4.	SP Register Format	6-7
Figure 6-5.	Status Register Format	6-10
Figure 6-6.	Operating Mode Register (OMR) Format.....	6-17
Figure 6-7.	Central Processor Programming Model.....	6-21

7 PROCESSING STATES

Figure 7-1.	Interrupt Priority Register C (IPRC)	7-6
Figure 7-2.	Interrupt Priority Register P (IPRP).....	7-7

8 DMA CONTROLLER

Figure 8-1.	DMA Control Register	8-8
Figure 8-2.	DMA Status Register	8-17

9 PLL and CLOCK GENERATOR

Figure 9-1.	PLL & CLOCK Block Diagram	9-1
Figure 9-2.	PLL Block Diagram	9-4
Figure 9-3.	PLL Control Register (PCTL)	9-5
Figure 9-4.	CLKGEN Block Diagram.....	9-10

10 ON-CHIP EMULATOR (OnCE™)

Figure 10-1.	OnCE™ Block Diagram.....	10-1
Figure 10-2.	OnCE Multiprocessor Configuration	10-2
Figure 10-3.	OnCE™ Controller	10-3
Figure 10-4.	OnCE™ Command Register	10-3
Figure 10-5.	OnCE™ Status and Control Register (OSCR)	10-5

List of Figures (Continued)		
Figure Number	Title	Page Number
Figure 10-6.	OnCE™ Memory Breakpoint Logic 0	10-8
Figure 10-7.	Breakpoint Control Register.....	10-9
Figure 10-8.	Circular Tags Buffer (TAGB).....	10-13
Figure 10-9.	OnCE™ Trace Logic Block Diagram.....	10-14
Figure 10-10.	OnCE™ Pipeline Information and GDB Registers	10-16
Figure 10-11.	OnCE™ Trace Buffer	10-19
11	JTAG (IEEE 1149.1) Test Access Port	
Figure 11-1.	JTAG Block Diagram	11-2
Figure 11-2.	TAP Controller State Machine	11-4
Figure 11-3.	Instruction Register	11-6
Figure 11-4.	Bypass Register.....	11-7
Figure 11-5.	Identification Register Configuration	11-7
12	OPERATING MODES AND MEMORY SPACES	
Figure 12-1.	DSP56300 Core Memory Map.....	12-2
Figure 12-2.	DSP56300 Core Memory Map (SC = 1)	12-8
Appendix A	INSTRUCTION SET	
Figure A-1.	General Formats of an Instruction Word.....	A-4
Figure A-2.	Reading and Writing the ALU Extension Registers	A-7
Figure A-3.	Reading and Writing Control Registers.....	A-8
Appendix B	INSTRUCTION EXECUTION TIMING	
Appendix C	BENCHMARK PROGRAMS	

LIST of TABLES

Table Number	Title	Page Number
1	CORE DESCRIPTION	
2	EXPANSION PORT	
3	DATA ARITHMETIC LOGIC UNIT	
Table 3-1.	Actions of the Arithmetic Saturation Mode (SM=1)	3-11
4	ADDRESS GENERATION UNIT	
Table 4-1.	Addressing Modes Summary	4-10
Table 4-2.	Address-Modifier-Type Encoding Summary	4-12
5	INSTRUCTION CACHE CONTROLLER	
6	PROGRAM CONTROL UNIT	
Table 6-1.	Seven Stage Pipeline.....	6-3
Table 6-2.	SP Register Values in the non-extended mode	6-8
Table 6-3.	Unnormalized Bit definition	6-11
Table 6-4.	Extension Bit definition.....	6-11
7	PROCESSING STATES	
Table 7-1.	Instruction Pipeline.....	7-2
Table 7-2.	Interrupt Sources	7-4
Table 7-3.	Status Register Interrupt Mask Bits	7-6
Table 7-4.	Interrupt Priority Level Bits.....	7-7
Table 7-5.	External Interrupt Trigger Mode Bits	7-7
Table 7-6.	Exception Priorities within an IPL.....	7-9
Table 7-7.	Fast Interrupt Pipeline.....	7-10
Table 7-8.	Long Interrupt Pipeline.....	7-11
8	DMA CONTROLLER	
Table 8-1.	DMA Controller Data Transfers.....	8-1
Table 8-2.	DMA Controller Programming Model - Channel 0	8-2
Table 8-3.	DMA Controller Programming Model - Channel 1	8-2
Table 8-4.	DMA Controller Programming Model - Channel 2	8-2

List of Tables (Continued)

Table Number	Title	Page Number
Table 8-5.	DMA Controller Programming Model - Channel 3.....	8-2
Table 8-6.	DMA Controller Programming Model - Channel 4.....	8-3
Table 8-7.	DMA Controller Programming Model - Channel 5.....	8-3
Table 8-8.	DMA Offset Registers.....	8-3
Table 8-9.	DMA Status Register.....	8-3
Table 8-10.	DMA Transfer Mode (DTM2-DTM0) Bits.....	8-9
Table 8-11.	DCR DMA Channel priority(DPR1-DPR0) Bits.....	8-10
Table 8-12.	Source Address Generation Mode (D3D = 0)	8-13
Table 8-13.	Destination Address Generation Mode (D3D = 0).....	8-14
Table 8-14.	Counter Mode (D3D = 1)	8-15
Table 8-15.	Address Mode Select (D3D = 1)	8-15
Table 8-16.	Address Generation Mode (D3D = 1).....	8-15
Table 8-17.	DCH Status bits encoding	8-18
9	PLL and CLOCK GENERATOR	
Table 9-1.	Multiplication Factor Bits MF0-MF11	9-6
Table 9-2.	Division Factor Bits DF0-DF2.....	9-7
Table 9-3.	PSTP and PEN Relationship.....	9-8
Table 9-4.	Predivision Factor Bits PD0-PD3	9-9
10	ON-CHIP EMULATOR (OnCE™)	
Table 10-1.	OnCE™ Register Addressing	10-4
Table 10-2.	Core Status Bits Description.	10-7
Table 10-3.	Memory Breakpoint 0 and 1 Select Table.	10-9
Table 10-4.	Breakpoint 0 Read/Write Select Table	10-9
Table 10-5.	Breakpoint 0 Condition Select Table	10-10
Table 10-6.	Breakpoint 1 Read/Write Select Table	10-10
Table 10-8.	Breakpoint 0 and 1 Event Select Table	10-11
Table 10-7.	Breakpoint 1 Condition Select Table	10-11
Table 10-9.	TMS Sequencing for DEBUG_REQUEST and poll the status	10-26
Table 10-10.	TMS Sequencing for ENABLE_ONCE	10-27
Table 10-11.	TMS Sequencing for reading pipeline registers	10-28
11	JTAG (IEEE 1149.1) Test Access Port	
Table 11-1.	JTAG Instructions.....	11-5
12	OPERATING MODES AND MEMORY SPACES	
Table 12-1.	DSP56300 Core reset vectors.....	12-1

List of Tables (Continued)

Table Number	Title	Page Number
-----------------	-------	----------------

Table 12-2.	DSP56300 Core operating modes	12-1
Table 12-3.	Internal X I/O Space Map.....	12-3

Appendix A INSTRUCTION SET

Table A-1.	Parallel Instructions Format	A-5
Table A-2.	NonParallel Instructions Format.....	A-5
Table A-3.	Arithmetic Instructions.....	A-9
Table A-4.	Logical Instructions	A-11
Table A-5.	Bit Manipulation Instructions	A-13
Table A-6.	Loop Instructions.....	A-13
Table A-7.	Move Instructions.....	A-15
Table A-8.	Program Control Instructions	A-15
Table A-9.	Instruction Description Notation	A-17
Table A-10.	Destination Accumulator Encoding	A-239
Table A-11.	Data ALU Operands Encoding.....	A-239
Table A-12.	Data ALU Source Operands Encoding	A-239
Table A-13.	Program Control Unit Register Encoding.....	A-239
Table A-14.	Data ALU Operands Encoding.....	A-240
Table A-15.	Data ALU operands encoding.....	A-240
Table A-16.	Effective Addressing Mode Encoding #1	A-241
Table A-17.	Memory/Peripheral Space	A-241
Table A-18.	Effective Addressing Mode Encoding #2	A-241
Table A-19.	Effective Addressing Mode Encoding #3	A-242
Table A-20.	Effective Addressing Mode Encoding #4	A-242
Table A-21.	Triple-Bit Register Encoding	A-242
Table A-22.	Six-Bit Encoding For all On-Chip Registers	A-243
Table A-23.	Long Move Register Encoding.....	A-243
Table A-24.	Data ALU Source Registers Encoding.....	A-243
Table A-25.	AGU Address and Offset Registers Encoding	A-244
Table A-26.	Data ALU Multiply Operands Encoding #1	A-244
Table A-27.	Data ALU Multiply Operands Encoding #2	A-244
Table A-28.	Data ALU Multiply Operands Encoding #3	A-244
Table A-29.	Data ALU Multiply Sign Encoding	A-244
Table A-30.	Data ALU Multiply Operands Encoding #3	A-245
Table A-31.	5-Bit Register Encoding #1	A-245
Table A-32.	Immediate Data ALU Operand Encoding.....	A-246
Table A-33.	Write Control Encoding	A-246
Table A-34.	ALU Registers Encoding.....	A-246
Table A-35.	X:R Operand Registers Encoding.....	A-247

List of Tables (Continued)

Table Number	Title	Page Number
Table A-36.	R:Y Operand Registers Encoding	A-247
Table A-37.	Single-Bit Special Register Encoding Tables	A-247
Table A-38.	X:Y: Move Operands Encoding Tables	A-248
Table A-39.	Signed/Unsigned partial encoding #1.....	A-248
Table A-40.	Signed/Unsigned partial encoding #2.....	A-249
Table A-41.	5-Bit Register Encoding.....	A-249
Table A-42.	Condition Codes Computation Equations	A-250
Table A-43.	Condition Codes Encoding.....	A-251
Table A-44.	Operation Code K0-2 Decode	A-251
Table A-45.	Nonmultiply Instruction Encoding	A-252
Table A-46.	Special Case #1	A-252

Appendix B INSTRUCTION EXECUTION TIMING

Table B-1.	Instruction Timing, Word Count and encoding	B-4
Table B-2.	Stack Extension Delays.....	B-18

Appendix C BENCHMARK PROGRAMS

1 CORE DESCRIPTION

This document describes the DSP56300 Core, a new core of Motorola's family of programmable CMOS digital signal processors. The DSP56300 is the powerful New DSP Engine (NDE) core capable of executing an instruction on every clock cycle, thus yielding a twofold performance increase as compared to the 56000 core while maintaining object code compatibility with it.

The DSP56300 core is composed of the Expansion Port and DRAM Controller, Data ALU, Address Generation Unit, Instruction Cache Controller, Program Control Unit, DMA Controller, PLL Clock Oscillator, On-chip Emulator and the Peripheral and Memory Expansion Bus.

The cost-effectiveness of the parts is the major factor in the economic success of the DSP family thus the provided solution must be low-cost but powerful enough to meet the computational demands, flexible enough to meet various demands of the customers/applications and have enough degree of integration to minimize the total system cost.

DSP applications require parts capable of very high execution speed in a real-time I/O intensive environment. The DSP56300 core with its capability of executing an instruction per clock cycle has the processing power to meet this demand.

To minimize the total system cost the DSP56300 core incorporates a versatile external memory interface that provides glueless interface to a variety of memories such as Dynamic RAMs (DRAMs), Static RAMs (SRAMs), Synchronous Static RAMs (SSRAMs) etc. by providing on-chip DRAM controller as well as chip select logic. The concurrent Six-Channel DMA Controller augments the data throughput that characterizes the DSP applications. Special attention is paid in the design stage to minimize the chip power consumption. Low power consumption is achieved both in active and in standby modes. Power consumption scale down with clock frequency reduction, use of on-chip memory, use of on-chip peripherals, and use of WAIT and STOP standby modes. External buses are driven only when required. On-chip memory expansion does not increase power dissipation significantly because only memory modules being accessed consume power.

The design priorities for the DSP56300 Core are:

1. Low-cost
2. Low-power dissipation
3. High-performance
4. High integration

DSP56300 Core Features

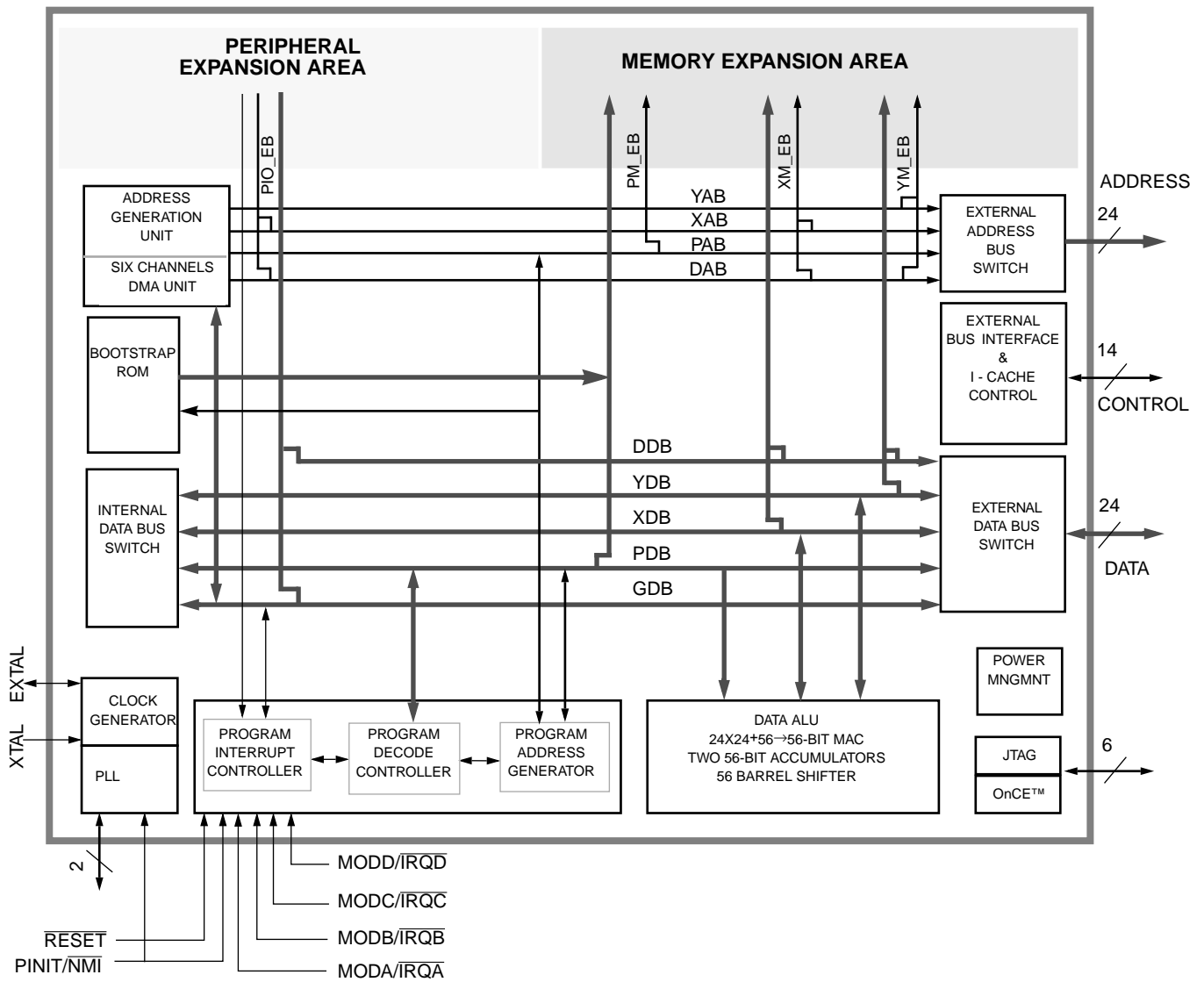
High performance CPU

- 66/80 Million Instructions per Second (Mips) with a 66/80 MHz clock
- Object Code Compatible with the 56K Core
- Fully pipelined 24 x 24 Bit Parallel Multiplier-Accumulator
- 56 Bit Parallel Barrel Shifter
- 16 Bit Arithmetic Support
- Highly Parallel Instruction Set
- Position Independent Code (PIC) support
- Unique DSP Addressing Modes
- On-Chip Memory-Expandable Hardware Stack
- Nested Hardware Do Loops
- Fast Auto-Return Interrupts
- On-Chip user-controllable Instruction Cache
- On-Chip Concurrent Six-Channel DMA Controller
- On-Chip PLL
- On-Chip Emulator (OnCE)
- Program Address Tracing Support
- JTAG port compatible with the IEEE 1149.1 Standard

Reduced power dissipation

- Very low power CMOS design
- Wait and Stop low power standby modes
- Fully-static logic, operation frequency down to DC.

Figure 1-1. DSP56300 Core Block Diagram



2 EXPANSION PORT

2.1 INTRODUCTION

The expansion port of the DSP56300 Core has the following features/functions:

- Interrupt And Mode Control
- Clock and Phase-Locked Loop (PLL)
- On-chip Emulator Interface (OnCE)/JTAG Interface
- Memory Expansion Port (Port A)
- Emulation Port

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

The port A data bus is 24 bit wide with a separate 24 bit address bus capable of sustained rate of one memory access per clock cycle for data space accesses (using synchronous static memory). External memory is divided into three 16M X 24 bit spaces - X, Y and P. An internal wait state generator can be programmed to insert up to 31 wait state if access to slower memory or I/O device is required. A bus wait signal allows an external device to control the number of wait states inserted in a bus access operation. Bus arbitration signals allow an external device use of the bus while internal operations continue using the internal memory.

2.2 EXPANSION PORT SIGNAL DESCRIPTION

2.2.1 Interrupt And Mode Control

$\overline{\text{RESET}}$	(Reset) - Active low, Schmitt trigger input. $\overline{\text{RESET}}$ is internally synchronized to the clock out (CLKOUT). When asserted, the chip is placed in the reset state and the internal phase generator is reset. The Schmitt trigger input allows a slowly rising input (such as a capacitor charging) to reliably reset the chip. If $\overline{\text{RESET}}$ is negated synchronous to the clock out (CLKOUT), exact start-up timing is guaranteed, allowing
---------------------------	---

multiple processors to start-up synchronously and operate together in “lock-step”. When the $\overline{\text{RESET}}$ pin is negated, the initial chip operating mode is latched from the MODA, MODB, MODC and MODD pins. $\overline{\text{RESET}}$ pin can tolerate 5V.

MODA/ $\overline{\text{IRQA}}$ (Mode Select A/External Interrupt Request A) - Active low Schmitt trigger input, internally synchronized to the clock out (CLKOUT). MODA/ $\overline{\text{IRQA}}$ selects the initial chip operating mode during hardware reset and becomes a level sensitive or negative edge triggered, maskable interrupt request input during normal instruction processing. MODA, MODB, MODC and MODD select one of 16 initial chip operating modes, latched into the operating mode register (OMR) when the $\overline{\text{RESET}}$ pin is negated. If $\overline{\text{IRQA}}$ is asserted synchronous to the clock out (CLKOUT), multiple processors can be re-synchronized using the WAIT instruction and asserting $\overline{\text{IRQA}}$ to exit the wait state. If the processor is in the STOP standby state and $\overline{\text{IRQA}}$ is asserted, the processor will exit the STOP state.
MODA/ $\overline{\text{IRQA}}$ pin can tolerate 5V.

MODB/ $\overline{\text{IRQB}}$ (Mode Select B/External Interrupt Request B) - Active low Schmitt trigger input, internally synchronized to the clock out (CLKOUT). MODB/ $\overline{\text{IRQB}}$ selects the initial chip operating mode during hardware reset and becomes a level sensitive or negative edge triggered, maskable interrupt request input during normal instruction processing. MODA, MODB, MODC and MODD select one of 16 initial chip operating modes, latched into the operating mode register (OMR) when the $\overline{\text{RESET}}$ pin is negated. If $\overline{\text{IRQB}}$ is asserted synchronous to the clock out (CLKOUT), multiple processors can be re-synchronized using the WAIT instruction and asserting $\overline{\text{IRQB}}$ to exit the wait state.
MODB/ $\overline{\text{IRQB}}$ pin can tolerate 5V.

MODC/ $\overline{\text{IRQC}}$ (Mode Select C/External Interrupt Request C) - Active low Schmitt trigger input, internally synchronized to the clock out (CLKOUT). MODC/ $\overline{\text{IRQC}}$ selects the initial chip operating mode during hardware reset and becomes a level sensitive or negative edge triggered, maskable interrupt request input during normal instruction processing. MODA, MODB, MODC and MODD select one of 16 initial chip operating modes, latched into the operating mode register (OMR) when the $\overline{\text{RESET}}$ pin is negated. If $\overline{\text{IRQC}}$ is asserted synchronous to the clock out (CLKOUT), multiple processors can be re-synchronized using the WAIT instruction and asserting $\overline{\text{IRQC}}$ to exit the wait state.
MODC/ $\overline{\text{IRQC}}$ pin can tolerate 5V.

MODD/ $\overline{\text{IRQD}}$ (Mode Select D/External Interrupt Request D) - Active low Schmitt

trigger input, internally synchronized to the clock out (CLKOUT). MODD/ $\overline{\text{IRQD}}$ selects the initial chip operating mode during hardware reset and becomes a level sensitive or negative edge triggered, maskable interrupt request input during normal instruction processing. MODA, MODB, MODC and MODD select one of 16 initial chip operating modes, latched into the operating mode register (OMR) when the $\overline{\text{RESET}}$ pin is negated. If $\overline{\text{IRQD}}$ is asserted synchronous to the clock out (CLKOUT), multiple processors can be re-synchronized using the WAIT instruction and asserting $\overline{\text{IRQD}}$ to exit the wait state. MODD/ $\overline{\text{IRQD}}$ pin can tolerate 5V.

2.2.2 Clock and Phase-Locked Loop (PLL)

EXTAL	(External Clock/Crystal Input) - This input connects the internal crystal oscillator input to an external crystal or an external clock.
XTAL	(Crystal Output) - This output connects the internal crystal oscillator output to an external crystal. If an external clock is used, XTAL should not be connected.
PCAP	(PLL capacitor) - This input connects the off-chip capacitor for PLL filter. One terminal of the capacitor is connected to PCAP while the other terminal is connected to PVCC.
CLKOUT	(Clock Output) - This output pin provides an output clock synchronized to the internal core clock phase.

NOTE 1: If PLL is enabled and both the multiplication and division factors are equal to one, then CLKOUT is also synchronized to EXTAL.

NOTE 2: If PLL is disabled, CLKOUT frequency and the chip frequency is half the frequency of EXTAL.

PINIT/ $\overline{\text{NMI}}$	(PLL Initial/Non Maskable Interrupt) - During the assertion of hardware reset, PINIT/ $\overline{\text{NMI}}$ is configured as PINIT and its value is written into the PEN bit of the PLL control register and determines whether the PLL is enabled or disabled. After hardware reset negation and during normal instruction processing the PINIT/ $\overline{\text{NMI}}$ Schmitt trigger input pin is configured as $\overline{\text{NMI}}$, a negative edge triggered, non maskable interrupt request, internally synchronized to the clock out (CLKOUT). PINIT/ $\overline{\text{NMI}}$ pin can tolerate 5V.
--------------------------------	--

2.2.3 On-Chip Emulator Interface (OnCE)/JTAG Interface

$\overline{\text{DE}}$	(Debug Event) - This open drain bidirectional active low pin provides, as an input, a means of entering the debug mode of operation from an external command controller, and as an output, a means of acknowledging that the chip has entered the debug mode. This pin when
------------------------	---

asserted as an input causes the DSP56300 core to finish the current instruction being executed, save the instruction pipeline information, enter the debug mode and wait for commands to be entered from the debug serial input line. This pin is asserted as an output for three clock cycles when the chip enters the debug mode as a result of a debug request or as a result of meeting a breakpoint condition.
 \overline{DE} pin can tolerate 5V.

TCK	(Test Clock) - The test clock input TCK pin is the test clock used to synchronize the JTAG test logic TCK pin can tolerate 5V.
TDI	(Test Data Input) - The test data input TDI pin is the serial input for test instructions and data. TDI is sampled on the rising edge of TCK and it has an internal pullup resistor. TDI pin can tolerate 5V.
TDO	(Test data output) - The test data output TDO pin is the serial output for test instructions and data. TDO is three-stateable and is actively driven in the shift-IR and shift-DR controller states. TDO changes on the falling edge of TCK.
TMS	(Test Mode Select) - The test mode select input (TMS) pin is used to sequence the test controller's state machine. The TMS is sampled on the rising edge of TCK and it has an internal pullup resistor TMS pin can tolerate 5V.
\overline{TRST}	(Test Reset) - This active low Schmitt trigger input pin \overline{TRST} is used to asynchronously initialize the test controller. The \overline{TRST} has an internal pullup resistor \overline{TRST} pin can tolerate 5V.

2.2.4 Expansion Port (Port A)

A0-A23	(Address Bus) - three-state. Active high outputs when a bus master, three-stated otherwise, specify the address for external program and data memory accesses. To minimize power dissipation, A0–A23 do not change state when external memory spaces are not being accessed. A0–A23 are three-stated during hardware reset.
D0-D23	(Data Bus) - three-state, active high, bidirectional input/outputs when a bus master. These pins provide the bidirectional data bus for external program and data memory accesses. D0–D23 are in the high impedance state when not a bus master, or when there is no external bus activity. They are also three-stated during hardware reset.

AA(3:0)/ $\overline{\text{RAS}}$ (3:0)(Address Attribute or Row Address Strobe) - three-state outputs with a programmable polarity. When defined as Address Attribute these signals can be used as chip selects or additional address lines. When defined as $\overline{\text{RAS}}$ these signals can be used as Row Address Strobe for DRAM interface. The AA/ $\overline{\text{RAS}}$ pins are three stated during hardware reset.

$\overline{\text{RD}}$ (Read Enable) - three-state. Active low output when bus master, three-stated otherwise. $\overline{\text{RD}}$ is asserted to read external memory on the data bus (D0–D23). $\overline{\text{RD}}$ is three-stated during hardware reset.

$\overline{\text{WR}}$ (Write Enable) - three-state. Active low output when bus master, three-stated otherwise. $\overline{\text{WR}}$ is asserted to write external memory on the data bus (D0–D23). $\overline{\text{WR}}$ is three-stated during hardware reset.

$\overline{\text{TA}}$ (Transfer Acknowledge) - active low input. If the DSP56300 core is the bus master and there is no external bus activity or the DSP56300 core is not the bus master, the $\overline{\text{TA}}$ input is ignored. The $\overline{\text{TA}}$ input is a synchronous/asynchronous (according to TAS bit in the OMR register) “DTACK” function which can extend an external bus cycle indefinitely. Any number of wait states (1, 2,..., infinity) may be added to the wait states inserted by the BCR by keeping $\overline{\text{TA}}$ negated. In typical operation, $\overline{\text{TA}}$ is negated at the start of a bus cycle, is asserted to enable completion of the bus cycle and is negated before the next bus cycle. The current bus cycle completes one clock period after $\overline{\text{TA}}$ is asserted synchronous to CLKOUT. The number of wait states is determined by the $\overline{\text{TA}}$ input or by the Bus Control Register (BCR), whichever is longer. The BCR can be used to set the minimum number of wait states in external bus cycles. If $\overline{\text{TA}}$ is tied low (asserted) and no wait states are specified in the BCR register, zero wait states will be inserted into external bus cycles.

NOTE1 In order to use the $\overline{\text{TA}}$ functionality the BCR must be programmed to at least one wait state, a zero wait state access can not be extended by negating $\overline{\text{TA}}$, Otherwise improper operation may result.

NOTE2 $\overline{\text{TA}}$ functionality may not be used while performing DRAM type accesses, Otherwise improper operation may result.

$\overline{\text{BR}}$ (Bus Request) - active low output, never three-stated. $\overline{\text{BR}}$ is asserted when the CPU or DMA is requesting bus mastership. $\overline{\text{BR}}$ is negated when the CPU or DMA no longer needs the bus. $\overline{\text{BR}}$ may be asserted or negated independent of whether the DSP56300 Core is a bus master or a bus slave. Bus “parking” allows $\overline{\text{BR}}$ to be negated even though the DSP56300 Core is the bus master. See the description of bus “parking” in the $\overline{\text{BB}}$ pin description. The BRH bit in the Bus Control Register (Section 2.5.2) allows $\overline{\text{BR}}$ to be asserted under software control even

though the CPU or DMA does not need the bus. \overline{BR} is typically sent to an external bus arbitrator which controls the priority, parking and tenure of each DSP56300 Core on the same external bus. \overline{BR} is only affected by CPU or DMA requests for the external bus, never for the internal bus. During hardware reset, \overline{BR} is negated and the arbitration is reset to the bus slave state.

\overline{BG} (Bus Grant) – active low input. \overline{BG} must be asserted/negated synchronous to the clock out (CLKOUT) for proper operation. \overline{BG} is asserted by an external bus arbitration circuit when the DSP56300 Core may become the next bus master. When \overline{BG} is asserted, the DSP56300 Core must wait until \overline{BB} is negated before taking bus mastership. When \overline{BG} is negated, bus mastership is typically given up at the end of the current bus cycle. This may occur in the middle of an instruction which requires more than one external bus cycle for execution. \overline{BG} is ignored during hardware reset.

\overline{BB} (Bus Busy) - bidirectional active low input/output, must be asserted and negated synchronous to the clock out (CLKOUT). This signal indicates that the bus is active. Only after this signal is negated the pending bus master can become the bus master (and then assert it again). The bus master may keep \overline{BB} asserted after ceasing bus activity regardless of whether \overline{BR} is asserted or negated, this is called “bus parking” and allows the current bus master to reuse the bus without re-arbitration until other device wants the bus. The negation of \overline{BB} is done by an “active pull-up” method i.e. \overline{BB} is driven high and then released and held high by an external pull-up resistor. \overline{BB} is an active low input during hardware reset.

NOTE: \overline{BB} requires an external pullup resistor.

\overline{BL} (Bus Lock) - active low output, never three-stated. Asserted at the start of an external indivisible Read-Modify-Write (RMW) bus cycle and negated at the end of the write bus cycle. \overline{BL} remains asserted between the read and write bus cycles of the RMW bus sequence. \overline{BL} may be used to “resource lock” an external multi-port memory for secure semaphore updates. The only instructions which automatically assert \overline{BL} are BSET, BCLR or BCHG instruction which accesses external memory. \overline{BL} can also be asserted by setting the BLH bit in the BCR register (see Section 2.5.2). \overline{BL} is negated during hardware reset.

\overline{BS} (Bus Strobe) - three-state. Active low output when a bus master, three-stated when not a bus master. Asserted at the start of a bus cycle (for half of a clock cycle) providing an “early bus start” signal for a bus

controller. If the external bus is not used during an instruction cycle \overline{BS} remains negated until the next external bus cycle. \overline{BS} is three-stated during hardware reset.

\overline{CAS} (Column Address Strobe) - active low output when bus master and three stated otherwise (if BME bit in DRAM control register is cleared), \overline{CAS} is used by DRAM memories to strobe column address. \overline{CAS} is three stated during hardware reset.

BCLK (Bus Clock) - three-state. Active high output when a bus master BCLK is used by synchronous SRAM to sample address, data and control signals. BCLK is active only during SSRAM accesses, when active BCLK is synchronized to CLKOUT by the internal Phase Lock Loop, BCLK precedes CLKOUT by 1/4 of a clock cycle. BCLK is three stated during hardware reset.

2.3 EXPANSION PORT OPERATION

The external bus timing is defined by the operation of the Address Bus, Data Bus and Bus Control pins described in the previous paragraph. The DSP56300 Core external ports are designed to interface with a wide variety of memory and peripheral devices, high speed synchronous static RAMs, high speed static RAMs and dynamic RAMs, as well as slower memory devices. For detailed explanation see the paragraphs on synchronous static RAM support, static RAM support and dynamic RAM support.

2.3.1 Static RAM support

External bus timing is controlled by the \overline{TA} control signal and by the Bus Control Register (BCR) that is described in Section 2.5.2. Insertion of wait states is controlled by the BCR to provide constant bus access timing, and by \overline{TA} to provide dynamic bus access timing. The number of wait states for each external access is determined by the \overline{TA} input or by the BCR, whichever is longer.

The external memory address is defined by the Address Bus A0-A23 and the Memory Address attribute signals AA(3:0). The Address Attribute signals have the same timing as the Address Bus and may be used as additional address lines. The Address Attribute signals are also used to generate chip select signals for the appropriate memory chips. These chip select signals change the memory chips from low power standby mode to active mode and begin the access time. This allows slower memories to be used since the Address Attribute signals are address-based rather than read or write enable-based.

2.3.1.1 Synchronous Static RAM (SSRAM) Support

Synchronous Static RAM devices can be easily interfaced to the DSP56300 Core bus timing. The Synchronous Static RAMs internal pipeline fits the DSP56300 Core pipeline, and therefore permits high speed data transfers (each clock cycle, zero wait states) with

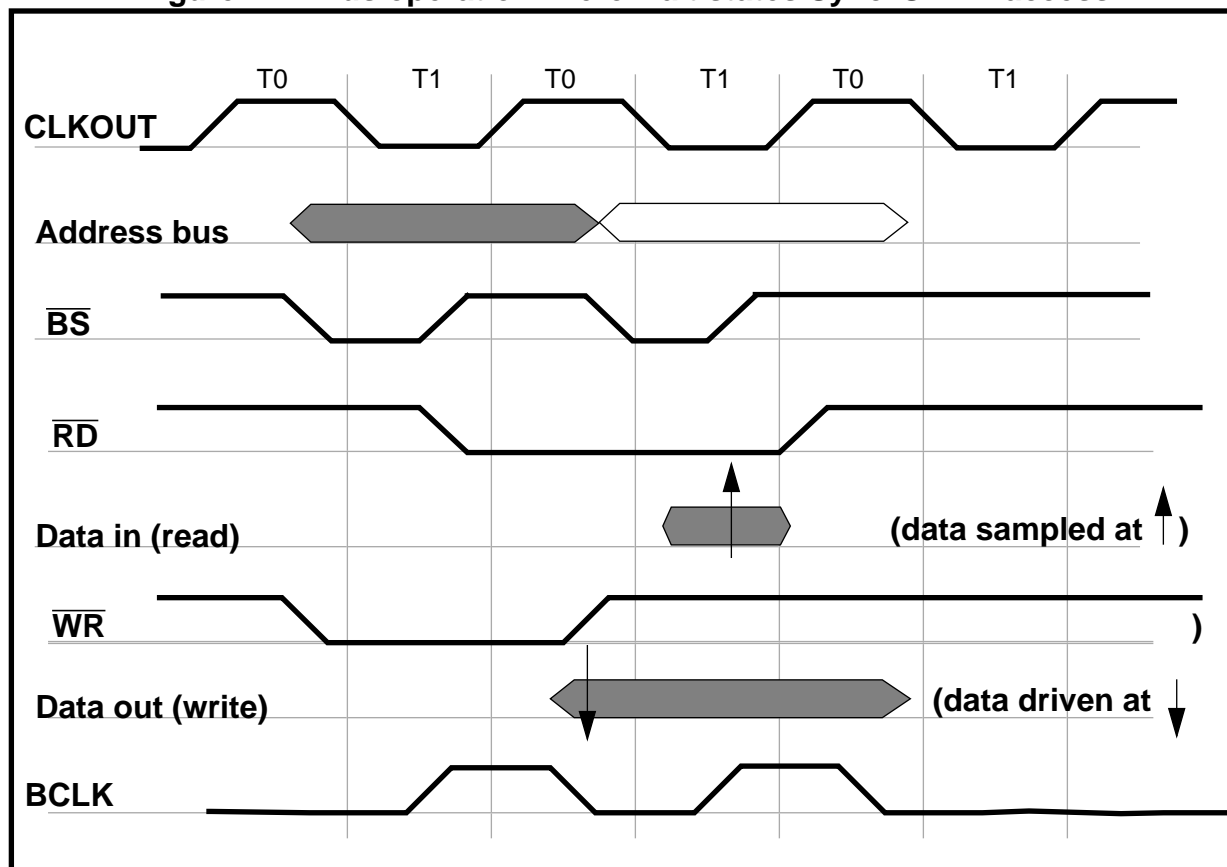
a reasonable access time. Due to the DSP56300 Core pipeline structure, one cycle stall is inserted after performing fetch from external SSRAM. In such a case, the effective number of stall states in the pipeline will be the number specified in the Bus Control Register (BCR) + 1 (although the external access itself will be performed exactly the same for fetches and for data moves).

Figure 2-3 shows a connection configuration (for a detailed timing information see the specific DSP56300 Core based chip technical data sheet). The synchronous SRAM access is composed from the following steps:

1. The address - $A(23:0)$, address attributes - $AA(3:0)$, bus strobe - BS , and write enable - \overline{WR} are asserted before the rising edge of BCLK. (\overline{WR} only for write accesses, for read accesses \overline{RD} is asserted asynchronously to BCLK).
2. Bus strobe is negated before the change of the new address thus enabling the sample of address and control (for devices that do not use BCLK).
3. The address - $A(23:0)$, address attributes - $AA(3:0)$, and write enable - \overline{WR} are negated (for write accesses) with the falling edge of BCLK (new address and controls are driven if another external SSRAM access is needed).
4. **For write operation:** Data is driven with the falling edge of BCLK
For read operation: Data is sampled with the leading edge of BCLK.

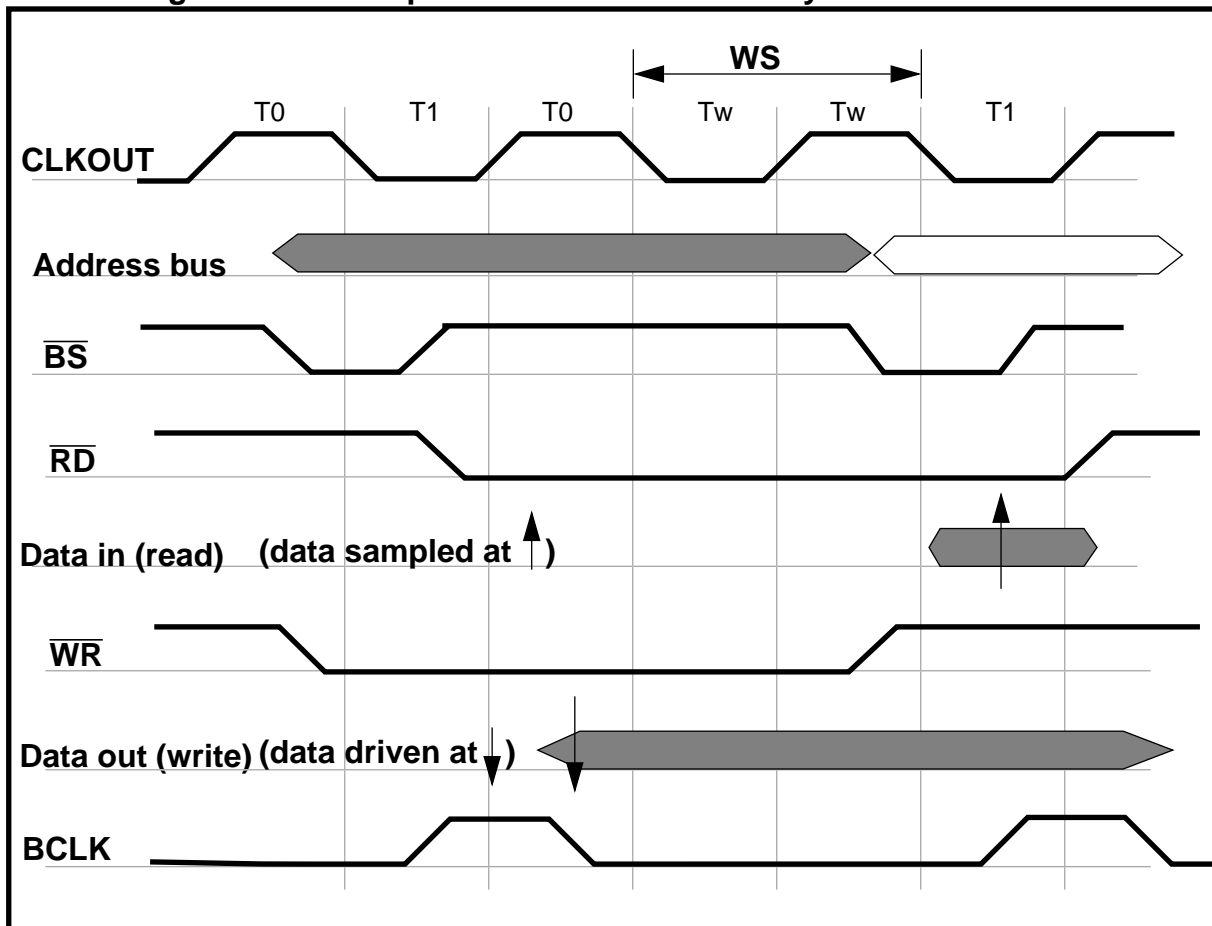
Wait states (from BCR or by \overline{TA} signal) will postpone the appearance of the next leading edge of BCLK thus increasing memory access time (see Figure 2-2 on page 2-10).

Figure 2-1. Bus operation - zero wait states Sync. SRAM access



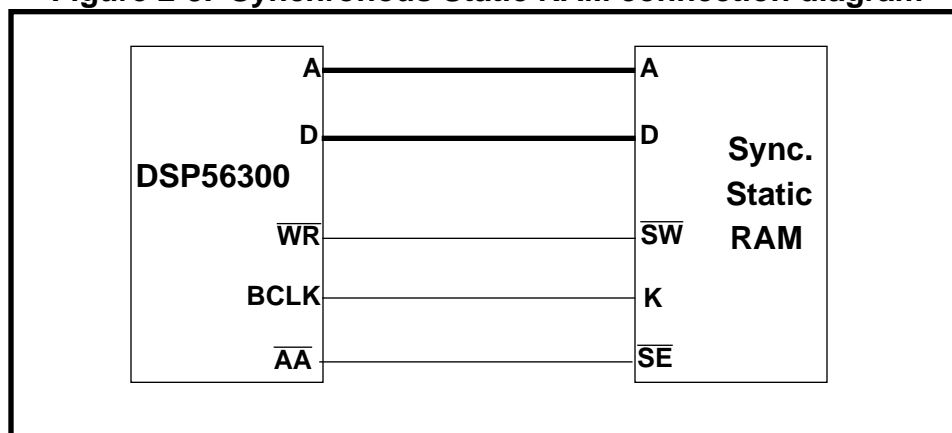
- for detailed timing specification see the specific data sheet

Figure 2-2. Bus operation - one wait state Sync. SRAM access



- for detailed timing specification see the specific data sheet

Figure 2-3. Synchronous Static RAM connection diagram



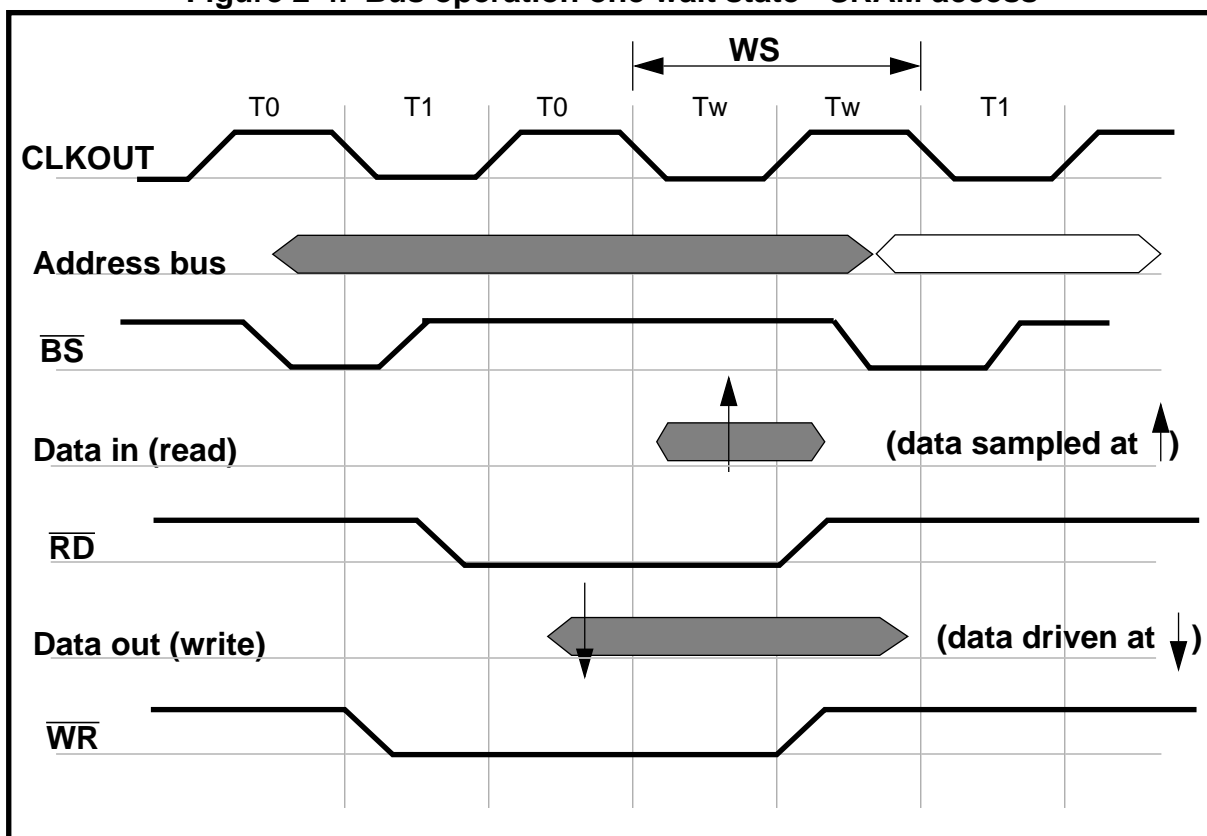
2.3.1.2 Asynchronous Static RAM (SRAM) Support

Static RAMs can be easily interfaced to the DSP56300 Core bus timing. Due to the Static RAM requirement to keep the address stable during the entire bus cycle, at least one wait state must be inserted to the bus operation. The next diagram shows a possible configuration (for a detailed timing information see the specific DSP56300 Core based chip technical data sheet). The static RAM access is composed from the following steps (see also Figure 2-4 on page 2-11)

1. The address - $A(23:0)$, address attributes - $AA(3:0)$, and bus strobe - \overline{BS} are asserted in the middle of CLKOUT high phase.
2. Write enable - \overline{WR} is asserted with the falling edge of CLKOUT (for a single wait state access). Read enable - \overline{RD} , is asserted in the middle of CLKOUT low phase.
3. **For write operation:** Data is driven in the middle of CLKOUT high phase.
For read operation: Data is sampled in the middle of CLKOUT last low phase of the external access.

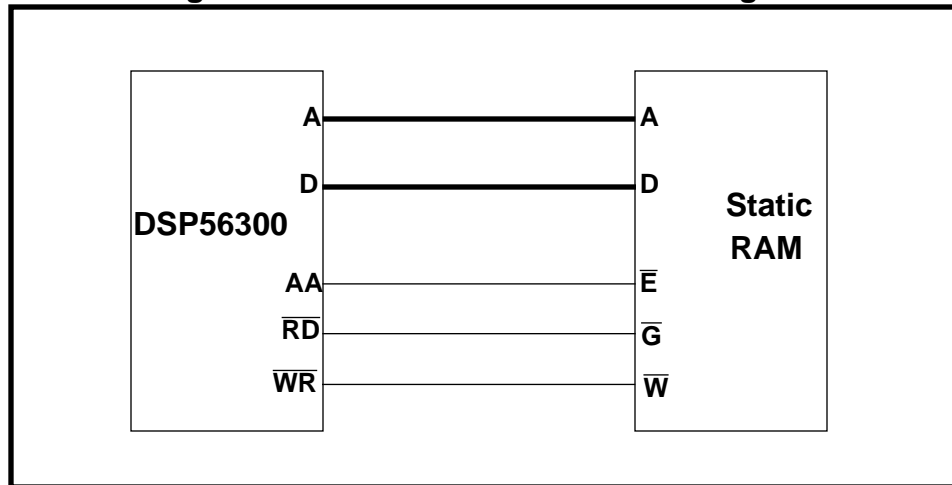
Wait states (from BCR or by \overline{TA} signal) will postpone the disappearance of the external address thus increasing memory access time. In any case, static RAM access requires at least one wait state.

Figure 2-4. Bus operation one wait state - SRAM access



- for detailed timing specification see the specific data sheet

Figure 2-5. Static RAM connection diagram



NOTE 1: BCLK is negated during asynchronous SRAM access and therefore all signals timing are related to CLKOUT.

NOTE 2: When the external access type is defined as SRAM, the assertion of \overline{WR} signal depends on the number of wait states programmed in the BCR. If a single wait state is programmed in the BCR, \overline{WR} signal is asserted with the falling edge of CLKOUT. If the number of wait states programmed is 2 or 3, \overline{WR} assertion is delayed by half of a clock cycle (half CLKOUT cycle). If the number of wait states programmed is 4 or more, \overline{WR} assertion is delayed by a full clock cycle. This feature enables the connection of slow external devices that require long address setup time before write assertion in order to prevent false write.

2.3.2 Dynamic Memories Support

External bus timing is controlled by the DRAM Control Register (DCR) that is described in Section 2.6.1. Insertion of wait states is controlled by the DCR to provide constant bus access timing.

The external memory address is defined by the Address Bus A0-A23. The n low order address bits are multiplexed inside the DSP56300 Core, and the new 24 bits address is driven to the external bus. The address multiplexing enable glue-less interface to dynamic memories by simply connecting the low order n bits to the memory address pins. The Address Attribute signals function as \overline{RAS} . An in page access is assumed and therefore \overline{RAS} is kept asserted unless one of the following occurs:

1. An out of page access is detected.
2. An access to another bank of dynamic memory is attempted.
3. A refresh access is attempted (\overline{CAS} before \overline{RAS}).

-
4. A write to of the following registers is detected: BCR, DCR, AAR3, AAR2, AAR1, AAR0.
 5. A lost of bus mastership is detected while the BME bit in the DCR register is cleared.
 6. Wait or stop instruction are detected.
 7. Hardware or software reset are detected.

Modern dynamic memory (DRAM) are becoming the preferred choice for a wide variety of computing systems based on

1. Cost per bit due to dynamic storage cell density.
2. Packaging density due to multiplexed address and control pins.
3. Improved price-performance relative to static RAMs due to fast access mode (page mode).
4. Commodity pricing due to high volume production.

Port A bus control signals are designed for efficient interface to DRAM devices in both random read/write cycles and fast access mode (page mode). An on-chip DRAM controller controls the page hit circuit, address multiplexing (row address and column address), control signal generation ($\overline{\text{CAS}}$ and $\overline{\text{RAS}}$) and refresh access generation ($\overline{\text{CAS}}$ before $\overline{\text{RAS}}$) for a large variety of DRAM module sizes and different access times. The DRAM controller operation and programming is described in Section 2.6. The next diagram shows a possible configuration (for a detailed timing information see the specific DSP56300 Core based chip technical data sheet). The dynamic RAM access is composed from the following steps (in page access):

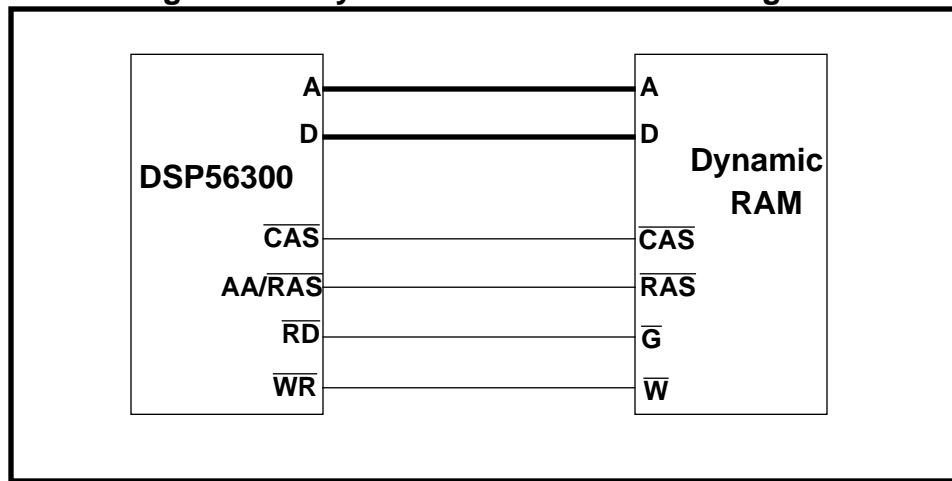
1. The column address - A(23:0), and bus strobe - BS are asserted in the middle of CLKOUT high phase.
2. Write enable - WR, or read enable - RD are asserted with the falling edge of CLKOUT.
3. $\overline{\text{CAS}}$ assertion timing depends on the number of in page wait states selected by BCW bits in DCR register and on the access purpose (read/write). (See Figure 2-7 on page 2-14 for DRAM in page 2 w.s example).
4. $\overline{\text{CAS}}$ is negated before the end of the external access in order to meet the $\overline{\text{CAS}}$ precharge timing.

In any case, DRAM access requires at least one wait state.

Out of page access: The out of page access will start with the negation of $\overline{\text{RAS}}$, the assertion of the control signals ($\overline{\text{WR/RD}}$) and after $\overline{\text{RAS}}$ precharge time the assertion of RAS. RAS assertion, and $\overline{\text{CAS}}$ timing, depend on the number of out of page wait states selected by BRW bit in DCR register.

NOTE: The 56300 Core does not support external DRAM devices overlapping, i.e. two or more external DRAM devices, connected to different AA pins, with common addresses.

Figure 2-6. Dynamic RAM connection diagram



- Address line are multiplexed inside the DSP56300 Core

Figure 2-7. Bus operation two wait states - DRAM read access (in-page)

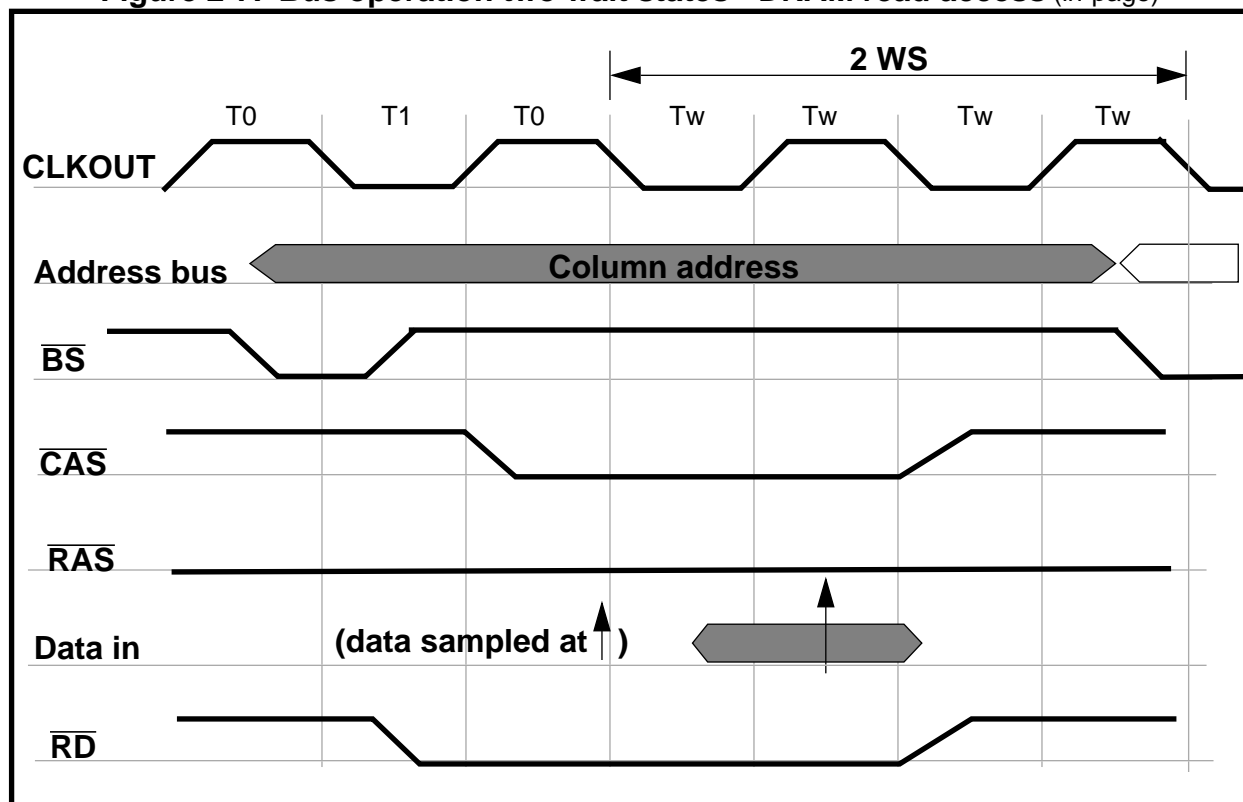
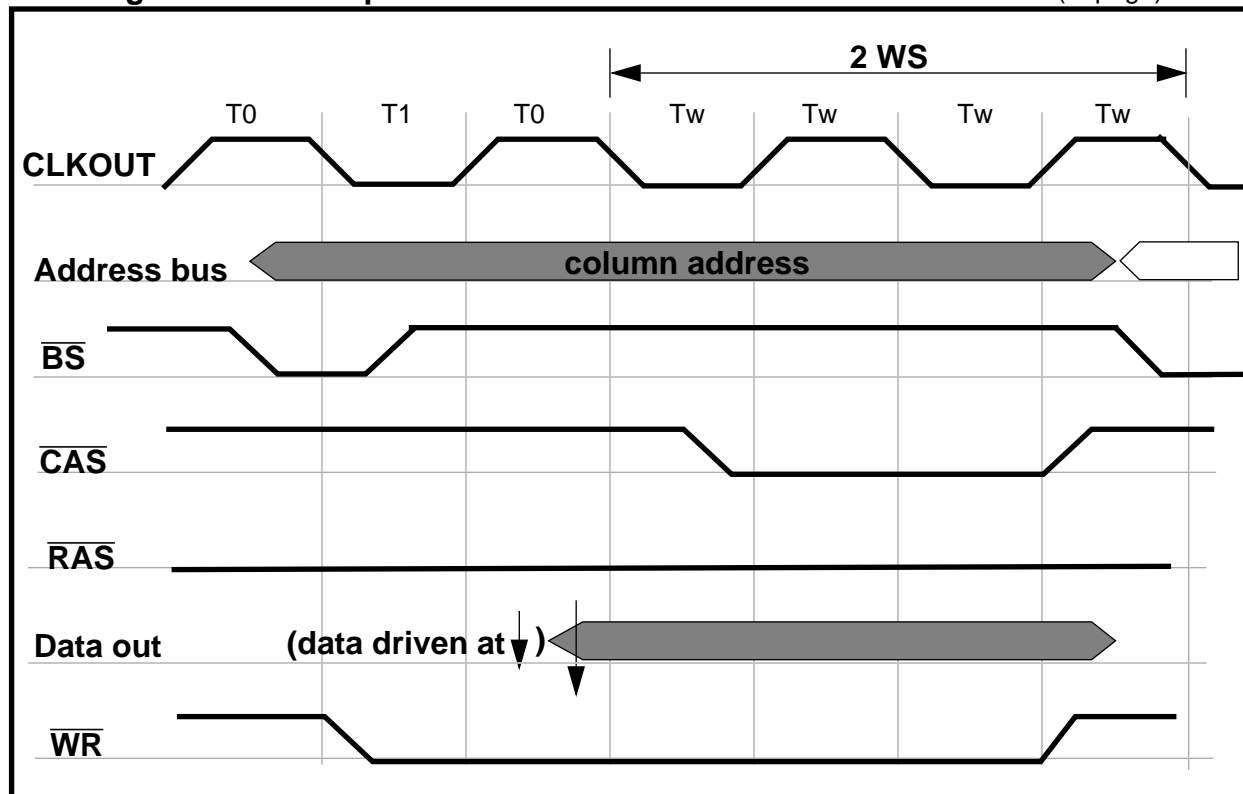


Figure 2-8. Bus operation two wait states - DRAM write access (in-page)



- for detailed timing specification see the specific data sheet

2.3.3 Expansion Port Stalls

In addition to the wait states that are controlled by the BCR, DCR and the \overline{TA} negation there are two more cases where the expansion port controller stalls the DSP56300 pipeline.

2.3.3.1 External Fetch From Synchronous SRAM.

Due to the DSP56300 pipeline any external fetch from Synchronous SRAM will add one cycle stall. This one cycle stall will be inserted even if the synchronous SRAM already needs wait states for access time. The instruction cache enable/disable has no effect on this stall.

2.3.3.2 Non Synchronous SRAM Access Immediately Following Synchronous SRAM access.

Due to the synchronous SRAM pipelined access there is a possibility of contention on the data bus in a case of non synchronous SRAM (DRAM or asynchronous SRAM) access immediately following a synchronous SRAM access. This sequence is automatically detected by the expansion port control hardware, and a one cycle stall is inserted in order to avoid contention.

In case of a default area (always SRAM) access immediately following a synchronous SRAM access, the user should be careful that the SRAM will not be activated in the

second cycle of the synchronous SRAM access, because the select of the default area SRAM is generated externally (not one of the AA signals).

2.3.4 Expansion port Disable

In many application that are sensitive to the power consumption there is no use of the expansion port because all the memory reside inside the chip itself. A special feature of the expansion port controller enables the user to reduce significantly the power consumption of the expansion port controller by setting the EBD bit in the OMR register. If this bit is set the expansion port controller is disabled, the DSP56300 will release the bus i.e. negate \overline{BR} and \overline{BL} , tristate \overline{BB} , and ignore \overline{BG} . Of course no external DMA accesses or refresh accesses can be performed. When EBD is set the user should not attempt to access the external memory, otherwise improper operation will result. Likewise, before EBD bit is set, the user should clear BREN (Refresh Enable - bit 13 in DCR) to prevent a refresh attempt to external DRAM, otherwise improper operation will result.

2.4 BUS HANDSHAKE AND ARBITRATION

Bus transactions are governed by a single bus master. Bus arbitration determines which device becomes the bus master. The arbitration logic implementation is system dependent, but must result in at most one device becoming the bus master (even if multiple devices request bus ownership). The arbitration signals permit simple implementation of a variety of bus arbitration schemes (e.g. fairness, priority, etc.). External logic must be provided by the system designer to implement the arbitration scheme.

2.4.1 Bus Arbitration Signals

Three signals are provided for bus arbitration. Two of them are considered as local arbitration signals and one as system arbitration signal. The local arbitration signals run between a potential bus master and the arbitration logic. The local signals are \overline{BR} and \overline{BG} . \overline{BB} is a system arbitration signal. These signals are described below.

\overline{BR}	Bus Request - Asserted by the requesting device to indicate that it wants to use the bus, and it is held asserted until the device no longer needs the bus. This includes time when it is the bus master as well as when it is not the bus master.
\overline{BG}	Bus Grant - Asserted by the bus arbitration controller to signal the requesting device that it is the bus master elect. \overline{BG} is valid only when the bus is not busy (Bus Busy signal - \overline{BB} is described below).

\overline{BB} Bus Busy - The system arbitration signal \overline{BB} is monitored by all potential bus masters and is driven by the current bus master. This signal controls the hand-over of bus ownership by the bus master at the end of bus possession. \overline{BB} is an active pull-up signal i.e. it is driven high before it is released (and then held high by an external pull-up resistor).

2.4.2 The Arbitration Protocol

The bus is arbitrated by a central bus arbitrator, using individual request/grant lines to each bus master. The arbitration protocol can operate in parallel with bus transfer activity so that the bus hand-over can be made without much performance penalty.

The arbitration sequence occurs as follows:

1. All candidates for bus ownership assert their respective \overline{BR} signals as soon as they need the bus.
2. The arbitration logic designates a bus master-elect by asserting the \overline{BG} signal for that device.
3. The master-elect tests \overline{BB} to ensure that the previous master has relinquished the bus. If \overline{BB} is negated, then the master-elect asserts \overline{BB} , which designates the device as the new bus master. If a higher priority bus request occurs before the \overline{BB} signal was negated, then the arbitration logic may replace the current master-elect with the higher priority candidate. However, only one \overline{BG} signal must be asserted at one time.
4. The new bus master begins its bus transfers after the assertion of \overline{BB} .
5. The arbitration logic signals the current bus master to relinquish the bus by negating \overline{BG} at any time. An DSP56300 Core bus master releases its ownership (drives \overline{BB} high and then release it) after completing the current external bus access except for the cases described in **NOTE2**. If an instruction is executing a Read-Modify-Write external access, an DSP56300 Core master asserts the \overline{BL} signal and will only relinquish the bus (and negate \overline{BL}) after completing the entire Read-Modify-Write sequence. When the current bus master release \overline{BB} , it first drives the \overline{BB} signal high and then the \overline{BB} signal is held by the pull-up resistor. The next bus master-elect has received its \overline{BG} signal and is waiting for \overline{BB} to be negated before claiming ownership
6. The possession of the bus by the new bus master is done by asserting the \overline{BB} signal.

The DSP56300 Core has 2 control bits and one status bit, located in the Bus Control Register (BCR - see Section 2.5.2) to permit software control of the \overline{BR} and \overline{BL} signals, and to verify when the chip is the bus master. If the BRH bit in the BCR register is cleared, the DSP56300 Core asserts its \overline{BR} signal only as long as requests for bus transfers are pending or being attempted. If the BRH bit is set, \overline{BR} will remain asserted. If the BLH bit in the BCR register is cleared, the DSP56300 Core asserts its \overline{BL} signal only during a read-modify-write bus access. If the BLH bit is set, \overline{BL} will remain asserted (even when not a bus master).

The DSP56300 core has a control bit located in the Operating Mode Register (BRT in OMR register) that enable fast/slow bus release mode. In fast bus release mode all port A pins are three stated in the same cycle. In slow bus release mode an extra cycle is add, all port A pins except \overline{BB} are released first and only in the next cycle \overline{BB} is released. Therefore, in slow mode it is guaranteed that \overline{BB} is the last pin that is three stated. This may be useful in systems where a possibility of contention exists. More detailed explanation (including timing diagrams) may be found in the data sheet.

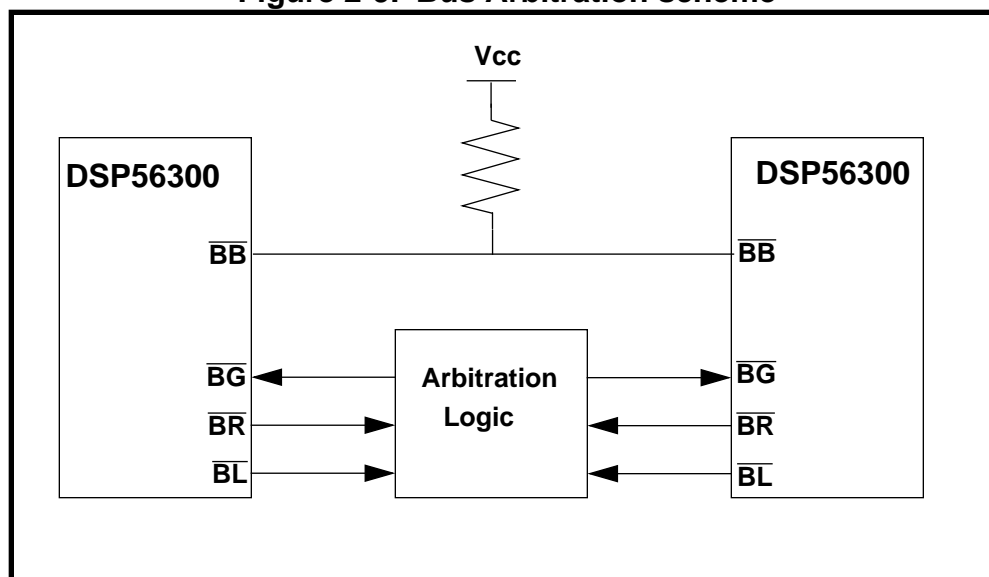
NOTE1 During the execution of WAIT and STOP instructions the DSP56300 will release the bus (i.e. negate \overline{BR} and \overline{BB}), and ignore \overline{BG} .

NOTE2 The three packing accesses, the two accesses of a read-modify-write instruction (BSET, BCLR, BCHG) and the up to four fetch burst accesses are treated as one access form an arbitration point of view, i.e. the bus mastership will not be released during the execution of these accesses.

2.4.3 Arbitration Scheme

The bus arbitration scheme is implementation dependent. The diagram in Figure 2-9 on page 2-18 illustrates a common method of implementing the bus arbitration scheme. The arbitration logic determines the device priorities and assigns bus ownership depending on those priorities. An implementation of a bus arbitration scheme may hold \overline{BG} asserted, for example, to the current bus owner if none of the other devices are requesting the bus. As a consequence, the current bus master may keep \overline{BB} asserted after ceasing bus activity, regardless of whether \overline{BR} is asserted or negated. This situation is called “bus parking” and allows the current bus master to use the bus repeatedly without re-arbitration until some other device requests the bus.

Figure 2-9. Bus Arbitration scheme



2.4.4 Bus Arbitration Example Cases

2.4.4.1 Case 1 – Normal

If the device requesting mastership asserts \overline{BR} , the arbiter asserts the requesting devices \overline{BG} and \overline{BB} is driven high and then released, indicating the bus is not busy. The requesting device will assert \overline{BB} .

2.4.4.2 Case 2 – Bus Busy

If the device requesting mastership asserts \overline{BR} , the arbiter responds by asserting the requesting devices \overline{BG} , however, the bus is busy because \overline{BB} is asserted. The requesting device will not assert \overline{BB} until \overline{BB} is driven high and then released by the current bus master.

2.4.4.3 Case 3 – Low Priority

If the device requesting mastership asserts \overline{BR} , the arbiter withholds asserting the requesting devices \overline{BG} because a higher priority device requested the bus. \overline{BB} of the requesting device will not be asserted.

2.4.4.4 Case 4 – Default

If a device does not request the bus and the arbiter, by design (i.e. default), asserts \overline{BG} and \overline{BB} is negated indicating the bus is not busy. The granted device will assert \overline{BB} . If the bus arbiter leaves \overline{BG} asserted because other requests are not pending, then \overline{BB} will remain asserted. This condition is called bus parking and eliminates the need for the default bus master to re-arbitrate for the bus during its next external access.

2.4.4.5 Case 5 – Bus Lock during RMW

If the device requesting mastership asserts \overline{BR} and the arbiter asserts the requesting devices \overline{BG} and \overline{BB} is negated, then the requesting device will assert \overline{BB} . If a read-modify-write (RMW) instruction which accesses external memory is being executed, and the bus arbiter negates \overline{BG} , then \overline{BB} will remain asserted until the entire RMW instruction completes execution. \overline{BB} will then be driven high and released thereby relinquishing the bus. Note that during external RMW instruction execution, \overline{BL} is asserted. In general, the \overline{BL} signal can be used to ensure that a multiport memory can only be written by one master at a time.

2.4.4.6 Case 6 – Bus Park

The device requesting mastership asserts \overline{BR} , the arbiter asserts the requesting devices \overline{BG} and \overline{BB} is negated indicating the bus is not busy – the requesting device will assert \overline{BB} . When the requesting device no longer requires the bus it will negate \overline{BR} . If the bus arbiter leaves \overline{BG} asserted because other requests are not pending, then \overline{BB} will remain asserted. This condition is called bus parking and eliminates the need for the last bus master to re-arbitrate for the bus during its next external access.

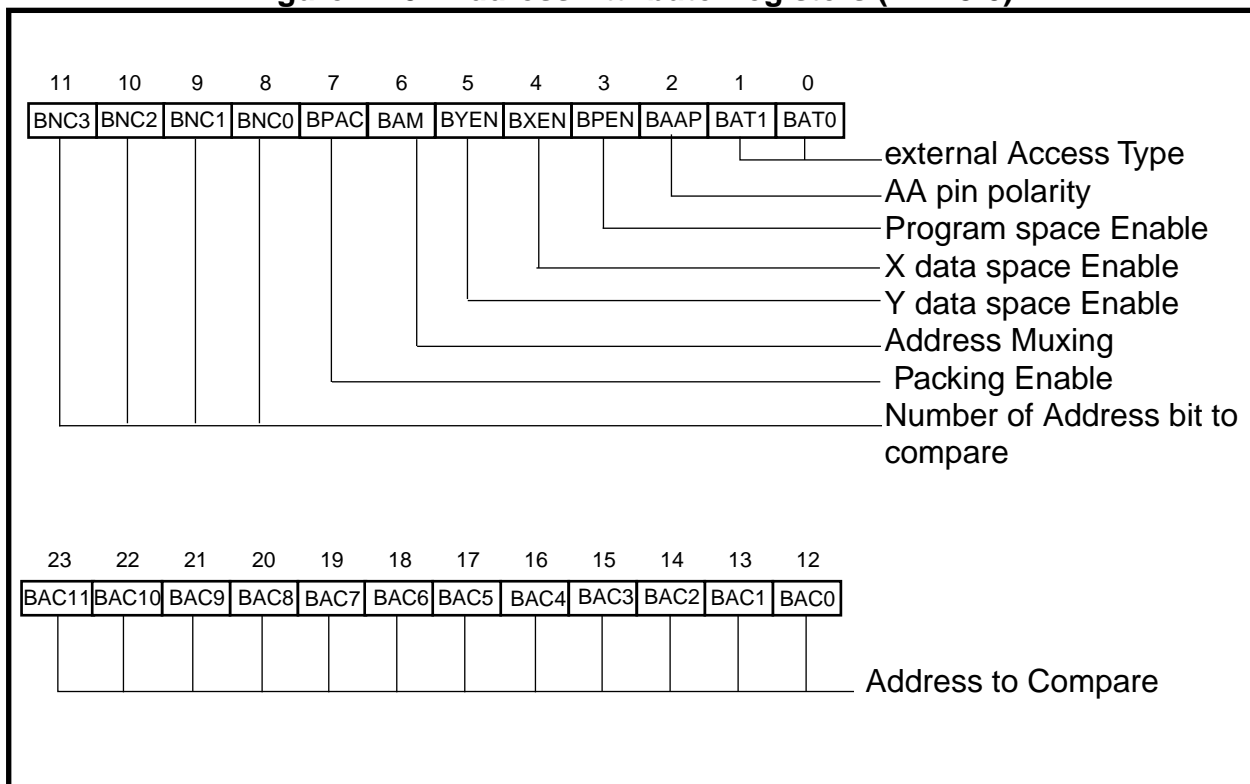
2.5 EXPANSION PORT CONTROL

The expansion port control consist of 4 Address Attribute Registers, DRAM control register and the Bus Control Register.

2.5.1 AA control Registers (one for each AA pin)

The **four** control registers (AAR3, AAR2, AAR1, AAR0) are 24 bit read write registers used to control the activity of the AA3-0/ $\overline{\text{RAS}}$ 3-0 pins. An AA/ $\overline{\text{RAS}}$ pin is asserted if the address in his appropriate AAR register (BAC bits) matches the external address (the exact number of address bits that are compared is determined by BNC bits) and if the external access is aimed to a space (X Y or P) that is enabled in the appropriate AAR register. All AAR registers are disabled (all the AAR bits are cleared) during hardware reset. The AAR bits are shown in the following figure and described in the following paragraphs.

Figure 2-10. Address Attribute Registers (AAR3-0)



NOTE 1 A priority mechanism exists among the four AAR control registers in order to resolve selection conflicts. AAR3 has the highest priority and AAR0 has the lowest priority, (e.g. if the external address matches the address and the space that is specified in both AAR1 and AAR2, the

external access type will be selected according to the AAR2 register)
The priority mechanism allows continues partition of the external address space.

NOTE 3 When the AA/ $\overline{\text{RAS}}$ pin functions as AA pin, it is negated at the start of the next clock cycle only if there is no external access that use the same AA pin (i.e. the AA pin will be kept asserted in a sequence of two consecutive external accesses that access the same memory bank). This method enables the use of low power standby mode in the external memories (these memories should be accessed first by a dummy access)

NOTE 4 The programmer should guarantee an AAR register is not changed while accessing the memory selected by this AAR, otherwise improper operation may result.

NOTE 5 A write operation to any AAR register will cause the DRAM controller to invalidate the page logic and will force the next DRAM access to be an out of page access.

2.5.1.1 BAT(1:0) - External Access Type and pin definition- bits 1-0

The read/write control bits BAT(1:0) define the external access type (DRAM, SRAM or SSRAM) to the area defined by BAC(11:0),BYEN, BXEN and BPEN bits. The encoding of BAT1 - BAT0 is described in the following table.

BAT1	BAT0	external access type
0	0	Synchronous SRAM access
0	1	Static RAM access
1	0	DRAM access
1	1	Reserved

When the external access type is defined as DRAM access ($BAT(1:0) = 10$) the AA/\overline{RAS} pin will act as a \overline{RAS} pin, otherwise it will act as a AA pin. External accesses to the default area will be always executed as if $BAT(1:0)$ of the default area equals 01, i.e. static RAM access.

$BAT(1:0)$ bits are cleared during hardware reset.

2.5.1.2 BAAP - AA pin Polarity - bit 2

The read/write control bit BAAP defines whether the AA/\overline{RAS} pin is an active low or an active high pin. When BAAP is cleared the AA/\overline{RAS} pin is an active low pin (useful for enabling memory modules, or for DRAM row address strobe), if BAAP is set the appropriate AA/\overline{RAS} pin is a active high pin (useful as additional address bit).

BAAP bit is cleared during hardware reset.

2.5.1.3 BPEN - Program space Enable - bit 3

The read/write control bit BPEN defines whether the AA/\overline{RAS} pin and logic should be activated during external program space accesses. BPEN when set enables the comparison of the external address to the BAC bits during external program space accesses. If BPEN is cleared no comparison of address is performed, during external Program space accesses.

BPEN bit is cleared during hardware reset.

2.5.1.4 BXEN - X data space Enable - bit 4

The read/write control bit BXEN defines whether the AA pin and logic should be activated during external X data space accesses. BXEN when set enables the comparison of the external address to the BAC bits during external X data space accesses. If BXEN is cleared no comparison of address is performed, during external X data space accesses.

BXEN bit is cleared during hardware reset.

2.5.1.5 BYEN - Y data space Enable - bit 5

The read/write control bit BYEN defines whether the AA pin and logic should be activated during external Y data space accesses. BYEN when set enables the comparison of the external address to the BAC bits during external Y data space accesses. If BYEN is cleared no comparison of address is performed during external Y data space accesses.

BYEN bit is cleared during hardware reset.

2.5.1.6 BAM - Address Muxing - bit 6

The read/write control bit BAM defines whether the 8 least significant bits of the address will appear on A7-A0 pins (LS portion of the external address bus) or on A23-A16 pins (MS portion of the external address bus). When BAM is set, the 8 LS bits will appear on A23-A16 pins. When BAM is cleared, the address will appeared normally and will occupy the entire external address bus (A23-A0). This feature enables to connect an external

peripheral to the most significant bits of the address thus decreasing the load on the LSP of the external address and enables more efficient interface to external memories. BAM is ignored during DRAM access (BAT1=1).
BAM bit is cleared during hardware reset.

2.5.1.7 BPAC- Packing Enable - bit 7

The read/write control bit BPAC enables (when set) the internal packing/unpacking logic. In this mode each DMA external access will initiate three external accesses to an eight bits wide external memory (the address of these accesses will be the original DMA address - the DAB, then DAB+1 and then DAB+2). The packing to a 24 bits word (or the unpacking from 24 bits word to 3 eight bits words) is done automatically by the expansion port control hardware. The external memory should reside in the eight least significant bits of the external data bus, and the packing (or unpacking for external write accesses) is done least significant byte first (i.e. the first data byte is the LS byte the second is the middle byte and the last is the MS byte). When this bit is cleared the expansion port control logic assumes a 24 bit wide external memory.
BPAC bit is cleared during hardware reset.

NOTE 1 BPAC is considered only for DMA accesses, and ignored during core accesses.

NOTE 2 In order to ensure sequential external accesses the DMA address should advance in steps of three. See example of the DMA channel programming in Chapter 8.1 - DMA CONTROLLER PROGRAMMING MODEL

NOTE 3 DMA address +1, and DMA address +2 should not cross the AAR bank borders otherwise improper operation may result.

NOTE 4 Arbitration is not allowed during the packing access, i.e. the three accesses are treated as one access from the arbitration point of view, and the bus mastership will not be released during these accesses.

NOTE 5 Packing Mode is not allowed to Synchronous SRAM with zero wait states, otherwise improper operation may result.

2.5.1.8 BNC(3:0) - Number of address bits to Compare - bits 11-8

The read/write control bits BNC(3:0) defines the number of bits (from the BAC bits) that are compared to the external address. If no bits should be compared (BNC(3:0) = 0000) the AA pin is activated only according to the space enable bits (BPEN, BXEN, BYEN). The combinations BNC(3:0) = 1111, 1110, 1101 are reserved.
BCN(3:0) bit are cleared during hardware reset.

2.5.1.9 BAC(11:0) - Address to compare - bits 23-12

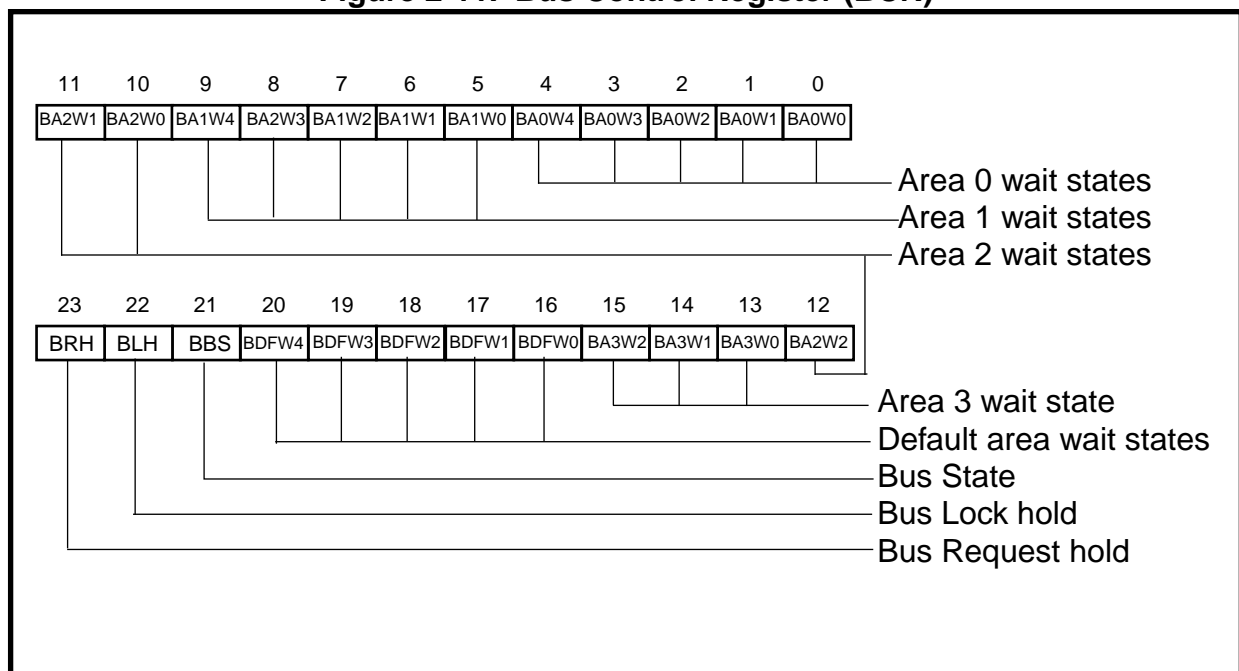
The read/write control bits BAC(11:0) defines the address that should be compared to the

external address in order to decide if to assert the $\overline{AA/RAS}$ pin. The number of bits that should be compared is defined by the BNC(3:0) bits, BAC bits are always compared to the most significant portion of the external address bus (e.g. if BNC(3:0) = 0011 then BAC(11:9) are compared to the 3 most significant bits of the external address). BAC(11:0) bits are cleared during hardware reset.

2.5.2 Bus Control Register

The Bus Control Register (BCR) is a 24 bit read write register used to control the external bus activity and Bus Interface Unit operation. The BCR bits are shown in Figure 2-11 on page 2-24 and described in the following paragraphs.

Figure 2-11. Bus Control Register (BCR)



2.5.2.1 BA0W(4:0) - Area 0 Wait control - bits 4-0

The read/write control bits BA0W(4:0) define the number of wait states (0 - 31) inserted in each external SRAM or synchronous SRAM accesses to area 0 (DRAM accesses are not affected by these bits). Area 0 is the area defined by AAR0 register.

For SRAM accesses only, the value of these bits should not be programmed as zero since SRAM memory access requires at least one wait state.

For SRAM accesses only, when selecting 4 to 7 wait states, one additional wait state will be inserted at the end of the access. When selecting 8 or more wait states, two additional wait states will be inserted at the end of the access. These trailing wait states increase the data hold time and the memory release time and do not increase the memory access time. BA0W(4:0) bits are set during hardware reset (i.e. 31 wait states).

2.5.2.2 BA1W(4:0) - Area 1 Wait control - bits 9-5

The read/write control bits BA1W(4:0) define the number of wait states (0 - 31) inserted in each external SRAM or synchronous SRAM accesses to area 1 (DRAM accesses are not affected by these bits). Area 1 is the area defined by AAR1 register.

For SRAM accesses only, the value of these bits should not be programmed as zero since SRAM memory access requires at least one wait state.

For SRAM accesses only, when selecting 4 to 7 wait states, one additional wait state will be inserted at the end of the access. When selecting 8 or more wait states, two additional wait states will be inserted at the end of the access. These trailing wait states increase the data hold time and the memory release time and do not increase the memory access time. BA1W(4:0) bits are set during hardware reset (i.e. 31 wait states).

2.5.2.3 BA2W(2:0) - Area 2 Wait control - bits 12-10

The read/write control bits BA2W(2:0) define the number of wait states (0 - 7) inserted in each external SRAM or synchronous SRAM accesses to area 2 (DRAM accesses are not affected by these bits). Area 2 is the area defined by AAR2 register.

For SRAM accesses only, the value of these bits should not be programmed as zero since SRAM memory access requires at least one wait state.

For SRAM accesses only, when selecting 4 to 7 wait states, one additional wait state will be inserted at the end of the access. These trailing wait states increase the data hold time and the memory release time and do not increase the memory access time.

BA2W(2:0) bits are set during hardware reset (i.e. 7 wait states).

2.5.2.4 BA3W(2:0) - Area 3 Wait control - bits 15-13

The read/write control bits BA3W(2:0) define the number of wait states (0 - 7) inserted in each external SRAM or synchronous SRAM accesses to area 3 (DRAM accesses are not affected by these bits). Area 3 is the area defined by AAR3 register.

For SRAM accesses only, the value of these bits should not be programmed as zero since SRAM memory access requires at least one wait state.

For SRAM accesses only, when selecting 4 to 7 wait states, one additional wait state will be inserted at the end of the access. These trailing wait states increase the data hold time and the memory release time and do not increase the memory access time.

BA3W(2:0) bits are set during hardware reset (i.e. 7 wait states).

2.5.2.5 BDFW(4:0)- Default Area Wait control - bits 20-16

The read/write control bits BDFW(3:0) define the number of wait states (0 - 31) inserted in each external accesses to an area which is not defined by any of the BAAR registers. The access type to this area is SRAM only. The value of these bits should not be programmed as zero since SRAM memory access requires at least one wait state.

When selecting 4 to 7 wait states, one additional wait state will be inserted at the end of the access. When selecting 8 or more wait states, two additional wait states will be inserted at the end of the access. These trailing wait states increase the data hold time and the memory release time and do not increase the memory access time.

BDFW(4:0) bits are set during hardware reset (e.g. 31 wait states).

2.5.2.6 BBS - Bus State - bit 21

The read only Bus State status bit - BBS is set when the DSP is the bus master and cleared otherwise. BBS bit is cleared during hardware reset.

2.5.2.7 BLH - Bus Lock Hold - bit 22

The read/write control bit Bus Lock Hold - BLH is used to assert the \overline{BL} pin even if no read-modify-write access is occurring. When BLH is set, \overline{BL} pin is always asserted. If BLH bit is cleared \overline{BL} pin is asserted only if a read-modify-write external access is attempted. BLH bit is cleared during hardware reset.

2.5.2.8 BRH - Bus Request Hold - bit 23

The read/write control bit Bus Request Hold - BRH is used to assert the \overline{BR} pin even if no external access is needed. When BRH is set \overline{BR} pin is always asserted, if BRH bit is cleared \overline{BR} pin is asserted only if external access is attempted or pending. BRH bit is cleared during hardware reset.

2.5.3 IDentification Register

The IDentification Register (IDR) is a 24 bit read only via programmed register used to identify the different DSP56300 core-based family members. This register specifies the chip number and revision and the DSP56300 core revision. The exact number for each DSP56300 core member can be found in the specific part data sheet.

2.6 DRAM CONTROLLER

The DRAM controller is designed for efficient interface to dynamic RAM devices in both random read/write cycles and fast access mode (page mode). An on-chip DRAM controller controls the page hit circuit, the address multiplexing (row address and column address), the control signal generation (\overline{CAS} and \overline{RAS}) and the refresh access generation (\overline{CAS} before \overline{RAS}) for a large variety of DRAM module sizes and different access times. The on-chip DRAM controller configuration is determined by the DRAM Control Register (DCR).

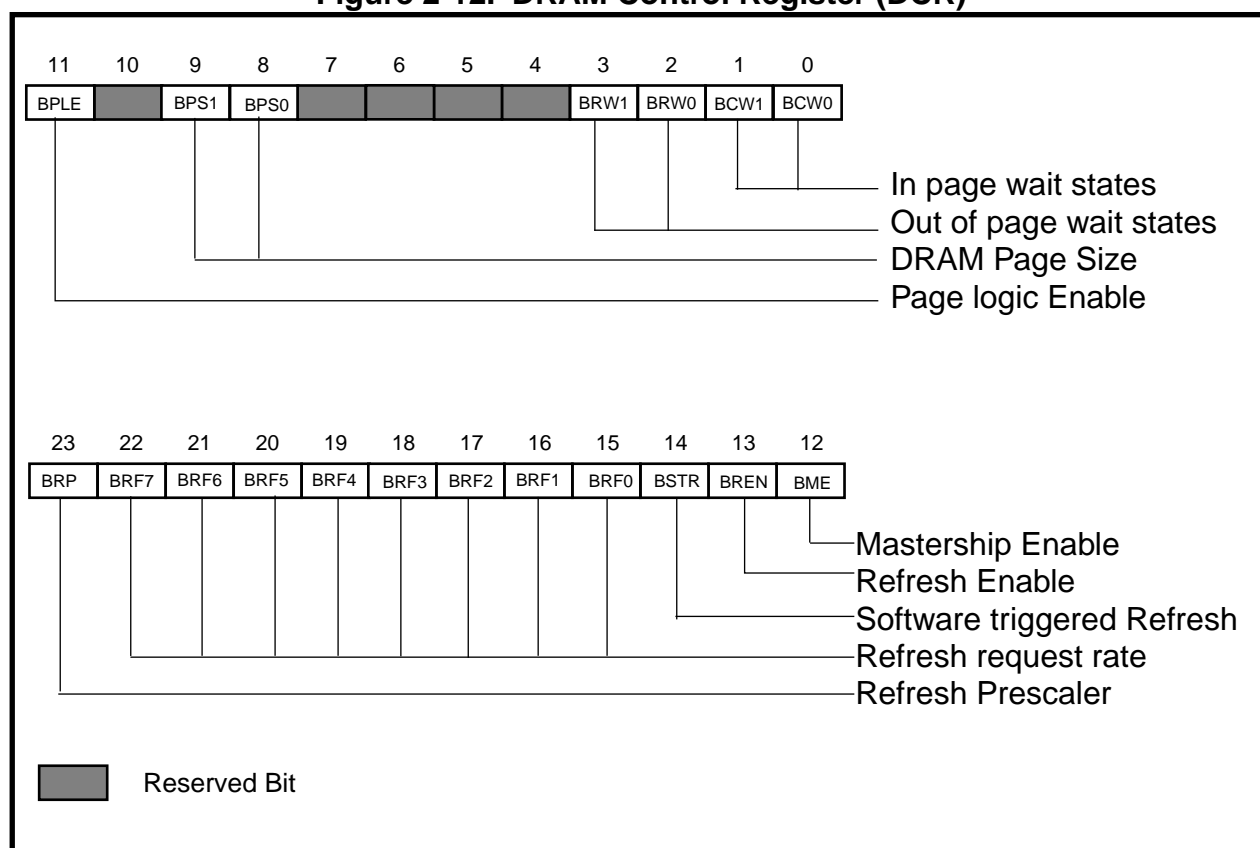
2.6.1 DRAM Control Register

The DRAM Control Register (DCR) is a 24 bit read write register used to control and configure the external DRAM accesses. The DCR bits are shown in Figure on page 2-27 and described in the following paragraphs.

NOTE The programmer must guarantee that all the DCR bits except BSTR are

not changed while accessing a DRAM. Otherwise improper operation may result.

Figure 2-12. DRAM Control Register (DCR)



2.6.1.1 BCW(1:0) - In page Wait states - bits 1-0

The read/write control bits - BCW(1:0) define the number of wait states that should be inserted in each DRAM in-page access.

The encoding of BCW1 and BCW0 is described in the following table. BCW(1:0) bits are cleared during hardware reset

BCW1	BCW0	DRAM External access
0	0	1 w.s for each in-page access
0	1	2 w.s for each in-page access
1	0	3 w.s for each in-page access
1	1	4 w.s for each in-page access

2.6.1.2 BRW(1:0) - Out of page Wait states- bits 3-2

The read/write control bits- BRW(1:0) define the number of wait states that should be inserted in each DRAM out of page access.

The encoding of BRW1 and BRW0 is described in the next table. BRW(1:0) bits are cleared during hardware reset.

BRW1	BRW0	DRAM External access
0	0	4 w.s for each out-of-page access
0	1	8 w.s for each out-of--page access
1	0	11 w.s for each out-of-page access
1	1	15 w.s for each out-of-page access

2.6.1.3 BPS(1:0) - DRAM Page Size - bits 9-8

The read/write control bits- BPS(1:0) define the size of the external DRAM page and thereby the number of the column address bits. The internal page mechanism works according to these bits only if the page logic is enabled (by BPLE bit). The four combinations of BPS(1:0) enable the use of many DRAM sizes (1Mbit, 4Mbit, 16Mbit and 64Mbit). The encoding of BPS1 and BPS0 is described in the following table. BPS(1:0) bits are cleared during hardware reset.

BPS1	BPS0	Column address width	DRAM Page size
0	0	9 bits	512
0	1	10 bits	1K
1	0	11 bits	2K
1	1	12 bits	4K

NOTE When driving the row address all the 24 address bits of the external address bus are driven. e.g: if BPS(1:0) = 01, when driving the row address the 14 MS bits of the internal address (XAB, YAB,PAB or DAB) will be driven on A(13:0) pins, and A(23:14) pins will be driven with the 10 MSB of the internal address. This method enables the use of different DRAMs with the same page size.

2.6.1.4 BPLE - Page logic Enable - bit 11

The read/write Page logic Enable - BPLE is used to enable/disable the in-page identifying logic. When this bit is set it enables the page logic (the page size is defined by BPS(1:0) bits), each in-page identification will cause the DRAM controller to drive only the column address (and the associate $\overline{\text{CAS}}$ signal). When this bit is cleared the page logic is disabled, and the DRAM controller will always access the external DRAM in out-of-page accesses

(e.g. row address with $\overline{\text{RAS}}$ assertion and then column address with $\overline{\text{CAS}}$ assertion). This mode is useful for low power dissipation. There is only one in-page identifying logic and therefore when switching from one DRAM external bank to another DRAM bank (the DRAM external banks are defined by the access type bits in the AAR registers, different external bank are accessed through different AA/ $\overline{\text{RAS}}$ pin) a page fault occurs. BPLE bit is cleared during hardware reset.

2.6.1.5 BME - Mastership Enable - bit 12

The read/write control bit Mastership Enable - BME is used to enable/disable interface to a local DRAM for the DSP. When BME is cleared, the $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ pins are three-stated when mastership is lost and therefore the user must connect an external pull-up resistor to these pins. In this case (BME = 0) the DSP DRAM controller assumes a page fault each time the mastership is lost and a DRAM refresh will require a bus mastership. If BME bit is set the $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ pins are always driven from the DSP and therefore DRAM refresh can be performed even if the DSP is not the bus master. BME bit is cleared during hardware reset.

2.6.1.6 BREN - Refresh Enable - bit 13

The read/write control bit Refresh Enable - BRE enables/disables the internal refresh counter. When this bit is set the refresh counter is enabled and a refresh request ($\overline{\text{CAS}}$ before $\overline{\text{RAS}}$) is generated each time the refresh counter reaches zero, a refresh cycle will occur to all DRAM banks together (all pins that were defined as $\overline{\text{RAS}}$ will be asserted together). When this bit is cleared the refresh counter is disabled and a refresh request may be software triggered by using the BSTR bit. BRE bit is cleared during hardware reset.

NOTE 1 In a system where more than one DSP share the same DRAM, the DRAM controller of more than one DSP may be active, but it is recommended that only one DSP will have its BREN bit set, and bus mastership will be requested for a refresh access.

NOTE 2 If BREN is set and a WAIT instruction is executed, periodic refresh will still be generated each time the refresh counter reaches zero.

NOTE 3 If BREN is set and a STOP instruction is executed, periodic refresh will not be generated and the refresh counter will be disabled.

2.6.1.7 BSTR - Software Triggered Refresh - bit 14

The read/write control/status bit Software triggered refresh - BSTR is used to generate a software triggered refresh request. When this bit is set a refresh request is generated and a refresh access will be executed to all DRAM banks (the exact timing of the refresh access depends on the pending external accesses and on BME bit). After the refresh access ($\overline{\text{CAS}}$ before $\overline{\text{RAS}}$) was executed BSTR bit is cleared by the DRAM controller hardware. The refresh cycle length depends on the BRW(1:0) bits (a refresh access is as long as the out-of-page access). BSTR bit is cleared during hardware reset.

2.6.1.8 BRF(7:0) Refresh rate - bits 22-15

The read/write control bits BRF(7:0) control the refresh request rate. The BRF(7:0) specify a divide rate of 1 (BRF(7:0) = \$00) to 256 (BRF(7:0) = \$FF). A refresh request will be generated each time the refresh counter reaches zero if the refresh counter is enabled (by setting BRE bit). BRF(7:0) bits are cleared during hardware reset.

2.6.1.9 BRP - Refresh Prescaler - bit 23

The read/write control bit - BRP controls a prescaler in series with the refresh clock divider. If BRP is set a divide by 64 prescaler is connected in series with the refresh clock divider, if BRP is cleared the prescaler is bypassed. The refresh request rate (in clock cycles) is the value written to BRF(7:0) bits + 1, multiplied by 64 (if BRP is set) or by 1 (if BRP is cleared). BRP is cleared during hardware reset.

NOTE 1 Refresh requests are not accumulated and therefore in a fast refresh request rate not all the refresh requests will be served (e.g. the combination BRF(7:0) = \$00 and BRP = 0 will generate refresh request every clock cycle, but a refresh access takes at least 5 clock cycles).

NOTE 2 When programming the periodic refresh rate the user must consider the $\overline{\text{RAS}}$ time-out period. There is no hardware support for the $\overline{\text{RAS}}$ time-out restriction.

3 DATA ARITHMETIC LOGIC UNIT

3.1 DATA ALU ARCHITECTURE

The Data ALU (see Figure 3-1) performs all the arithmetic and logical operations on data operands in the DSP56300 Core.

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. A sixteen bit arithmetic mode of operation is available by setting the SA bit in the status register (SR).

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

The components of the Data ALU are as follows:

- Four 24-bit input registers
- A parallel, fully pipelined multiply-accumulator unit (MAC)
- Two 48-bit accumulator registers
- Two 8-bit accumulator extension registers
- A Bit Field Unit (BFU) with a 56-bit barrel shifter
- An accumulator shifter
- Two data bus shifter/limiter circuits

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

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

3.1.2 MAC Unit

The MAC unit comprises the main arithmetic processing unit of the DSP56300 Core and performs all of the calculations on data operands. In the case of arithmetic instructions, the unit accepts up to three input operands and outputs one 56-bit result of the following form, extension:most significant product:least significant product (EXT:MSP:LSP). The operation of the MAC unit occurs independently and in parallel with XDB and YDB activity, and its registers facilitate buffering for both Data ALU inputs and outputs. Latches are provided on the MAC unit input to permit writing an input register, which is the source for a Data ALU operation in the same instruction. The input to the multiplier can only come from the X or Y registers. The multiplier executes 24-bit x 24-bit, parallel, fractional multiplies, between two's-complement signed, unsigned or mixed operands. 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. The multiply/accumulate operation is fully pipelined and takes two clock cycles to complete. In the first clock the multiply is performed and the product is stored in the pipeline register. In the second clock the accumulator is added or subtracted. If a multiply without accumulation (MPY) is specified in the instruction, the MAC clears the accumulator and then adds the contents to the product. When a 56-bit result is to be stored as a 24-bit operand, the LSP can be simply truncated, or it can be rounded into the MSP. Rounding is performed if specified in the DSP instruction (e.g., the signed multiply-accumulate and round (MACR) instruction). The rounding performed is either convergent rounding (round-to-nearest-even) or two's-complement rounding. The type of rounding is specified by the rounding bit in the status register. The bit in the accumulator that is rounded is specified by the scaling mode bits in the status register.

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

The six Data ALU registers (A2, A1, A0, B2, B1, and B0) form two general-purpose, 56-bit accumulators, A and B. Each of these two accumulators consists of three concatenated registers (A2:A1:A0 and B2:B1:B0, respectively). The 24-bit MSP is stored in A1 or B1; the 24-bit LSP is stored in A0 or B0. The 8-bit EXT is stored in A2 or B2.

Reading the A or B accumulators over the XDB and YDB is protected against overflow by substituting a limiting constant for the data that is being transferred. The content of A or B is not affected should limiting occur; only the value transferred over the XDB or YDB is limited. This process is commonly referred to as transfer saturation and should not be confused with the arithmetic saturation mode.

The overflow protection is performed after the contents of the accumulator have been shifted according to the scaling mode. Shifting and limiting will be performed only when the entire 56-bit A or B register is specified as the source for a parallel data move over the XDB or YDB. When A0, A1, A2, B0, B1, or B2 are specified as the source for a parallel data move, shifting and limiting are not performed. When the 8-bit wide accumulator extension register (A2 or B2) is specified as the source for a parallel data move, it is sign extended to produce the full 24-bit wide word. The accumulator registers (A or B) serve as buffer registers between the arithmetic unit and the XDB and/or YDB. These registers are used as both Data ALU source and destination operands.

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

3.1.4 Accumulator Shifter

The accumulator shifter 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)
- 24-Bit Right Shift (Arithmetic) for DMAC
- 16-Bit Right Shift (Arithmetic) for DMAC in
Sixteen Bit Arithmetic Mode
- Force to zero

3.1.5 Bit Field Unit (BFU)

The bit field unit contains a 56-bit parallel bidirectional shifter with a 56-bit input and a 56-bit output, mask generation unit and logic unit. The bit field unit is used in the following operations:

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

3.1.6 Data Shifter/Limiter

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

3.1.7 Scaling

The data shifters (in the shifters/limiters unit), controlled by the scaling mode bits in the status register, are capable of shifting data one bit to the left (scale up) or one bit to the right (scale down) as well as passing the data unshifted (no scaling). Each data shifter has a 24-bit output with overflow indication. These shifters permit dynamic scaling of fixed-point data without modifying the program code. For example, this permits block floating-point algorithms such as fast Fourier transforms to be implemented in a regular fashion.

3.1.8 Limiting

In the DSP56300 Core, the Data ALU accumulators A and B have eight extension bits. Limiting will occur when the extension bits are in use and either A or B is the source being read over XDB or YDB. The limiters in the DSP56300 Core place a shifted and limited value on XDB or YDB without changing the contents of the A or B registers. Having two limiters allows two-word operands to be limited independently in the same instruction cycle. The two data limiters can also be combined to form one 48-bit data limiter for long-word operands.

If the contents of the selected source accumulator can be represented without overflow in the destination operand size (i.e. signed integer portion of the accumulator is not in use), the data limiter is disabled, and the operand is not modified. If the contents of the selected source accumulator cannot be represented without overflow in the destination operand size, the data limiter will substitute a limited data value having maximum magnitude (saturated) and having the same sign as the source accumulator contents: \$7FFFFFFF for 24-bit or \$7FFFFFFF FFFFFFFF for 48-bit positive numbers, \$800000 for 24-bit or \$800000 000000 for 48-bit negative numbers. This process is called transfer saturation. The value

in the accumulator register is not shifted or limited and can be reused within the Data ALU. When limiting does occur, a flag is set and latched in the status register.

3.2 DATA ALU ARITHMETIC AND ROUNDING

3.2.1 Data Representation

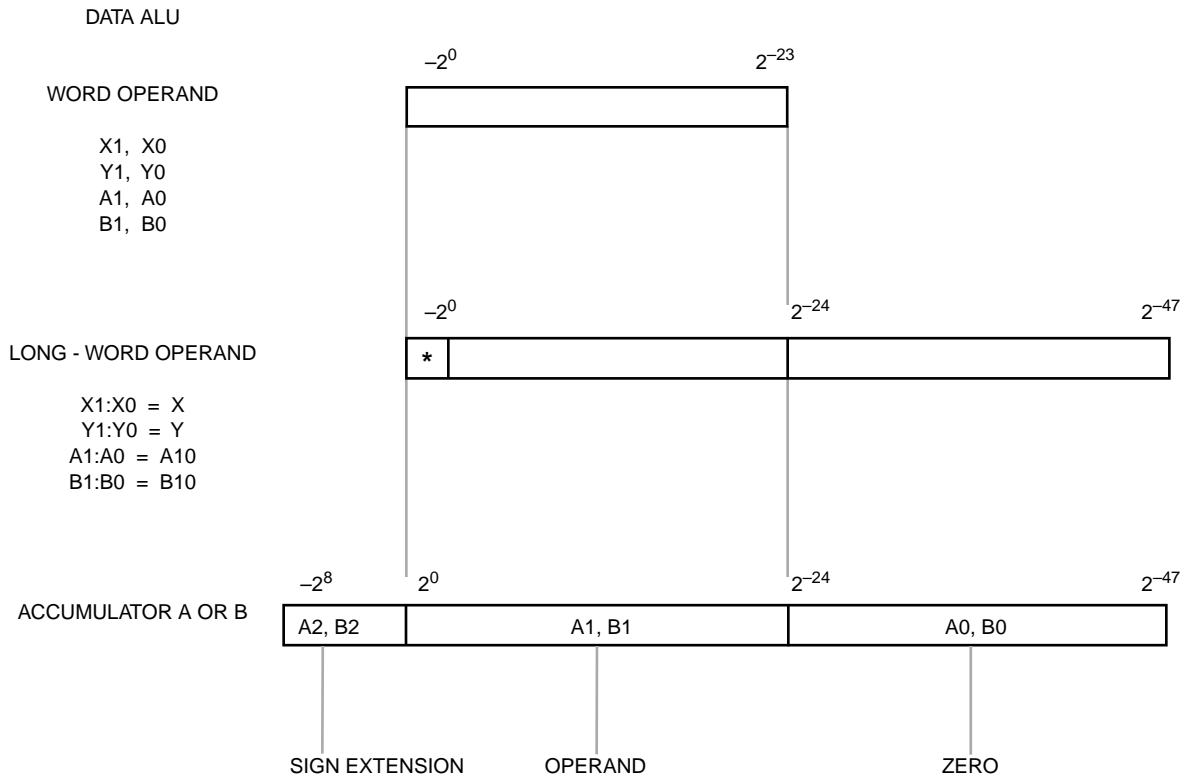
The DSP56300 Core uses a fractional data representation for all Data ALU operations. Figure 3-2 shows the bit weighting of words, long words, and accumulator operands for this representation. The decimal points are all aligned and are left justified.

For words and long words, the most negative number that can be represented is -1.0 whose internal representation is \$800000 and \$800000000000, respectively.

The most positive word is \$7FFFFFFF or $1-2^{-23}$ and the most positive long word is \$7FFFFFFFFFFFFFFF or $1-2^{-47}$. These limitations apply to all data stored in memory and to data stored in the Data ALU input buffer registers. The extension registers associated with the accumulators allow word growth so that the most positive number that can be used is approximately 256 and the most negative number is -256.

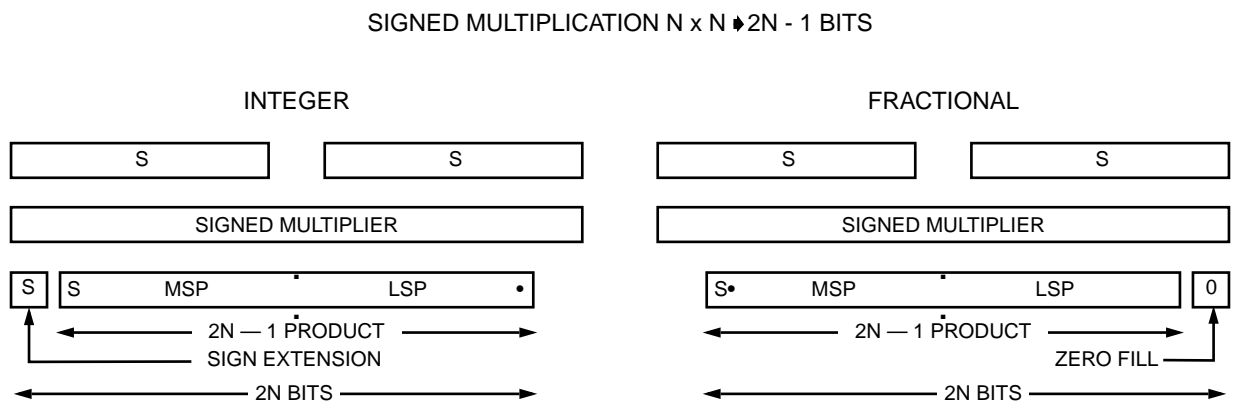
To maintain alignment of the binary point when a word operand is written to accumulator A or B, the operand is written to the most significant accumulator register (A1 or B1), and its MSB is automatically sign extended through the accumulator extension register (A2 or B2). The least significant accumulator register (A0 or B0) is automatically cleared. When a long-word operand is written to an accumulator, the least significant word of the operand is written to the least significant accumulator register (see Figure 3-2).

Figure 3-2. Bit Weighting and Alignment of Operands



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 3-3.

Figure 3-3. Integer/Fractional Multiplication



The key difference is in the alignment of the $2N-1$ bit product. In fractional multiplication the $2N-1$ significant product bits should be left aligned, and a zero is filled in the least

significant bit (LSB), to maintain fractional representation. In integer multiplication the $2N-1$ significant product bits should be right aligned, and the sign bit should be duplicated, to maintain integer representation. Since the DSP56300 Core incorporates a fractional array multiplier, it always aligns the $2N-1$ significant product bits to the left. The user should be aware of this when multiplying integer numbers.

3.2.2 Rounding Modes

The DSP56300 Core's DATA ALU performs rounding of the accumulator register to single precision if requested in the instruction. The upper portion of the accumulator is rounded according to the contents of the lower portion of the accumulator. The boundary between the lower portion and the upper portion is determined by the scaling mode bits S0 and S1 in the status register (SR). Two types of rounding are implemented: convergent rounding and two's complement rounding. The type of rounding is selected by the rounding mode bit (RM) in the EMR portion of the status register.

3.2.2.1 Convergent Rounding

This is the default rounding mode. Convergent rounding is also called round-to-nearest (even) number. 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 biased in that direction. Convergent rounding solves the problem by rounding down if the number is even (LSB=0) and rounding up if the number is odd (LSB=1). Figure 3-4 shows the four cases for rounding a number in the A1 (or B1) register. If scaling is set in the status register, the rounding position is updated to reflect the alignment of the result when it will be put on the data bus. However, the contents of the register are not scaled.

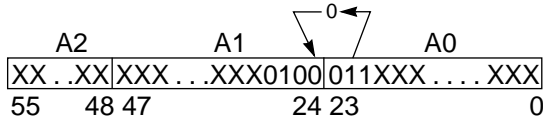
3.2.2.2 Two's Complement Rounding

When twos-complement rounding is selected by setting the rounding mode bit in the MR, all values equal or above one-half are rounded up and all values below one-half are rounded down. Therefore a small positive bias is introduced. Figure 3-4 shows the four cases for rounding a number in the A1 (or B1) register. If scaling is set in the status register, the rounding position is updated to reflect the alignment of the result when it will be put on the data bus. However, the contents of the register are not scaled.

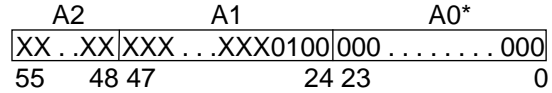
Figure 3-4. Convergent Rounding (no scaling)

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

BEFORE ROUNDING

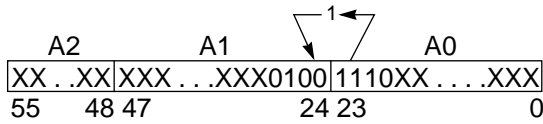


AFTER ROUNDING

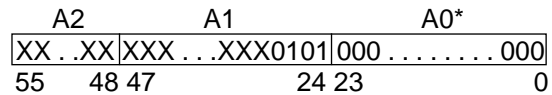


CASE II: IF $A0 > \$800000$ ($1/2$), THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING

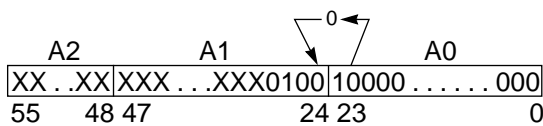


AFTER ROUNDING

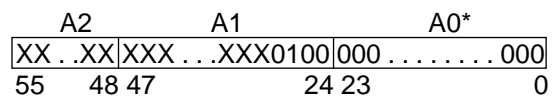


CASE III: IF $A0 = \$800000$ ($1/2$), AND THE LSB OF A1 = 0, THEN ROUND DOWN (ADD NOTHING)

BEFORE ROUNDING

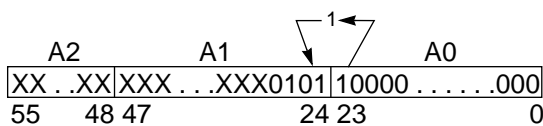


AFTER ROUNDING

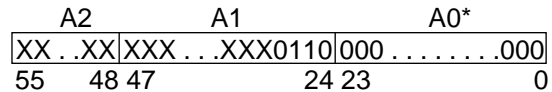


CASE IV: IF $A0 = \$800000$ ($1/2$), AND THE LSB = 1, THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING



AFTER ROUNDING

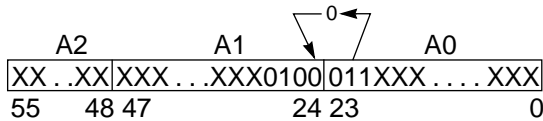


*A0 is always clear; performed during RND, MPYR, MACR

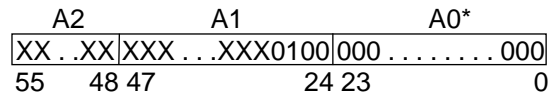
Figure 3-5. Two's Complement Rounding (no scaling)

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

BEFORE ROUNDING

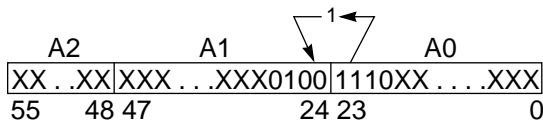


AFTER ROUNDING

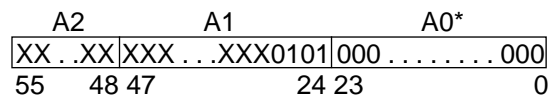


CASE II: IF $A0 > \$800000$ ($1/2$), THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING

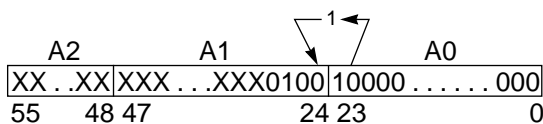


AFTER ROUNDING

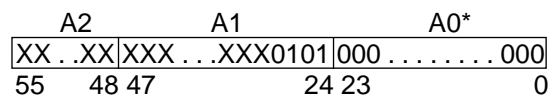


CASE III: IF $A0 = \$800000$ ($1/2$), AND THE LSB OF A1 = 0, THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING

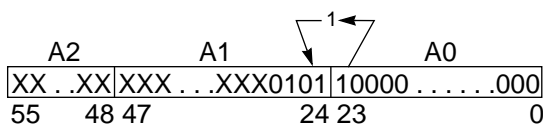


AFTER ROUNDING

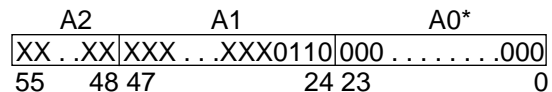


CASE IV: IF $A0 = \$800000$ ($1/2$), AND THE LSB OF A1 = 1, THEN ROUND UP (ADD 1 TO A1)

BEFORE ROUNDING



AFTER ROUNDING



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

3.2.3 Arithmetic Saturation Mode

By setting the Arithmetic Saturation Mode (SM) bit in the status register (SR), the arithmetic unit's result is limited to 48 bits (MSP and LSP). The highest dynamic range of the machine is then limited to 48 bits. The purpose of this bit is to provide a saturation mode for algorithms which do not recognize or cannot take advantage of the extension accumulator.

The arithmetic saturation logic operates by checking three bits of the 56-bit result after rounding: two bits of the extension byte (EXT[7] and EXT[0]) and one bit on the MSP (MSP[23]). The result obtained in the accumulator when SM =1 is shown in Table 3-1.:

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

EXT[7]	EXT[0]	MSP[23]	result in accumulator
0	0	0	unchanged
0	0	1	\$00 7FFFFFFF FFFFFFFF
0	1	0	\$00 7FFFFFFF FFFFFFFF
0	1	1	\$00 7FFFFFFF FFFFFFFF
1	0	0	\$FF 800000 000000
1	0	1	\$FF 800000 000000
1	1	0	\$FF 800000 000000
1	1	1	unchanged

The two saturation constants \$007FFFFFFFFFFFFFFF and \$FF80000000000000 are not affected by the scaling mode. In the same way, the rounding of the saturation constant (during MPYR, MACR, RND) is independent of the scaling mode: \$007FFFFFFFFFFFFFFF is rounded to \$007FFFFFFF000000 and \$FF80000000000000 to \$FF80000000000000.

When in Arithmetic Saturation Mode, the Overflow Bit (V bit) in the status register is set if the Data ALU result is not representable in the 48-bit accumulator, i.e. an arithmetic saturation has occurred. This also implies that the limiting bit (L bit) in the status register is set when an arithmetic saturation occurs.

Caution: The arithmetic saturation mode is **ALWAYS** disabled during the execution of the following instructions: TFR, Tcc, DMACsu, DMACuu, MACsu, MACuu, MPYsu, MPYuu, CMPU, and all BIT FIELD UNIT operations (see 3.1.5). If the result of these instructions should be saturated, a MOVE A,A (or B,B) instruction must be added following the original instruction (provided no scaling is set). However, the "V" bit of the status register will never be set by the arithmetic saturation of the accumulator during the MOVE A,A (or B,B). Only the "L" bit will then be set.

3.2.4 Multiprecision Arithmetic Support

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

- **MPY/MAC su:** multiplication and multiply-accumulate with signed times unsigned operands
- **MPY/MAC uu:** multiplication and multiply-accumulate with unsigned times unsigned operands
- **DMACss:** multiplication with signed times signed operands and 24-bit arithmetic right shift of the accumulator before accumulation
- **DMACsu:** multiplication with signed times unsigned operands and 24-bit arithmetic right shift of the accumulator before accumulation
- **DMACuu:** multiplication with unsigned times unsigned operands and 24-bit arithmetic right shift of the accumulator before accumulation

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

Figure 3-6. DMAC Implementation

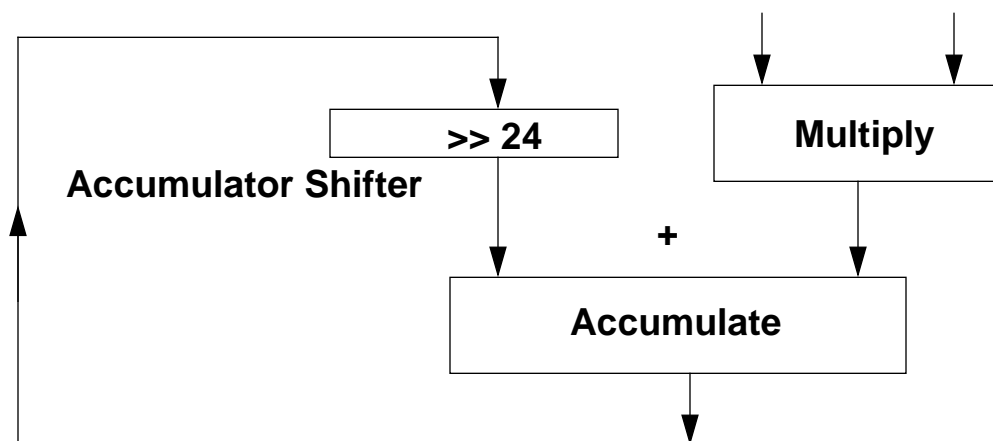
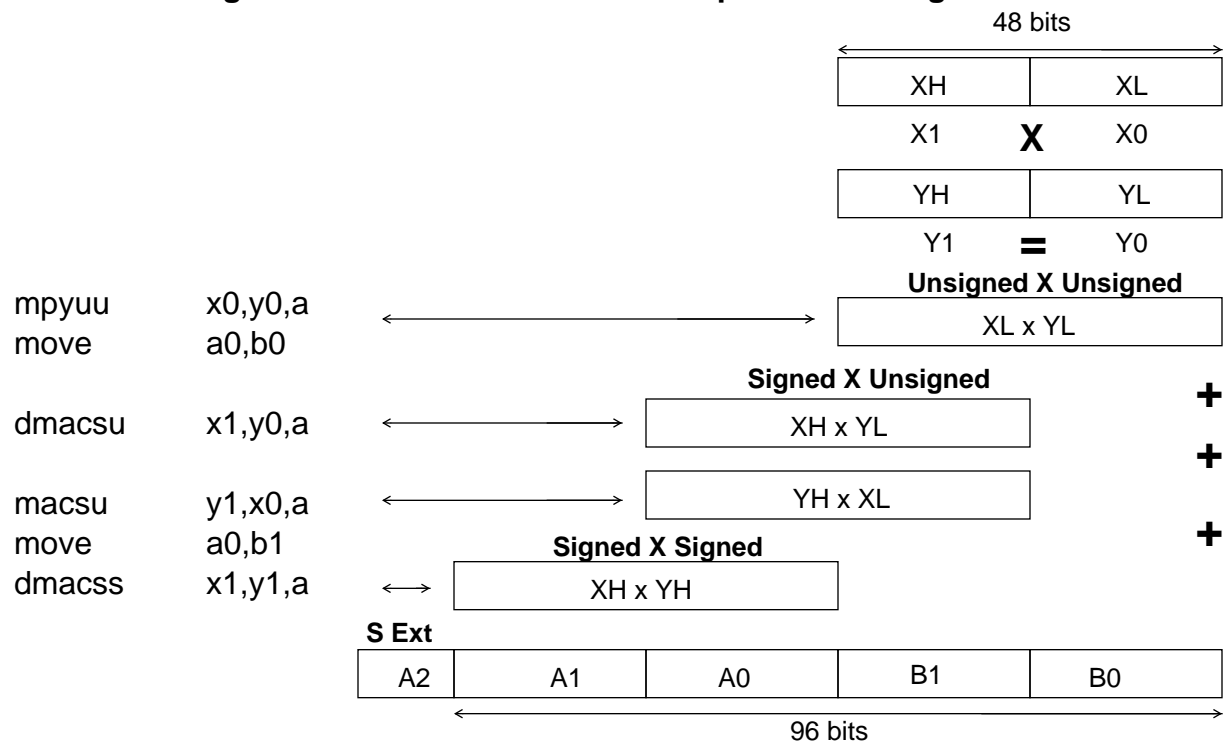


Figure 3-7 illustrates the use of these instructions in the case of a double precision multiplication. The signed x signed operation is used to multiply or multiply-accumulate the two upper, signed, portions of two signed double precision numbers. The unsigned x signed operation is used to multiply or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

Figure 3-7. Double Precision Multiplication Using DMAC



3.2.4.1 Double Precision Multiply Mode

To support existing 56K code, double precision multiply can also be performed by a double precision algorithm which uses four multiply operations, after entering a dedicated “Double Precision Multiply” mode. The mode is entered by setting bit 14 (DM) of the Status Register (bit 6 of the MR register). The mode is disabled by clearing the DM bit.

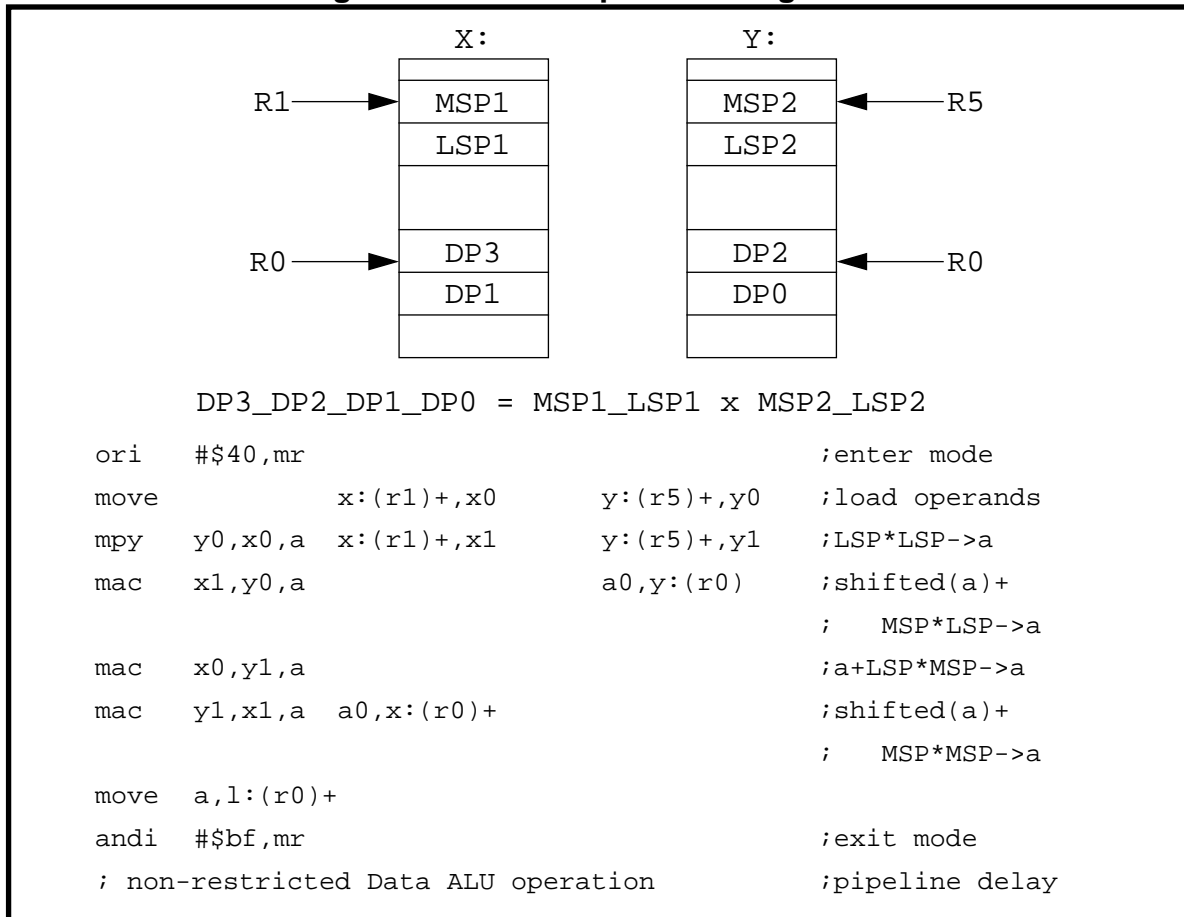
The algorithm is shown in Figure 3-8. The ORI instruction sets the DM mode bit in the MR register, but due to the instruction execution pipeline the Data ALU enters the Double Precision Multiply mode only after one cycle. The ANDI instruction clears the DM mode bit in the MR register, but due to the instruction execution pipeline the Data ALU leaves the mode after one cycle; the ANDI instruction should not be immediately followed by a restricted Data ALU instruction, to allow for the pipeline delay.

While in Double Precision Multiply mode, the behavior of the four specific operations listed in the double precision algorithm is modified. Therefore these operations (with those specific register combinations) should not be used, while in Double Precision Multiply mode, for any other purpose but for the double precision multiply algorithm. All other Data ALU operations (or the four listed operations but with other register combination) may not be used.

The double precision multiply algorithm uses the Y0 register at all stages. Therefore Y0 should not be changed when running the double precision multiply algorithm. 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.

Figure 3-8. Double precision algorithm



3.2.5 Block Floating Point FFT Support

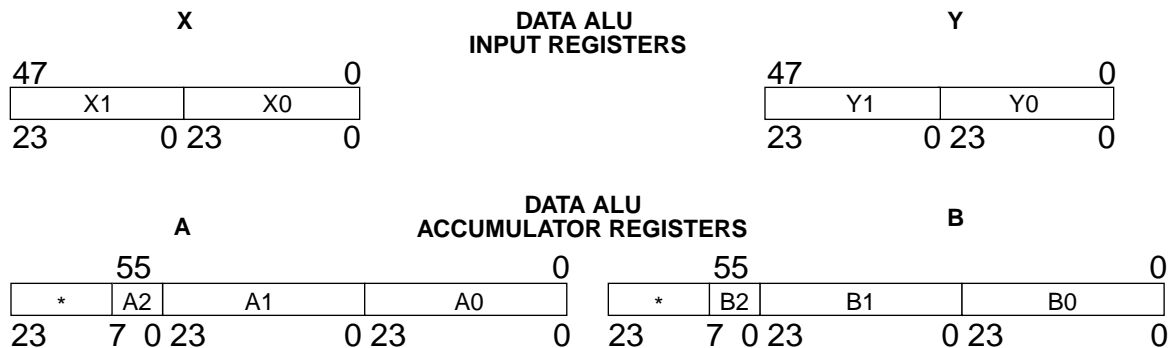
Block Floating Point FFT operation requires the early detection of data growth between FFT butterfly passes. If data growth is detected, suitable down scaling must be applied to ensure that no overflow will occur during the next butterfly calculation pass. The total scaling applied is the block exponent of the FFT output. The Block Floating Point FFT algorithm is described in the Motorola application note APR4/D, "Implementation of Fast Fourier Transforms on Motorola's DSP56000/DSP56001 and DSP96002 Digital Signal Processors".

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

3.3 DATA ALU PROGRAMMING MODEL

The Data ALU features 24-bit input/output data registers that can be concatenated to accommodate 48-bit data and two 56-bit accumulators, which are segmented into three 24-bit pieces that can be transferred over the buses. Figure 3-9 illustrates how the registers in the programming model are grouped.

Figure 3-9. DSP56300 Core Programming Model



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

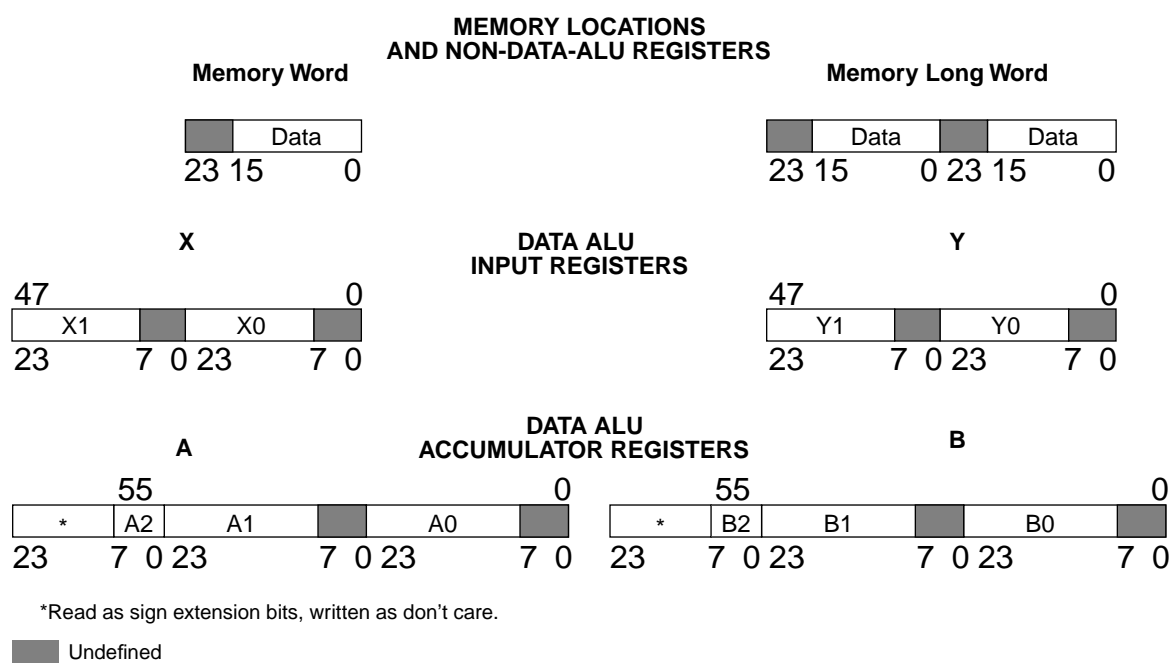
3.4 SIXTEEN BIT ARITHMETIC MODE

Setting the SA bit in the status register (SR) enables the Sixteen Bit Arithmetic mode of operation. The 16 bit data is right aligned in the 24 bit memory word, that is in the 16 least significant bits of the 24 bit word. The user may use 16 bit wide data memories with either leaving the 8 most significant bits unconnected, or tying these bits to GND.

In the Sixteen Bit Arithmetic mode of operation the source operands can be 16, 32 or 40 bit. The numerical results have 40 bits accuracy. These 40 bits are composed of 16 bit LSP, 16 bit MSP and 8 bit EXT.

Figure 3-10 shows the bit positions in the memory and Data ALU registers when in Sixteen Bit Arithmetic mode.

Figure 3-10. Sixteen Bit Arithmetic Mode Data Organization



Note 1: When switching to and from sixteen bit arithmetic mode, for two instruction cycles, no arithmetic instruction or a move instruction should be performed.

Note 2: The programmer should be cautious about exchanging data between 16-bit arithmetic mode and 24-bit arithmetic mode via write-read operations on data-ALU registers and accumulators. Since the write operations in 16 bit arithmetic mode corrupt the information in the least significant bytes of the registers or accumulators, these registers or accumulator should not be used as 24-bit data without some processing.

3.4.1 Moves in sixteen bit arithmetic mode

In the Sixteen Bit Arithmetic mode of operation, the Data ALU registers may still be read or written over the XDB and the YDB as 24 or 48 bit operations. No 16 or 32 bit moves are provided. The mapping of the 16-bit data to the 24-bit buses is described in the following paragraphs.

3.4.1.1 Moves into registers or accumulators

When moving XDB or YDB into a full Data ALU accumulator (A or B) the 16 LS bits of the bus will be placed in bits 32-47 of the accumulator (16 MS bits of A1 or B1). Bits 8-23 of the accumulator (16 MS bits of A0 or B0) will be cleared and the EXT of the accumulator (A2 or B2) will be loaded with sign extension.

When moving XDB and YDB (48 bits) into a full Data ALU accumulator (A or B) the 16 LS bits from XDB will be placed in bits 32-47 of the accumulator (16 MS bits of A1 or B1). The

16 LS bits from YDB will be placed in bits 8-23 of the accumulator (16 MS bits of A0 or B0). The EXT of the accumulator (A2 or B2) will be loaded with sign extension.

When moving XDB or YDB into a register (X0, X1, Y0 or Y1) or partial accumulator (A0, A1, B0 or B1) the 16 LS bits of the bus will be loaded into the 16 MS bits of the destination register. No other portion of the accumulator will be affected.

When moving XDB or YDB into the accumulator extension register (A2 or B2) the 8 LS bits of the bus will be loaded into the 8 LS bits of the destination register and the 16 MS bits of the bus will not be used. The remaining parts of the accumulator will not be affected.

When moving XDB and YDB into a 48-bit register (X or Y) or partial accumulator (A10 or B10) the 16 LS bits of XDB bus will be loaded into the 16 MS bits of the MSP (X1, Y1, A1 or B1) and the 16 LS bits of YDB bus will be loaded into the 16 MS bits of the LSP (X0, Y0, A0 or B0). The EXT part of the accumulator (A2 or B2) will not be affected.

3.4.1.2 Moves from registers or accumulators

When moving a partial accumulator (A0, A1, B0 or B1) to XDB or YDB, the 16 MS bits of the source will be transferred to the 16 LS bits of the bus with 8 zeros in the MS bits. No scaling or limiting will be performed. When the source is the accumulator extension register (A2 or B2) it occupies the 8 LS bits of the bus while the next 16 bits are sign extension of bit number 7.

When moving a partial accumulator (A10 or B10) to XDB and YDB, the 16 MS bits of the MSP of the source (A1 or B1) will be transferred to the 16 LS bits of XDB with 8 zeros in the MS bits, while the 16 MS bits of the LSP of the source (A0 or B0) will be transferred to the 16 LS bits of YDB with 8 zeros in the MS bits. No scaling or limiting will be performed.

When moving a full Data ALU accumulator (A or B) to XDB or YDB, scaling and limiting will be performed, and then the 16-bit scaled and limited word will be placed on the bus LS bits and sign extension will be placed on the bus 8 MS bits.

When moving a full Data ALU accumulator (A or B) to XDB and YDB, scaling and limiting will be performed, and then the 16 MS bits of the 32-bit scaled and limited double word will be placed on XDB 16 LS bits and sign extension will be placed on the bus 8 MS bits. The 16 LS bits of the 32-bit scaled and limited double word will be placed on YDB 16 LS bits with 8 zeros on the bus 8 MS bits.

When moving a register (X0, X1, Y0 or Y1) to XDB or YDB, the 16 MS bits of the source will be transferred to the 16 LS bits of the bus with 8 zeros in the MS bits.

When moving a 48-bit register (X or Y) to XDB and YDB, the 16 MS bits of the high register (X1 or Y1) will be placed on XDB 16 LS bits and 8 zeroes will be placed on the bus 8 MS bits. The 16 LS bits of the low register (X0 or Y0) will be placed on YDB 16 LS bits with 8 zeros on the bus 8 MS bits.

Note: When a read operation of a Data ALU register (X, Y, X0, X1, Y0 or Y1) immediately follows a write operation to the same register, then the value placed on the 8 MS bits of the XDB or YDB will be undefined.

3.4.1.3 Short Immediate moves

When an Immediate Short Data MOVE is performed while in Sixteen Bit Arithmetic mode of operation and the destination register is A0, A1, B0 or B1, the 8 bit immediate short operand is interpreted as an unsigned integer and is therefore stored in bits 15-8 of the register (which correspond to the 8 LS bits of a 16-bit number). If the destination register is A2 or B2, the 8 bit immediate short operand is stored in bits 7-0 of the register.

When the destination register is A, B, X0, X1, Y0 or Y1, the 8 bit immediate short operand is interpreted as a signed fraction and is therefore stored in bits 47-40 of the accumulator, or bits 23-16 of a register (which correspond to the 8 MS bits of a 16-bit number).

3.4.1.4 Scaling and Limiting

If scaling is specified, the data shifter virtually concatenates the 16 bit LSP to the 16 bit MSP so as to provide numerically correct shift.

During the Sixteen Bit Arithmetic mode of operation the limiting will be affected as described below: the maximum positive value will be \$007FFF (\$007FFF00FFFF for double precision). The maximum negative value will be \$008000 (\$008000000000 for double precision).

3.4.2 Sixteen bit arithmetic

When reading an operand from a Data ALU register or accumulator to the arithmetic unit, the 8 least significant bits of the 24 bit word, are ignored, i.e. read as zeros. The arithmetic unit will force these bits to zero when generating a result.

The arithmetic unit virtually concatenates the 16 bit LSP with the 16 bit MSP to form a continuous number. Therefore all arithmetic operations, including shifts, are numerically correct.

The execution of Data ALU instructions when in sixteen bit arithmetic mode is not affected, except for the following:

1. The operands and results width (16/32/40 instead of 24/48/56).
2. The rounding, if specified by the operation, will be performed on the most significant bit of the 16-bit LS portion of the result, that is on the bit corresponding to bit number twenty-three of A0/B0 (the scaling mode will affect this position accordingly). See RND instruction for details.
3. The arithmetic saturation detection is unchanged, but the saturated values change to \$007FFF00FFFF00 and \$FF800000000000.
4. The carry bit C will be added/subtracted, in ADC/SBC instructions, to the least significant bit of the 16-bit LSP.

-
5. Logic operations will only affect the 16-bit wide word.
 6. Rotation in rotate instructions will be performed on a 16-bit wide word.
 7. The possible normalization range changes, thus affecting the CLB instruction.
 8. DMAC instruction will perform a 16-bit arithmetic right shift of the accumulator before accumulation.
 9. Double Precision Multiplication Algorithm, with the Double Precision Multiply Mode bit is set, will not be supported.
 10. The bit parsing instructions (MERGE, EXTRACT, EXTRACTU and INSERT) are affected by the sixteen bit arithmetic mode so as to perform on the appropriate bit positions of the sixteen bit data. In INSERT the user has to update the offset by adding a bias value of 16. For further details refer to the specific instruction details in appendix A.
 11. In the Read Modify Write instructions (BCHG, BCLR, BSET and BTST) and in the Jump/Branch on bit instructions (BRCLR, BRSET, BSCLR, BSSET, JCLR, JSET, JSCLR and JSSET) the bit numbering in sixteen bit arithmetic mode is relative to 16-bit wide words, i.e. bit number 0 is the least significant bit and bit number 15 is the most significant bit. Bit numbers over 15 should not be used.

3.5 PIPELINE CONFLICTS

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

Since every Data ALU instruction takes two clock cycles to complete, an interlock condition occurs when trying to read an accumulator (or parts of an accumulator) while the preceding instruction was a Data ALU instruction that specified that same accumulator as the destination. This interlock condition, named “arithmetic stall”, is detected in hardware and an idle cycle (no op) is inserted, thereby the correctness is guaranteed. The user can optimize his code by inserting a useful instruction before the read instruction. Figure 3-13 describes the cases in which the pipelined nature of the Data ALU generates arithmetic stall cases.

Figure 3-11. Pipeline Conflicts - Arithmetic stall

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

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

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

A second interlock condition, named “status stall”, occurs when trying to read the status register (SR) while the preceding or the second preceding instruction was a Data ALU instruction or an accumulator read (which updates the S and L condition codes in the status register). The hardware will insert two or one idle cycles (no op) accordingly, thereby the correctness is guaranteed. Notice that “read status register” implies a MOVE status register, Bit Manipulation Instructions (e.g. BSET) on a status register bit, or Program Control Instructions (e.g. BSCLR) which test for a bit in the status register. Figure 3-12 describes the cases in which the pipelined nature of the Data ALU generates stall interlock cases.

Figure 3-12. Pipeline Conflicts - Status stall

```

;following example illustrates a two-clock pipeline delay when
;trying to read the status register as source for move:
mac    x0,y0,a                ;data ALU operation
move   sr,x:(r0)+             ;TWO clock delay is added to
                                ;allow mac to update SR

;following example illustrates a one-clock pipeline delay when
;trying to read the status register as source for bit
;manipulation instruction:
move   a,x:(r0)+              ;read full accumulator
nop
btst   #5,sr                  ;ONE clock delay is added (and
                                ;not two) due to the previous nop

;following example illustrates a one-clock pipeline delay when
;trying to read the status register as source for program control
;instruction:
insert x0,y1,a                ;data ALU operation
bsclr  #5,sr,$ff00ff          ;ONE clock delay is added (and not
                                ;two) since bsclr is a two word
                                ;instruction

```

A third interlock condition, named “transfer stall”, occurs when the source Data ALU accumulator of the move portion of an instruction is “identical” to the destination Data ALU accumulator of the move portion of the preceding instruction. “Identical” accumulators for this matter are any combination of portions (including the full width) of the same Data ALU accumulator, e.g. A1 and A, A2 and A0 etc. The hardware will insert one idle cycle (no op) thereby the correctness is guaranteed.

Figure 3-13. Pipeline Conflicts - Transfer stall

```
;following example illustrates a one-clock pipeline delay when
;trying to read an accumulator that was written by the preceding
;instruction:
move  y:(r1)+,a1                ;write into partial accumulator
move  a2,x:(r0)+                ;one clock delay is added

;following example illustrates a way to find useful usage of
;the pipeline delay clock:
move  y:(r1)+,a1                ;write into partial accumulator
mac   x1,y1,b                   ;insert a useful instruction
move  a,x:(r0)+                 ;no time penalty for this read
```

Note: A special case of interlock occurs when using a 24 bit logic instruction and writing concurrently to the EXT or the LSP of the same accumulator. The hardware will insert one idle cycle (no op), thereby the correctness is guaranteed. For example:

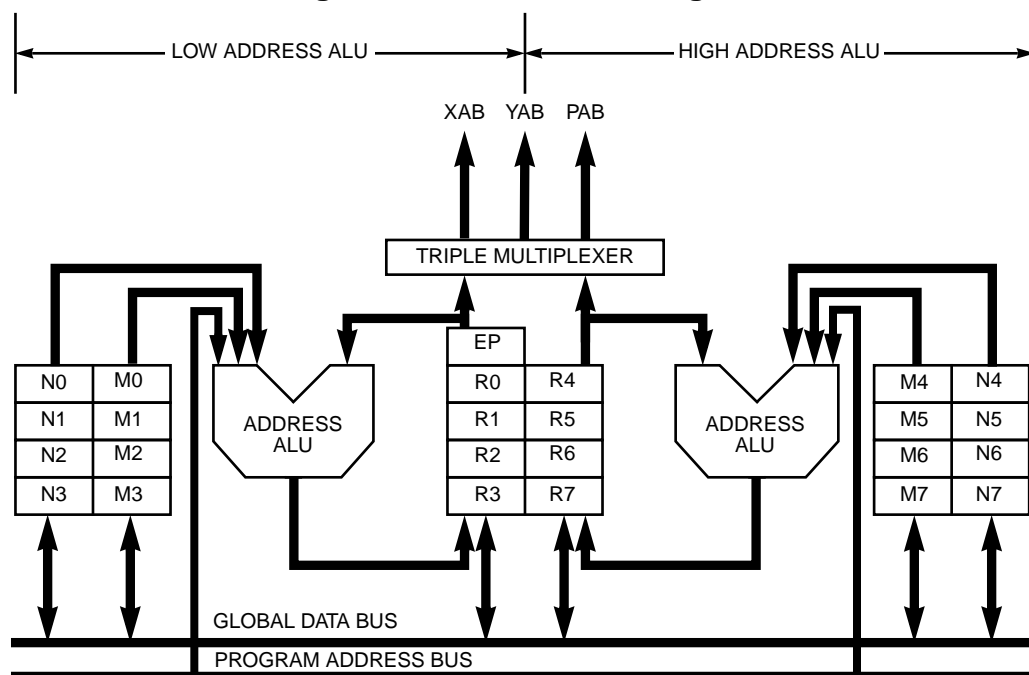
or x1,a y1,a0

4 ADDRESS GENERATION UNIT

4.1 AGU ARCHITECTURE

The AGU is one of the three execution units on the DSP56300 Core. The AGU performs the effective address calculations (using integer arithmetic) necessary to address data operands in memory and contains the registers used to generate the addresses. It implements four types of arithmetic: linear, modulo, multiple wrap-around modulo and reverse-carry and operates in parallel with other chip resources to minimize address-generation overhead. The AGU is divided into two halves, each of which has an address arithmetic logic unit (ALU) and four sets of registers (see Figure 4-1).

Figure 4-1. AGU Block Diagram



These registers are the address registers (R0 - R3 and R4 - R7), offset registers (N0 - N3 and N4 - N7), and the modifier registers (M0 - M3 and M4 - M7). The eight R_n , N_n , and M_n 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. Each register may be read or written by the global data bus (GDB).

The two arithmetic units can generate two 24-bit addresses every instruction cycle — one

for any two of the XAB and YAB, or one PAB address. The AGU can directly address 16,777,216 locations on the XAB, 16,777,216 locations on the YAB, and 16,777,216 locations on the PAB. The two independent address ALUs work with the two data memories to feed two operands to the data ALU in a single cycle. Each operand may be addressed by an Rn, Nn, and Mn triplet.

The two address ALUs are identical (see Figure 4-1); each one of them contains a 24-bit full adder (called offset adder), which can add 1) plus one, 2) minus one, 3) the contents of the respective offset register N, or 4) minus N to the contents of the selected address register. A second full adder (called a modulo adder) adds the summed result of the first full adder to a modulo value, M or minus M, where M is stored in the respective modifier register. A third full adder (called a reverse-carry adder) can 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. 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 address output multiplexers (see Figure 4-1) 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.

4.2 SIXTEEN-BIT COMPATIBILITY MODE

When the SIXTEEN-BIT COMPATIBILITY mode bit (see Figure 6-5 on page 6-10) is turned on, the following occur in the AGU:

- Move operations to/from any of the AGU registers (R0-R7, N0-N7 and M0-M7) clear the 8 MSBits of the destination.
- The 8 MSBits of any AGU address calculation result are cleared.
- The sign bit of the selected N register is bit15 instead of bit 23.
- The 8 MSBits of the address are ignored in the calculations of memory regions.

Note: Proper memory access operation is not guaranteed if an address register had non-zero bits in its 8 MSBits before changing to 16 bit com-

patibility mode. This is because the 8 MSBits will not be cleared when the register is the source for the address.

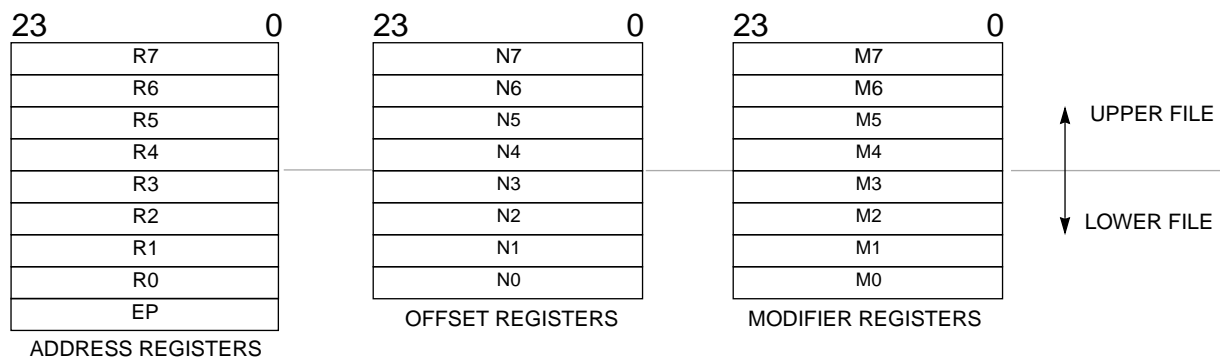
This mode of operation supports compatibility to object code written for the DSP56000 Family of Digital Signal Processors.

Due to pipelining, a change in the bit takes affect only after the following three instruction cycles. Inserting three NOP instructions after the instruction that changes the value of this bit will ensure proper operation.

4.3 PROGRAMMING MODEL

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

Figure 4-2. AGU Programming Model



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

The eight 24-bit address registers, R0 - R7, can contain addresses or general-purpose data. The 24-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.

4.3.1.1 Stack Extension Pointer (EP)

The contents of the 24-bit EP register is used to point to the stack extension in data memory whenever the stack extension is enabled and move operations to/from the on-chip hardware stack are needed. The EP register is a read/write register and is referenced implicitly by some instructions (DO, JSR, RTI, etc.) or directly e.g. by using the MOVEC instruction. The stack extension pointer is not initialized during hardware reset, and must be set (using a MOVEC instruction) prior to enabling the stack extension. For a more detailed description of the stack extension mode of operation, please refer to Section 6.3.5.

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

The eight 24-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 24-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.

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

The eight 24-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 \$FFFFFF on processor reset, which specifies linear arithmetic as the default type for address register update calculations.

4.4 ADDRESSING MODES

The DSP56300 Core provides four different addressing modes: register direct, address register indirect, PC relative and special (see Table 4-1).

4.4.1 Register Direct Mode

These effective addressing modes specify that the operand is in one (or more) of the 10 Data ALU registers, 24 address registers or 7 control registers.

4.4.1.1 Data or Control Register Direct

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

4.4.1.2 Address Register Direct

The operand is in one of the 24 address registers specified by an effective address in the instruction. This reference is classified as a register reference.

4.4.2 Address Register Indirect Modes

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.

4.4.2.1 No Update (Rn)

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.

4.4.2.2 Postincrement By 1 (Rn)+

The address of the operand is in the address register, Rn (see Table 4-1). After the operand address is used, it is incremented by 1 and stored in the same address register. The type of arithmetic used to calculate is determined by Mn. The Nn register is ignored.

4.4.2.3 Postdecrement By 1 (Rn)-

The address of the operand is in the address register, Rn (see Table 4-1). After the operand address is used, it is decremented by 1 and stored in the same address register. The type of arithmetic used to calculate is determined by Mn. The Nn register is ignored.

4.4.2.4 Postincrement By Offset Nn (Rn)+Nn

The address of the operand is in the address register, Rn (see Table 4-1). After the operand address is used, it is incremented by the contents of the Nn register and stored in the same address register. The type of arithmetic used to calculate is determined by Mn. The contents of the Nn register are unchanged.

4.4.2.5 Postdecrement By Offset Nn (Rn)-Nn

The address of the operand is in the address register, Rn (see Table 4-1). After the operand address is used, it is decremented by the contents of the Nn register and stored in the same address register. The type of arithmetic used to calculate is determined by

Mn. The contents of the Nn register are unchanged.

4.4.2.6 Indexed By Offset Nn ($Rn+Nn$)

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). The type of arithmetic used to calculate is determined by Mn. The contents of the Rn and Nn registers are unchanged.

4.4.2.7 Predecrement By 1 $-(Rn)$

The address of the operand is the contents of the address register, Rn, decremented by 1 (see Table 4-1). The contents of Rn are decremented and stored in the same address register. The type of arithmetic used to calculate is determined by Mn. The Nn register is ignored.

4.4.2.8 Short displacement ($Rn+\text{short displacement}$)

In this addressing mode the address of the operand is the sum of the contents of the address register Rn and a short displacement occupying 7 bits in the instruction word. The displacement is first sign extended to 24 bits and then added to Rn to obtain the address of the operand. The contents of the Rn register is unchanged. The type of arithmetic used to calculate is determined by Mn. The Nn register is ignored. This reference is classified as a memory reference.

4.4.2.9 Long displacement ($Rn+\text{long displacement}$)

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

4.4.3 PC Relative Modes

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

4.4.3.1 Short Displacement PC Relative

The short displacement occupies 9 bits in the instruction operation word. The displacement is first sign extended to 24 bits and then added to the PC to obtain the address of the operand.

4.4.3.2 Long Displacement PC Relative

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

4.4.3.3 Address Register PC Relative

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

4.4.4 Special Address Modes

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

4.4.4.1 Immediate Data

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

4.4.4.2 Immediate Short Data

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

4.4.4.3 Absolute Address

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

4.4.4.4 Absolute Short Address

For the Absolute Short addressing mode the address of the operand occupies 6 bits in the instruction operation word and it is zero extended. This reference is classified as a memory reference.

4.4.4.5 Short Jump Address

The operand occupies 12 bits in the instruction operation word. The address is zero extended to 24 bits. This reference is classified as a program reference.

4.4.4.6 I/O Short Address

For the I/O short addressing mode the address of the operand occupies 6 bits in the instruction operation word and it is one extended. I/O short is used with the bit manipulation and move peripheral data instructions.

4.4.4.7 Implicit Reference

Some instructions make implicit reference to the program counter (PC), system stack

(SSH, SSL), loop address register (LA), loop counter (LC) or status register (SR). The registers implied and their use is defined by the individual instruction descriptions (Appendix A).

4.5 ADDRESS MODIFIER TYPES

The DSP56300 Core address ALU supports linear, modulo, and reverse-carry arithmetic types for all address register indirect modes. These arithmetic types easily allow the creation of data structures in memory for FIFOs (queues), delay lines, circular buffers, stacks, and bit-reversed FFT buffers. Data is manipulated by updating address registers (pointers) rather than moving large blocks of data. The contents of the address modifier register, Mn, define the type of arithmetic to be performed for addressing mode calculations; for modulo arithmetic, the contents of Mn also specify the modulus. All address register indirect modes can be used with any address modifier. Each address register, Rn, has its own modifier register, Mn, associated with it.

4.5.1 Linear Modifier (Mn=\$XXFFFF)

Address modification is performed using normal 24-bit linear (modulo 16,777,216) arithmetic. A 24-bit offset, Nn, and ± 1 can be used in the address calculations. The range of values can be considered as signed (Nn from $-8,388,608$ to $+8,388,607$) or unsigned (Nn from 0 to $+16,777,216$) since there is no arithmetic difference between these two data representations.

4.5.2 Reverse-Carry Modifier (Mn=\$000000)

Reverse carry is selected by setting the modifier register to zero. 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 2^k -point FFT addressing and to unscramble 2^k -point FFT data. The range of values for Nn is 0 to $+8M$ (i.e., $Nn=2^{23}$), which allows bit-reverse addressing for FFTs up to 16,777,216 points.

4.5.3 Modulo Modifier (Mn=MODULUS-1)

The address modification is performed modulo M, where M ranges from 2 to $+32,768$ (see Table 4-2). Modulo M arithmetic causes the address register value to remain within an address range of size M, defined by a lower and upper address boundary.

The value $m=M-1$ is stored in the modifier register, Mn. The lower boundary (base

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

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 M_n . The boundaries are determined by the contents of R_n . Assuming the $(R_n)+$ 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 $(R_n)-$ 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).

Table 4-1. Addressing Modes Summary

Addressing Modes	Uses Mn Modifier	Operand Reference										Assembler Syntax
		S	C	D	A	P	X	Y	L	XY		
Register Direct												
Data or Control Register	No		✓	✓								
Address Register Rn	No				✓							
Address Modifier Register Mn	No				✓							
Address Offset Register Nn	No				✓							
Address Register Indirect												
No Update	No					✓	✓	✓	✓	✓		(Rn)
Postincrement by 1	Yes					✓	✓	✓	✓	✓		(Rn)+
Postdecrement by 1	Yes					✓	✓	✓	✓	✓		(Rn)–
Postincrement by Offset Nn	Yes					✓	✓	✓	✓	✓		(Rn)+Nn
Postdecrement by Offset Nn	Yes					✓	✓	✓	✓			(Rn)–Nn
Indexed by Offset Nn	Yes					✓	✓	✓	✓			(Rn+Nn)
Predecrement by 1	Yes					✓	✓	✓	✓			–(Rn)
Short/Long Displacement	Yes						✓	✓	✓			(Rn+displ)
PC Relative												
Short/Long Displacement PC Relative	No					✓						(PC+displ)
Address Register	No					✓						(PC+Rn)
Special												
Short/Long Immediate Data	No					✓						
Absolute Address	No					✓	✓	✓	✓			
Absolute Short Address	No						✓	✓	✓			
Short Jump Address	No					✓						
I/O Short Address	No						✓	✓				
Implicit	No	✓	✓			✓						
Notes:	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										

If an offset, Nn, is used in the address calculations, the 24-bit absolute value, |Nn|, must

be less than or equal to M for proper modulo addressing. If $N_n > M$, the result is data dependent and unpredictable, except for the special case where $N_n = P \times 2^k$, a multiple of the block size where P is a positive integer. For this special case, when using the $(R_n) + N_n$ addressing mode, the pointer, R_n , will jump linearly to the same relative address in a new buffer, which is P blocks forward in memory. Similarly, for $(R_n) - N_n$, 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 N_n is $-8,388,608$ to $+8,388,607$. The modulo arithmetic unit will automatically wrap around the address pointer by the required amount. This type address modification is useful for creating circular buffers for FIFOs (queues), delay lines, and sample buffers up to $8,388,607$ words long as well as for decimation, interpolation, and waveform generation. The special case of $(R_n) \pm N_n$ modulo M with $N_n = P \times 2^k$ is useful for performing the same algorithm on multiple blocks of data in memory — e.g., parallel infinite impulse response (IIR) filtering.

4.5.4 Multiple Wrap-Around Modulo Modifier

The multiple wrap-around addressing mode is selected by setting bit 15 of modifier register to one (see Table 4-2). 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 M_n least significant 15 bits while the 16th bit (bit 15) is set to one and the rest of the most significant 8 bits are considered don't care. 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 $M-1$).

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 N_n 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 $(R_n) + N_n$, $(R_n) - N_n$ and $(R_n + N_n)$ address updates (multiple wrap-around cannot occur with $(R_n) +$, $(R_n) -$ and $-(R_n)$ addressing modes).

4.5.5 Address-Modifier-Type Encoding Summary

Table 4-2 is a summary of the address modifier types discussed in the previous paragraphs. There are four modifier types:

- Linear Addressing
- Reverse-Carry Addressing
- Modulo Addressing
- Multiple Wrap-Around 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. The linear addressing is useful for general-purpose addressing. 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.

Table 4-2. Address-Modifier-Type Encoding Summary

Modifier Mn	Address Calculation Arithmetic
XX0000	Reverse-Carry (Bit-Reverse)
XX0001	Modulo 2
XX0002	Modulo 3
:	:
XX7FFE	Modulo 32767 ($2^{15}-1$)
XX7FFF	Modulo 32768 (2^{15})
XX8001	Multiple Wrap-Around Modulo 2
XX8003	Multiple Wrap-Around Modulo 4
XX8007	Multiple Wrap-Around Modulo 8
:	:
XX9FFF	Multiple Wrap-Around Modulo 2^{13}
XXBFFF	Multiple Wrap-Around Modulo 2^{14}
XXFFFF	Linear (Modulo 2^{24})

Notes:

XX means don't care

All other combinations are reserved

5 INSTRUCTION CACHE CONTROLLER

5.1 INTRODUCTION

The instruction cache may be viewed as a buffer memory between the main (external and probably slow) memory, and the fast CPU. The cache is used to store the program instructions that are frequently used. An increase in throughput may result when instruction words required by a program are available in the on-chip cache, and the time required to access them on the external bus is eliminated. Cache specific instructions are provided in the instruction set, permitting the user to lock sectors of the cache, and to flush the cache contents under software control. The instruction cache controller in the DSP56300 core is capable of controlling 1k or 2k of instruction cache memory array, with the following features:

Following is a summary of the instruction cache features:

- Software controlled enable bit in the chip extended mode register
- Mask programming selection between 1024 or 2048 locations
- 8 Way, Fully Associative, Sector Placement Policy
- One to Four Word Transfer Granularity
- LRU Sector Replacement Algorithm
- User Transparent - No user management required
- Individual Sector Locking/Unlocking
- Software controlled Global Cache Flush
- Cache controller status observability via the On-Chip Emulator

5.2 INSTRUCTION CACHE ARCHITECTURE

5.2.1 Instruction Cache Structure

The Instruction Cache is composed of the Memory Array and the Cache Controller. Figure 5-1 shows a block diagram of the instruction cache controller.

The Instruction Cache memory array contains 1024 (or 2048, mask programmed) 24-bit words, logically divided into eight 128 (256)-word cache sectors. Since there are 8 sectors of 128 (256) words each, in the internal program RAM, the 24 bit address is divided into the following two fields:

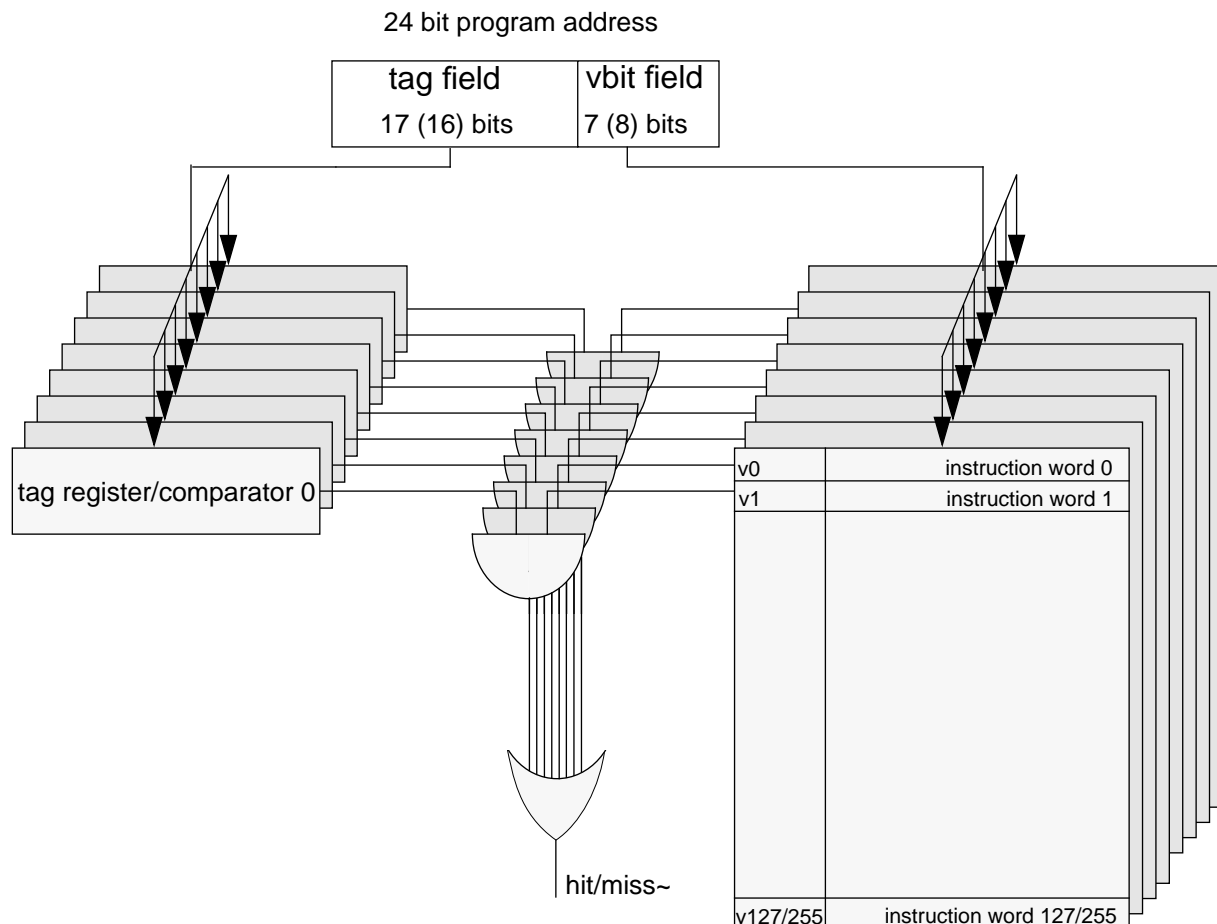
- vbit field; 7 (8) LSBits for the word displacement in the sector
- tag field; 17 (16) MSBits for the sector base address

The sectors placement algorithm is fully associative.

A 17 (16) bit tag is associated with every one of the 8 internal program memory RAM sectors. When the Cache Controller searches for a tag equal to the tag field of the current address, it compares it to the 8 tags in parallel using the 8 comparators.

Each word in each cache sector is associated with a cache word valid bit, that specifies whether or not the data in that word has already been fetched from external memory and is therefore valid. There is a total of 2048 (1024 of them are not used if the Icache size is 1024 bytes) valid bits, arranged as 8 banks of 128 or 256 valid bits each, one bank for every sector. Note that these valid bits are not available to the user, for direct use. The valid bits are cleared by the processor hardware RESET to indicate that the PRAM context has not been initialized.

Figure 5-1. Instruction Cache Block Diagram



5.2.2 Cache Programmer's Model

The Instruction Cache is controlled by two control bits:

-
- Cache Enable (CE) bit in the EMR part of the Chip Status Register (SR, bit 19). When CE is cleared the Instruction Cache is disabled. When CE is set the Instruction Cache is enabled.
 - Burst Enable (BE) bit in the EOM part of the Chip Operating Mode Register (OMR, bit 10). When BE is cleared the Instruction Cache Transfer Granularity on miss is a single-word. When BE is set, the Instruction Cache Transfer Granularity on miss is increased to a one to four burst block.

NOTE To assure proper operation, cache enable mode (CE bit in SR) should not be cleared while burst mode is enabled (BE bit in OMR is set).

- The Instruction Cache is supported by the instruction set via the following instructions: PLOCK, PLOCKR, PUNLOCK, PUNLOCKR, PFREE, PFLUSH, PFLUSHUN.

5.2.3 Cache Operation

5.2.3.1 program fetch

When the core generates an address for an instruction fetch, the cache controller compares its tag field to the tag values currently stored in the tag register file. This tag values are the tag fields of the base addresses of the memory sectors currently mapped into the cache.

5.2.3.2 hit

If there is a tag match, i.e. sector hit, then the valid bit of the corresponding word in that cache sector is checked by using the vbit field as an address to the valid bit array. If the valid bit is set, meaning the word in the cache has already been brought to the cache and is valid, then that word is fetched from the cache location corresponding to the desired address. This situation is called a cache hit meaning that both corresponding sector and corresponding instruction word are present and valid in the instruction cache. The Sector Replacement Unit (SRU) flags the sector as the Most Recently Used (MRU).

5.2.3.3 word miss when burst mode is disabled

If there is a tag match, i.e. sector hit, but the desired word is not flagged as valid (corresponding valid bit is cleared), then the cache initiates a read access to the external program memory, introducing wait states into the pipeline. The amount of wait states will be 1 plus the number of wait states that are programmed into the bus interface unit's control registers, reflecting the type of memory used. The Sector Replacement Unit (SRU) flags the sector as the Most Recently Used (MRU) and the fetched instruction is sent both to the core and copied to the relevant sector location. Then the valid bit of that word is set.

5.2.3.4 word miss when burst mode is enabled

If there is a tag match, i.e. sector hit, but the desired word is not flagged as valid (corresponding valid bit is cleared), then the cache initiates a burst of up to 4 read accesses to the external program memory. The exact number of fetch requests depends

on the two least significant bits of the address of the initiating fetch that was detected as miss -

- '11' - only one request will be initiated as if the burst mode was disabled
- '10' - two requests will be initiated and the number of wait states required by the memory type and speed will be added by two.
- '01' - three requests will be initiated and the number of wait states required by the memory type and speed will be added by three.
- '00' - four requests will be initiated and the number of wait states required by the memory type and speed will be added by four.

These external read accesses will introduce wait states into the pipeline. The amount of wait states for each fetch will be 1 plus the number of wait states that are programmed into the bus interface unit's control registers, reflecting the type of memory used. The Sector Replacement Unit (SRU) flags the sector as the Most Recently Used (MRU) and each of the fetched instruction is copied to the relevant sector location. Then the valid bit of that word is set.

5.2.3.5 sector miss

If there is no match between the tag field and all sector tag registers, meaning that the memory sector containing the requested word is not present in the cache, the situation is called a sector miss, which is another form of a cache miss. If a sector miss occurred, the cache's SRU selects the sector to be replaced. The cache controller then flushes the selected cache sector by clearing all corresponding valid bits, loads the corresponding tag register with the new tag field, and at the same time initiates an access to the external program memory, as described in Section 5.2.3.3 and Section 5.2.3.4. The sector is flagged as MRU, the fetched instruction is sent both to the core and copied to the relevant sector location and the valid bit of that word is set.

5.2.4 Default Mode On Hardware Reset

After hardware RESET the Instruction Cache is disabled and the DSP56300 Core operates in the PRAM Mode. The cache is initialized to the following initial condition:

- All valid bits will be cleared.
- All tag registers are initialized to hold the value \$1FFFF (\$FFFF).
- The LRU stack will hold a default descending order of sectors.
- All cache sectors are in the unlocked state.

5.2.5 Cache Locking

Cache locking is useful for locking some time-critical code parts in the cache memory. When a cache sector is locked, the Sector Replacement Unit (SRU) can not replace this sector even if it becomes the least recently used sector (bottom of LRU stack).

A sector can be locked by the instructions PLOCK or PLOCKR. Their operand is an effective memory address (absolute or PC-relative). The cache sector to which this address belongs (if there is such one), is locked. If the specified effective address does not belong to one of the current cache sectors, a memory sector containing this address will be allocated into the cache, thereby replacing the least recently used cache sector. This cache sector will be locked but empty. If all the cache sectors are already locked, this memory sector will not be allocated into the cache and the lock operation will not be executed. The locked cache sector becomes MRU.

Locking a cache sector, if it is already in the cache, does not affect the contents of it, the value of its valid bits or the corresponding tag register contents.

Note: PLOCK and PLOCKR are detected as illegal opcodes in PRAM Mode. Issuing these instructions when the cache is disabled will initiate the Illegal Interrupt. A distance of at least three instruction cycles (equivalent to three NOP instructions) should be maintained between an instruction that changes the value of the cache enable bit (CE) and one of the instructions PLOCK and PLOCKR.

5.2.6 Cache Unlocking

A locked sector can be unlocked to allow sector replacement from that cache sector. Unlocking can be performed in three different ways.

A locked sector can be unlocked by the PFREE, PUNLOCK or PUNLOCKR instructions. PUNLOCK/R's operand is an effective memory address (absolute or PC-relative). The memory sector containing this address is allocated into a cache sector (if its not already in a cache sector) and this cache sector is unlocked. If all the cache sectors are already locked, this memory sector will not be allocated into the cache and the unlock operation will not be executed. The unlocked cache sector becomes MRU, and is enabled for replacement by the LRU algorithm. Unlocking a locked cache sector using these instructions does not affect its contents, its tag, or its valid bits.

All the locked sectors can be unlocked simultaneously using the instruction PFREE, which provides the user the ability to reset the locking mechanism. Unlocking the sectors using PFREE, does not affect their contents (instructions already fetched into the sector storage area), their valid bits, their tag register contents or the LRU stack status. If a PFREE instruction is executed when none of the sectors are locked, than none of the tag registers, valid bits and LRU status will be changed.

The locked sectors could also be unlocked by the PFLUSH instruction. Unlocking the sectors, via PFLUSH, clears all the sector's valid bits and sets the LRU stack and tag

registers to their default values.

Note: PFREE, PUNLOCK and PUNLOCKR are detected as illegal opcodes in PRAM Mode. Issuing these instructions when the cache is disabled will initiate the Illegal Interrupt. A distance of at least three instruction cycles (equivalent to three NOP instructions) should be maintained between an instruction that changes the value of the cache enable bit (CE) and one of the instructions PFREE, PUNLOCK and PUNLOCKR.

5.2.7 Cache Flush

This operation is performed by executing the instructions PFLUSH or PFLUSHUN. The execution of PFLUSH causes a global cache flush that brings the cache to the hardware reset initial condition:

- All valid bits will be cleared.
- All tag registers are initialized to hold the value \$1FFFF (\$FFFF).
- The LRU stack will hold a default descending order of sectors.
- All cache sectors are in the unlocked state.

The execution of PFLUSHUN causes a flush only to the unlocked sectors and brings the cache to the following initial condition:

- All valid bits of the unlocked sectors will be cleared.
- All tag registers of the unlocked sectors are initialized to hold the value \$1FFFF (\$FFFF).
- The LRU stack will hold a default descending order of sectors.

Note: Coherency between PRAM mode and CACHE mode is not supported by the Instruction Cache Controller. It is not possible to fill the cache while in PRAM mode, and use the contents after switching to CACHE mode. The cache is automatically flushed when switching from CACHE to PRAM mode.

Note: PFLUSH and PFLUSHUN are detected as illegal opcodes in PRAM Mode. Issuing these instructions when the cache is disabled will initiate the Illegal Interrupt. A distance of at least three instruction cycles (equivalent to three NOP instructions) should be maintained between an instruction that changes the value of the cache enable bit (CE) and one of the instructions PFLUSH and PFLUSHUN.

5.2.8 Sector Replacement Unit

When a sector miss occurs, a cache sector must be selected to contain the new memory sector. The Sector Replacement Unit (SRU) determines which sector would be flushed from the cache, by constantly monitoring the requested addresses and the sectors usage.

The sector replacement policy is to replace the Least Recently Used (LRU) sector.

The LRU stack status is effected only in Cache Enable Mode by instruction fetch operations and by execution of the PFLUSH, PLOCK and PUNLOCK instructions. Locked cache sectors continue to “move” up and down the LRU stack. This implies that when picking the Least Recently Used sector (the one at the bottom of the LRU stack), locked sectors are skipped.

When the cache is initialized to the reset condition, the LRU stack holds a default descending order of sectors, i.e. sector number 0 is the Most Recently Used and sector number 7 is the Least Recently Used.

5.2.9 Data Transfers to/from ICACHE Space

Data transfers to/from the program memory can be accomplished by the DMA or by software, using MOVE instructions.

5.2.9.1 DMA transfers

DMA transfers have no effect on the Tag Register File, Valid Bit Array and LRU Stack even when the cache is enabled.

When the cache is disabled, the instruction cache memory space is considered a part of the internal program memory space. DMA transfers to/from this space will be executed without any limitation.

When the cache is enabled, the instruction cache memory space is considered a part of the external program memory space. DMA transfers to/from this space will be executed through the external memory expansion port. Coherency between the external program memory and the contents of the instruction cache is not maintained.

5.2.9.2 Software-Controlled transfers

The term “PMOVE” is used to indicate a MOVE instruction used to transfer data between the program memory space and any other source/destination. PMOVE transfers have no effect on the Tag Register File and LRU Stack even if the cache is enabled. The term “PMOVEW” is used to indicate a PMOVE transfer with the program memory space being the destination. The term “PMOVER” is used to indicate a PMOVE transfer with the program memory space being the source.

When the cache is disabled, the instruction cache memory space is considered a part of the internal program memory space. PMOVER from this space or PMOVEW to this space

will be executed without any limitation.

When the cache is enabled, PMOVE transfers are checked for a HIT or MISS:

- If the cache controller generates a HIT on the program memory space address, the data will be read from the cache memory array. Since PMOVE is not considered an instruction fetch operation, the LRU state will not be changed due to this transfer.
- If the cache controller generates a MISS on the program memory space address, the data will be read from the external program memory causing the amount of wait states as specified in the bus control register of the memory expansion port. The Cache state will not be changed due to this transfer. When in the Burst Mode, no burst will be initiated.

When the cache is enabled, PMOVEW transfers are also checked for a HIT or MISS:

- If the cache controller generates a SECTOR-HIT on the program memory space address, the data will be written both to the cache memory array and to the external program memory causing the amount of wait states as specified in the bus control register of the memory expansion port. The valid bit of the word will be set. The LRU stack will not be changed due to this transfer.
- If the cache controller generates a SECTOR-MISS on the program memory space address, the data will be written only to the external program memory causing the amount of wait states as specified in the bus control register of the memory expansion port. The Cache state will not be changed due to this transfer. When in the Burst Mode, no burst will be initiated.

WARNING

For proper operation, none of the three instructions before a PMOVE transfer should clear or set the Cache-Enable bit in the Status Register.

5.2.10 Cache Observability Via OnCE

The user is supplied with full non-intrusive system debug capability when in cache mode, having the ability to observe the cache status:

- what are the memory sectors that are currently mapped into cache
- which cache sectors are locked
- which cache sector is the Least Recently Used

-
- indication on the occurrence of HIT

This is accomplished in the debug mode, by reading the tag registers contents, lock bits, LRU bits and hit-status serially via the OnCE.

It is also possible to read the valid bits of specific cache locations. To check, whether an address, which MSBs are in a tag register, is in the cache, one should send the opcode of a MOVEM from this address. The bit 5 of OSCR will indicate the value of the valid bit.

- Each read of the cache status via the OnCE should access all the 9 registers, so that such a read starts every time from the tag #0.
- Each read of the cache status via the OnCE should be made only when the chip is in the debug mode of operation.

6 PROGRAM CONTROL UNIT

This section describes the program control unit (PCU) hardware and its programming model. The instruction pipeline description is also included since understanding the pipeline is particularly important in understanding the DSP56300 Core. Note that the pipelined operation remains essentially hidden from the user, thus easing programmability.

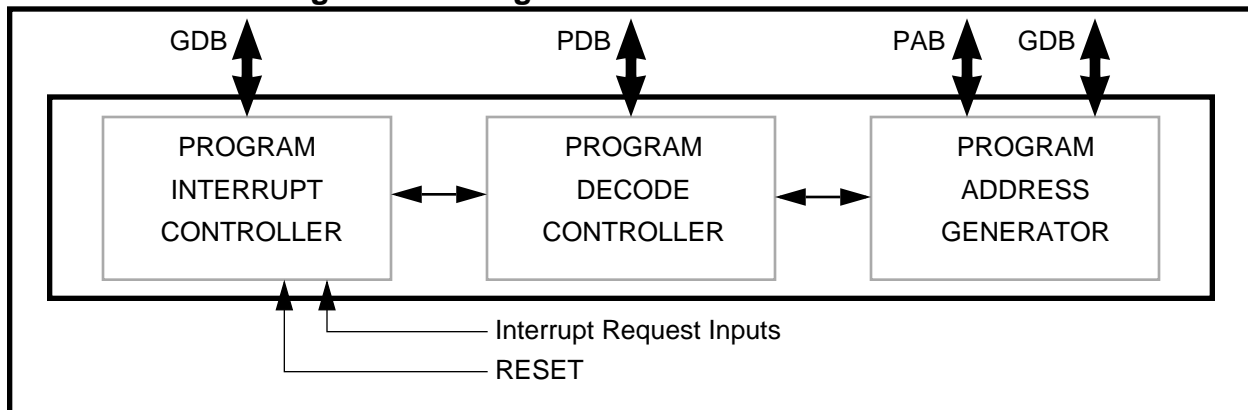
6.1 OVERVIEW

The program control unit performs instruction prefetch, instruction decoding, hardware DO loop control and exception processing. Its programmers model consists of eight read/write 24-bits registers, one read/write 5-bits register and a hardware system stack (SS). In addition to the standard program flow-control resources (e.g. interrupts, jumps), the program control unit support hardware DO loop and REPEAT mechanism.

The SS is a 16-level by 48-bit separate internal memory used to automatically store the PC and SR during subroutine calls and long interrupts. The SS automatically stores the LC and LA registers in addition to the PC and SR registers for hardware loops. All other data and control registers can be stored in the SS via software control. Each location in the SS is addressable as two 24-bit registers, system stack high (SSH) and system stack low (SSL), which are pointed to by the four LSBs of a 24-bit stack pointer (SP). The SS is extended in the data memory, in a space specified by the stack control registers that monitor the accesses to the SS. This hardware will copy the Least-Recently-Used location of the SS to data memory in case the on-chip hardware stack is full, and will bring data from data memory in case the on-chip hardware stack is empty.

The program control unit implements a seven-stage (prefetch-I, prefetch-II, decode, address gen-I, address gen-II, execute-I, execute-II) pipeline and controls the five processing states of the DSP56300 Core: normal, exception, reset, wait, and stop.

Figure 6-1. Program Control Unit Architecture



6.2 PROGRAM CONTROL UNIT ARCHITECTURE

The program control unit (Figure 6-1) consists of three hardware blocks:

Program Decode Controller (PDC), that decodes the 24-bit instruction loaded into the instruction latch and generates all signals necessary for pipeline control.

Program Address Generator (PAG), which contains all the hardware needed for program address generation, system stack and loop control.

Program Interrupt Controller (PIC) which arbitrates among all interrupt requests (internal and the five external: IRQA, IRQB, IRQC, IRQD and NMI) and generates the appropriate interrupt vector address.

6.2.1 Instruction Pipeline

The program control unit implements a seven-stage pipelined architecture in which concurrent stages of this pipeline occur. These seven stages consist of two prefetch stages, one decode stage, two address generation stages and two execute stages, as illustrated in Figure 6-2 and described in Table 6-1.

Although composed of many stages, the pipelined operation remains essentially hidden from the user, thus easing programmability. This is achieved by means of interlock hardware that is present in every execution unit of the processor. Due to this feature, programs written for the DSP56000/1/2 will execute correctly on the DSP56300 Core without any need for modification. Modification of the program may reduce the occurrence

of interlocks and improve execution speed.

Figure 6-2. Seven Stage Pipeline

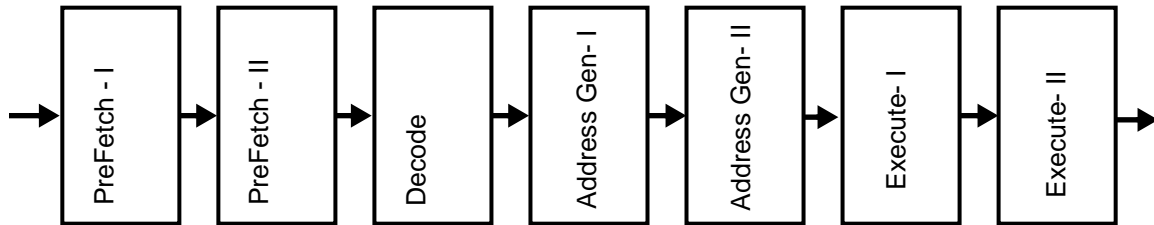


Table 6-1. Seven Stage Pipeline

Pipeline Stage	Description of Pipeline Stage
PreFetch-I	<ul style="list-style-type: none">• Address generation for Program Fetch• Increment PC
PreFetch-II	<ul style="list-style-type: none">• Instruction word read from memory
Decode	<ul style="list-style-type: none">• Instruction Decode
Address Gen-I	<ul style="list-style-type: none">• Address generation for Data Load/Store operations
Address Gen-II	<ul style="list-style-type: none">• Address pointer update
Execute-I	<ul style="list-style-type: none">• Read source operands to Multiplier and Adder• Read source register for memory store operations• Multiply• Write destination register for memory load operations
Execute-II	<ul style="list-style-type: none">• Read source operands for Adder if written by previous ALU operation• Add• Write Adder results to the Adder destination operand• Write Multiplier results to the Multiplier destination operands

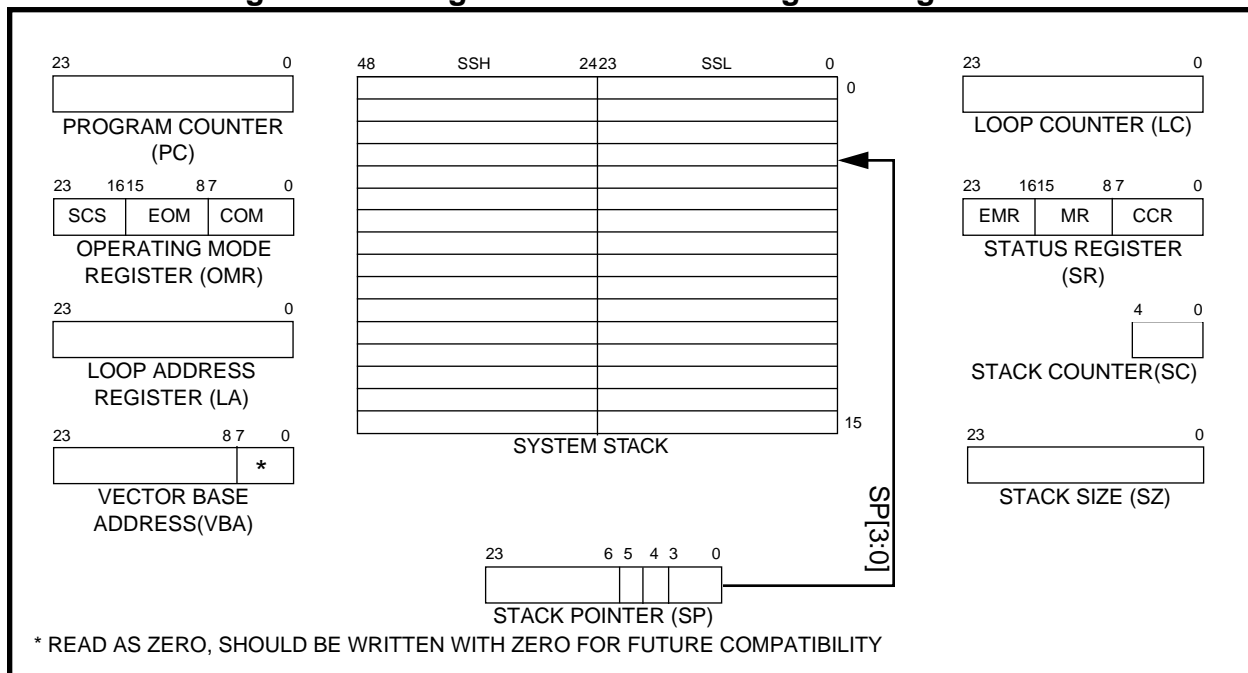
6.2.2 Clock Oscillator

The DSP56300 Core uses a two-phase clock for instruction execution; therefore, the clock runs at the same rate as the instruction execution. The clock can be provided by connecting an external crystal between XTAL and EXTAL, or by an external oscillator connected to EXTAL. The PLL can be used in order to determine the internal frequency related to the external.

6.3 PROGRAMMING MODEL

The program control unit features LA and LC registers dedicated to supporting the hardware DO loop instruction in addition to the standard program flow-control resources, such as a PC, SR, and SS. All registers are read/write to facilitate system debugging. Figure 6-3 shows the program control unit programming model with the registers and the SS. The following paragraphs give a detailed description of each register.

Figure 6-3. Program Control Unit Programming Model.



6.3.1 Program Counter (PC)

This is a special-purpose 24-bit address register that contains the address of instruction words in the program memory space. The PC can point to instructions, data operands, or addresses of operands. References to this register are always inherent and are implied by most instructions. The PC is stacked when hardware loops are initialized, when a JSR is performed, or when a long interrupt occurs. The PC is the source for the calculation of the real address in all position-independent instructions (like BRA).

6.3.2 Vector Base Address Register (VBA)

The VBA is a 24-bit register, 8 of them (bits 7-0) are read-only and are always cleared. The VBA is used as a base address of the interrupt vector and interrupt vector+1. When executing a fast or long interrupt, the vector address bits 7-0 are driven from the program interrupt control unit while bits 23-8 are driven from the VBA. The VBA register is a read/

write register that is referenced implicitly by interrupt processing or directly by the MOVEC instruction. The VBA is cleared during hardware reset.

6.3.3 Loop Counter Register (LC)

The LC register is a special read/write 24-bit counter used to specify the number of times a hardware program loop is to be repeated, in the range of 0 to $(2^{24}-1)$. This register is stacked into the SSL by a DO instruction and unstacked by end-of-loop processing or by execution of an ENDDO and BRKcc instructions. The LC is used also in the REP instruction, to specify the number of times the repeated instruction is to be repeated.

6.3.4 Loop Address Register (LA)

The contents of the 24-bit LA register indicate the location of the last instruction word in a hardware loop. This register is stacked into the SSH by a DO instruction and is unstacked by end-of-loop processing or by execution of an ENDDO and BRKcc instructions. The LA register, a read/write register, is written by a DO instruction and read by the SS when stacking the register.

6.3.5 System Stack (SS)

The System Stack (SS) is a separate 16x48-bit internal memory divided into two banks: System Stack High (SSH) and System Stack Low (SSL), 24 bits wide each. The SS is used for the following main tasks:

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

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

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

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

The SS may be extended in the data memory by means of control hardware that monitors the accesses to the SS. This extension is enabled by Stack Extension Enable (SEN) bit in the chip Operating Mode Register (OMR). If this bit is cleared, the extension of the system stack is disabled and the amount of nesting is determined by the limited level of the hardware stack (15, one location is unusable when the stack extension is disabled). Up to 15 long interrupts, 7 DO loops, 15 JSRs, or combinations of these can be accommodated by the SS when its extension in data memory is disabled. When the SS limit is exceeded (either in the extended or in the non-extended mode), a nonmaskable stack error interrupt occurs.

By enabling the Stack extension, the limits on the level of nesting of subroutines or DO loops can be set to any desired value.

A stack extension algorithm is applied to all accesses to the stack:

- If an explicit (e.g. move to ssh) or implicit (e.g. jsr) push operation is performed, then the stack is examined by the stack extension control logic after that push has finished. If the on-chip hardware stack is full, then the least recently used word is moved into data memory to the location specified by the stack extension pointer (EP).
- If an explicit (e.g. move from ssh) or implicit (e.g. rts) pop operation is performed, then the stack is examined by the stack extension control logic after that pop has finished. If the on-chip hardware stack is empty, then the stack is loaded from the location (in data memory) specified by the stack extension pointer (EP).

6.3.6 Stack Extension Pointer (EP)

The contents of the 24-bit EP register is used to point to the stack extension in data memory whenever the stack extension is enabled and move operations to/from the on-chip hardware stack are needed. The EP register is located in the Address Generation Unit (AGU). For more details, please refer to Section 4.3.1.

6.3.7 Stack Size Register (SZ)

The 24-bit SZ register determines the number of data words allocated in memory for the stack in the extended mode. The extended stack overflow flag is generated when the value in SP equals the value in SZ. The stack size register is not initialized during hardware reset, and must be set (using a MOVEC instruction) prior to enabling the stack extension.

6.3.8 Stack Counter Register (SC)

The 5-bit SC register is used to monitor how many entries of the hardware stack are in use. The SC register is a read/write register and is referenced implicitly by some

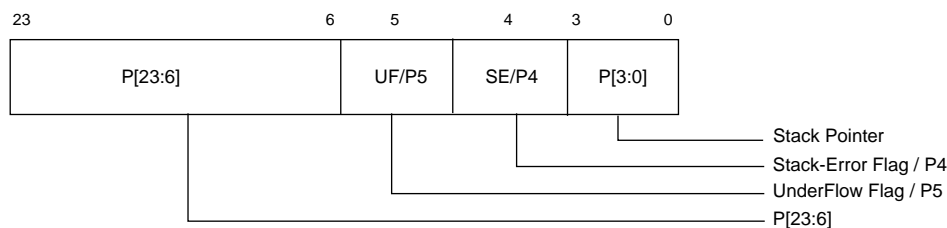
instructions (DO, JSR, RTI etc.) or directly by the MOVEC instruction. The stack counter register is cleared during hardware reset.

Note: During normal operation, the Stack Counter register should not be written. If a task switch is needed, writing a value greater than 14 or smaller than 2 will automatically activate the stack extension control hardware.
For proper operation, the SC should not be written with values greater than 16.

6.3.9 Stack Pointer Register (SP)

The 24-bit SP register indicates the location of the top of the SS. The status of the SS is also indicated in SP, when the extended mode is disabled (underflow, empty, full, and overflow). The SP register is referenced implicitly by some instructions (DO, JSR, RTI, etc.) or directly by the MOVEC instruction. The SP register format, shown in Figure 6-4, is described in the following paragraphs. The SP register is implemented as a 24-bit counter that addresses (selects) a 16-locations stack with its four LSBs. The possible SP values in the non-extended mode are shown in Table 6-2.

Figure 6-4. SP Register Format



6.3.9.1 Stack Pointer (Bits 0–3)

The SP points to the last used location on the SS. Immediately after hardware reset, these bits are cleared (SP=0), indicating that the SS is empty.

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

6.3.9.2 Stack Error Flag / P4 bit (Bit 4)

This is a dual function bit. In the extended mode it acts as bit 4 of the Stack Pointer, as part of a 24-bit up/down counter. In the non-extended mode, it serves as the stack error (SE) flag that indicates that a stack error has occurred. The transition of the stack error flag from zero to one in the non-extended mode causes a priority level-3 stack error exception.

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

Any implied pull operation with SP equal to zero will cause a stack error exception, and the SP will read \$0000FF (or \$0000FE if an implied double pull occurs). During such case, the stack error bit is set as shown in Table 6-2.

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

Table 6-2. SP Register Values in the non-extended mode

UF	SE	P3	P2	P1	P0	Description
1	1	1	1	1	0	Stack Underflow condition after double pull
1	1	1	1	1	1	Stack Underflow condition
0	0	0	0	0	0	Stack Empty (RESET); Pull causes underflow
0	0	0	0	0	1	Stack Location 1
.	Stack Locations 2 - 13
0	0	1	1	1	0	Stack Location 14
0	0	1	1	1	1	Stack Location 15; Push causes overflow
0	1	0	0	0	0	Stack Overflow condition
0	1	0	0	0	1	Stack Overflow condition after double push

6.3.9.3 Underflow Flag / P5 bit (Bit 5)

This is a dual function bit. In the extended mode it acts as bit 5 of the Stack Pointer, as part of a 24-bit up/down counter. In the non-extended mode, the underflow flag is set when a stack underflow occurs. The stack underflow flag is a “sticky bit”, i.e. once the stack error flag is set, the underflow flag will not change state until explicitly written by a move instruction. The combination of “underflow=1” and “stack error=0” is an illegal combination and will not occur unless it is forced by the user. Also see the description for the stack error flag (Section 6.3.9.2) for additional information.

6.3.10 Status Register (SR)

This 24-bit register consists of an 8-bit condition code register (CCR), 8-bit mode register (MR) and an 8-bit extended mode register (EMR). The SR is stacked when program

looping is initialized, when a JSR is performed, or when interrupts occur (except for no-overhead fast interrupts). The SR format is shown in Figure 6-5.

Each of the SR bits are masked programmable. They can be programmed to one of the following configurations:

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

The two reserved bits are also outputs of the DSP56300 core, with derivative-dependent functionality. These outputs are also mask programmed to one of the following states:


- connected to the SR bit, reflecting its state
- connected to GND (forced to '0')

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

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

Figure 6-5. Status Register Format

EMR								MR								CCR								
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
CP1:0				RM	SM	CE		SA	FV	LF	DM	SC		S1:0		I1:0	S	L	E	U	N	Z	V	C
CP1		- Core Priority Bit 1						LF		- DO-Loop Flag						S		- Scaling Bit						
CP0		- Core Priority Bit 0						DM		- Double Precision Multiply						L		- Limit						
RM		- Rounding Mode						SC		- Sixteen-Bit Compatibility						E		- Extension						
SM		- Arithmetic Saturation						S1		- Scaling Mode Bit 1						U		- Unnormalized						
CE		- Instruction Cache Enable						S0		- Scaling Mode Bit 0						N		- Negative						
SA		- Sixteenth-Bit Arithmetic						I1		- Interrupt Mask Bit 1						Z		- Zero						
FV		- DO-Forever Flag						I0		- Interrupt Mask Bit 0						V		- Overflow						
																C		- Carry						

 - Reserved bit. Read as zero, should be written with zero for future compatibility

The EMR is a special-purpose control register defining the current system state of the processor. The EMR bits are affected by processor reset, exception processing, DO FOREVER, ENDDO (end current DO loop), BRKcc, RTI (return from interrupt) and TRAP instructions, and by instructions that specify SR as its destination (e.g. MOVEC). During processor reset, all EMR bits will be cleared.

6.3.10.1 Carry (Bit 0)

The carry (C) bit is set if a carry is generated out of the MSB of the result in an addition operation. This bit is also set if a borrow is generated in a subtraction operation; Otherwise, this bit is cleared. 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.

6.3.10.2 Overflow (Bit 1)

The overflow (V) bit is set if an arithmetic overflow occurs in the 56-bit result; Otherwise, this bit is cleared. This bit indicates that the result cannot be represented in the accumulator register; thus, the register has overflowed. In Arithmetic Saturation Mode, an arithmetic overflow occurs if the Data ALU result is not representable in the accumulator without the extension part, i.e. 48-bit accumulator (32-bit in Sixteen Bit Mode).

6.3.10.3 Zero (Bit 2)

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

6.3.10.4 Negative (Bit 3)

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

6.3.10.5 Unnormalized (Bit 4)

The unnormalized (U) bit is set if the two MSBs of the most significant (MSP) portion of the result are identical. Otherwise, this bit is cleared. The MSP portion of the A or B accumulators is defined by the scaling mode. The U bit is computed as described in Table 6-3.

6.3.10.6 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 is defined by the scaling mode as described in Table 6-4. 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.

Table 6-3. Unnormalized Bit definition

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

Table 6-4. Extension Bit definition

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

6.3.10.7 Limit (Bit 6)

The limit (L) bit is set if the overflow bit is set or if the data shifter/limiter circuits perform a

limiting operation. In arithmetic Saturation Mode, the limit bit is also set when an arithmetic saturation occurs in the Data Alu result; otherwise, it is not affected. The L bit is cleared only by a processor reset or by an instruction that specifically clears it, which allows the L bit to be used as a latching overflow bit (i.e., a “sticky” bit). L is affected by data movement operations that read the A or B accumulator registers.

6.3.10.8 Scaling (Bit 7)

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

S0	S1	Scaling Mode	S equation
0	0	No scaling	$S = (A_{46} \text{ XOR } A_{45}) \text{ OR } (B_{46} \text{ XOR } B_{45}) \text{ OR } S \text{ (previous)}$
0	1	Scale down	$S = (A_{47} \text{ XOR } A_{46}) \text{ OR } (B_{47} \text{ XOR } B_{46}) \text{ OR } S \text{ (previous)}$
1	0	Scale up	$S = (A_{45} \text{ XOR } A_{44}) \text{ OR } (B_{45} \text{ XOR } B_{44}) \text{ OR } S \text{ (previous)}$
1	1	Reserved	S undefined

6.3.10.9 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

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

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

6.3.10.11 Reserved SR Bit (Bit 12)

This bit is reserved for future expansion; It will read as zero during DSP56300 Core read operations and should be written with zero for future compatibility.

6.3.10.12 Sixteen-Bit Compatibility Mode (Bit 13)

The Sixteen-Bit Compatibility Mode (SC) enables full compatibility to object code written for the DSP56000 Family of Digital Signal Processors. When the SC bit is set, move operations to/from any of the AGU registers and to/from any of the PCU registers clear the 8 MSBits of the destination. The SC is cleared during processor reset.

6.3.10.13 Double Precision Multiply Mode (Bit 14)

The double precision multiply (DM) bit enables the operation of four multiply/multiply-accumulate operations, for the implementation of a double precision algorithm. This algorithm involves the multiplication of two 48-bit operands with a 96-bit result. The mode is disabled by clearing the DM bit.

Note: The Double Precision Multiply Mode is supported in order to maintain object code compatibility with the 56k Family of Digital Signal Processors. For a more efficient way of executing double precision multiply, please refer to Section 3.

While in Double Precision Multiply mode, the behavior of the four specific operations listed in the double precision algorithm is modified. Therefore these operations (with those

specific register combinations) should not be used, while in Double Precision Multiply mode, for any other purpose but for the double precision multiply algorithm. All other Data ALU operations (or the four listed operations but with other register combination) may be used.

The double precision multiply algorithm uses the Y0 register at all stages. Therefore Y0 should not be changed when running the double precision multiply algorithm. 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. The DM is cleared during a processor reset.

6.3.10.14 DO-Loop Flag (Bit 15)

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

6.3.10.15 DO-Forever flag (Bit 16)

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

6.3.10.16 Sixteen-Bit Arithmetic Mode (Bit 17)

The Sixteen-Bit Arithmetic Mode (SA) when set, enables the sixteen bit arithmetic mode of operation. In this mode the rounding of the arithmetic operation will be performed on bit 15 of the accumulator A1/B1 instead of the usual bit 23 of A0/B0. The scaling, as well as the shifting/limiting operation of the Data-ALU will be affected accordingly. The SA bit is cleared during processor reset. At the start of a long interrupt service routine, the SR (including the SA) is pushed on the SS and the SA is cleared. The SA is cleared during a processor reset.

6.3.10.17 Reserved SR Bit (Bit 18)

This bit is reserved for future expansion; It will read as zero during DSP56300 Core read operations and should be written with zero for future compatibility.

6.3.10.18 Cache Enable (Bit 19)

The Cache Enable (CE) bit is used to enable or to disable the operation of the instruction cache controller. If the bit is set, the cache is enabled, instructions are cached into the internal PRAM and fetch from there. If the bit is cleared, the cache is disabled and the DSP56300 Core will fetch instructions from external or internal program memory, according to the memory space table of the specific DSP56300 Core-based chip. This bit is cleared during a processor reset.

NOTE To guarantee proper operation cache enable mode (CE bit in SR) should not be cleared while burst mode is enabled (BE bit in OMR is set).

6.3.10.19 Arithmetic Saturation Mode (Bit 20)

The Arithmetic Saturation Mode (SM) bit, when set, selects automatic saturation on 48 bits for the results going to the accumulator. This saturation is done by a special circuit inside the MAC unit. The purpose of this bit is to provide an arithmetic saturation mode for algorithms which do not recognize or cannot take advantage of the extension accumulator. This bit is cleared during processor reset.

6.3.10.20 Rounding Mode (Bit 21)

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

6.3.10.21 Core Priority (Bits 22 and 23)

Under the control of CDP1:0 bits in the Operating Mode Register (OMR), see Section 6.3.11.6, the CORE priority bits, CP1 and CP0, specify the priority of CORE accesses to external memory. These bits are compared against the priority bits of the active DMA

OMR - CDP1:0	CP1:0	Core Priority
00	00	0 (lowest)
00	01	1
00	10	2
00	11	3 (highest)
01	xx	see Section 6.3.11.6
10	xx	see Section 6.3.11.6
11	xx	see Section 6.3.11.6

channel. If CORE priority is greater than DMA priority, the DMA will wait for a free access slot in the external bus. If CORE priority is less than DMA priority, the CORE will wait for a free access slot in the external bus. If CORE priority equals to the DMA priority, CORE and DMA will access the external bus in a linear fix pattern. The Core priority bits are set during processor reset.

6.3.11 Operating Mode Register

The OMR is a 24-bit register, partitioned into three bytes. The least significant byte of OMR (bits 7-0) is the Chip Operating Mode byte (COM) which is used to determine the operating mode of the chip. This byte is only affected by processor reset and by instructions directly referencing the OMR: ANDI, ORI or other instructions that specify OMR as a destination (e.g. MOVEC). During processor reset, the chip operating mode bits (MD,MC, MB and MA) will be loaded from the external mode select pins MODD,MODC, MODB and MODA respectively.

Each of the OMR bits are masked programmable. They can be programmed to one of the following configurations:

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

Some of the reserved bits, as described later, are also outputs of the DSP56300 core, with derivative-dependent functionality. These outputs are also mask programmed to one of the following states:


- connected to the SR bit, reflecting its state
- connected to GND (forced to '0')

The middle part of OMR (bits 15-8) is the Extended Chip Operating Mode byte (EOM) which is used to determine the operating mode of the chip. This byte is only affected by processor reset and by instructions directly referencing the OMR: ANDI, ORI or other instructions that specify OMR as a destination (e.g. MOVEC).

The most significant byte of OMR (bits 23-16) is the System Stack Control Status byte (SCS) which is used to control and monitor the Stack extension in the data memory. The SCS byte is referenced implicitly by some instructions (DO, JSR, RTI, etc.) or directly by the MOVEC instruction.

Figure 6-6. Operating Mode Register (OMR) Format

SCS								EOM								COM							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			SEN	WRP	EOV	EUN	XYS				BRT	TAS	BE	CDP1:0		MS	SD		EBD	MD	MC	MB	MA
SEN - Stack Extension Enable								BRT - Bus Release Timing								MS - Memory Switch Mode							
WRP - Extended Stack Wrap Flag								TAS - TA Synchronize Select								SD - Stop Delay							
EOV - Extended Stack Overflow Flag								BE - Burst Mode Enable								EBD - External Bus Disable							
EUN - Extended Stack Underflow Flag								CDP1 - Core-Dma Priority 1								MD - Operating Mode D							
XYS - Stack Extension Space Select								CDP0 - Core-Dma Priority 0								MC - Operating Mode C							
																MB - Operating Mode B							
																MA - Operating Mode A							

 - Reserved bit. Read as zero, should be written with zero for future compatibility

6.3.11.1 Chip Operating Mode (Bits 0, 1,2 and 3)

The chip operating mode bits MD,MC, MB and MA, indicate the operating mode of the DSP56300 Core. On processor reset, these bits are loaded from the external mode select pins, MODD, MODC, MODB and MODA, respectively. After the DSP56300 Core leaves the reset state, MD, MC, MB and MA can be changed under program control.

6.3.11.2 External Bus Disable (Bit 4)

The External Bus Disable (EBD) bit is use to disable the external bus controller, in order to reduce the power consumption when external memories are not used. When EBD bit is set the external bus controller is disabled and external memory should not be accessed. When the EBD bit is cleared the external bus controller is enabled and external access may be performed. The EBD bit is cleared on hardware reset. For a detailed description of the EBD and it's applications please refer to Chapter 2.

6.3.11.3 Reserved COM Bit (Bit 5)

This bit is reserved for future expansion; it will read as zero during DSP56300 Core read operations and should be written with zero for future compatibility. The bit is also output of the DSP56300 core, with derivative-dependent functionality. The output is mask programmed to one of the states mentioned in previous paragraph.

6.3.11.4 Stop Delay (Bit 6)

The STOP instruction causes the DSP56300 Core to indefinitely suspend processing in the middle of the STOP instruction. When exiting the stop state, if the stop delay bit is cleared, a 128K clock cycle delay is selected before continuing the stop instruction cycle. However, if the stop delay bit is set, the delay before continuing the instruction cycle is 16 clock cycles. The long delay allows a clock stabilization period for the internal clock to begin oscillating and to stabilize. When a stable external clock is used, the shorter delay allows faster start-up of the DSP56300 Core. The Stop Delay bit is cleared during processor reset.

6.3.11.5 Memory Switch (Bit 7)

The Memory Switch (MS) Mode bit is used to turn on the memory space switch mode in which some addresses of the internal data memory (X, Y or both) become part of the chip internal program RAM. This bit is cleared during processor reset.

NOTE 1 Program data placed in PRAM/I-Cache area changes its placement after the setting of MS bit, such as it always occupies the top most internal PRAM addresses.

NOTE 2 To assure proper operation, six NOP instructions should be placed after instruction that changes MS bit.

NOTE 3 To assure proper operation, MS bit should not be set while I-Cache is enabled (CE bit is set in SR).

6.3.11.6 Core-Dma Priority Bits (Bits 9 and 8)

The Core-Dma Priority (CDP1, CDP0) bits specify the priority between the Core accesses and DMA accesses to external bus.

CDP1:0	Core-Dma Priority
00	determined by comparing CP1:0 with the active DMA channel priority
01	DMA accesses have higher priority than CORE accesses
10	DMA accesses have the same priority as the CORE accesses
11	DMA accesses have lower priority than the CORE accesses

These bits are set during processor reset.

6.3.11.7 Burst Mode Enable (Bit 10)

The burst mode enable (BE) bit is used to enable or to disable the burst mode in the memory expansion port during instruction cache miss. If the bit is cleared, the burst mode is disabled and only one program word will be fetched from the external memory when an instruction cache miss condition is detected. If the bit is set, the burst mode is enabled, and up to four program words will be fetched from the external memory when an instruction cache miss is detected. For detailed description of the Burst Mode, refer to Chapter 5. This bit is cleared by hardware reset.

6.3.11.8 TA Synchronize Select (Bit 11)

The TA synchronize select (TAS) bit is used to select the synchronization method for the input Port A pin - \overline{TA} (Transfer Acknowledge). If TAS is cleared, the user is responsible to assert the TA pin synchronously to the chip clock, as described in the detailed data sheet. If TAS is set, the \overline{TA} input pin is synchronized inside the chip, thus eliminating the need for an off-chip synchronizer. The user must negate TA pin synchronously to the chip clock, independently of the TAS bit value. See Section 2.2.4 for more details. The TAS bit is cleared on hardware reset.

6.3.11.9 Bus Release Timing (Bit 12)

The Bus Release Timing (BRT) bit is used to select between fast or slow bus release. If BRT is cleared, a fast bus release mode is selected (i.e. no additional cycles are added to the access and BB# is not guaranteed to be the last port A pin that is tri-stated at the end of the access). If BRT is set, a slow bus release mode is selected (i.e. additional one cycle is added to the access, and BB# is the last port A pin that is tri-stated at the end of the access). The BRT bit is cleared on hardware reset. For a detailed description of the Bus Release Modes and their applications please refer to Chapter 2.

6.3.11.10 Reserved EOM Bits (Bits 15, 14 and 13)

These bits are reserved for future expansion; they will read as zero during DSP56300 Core read operations and should be written with zero for future compatibility. These bits are also outputs of the DSP56300 core, with derivative-dependent functionality. The outputs are mask programmed to one of the states mentioned in previous paragraph.

6.3.11.11 XY Select for Stack extension (Bit 16)

The XY Select bit for the Stack extension determines if the extension will be mapped onto the X memory space or onto the Y memory space. If the bit is clear, then the stack extension is mapped onto the X memory space. If it is set, the stack extension is mapped to the Y memory space. This bit is cleared by hardware reset.

6.3.11.12 Extended Stack Underflow Flag (Bit 17)

The extended stack underflow (EUN) flag is set when a stack underflow occurs in the stack extended mode. Extended stack underflow is recognized when a pull operation is requested when SP equals to 0, and the extended mode is enabled by the EN bit. The extended stack underflow flag is a “sticky bit” i.e. the only way to clear this bit is by hardware reset or by an explicit move operation to the OMR. The transition of the extended stack underflow flag from zero to one causes a priority level-3 stack error exception. The extended stack underflow flag is cleared by hardware reset.

6.3.11.13 Extended Stack Overflow Flag (Bit 18)

The extended stack overflow (EOV) flag is set when a stack overflow occurs in the stack extended mode. Extended stack overflow is recognized when a push operation is requested while SP equals to SZ (Stack Size register), and the extended mode is enabled by the EN bit. The extended stack overflow flag is a “sticky bit” i.e., the only way to clear this bit is by hardware reset or by an explicit move operation to the OMR. The transition of the extended stack overflow flag from zero to one causes a priority level-3 stack error exception. The extended stack overflow flag is cleared by hardware reset.

6.3.11.14 Extended Stack Wrap Flag (Bit 19)

The extended stack wrap (WR) flag is set when it is first recognized that a copy from the on-chip hardware stack to the stack extension memory is needed. This flag may be used during the debugging phase of the software as means of evaluating and increasing the speed of the software implemented algorithms. The extended stack wrap flag is a “sticky bit” i.e., the only way to clear this bit is by hardware reset or by an explicit move operation to the OMR. The extended stack wrap flag is cleared by hardware reset.

6.3.11.15 Extended Stack Enable (Bit 20)

The extended stack enable (EN) bit is used to enable or to disable the stack extension in the data memory. If the EN bit is set, the extension is enabled. This bit is cleared by hardware reset, thus disabling the stack extension as default.

6.3.11.16 Reserved SCS Bits (Bits 21-23)

These OMR bits, reserved for future expansion, will read as zero during DSP56300 Core read operations, and should be written with zero for future compatibility. These bits are not outputs of the DSP56300 core.

6.4 SIXTEEN-BIT COMPATIBILITY MODE

When the SIXTEEN-BIT COMPATIBILITY mode bit (SC, see Figure 6-5 on page 6-10) is set, move operations to/from any of the following PCU registers clear the 8 MSBits of the destination: LA, LC, SP, SSL, SSH, EP, SZ, VBA and SC. This guarantees compatibility for object code written for the DSP56000 Family of Digital Signal Processors.

If the source is either SR or OMR, then the 8 MSBits of the destination will also be cleared. If the destination is either SR or OMR, then the 8 MSBits of the destination will be left unchanged.

In order to change the value of one of the 8 MSBits of SR or OMR, the SIXTEEN-BIT COMPATIBILITY mode bit (SC) should be cleared.

The LOOP count is also affected by the SIXTEEN-BIT COMPATIBILITY mode bit. If it is cleared (normal operation), then loop count value of 0 will cause the loop body to be skipped, and loop count value of 0xFFFFF will cause execution of the loop the maximum number of $2^{24}-1$ times. If the bit is set, the loop count value of 0 will cause the loop to be executed 2^{16} times, and loop count value of 0x00FFFF will cause execution of the loop the maximum number of $2^{16}-1$ times.

Due to pipelining, a change in the SC bit takes affect only after three instruction cycles. Inserting three NOP instructions after the instruction that changes the value of this bit will ensure proper operation.

7 PROCESSING STATES

The DSP56300 Core is always in one of five processing states: normal, exception, reset, wait, or stop. These states are described in the following paragraphs.

7.1 NORMAL PROCESSING STATE

The normal processing state is associated with instruction execution. Instructions are executed using a seven-stage pipeline, which is described in the following paragraphs.

7.1.1 Instruction Pipeline

DSP56300 Core instruction execution is performed using a seven-stage pipeline, allowing most instructions to execute at a rate of one instruction every clock cycle. However, certain instructions require additional time to execute: all of the double-word instructions, instructions using an addressing mode that requires more than one cycle for the address calculation, and instructions causing a change of flow.

Instruction pipelining allows overlapping of instruction execution so that a pipeline stage of a given instruction occurs concurrently with other pipeline stages of other instructions. Only one word is fetched per cycle, so that in the case of double-word instructions, the second word of an instruction will be fetched before the next instruction is fetched. Table 7-1 describes the DSP56300 Core pipeline. The pipeline consists of seven stages: fetch 1, fetch 2, decode, address generation 1, address generation 2, execute1 and execute 2. $n1$ and $n2$ refer to first and second instructions respectively. The third instruction ($n3$), which contains an instruction extension word ($n3e$) takes two clock cycles to execute. The extension word will be either an absolute address or immediate data. Although it takes seven clock cycles for the pipeline to fill and the first instruction to execute, further instructions usually completes on each clock cycle.

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

Table 7-1. Instruction Pipeline

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

7.2 EXCEPTION PROCESSING STATE (INTERRUPT PROCESSING)

The exception processing state is associated with interrupts that can be generated by conditions inside the DSP or from external sources. There are many sources for interrupts to the DSP56300 Core; some of these sources can generate more than one interrupt. An interrupt vector scheme with 128 vectors of predefined priorities is used to provide fast interrupt service. The following list outlines how interrupts are processed by the DSP56300 Core:

1. A hardware interrupt is synchronized with the DSP56300 Core clock, and the interrupt pending flag for that particular hardware interrupt is set. An interrupt source can have only one interrupt pending at any given time.
2. All pending interrupts (external and internal) are arbitrated to select which interrupt will be processed. The arbiter automatically ignores any interrupts with an Interrupt Priority Level (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 inserts 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 of the two instructions is a Jump To Subroutine (JSR) instruction (e.g. a JSCLR), the state of the machine is not saved on the stack, and a fast interrupt is executed. A long interrupt is executed if one of the interrupt instructions fetched is a JSR instruction. The PC is immediately released, the SR and the PC are saved in the stack, and the jump instruction controls where the next instruction is fetched from.

Note: All the various types of Jump To Subroutine instructions may be used as the “JSR” needed to make the interrupt long, e.g. JScC, BSSET etc.

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

7.2.1 Interrupt Sources

Exceptions may originate from any of the 128 vector addresses listed in Table 7-2. Exceptions may originate from one of two groups: core and peripherals. Table 7-2 lists only the core-originating sources. The peripheral-originating sources are described in the specific chip configuration specification document. The corresponding interrupt starting address for each interrupt source is shown. These addresses are located in the 256 locations of program memory pointed to by the VBA (Vector Base Address) register in the program control unit. When an interrupt is serviced, the instruction at the interrupt starting address is fetched first. Because the program flow is directed to a different starting address for each interrupt, the interrupt structure of the DSP56300 Core is said to be vectored. A vectored interrupt structure has low overhead execution. If it is known a-priori that certain interrupts will not be used, those interrupt vector locations can be used for program or data storage.

Table 7-2. Interrupt Sources

Interrupt Starting Address	IPL	Interrupt Source
VBA:\$00	-	Reserved
VBA:\$02	3	Stack Error
VBA:\$04	3	Illegal Instruction
VBA:\$06	3	Debug Request Interrupt
VBA:\$08	3	Trap
VBA:\$0A	3	Non-Maskable Interrupt ($\overline{\text{NMI}}$)
VBA:\$0C	3	Reserved for Future Level-3 Interrupt Source
VBA:\$0E	3	Reserved for Future Level-3 Interrupt Source
VBA:\$10	0 - 2	$\overline{\text{IRQA}}$
VBA:\$12	0 - 2	$\overline{\text{IRQB}}$
VBA:\$14	0 - 2	$\overline{\text{IRQC}}$
VBA:\$16	0 - 2	$\overline{\text{IRQD}}$
VBA:\$18	0 - 2	DMA Channel 0
VBA:\$1A	0 - 2	DMA Channel 1
VBA:\$1C	0 - 2	DMA Channel 2
VBA:\$1E	0 - 2	DMA Channel 3
VBA:\$20	0 - 2	DMA Channel 4
VBA:\$22	0 - 2	DMA Channel 5
VBA:\$24	0 - 2	Peripheral interrupt request 1
VBA:\$26	0 - 2	Peripheral interrupt request 2
:	:	
VBA:\$FE	0 - 2	Peripheral interrupt request 110

The 128 interrupts are prioritized into four levels. Level 3, the highest priority level, is not maskable. Levels 0 – 2 are maskable. The interrupts within each level are prioritized according to a predefined priority.

7.2.1.1 Hardware Interrupt Source

There are two types of hardware interrupts to the DSP56300 Core: internal and external. The internal interrupts include the on-chip sources, namely: Stack Error, Illegal Instruction, Debug Request, Trap, the DMAs and the peripherals. Each internal interrupt source is serviced if it is not masked. When serviced, the interrupt request is cleared. Each maskable internal hardware source has independent enable control.

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

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

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

7.2.1.2 Software Interrupt Source

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

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

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

7.2.2 Interrupt Priority Structure

Four levels of interrupt priority are provided. IPLs numbered 0, 1, and 2 are maskable (level 0 is the lowest level). Level 3 (highest level) is nonmaskable. The IPL 3 interrupts are: Hardware Reset, III, Stack Error and TRAP. The interrupt mask bits (I1, I0) in the SR reflect the current processor priority level and indicate the IPL needed for an interrupt source to interrupt the processor (see Table 7-3). Interrupts are inhibited for all priority levels less than the current processor priority level. However, level 3 interrupts are not maskable and therefore can always interrupt the processor.

Table 7-3. Status Register Interrupt Mask Bits

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

7.2.2.1 Interrupt Priority Levels

There are two Interrupt priority registers in the DSP56300 Core: IPRC that is dedicated for DSP56300 Core interrupt sources and IPRP that is dedicated for the specific chip peripherals interrupt sources. These control registers are mapped on the internal X I/O memory space.

The IPL for each interrupting source is software programmable. Each on-chip or external peripheral device can be programmed to one of the three maskable priority levels (IPL 0, 1, or 2). IPLs are set by writing to the interrupt priority registers shown in Figure 7-1 and Figure 7-2. These two read/write registers specify the IPL for each of the interrupting devices. In addition, IPRC register specifies the trigger mode of each external interrupt source and is used to enable or disable the individual external interrupts. These registers are cleared on hardware RESET or by the RESET instruction. Table 7-4 defines the IPL bits. Table 7-5 defines the external interrupt trigger mode bits.

Figure 7-1. Interrupt Priority Register C (IPRC)

11	10	9	8	7	6	5	4	3	2	1	0
IDL2	IDL1	IDL0	ICL2	ICL1	ICL0	IBL2	IBL1	IBL0	IAL2	IAL1	IAL0
IxL2			IRQ A/B/C/D mode								
IxL1:0			IRQ A/B/C/D IPL								
23	22	21	20	19	18	17	16	15	14	13	12
D5L1	D5L0	D4L1	D4L0	D3L1	D3L0	D2L1	D2L0	D1L1	D1L0	D0L1	D0L0
DxL1:0			DMA 0/1/2/3/4/5 IPL								

Figure 7-2. Interrupt Priority Register P (IPRP)

11	10	9	8	7	6	5	4	3	2	1	0
Per6L1	Per6L0	Per5L1	Per5L0	Per4L1	Per4L0	Per3L1	Per3L0	Per2L1	Per2L0	Per1L1	Per1L0

23	22	21	20	19	18	17	16	15	14	13	12
PerCL1	PerCL0	PerBL1	PerBL0	PerAL1	PerAL0	Per9L1	Per9L0	Per8L1	Per8L0	Per7L1	Per7L0

Table 7-4. Interrupt Priority Level Bits

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

Table 7-5. External Interrupt Trigger Mode Bits

IxL2	Trigger Mode
0	Level
1	Negative Edge

7.2.2.2 Exception Priorities within an IPL

If more than one exception is pending when an instruction is executed, the interrupt with the highest priority level is serviced first. When multiple interrupt requests having the same IPL are pending, a second fixed-priority structure within that IPL determines which interrupt is serviced. The fixed priority of interrupts within an IPL and the interrupt enable bits for all interrupts are shown in Table 7-6.

7.2.3 Instructions Preceding the Interrupt Instruction Fetch

Every instruction which takes more than one cycle to execute is aborted when it is fetched in the cycle preceding the fetch of the first interrupt instruction word.

Aborted instructions are refetched again when program control returns from the interrupt routine. The PC is adjusted appropriately before the end of the decode cycle of the aborted instruction.

If the first interrupt word fetch occurs in the cycle following the fetch of a one-word-one-

cycle instruction, that instruction will complete normally before the start of the interrupt routine.

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

7.2.4 Interrupt Types

Two types of interrupt routines may be used: fast and long. The fast routine consists of the two automatically inserted interrupt instruction words. These words can contain any unrestricted, single two-word instruction or any two unrestricted one-word instructions. 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 sixteen-bit mode bit is reset.
5. The IPL is raised to disallow further interrupts of the same or lower levels (except Illegal Instruction, stack error and TRAP that can always interrupt).

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

7.2.5 Interrupt Arbitration

External interrupts are internally synchronized with the processor clock before their interrupt-pending flags are set. Each external interrupt and internal interrupt has its own flag. After each instruction is executed, all interrupts are arbitrated — i.e., all hardware interrupts that have been latched into their respective interrupt-pending flags and all internal interrupts. During arbitration, each interrupt's IPL is compared with the interrupt mask in the SR, and the interrupt is either allowed or disallowed. The remaining interrupts are prioritized according to the priority shown in Table 7-6, and the highest priority interrupt is chosen. The interrupt vector is then calculated so that the program interrupt controller can fetch the first interrupt instruction. The interrupt-pending flag for the chosen interrupt is not cleared until the second interrupt vector of the chosen interrupt is being fetched. A new interrupt from the same source will not be accepted for the next interrupt arbitration until that time.

Table 7-6. Exception Priorities within an IPL

Priority	Exception
Level 3 (Nonmaskable)	
Highest	Stack Error
	Illegal Instruction
	Debug Request Interrupt
	Trap
	Non-Maskable Interrupt ($\overline{\text{NMI}}$)
Lowest	Non-Maskable Peripheral Interrupt
Levels 0, 1, 2 (Maskable)	
Highest	$\overline{\text{IRQA}}$ (External Interrupt)
	$\overline{\text{IRQB}}$ (External Interrupt)
	$\overline{\text{IRQC}}$ (External Interrupt)
	$\overline{\text{IRQD}}$ (External Interrupt)
	DMA Channel 0 Interrupt
	DMA Channel 1 Interrupt
	DMA Channel 2 Interrupt
	DMA Channel 3 Interrupt
	DMA Channel 4 Interrupt
	DMA Channel 5 Interrupt
Lowest	Peripheral interrupt sources

7.2.6 Interrupt Instruction Fetch

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

7.2.7 Interrupt Instruction Execution

Interrupt instruction execution is considered “fast” if neither of the instructions of the interrupt service routine cause a change of flow. A JSR within a fast interrupt routine forms a long interrupt, which is terminated with an RTI instruction to restore the PC and SR from the stack and return to normal program execution. Reset is a special exception, which will normally contain only a JMP instruction at the exception start address. At the programmer's option, almost any instruction can be used in the fast interrupt routine. A fast interrupt routine may contain either two single-word instructions or one double-word instruction. Table 7-7 shows the effect of a fast interrupt routine on the instruction pipeline. The fast interrupt executes only two instructions (ii1 and ii2) and then automatically resumes execution of the main program. Table 7-8 shows the effect of a long interrupt routine on the instruction pipeline. A short JSR (ii1) is used to call the long interrupt routine which includes the 4 instructions sr1, sr2, sr3 and an rti. Instructions ii2, n3, sr5 and sr6 are not decoded nor executed.

Table 7-7. Fast Interrupt Pipeline

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

Table 7-8. Long Interrupt Pipeline

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

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

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

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

1. A JSR to the starting address of the interrupt service routine is located at one of the two interrupt vector addresses.
2. During execution of the JSR instruction, the PC and SR are stacked. The interrupt mask bits of the SR are updated to mask interrupts of the same or lower priority. The loop flag and scaling mode bits are cleared.
3. The interrupt service routine can be interrupted — i.e., nested interrupts are supported.
4. The long interrupt routine, which can be any length, should be terminated by an RTI, which restores the PC and SR from the stack.

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

A REP instruction is treated as a single two-word instruction, regardless of how many times it repeats the second instruction of the pair. Instruction fetches are suspended and will be reactivated only after the LC is decremented to one. During the execution of the repeated instruction, no interrupts will be serviced. When LC finally decrements to one, the fetches are reinitiated, and pending interrupts can be serviced.

7.3 RESET PROCESSING STATE

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

1. Internal peripheral devices are reset.
2. The modifier registers (M0-M7) are set to \$FFFFFF.
3. The interrupt priority registers are cleared.
4. The Bus Control Register (BCR), the Address Attribute Registers (AAR3-AAR0) and the Dram Control Register (DCR) are set to their initial values as described in Chapter 2. The initial value causes a maximum number of wait states to be added to every external memory access.
5. The Stack Pointer (SP) and the Stack Counter (SC) are cleared.
6. The scaling mode, loop flag, sixteen-bit mode, double precision mode and condition code bits of the SR are cleared, and the interrupt mask bits of the SR are set.
7. The Instruction Cache Controller is initialized as described in Chapter 5.
8. The cache-enable (CE) bit in SR and the burst-mode bit in OMR are cleared.
9. The PLL Control register is initialized as described in Chapter 9.
10. The Vector Base Address (VBA) register is cleared.

The DSP56300 Core remains in the reset state until RESET is deasserted. Upon leaving the reset state, the chip operating mode bits of the OMR are loaded from the external mode select pins (MODA, MODB, MODC,MODD), and program execution begins at the program memory address as described in Chapter 12.

7.4 WAIT PROCESSING STATE

The wait processing state is a low power-consumption state entered by execution of the WAIT instruction. In the wait state, the internal clock is disabled from all internal circuitry except the internal peripherals. All internal processing is halted until an unmasked interrupt occurs, the DSP is reset, or \overline{DE} is asserted. If exit from wait state was caused by asserting \overline{DE} , the processor will enter the debug mode.

7.5 STOP PROCESSING STATE

The stop processing state is the lowest power consumption mode and is entered by the execution of the STOP instruction. In the stop mode, the clock oscillator activity depends on the PSTP bit in the PLL control register. If this bit is cleared, the clock oscillator is turned off, while if the bit is set, the VCO remains active and the global clock to the entire chip is gated off.

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

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

Either of these actions will gate on the oscillator and, after a clock stabilization delay, clocks to the processor and peripherals will be re-enabled.

When the clocks to the processor and peripherals are re-enabled then the processor will enter the reset processing state if the exit from stop state was caused by a low level on the $\overline{\text{RESET}}$ pin.

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 (i. e. $\overline{\text{IRQA}}$ was negated before interrupts were arbitrated) or if no interrupt is enabled then the processor resumes execution at the instruction following the STOP instruction that caused the entry into the stop state.

If the exit from stop state was caused by a low level on the $\overline{\text{DE}}$ pin, the processor will enter the debug mode.

For minimum power consumption during the STOP state at the cost of longer recovery time, PSTP bit of the PLL Control register should be cleared. To enable rapid recovery when exiting the STOP state, at the cost of higher power consumption, PSTP should be set. PSTP is cleared by hardware reset.

8 DMA CONTROLLER

The Direct Memory Access (DMA) Controller is an on-chip device that permits data transfers between internal/external memory and/or internal/external I/O in any combination, without intervention of the program. Due to dedicated DMA address and data buses as well as internal memories partition, a high level of isolation is achieved where the DMA operation does not interfere or slow down the core operation.

The DMA Controller has six channels, each one having its own register set. All the registers are memory-mapped in the internal I/O memory space.

Table 8-1 shows the various types of data transfers that the DMA Controller can perform.

Table 8-1. DMA Controller Data Transfers

			Clock cycles per single word transfer
Internal Memory	→	Internal Memory	2
External Memory	↔	Internal Memory	2+wait states
External Memory	→	External Memory	2+wait states
Internal Memory	↔	Internal I/O	2
External Memory	↔	Internal I/O	2+wait states
Internal I/O	→	Internal I/O	2

Data transfer for one channel takes minimum two clock cycles per single word.

The DMA can execute data transfers with various types of address generation schemes such as:

1. constant addressing, where the address is unchanged throughout the data transfer.
2. uni-dimensional addressing, where one block is transferred using consecutive addresses.
3. two-dimensional addressing, where equally spaced blocks are transferred, using consecutive addresses within each block. The spacing between the blocks is programmed into an offset register.
4. three-dimensional addressing, where equally spaced groups of equally spaced blocks are transferred, using consecutive addresses within each block. The two spacings are programmed into two offset registers.

-
5. special cases of the above mentioned modes allow many other address generation patterns such as linear buffers with non-unit stride, circular buffers, etc.

8.1 DMA CONTROLLER PROGRAMMING MODEL

The registers comprising the DMA Controller are shown in Table 8-2 through Table 8-8.

Table 8-2. DMA Controller Programming Model - Channel 0

DSR0 - DMA Source Address Register for channel 0
DDR0 - DMA Destination Address Register for channel 0
DCO0 - DMA Counter for channel 0
DCR0 - DMA Control Register for channel 0

Table 8-3. DMA Controller Programming Model - Channel 1

DSR1 - DMA Source Address Register for channel 1
DDR1 - DMA Destination Address Register for channel 1
DCO1 - DMA Counter for channel 1
DCR1 - DMA Control Register for channel 1

Table 8-4. DMA Controller Programming Model - Channel 2

DSR2 - DMA Source Address Register for channel 2
DDR2 - DMA Destination Address Register for channel 2
DCO2 - DMA Counter for channel 2
DCR2 - DMA Control Register for channel 2

Table 8-5. DMA Controller Programming Model - Channel 3

DSR3 - DMA Source Address Register for channel 3
DDR3 - DMA Destination Address Register for channel 3
DCO3 - DMA Counter for channel 3
DCR3 - DMA Control Register for channel 3

Table 8-6. DMA Controller Programming Model - Channel 4

DSR4 - DMA Source Address Register for channel 4
DDR4 - DMA Destination Address Register for channel 4
DCO4 - DMA Counter for channel 4
DCR4 - DMA Control Register for channel 4

Table 8-7. DMA Controller Programming Model - Channel 5

DSR5 - DMA Source Address Register for channel 5
DDR5 - DMA Destination Address Register for channel 5
DCO5 - DMA Counter for channel 5
DCR5 - DMA Control Register for channel 5

Table 8-8. DMA Offset Registers

DOR0 - DMA Offset Register 0
DOR1 - DMA Offset Register 1
DOR2 - DMA Offset Register 2
DOR3 - DMA Offset Register 3

Table 8-9. DMA Status Register

DSTR - DMA Status Register

8.1.1 DMA Source Address Register (DSR)

The DMA Source Address Register (DSR) is a 24-bit read/write register that contains the source address for the next DMA transfer. The DMA controller has one source address register for each DMA channel - DSR0, DSR1, DSR2, DSR3, DSR4 and DSR5.

8.1.2 DMA Destination Address Register (DDR)

The DMA Destination Address Register (DDR) is a 24-bit read/write register that contains the destination address for the next DMA transfer. The DMA controller has one destination address register for each DMA channel - DDR0, DDR1, DDR2, DDR3, DDR4 and DDR5.

8.1.3 DMA Offset Register (DOR)

The DMA Offset Register is a 24-bit read/write register that contains the offset value to be used in some of the DMA addressing modes. The DMA controller has four common offset registers (DOR0, DOR1, DOR2 and DOR3) that can be used by all the channels according to their address generation mode.

8.1.4 DMA Counter (DCO)

The DMA Counter is a 24-bit read/write register that contains the number of DMA data transfers to be done. The DCO has five modes of operations determined by the DMA channel's address generation mode that are defined in the DMA channel's Control Register.

The following paragraphs explain the various modes of the DMA Counter. During DMA operation, a Source Address Register (DSR) is associated with one of the counter modes, while the Destination Address Register (DDR) can be associated with another counter mode. The examples below use DSR as an example of the address register used, but the same example is valid for the destination register also.

8.1.4.1 DMA counter mode A - single counter



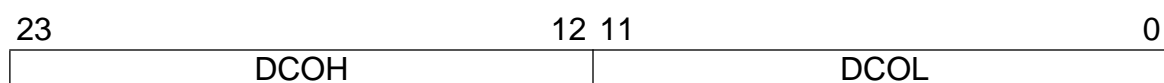
In this mode of operation, the number of transfers is equal to the value loaded into DCO plus one (DCO+1). Before each DMA transfer, the DCO is tested for zero, and the following actions occur based on the test result:

- DCO > 0: A transfer is initiated with an address equal to the address register, then DCO is decremented by one and the address register is updated according to the address generation mode.
- DCO = 0: The last transfer is initiated with an address equal to the address register, the address register is updated according to the address generation mode and DCO is loaded with its preloaded value.

If, for example, DCO is preloaded with the value 5, DSR is loaded with the value S and the address generation mode is postincrement by 1, the following DMA transfers will be initiated by the DMA controller:

before the transfer			after the transfer	
DSR	DCO	transfer source address	DSR	DCO
S	5	S	S+1	4
S+1	4	S+1	S+2	3
S+2	3	S+2	S+3	2
S+3	2	S+3	S+4	1
S+4	1	S+4	S+5	0
S+5	0	S+5	S+6	5

8.1.4.2 DMA counter mode B - dual counter



In this mode of operation, which is useful for two dimensional block transfers, the DCO is separated to two sections: DCOL (bits 0-11) and DCOH (bits 12-23). Before each DMA transfer the DCO is tested for zero and the following actions occur based on the test result:

- DCOL > 0: A transfer is initiated with an address equal to the address register, then DCOL is decremented by one and the address register is incremented by one.
- DCOH > 0; DCOL = 0: A transfer is initiated with an address equal to the address register, the address register is incremented with the specified offset register, DCOH is decremented by one and DCOL is loaded with its preloaded value.
- DCOH = 0; DCOL = 0: The last transfer is initiated with an address equal to the address register, the address register is incremented with the specified offset register and both DCOH and DCOL are loaded with their preloaded value.

The number of transfers in this mode is equal to (DCOL+1) x (DCOH+1).

If, for example, DCOH is preloaded with the value 1, DCOL is preloaded with the value 2, DOR is preloaded with the value O and DSR with the value S, the following DMA transfers will be initiated by the DMA controller:

before the transfer			after the transfer	
DSR	DCO	transfer source address	DSR	DCO
S	1:2	S	S+1	1:1
S+1	1:1	S+1	S+2	1:0
S+2	1:0	S+2	S+O+2	0:2
S+O+2	0:2	S+O+2	S+O+3	0:1
S+O+3	0:1	S+O+3	S+O+4	0:0
S+O+4	0:0	S+O+4	S+2O+4	1:2

8.1.4.3 DMA counter modes C, D and E- triple counter

In this mode of operation, which is useful for three dimensional block transfers, the DCO is separated to three sections: DCOL, DCOM and DCOH. Before each DMA transfer the

DCO is tested for zero and the following actions occur based on the test result:

- DCOL > 0: A transfer is initiated with an address equal to the address register, then DCOL is decremented by one and the address register is incremented by one.
- DCOM > 0; DCOL = 0: A transfer is initiated with an address equal to the address register, the address register is incremented with the first specified offset register, DCOM is decremented by one and DCOL is loaded with its preloaded value.
- DCOH > 0; DCOM = 0; DCOL = 0: A transfer is initiated with an address equal to the address register, the address register is then incremented with the second specified offset register, DCOH is decremented by one and both DCOM and DCOL are loaded with their preloaded value.
- DCOH = 0; DCOM = 0; DCOL = 0: The last transfer is initiated with an address equal to the address register, the address register is then incremented with the second specified offset register and both DCOH, DCOM and DCOL are loaded with their preloaded value.

The number of transfers in this mode is equal to $(DCOL+1) \times (DCOM+1) \times (DCOH+1)$.

If, for example, DCOH is preloaded with the value 1, DCOM is also preloaded with the value 1, DCOL is preloaded with the value 2, DOR0 is preloaded with the value 00, DOR1 is preloaded with the value 01 and DSR with the value S, the following DMA transfers will be initiated by the DMA controller:

before the transfer			after the transfer	
DSR	DCO	transfer source address	DSR	DCO
S	1:1:2	S	S+1	1:1:1
S+1	1:1:1	S+1	S+2	1:1:0
S+2	1:1:0	S+2	S+00+2	1:0:2
S+00+2	1:0:2	S+00+2	S+00+3	1:0:1
S+00+3	1:0:1	S+00+3	S+00+4	1:0:0
S+00+4	1:0:0	S+00+4	S+00+01+4	0:1:2
S+00+01+4	0:1:2	S+00+01+4	S+00+01+5	0:1:1
S+00+01+5	0:1:1	S+00+01+5	S+00+01+6	0:1:0
S+00+01+6	0:1:0	S+00+01+6	S+200+01+6	0:0:2
S+200+01+6	0:0:2	S+200+01+6	S+200+01+7	0:0:1
S+200+01+7	0:0:1	S+200+01+7	S+200+01+8	0:0:0
S+200+01+8	0:0:0	S+200+01+8	S+200+201+8	1:1:2

1. DMA counter mode C structure:

The structure of DMA counter mode C is as follows: DCOL (bits 0-5), DCOM (bits 6-11) and DCOH (bits 12-23).

23	12 11	6 5	0
DCOH	DCOM	DCOL	

2. DMA counter mode D structure:

The structure of DMA counter mode D is as follows: DCOL (bits 0-5), DCOM (bits 6-17) and DCOH (bits 18-23).

23	18 17	6 5	0
DCOH	DCOM	DCOL	

3. DMA counter mode E structure:

The structure of DMA counter mode E is as follows: DCOL (bits 0-11), DCOM (bits 12-17) and DCOH (bits 18-23).

23	18 17	12 11	0
DCOH	DCOM	DCOL	

8.1.5 DMA Control Register (DCR)

The DMA Control Register (DCR) is a 24-bit read/write register that controls the DMA operation. Each bit is shown in Figure 8-1 and described in the following paragraphs. All DCR bits are cleared during processor reset.

Figure 8-1. DMA Control Register

23	22	21	20	19	18	17	16	15	14	13	12
DE	DIE	DTM2	DTM1	DTM0	DPR1	DPR0	DCON	DRS4	DRS3	DRS2	DRS1
11	10	9	8	7	6	5	4	3	2	1	0
DRS0	D3D	DAM5	DAM4	DAM3	DAM2	DAM1	DAM0	DDS1	DDS0	DSS1	DSS0

8.1.5.1 DCR DMA Channel Enable Bit (DE) Bit 23

The DE bit enables the channel operation. Setting DE will trigger a single block DMA transfer in the DMA transfer mode that uses DE as a trigger, and will enable a single block, a single “line” or a single word DMA transfer in the transfer modes which use a requesting device as a trigger. DE is cleared by Hardware Reset, and by the end of DMA transfer in some of the transfer modes as defined by the DTM(2:0) bits. Clearing DE explicitly by software during a DMA operation will stop the channel operation only after the current DMA transfer has been completed (the current word has been stored into the destination).

DE	DMA Operation
0	Disabled
1	Enabled

8.1.5.2 DCR DMA Interrupt Enable Bit (DIE) Bit 22

When the DMA Interrupt Enable bit is set, a DMA interrupt will be generated at the end of a DMA block transfer, that is after the counter is loaded with its preloaded value and the DTD bit in the DMA status register is set. A DMA interrupt will also be generated when DE is cleared explicitly by software during a DMA operation, as described in Section 8.1.5.1.

When DIE is cleared, the DMA interrupt is disabled.

DIE	DMA Interrupt
0	Disabled
1	Enabled

8.1.5.3 DCR DMA Transfer Mode (DTM2-DTM0)- bits 21:19

DMA Transfer mode bits specify the modes of operation of the DMA channel.

When DTM2-DTM0=000, a block of data is transferred, the length of the block is determined by the counter, the transfer is enabled by DE and initiated by the first DMA request. The transfer is completed after the counter decrements to zero, then it reloads itself with the original value and clears the DE bit.

Table 8-10. DMA Transfer Mode (DTM2-DTM0) Bits

DTM(2:0)	triggered by	DE cleared at end of block	Transfer Mode
000	request	yes	block transfer
001	request	yes	word transfer
010	request	yes	line transfer
011	DE	yes	block transfer
100	request	no	block transfer
101	request	no	word transfer
110	reserved		
111	reserved		

When DTM2-DTM0=001, a block of data is transferred, the length of the block is determined by the counter and each DMA request will transfer a single word while enabled by DE. The transfer is completed after the counter decrements to zero, then it reloads itself with the original value and clears the DE bit.

When DTM2-DTM0=010, a block of data is transferred, the length of the block is determined by the counter and each DMA request will transfer a "line", i.e. the number of words as defined at DCOL, while enabled by DE. The transfer is completed after the whole counter decrements to zero, then it reloads itself with the original value and clears the DE bit.

When DTM2-DTM0=011, a block of data is transferred, the length of the block is determined by the counter and the transfer is initiated by setting DE. The transfer is completed when the counter decrements to zero, then it reloads itself with the original value and clears the DE bit.

When DTM2-DTM0=100, a block of data is transferred, the length of the block is determined by the counter, the transfer is enabled by DE and initiated by the first DMA request. The transfer is completed when the counter decrements to zero, then it reloads itself with the original value. The DE bit is not cleared at the end of the block, therefore the DMA channel is waiting for a new request.

When DTM2-DTM0=101, a single word transfer is enabled by DE and initiated by every DMA request. When the counter decrements to zero, it is reloaded with its original value. The DE bit is not automatically cleared, therefore the DMA channel is waiting for a new request.

8.1.5.4 DCR DMA Channel priority(DPR1-DPR0) - bits 18:17

The DMA Channel Priority control bits define the priority level of the DMA channel relative to the other DMA channels as well as to the priority level of the core when external bus access is required. When DMA transfers are pending, the DMA channel priority level of all the channels are compared to decide which channel will be activated in the next word transfer. This decision must be made since all channels use common resources such as the DMA address generation logic, the address and data buses etc.

Table 8-11. DCR DMA Channel priority(DPR1-DPR0) Bits

DPR(1:0)	Channel Priority
00	Priority Level 0 (lowest)
01	Priority Level 1
10	Priority Level 2
11	Priority Level 3 (highest)

If all or some of the channels have the same priority, then the channels will be activated in a round-robin fashion: channel 0 will be activated to transfer one word out of its programmed stream, followed by channel 1, followed by channel 2 and so on.

If the channel priorities are different, the channel with the highest priority will start executing DMA transfers and will remain doing so as long as there are DMA transfers pending. In the event that a lower priority channel is executing DMA transfers when a higher priority channel receives a transfer request, the lower priority channel will finish the transfer of the current word and arbitration will start again. If some channels with the same priority are activated in a round-robin fashion and a new channel with higher priority interferes, then after this channel finishes its pending transfers the order of the transfers in the round-robin mode may change, but the algorithm remains the same.

The DPR(1:0) bits are also used to determine the DMA priority relative to the core priority when an external bus access is required. This function involves the DMA priority level defined by the current active DMA channel, the core priority defined by bits CP1-CP0 in the DSP56300 Core Status Register (SR) and the core-DMA priority defined by bits CDP1-CDP0 in the DSP56300 Core Operating Mode Register (OMR).

When the priority of the DMA is higher than the priority of the core (CDP=01; CDP=00 and

DPR > CP) and both the DMA and the core require an external access, the DMA will perform the external bus access and the core will wait for the DMA to complete the programmed current transfer.

When the priority of the DMA is equal to the priority of the core (CDP=10; CDP=00 and DPR = CP) and both the DMA and the core require an external access, the core will perform all its external accesses pertaining to the current instruction and then the DMA will perform its access.

When the priority of the DMA is lower than the priority of the core (CDP=11; CDP=00 and DPR < CP) and both the DMA and the core require an external access, the core will always perform its external accesses and the DMA will wait for a free slot in which the core does not require the external bus.

In the dynamic priority mode (CDP=00), it is possible that a DMA channel will be halted before executing both the source and destination accesses when the core has higher priority over the external bus. In this case, if another, higher priority DMA channel will request an access, the halted channel will finish its previous access with the new higher priority before the new requesting DMA channel will be serviced.

8.1.5.5 DCR DMA Continuous Mode (DCON) - bit 16

When DMA Continuous Mode bit is set the channel will enter the continuous transfer mode, in which it will not be interrupted throughout the transfer by any other DMA channel of equal priority. The DMA transfers in Continuous Mode of operation can be interrupted if a DMA channel of higher priority has been enabled after the Continuous Mode transfer was started. If the priority of the DMA is higher than the priority of the core (CDP=01; CDP=00 and DPR>CP) and DCON bit is set, then if the DMA requires an external access, it will get the external bus and the core will not be able to use the external bus in the next cycle after the DMA access even if the DMA does not need the bus in this cycle. However, if a refresh cycle from the DRAM controller is requested in such a case, the DMA will be interrupted by the refresh cycle.

When the DCON bit is cleared the priority algorithm operates as described in the Section 8.1.5.4.

8.1.5.6 DCR DMA Request Source (DRS0-DRS4) Bits 15-11

The DMA Request Source bits encode the source of DMA requests used to trigger the DMA transfers. The DMA request sources may be the internal peripherals, external devices requesting service through the \overline{IRQA} , \overline{IRQB} , \overline{IRQC} and \overline{IRQD} pins or triggering by transfer done from a DMA channel. All the request sources behave as edge-triggered synchronous inputs.

Peripheral requests 18-21 (DRS[4:0]=111xx) are special because in addition to the regular behavior of all the requesting devices, they can serve as “fast request sources”. In a regular request from a peripheral, the trigger to the DMA remains set until the appropriate register at the peripheral is accessed by the DMA, therefore the peripheral

cannot generate a second request until the first one was served. Another method is when the peripheral (i.e. timer) gives a triggering pulse without taking care whether the DMA served this trigger. The “fast peripheral” has a full duplex handshake to the DMA, enabling a maximum throughput of a trigger every two clock cycles. This mode is functional only in the “word transfer mode” (DTM = 001 or 101). In the “fast request mode” the DMA sets an “enable line” to the peripheral. If the peripheral wants, he sends the DMA a one cycle triggering pulse. This pulse resets the enable line. If the DMA decides by the priority algorithm that this trigger will be served in the next cycle, the enable line is set again even before the corresponding register in the peripheral is accessed.

DMA Request Source Bits DRS(4:0)	Requesting Device
00000	External ($\overline{\text{IRQA}}$ pin)
00001	External ($\overline{\text{IRQB}}$ pin)
00010	External ($\overline{\text{IRQC}}$ pin)
00011	External ($\overline{\text{IRQD}}$ pin)
00100	Transfer Done from channel 0
00101	Transfer Done from channel 1
00110	Transfer Done from channel 2
00111	Transfer Done from channel 3
01000	Transfer Done from channel 4
01001	Transfer Done from channel 5
01010	Peripheral Request MDRQ0
...	...
11111	Peripheral Request MDRQ21

8.1.5.7 DCR DMA three Dimensional mode (D3D)- bit 10

When this bit is set the addressing mode, determined by DAM(5:0) is three-dimensional. When this bit is cleared the addressing mode is non three-dimensional.

8.1.5.8 DCR DMA Address Mode (DAM5-DAM0)- bits 9:4

These bits define the address generation mode for the DMA transfer. These bits are encoded in two different ways according to D3D bit.

Non three dimensional modes (D3D = 0).

In this case DAM bits are separated into two groups: DAM(5:3), that defines the address generation mode for destination transfers and DAM(2:0), that defines the address generation mode for source transfers. The encoding is defined in Table 8-12 and in Table 8-13. The address generation mode can be no update, postincrement by 1 or two-dimensional.

In the no update addressing mode the DMA is accessing a constant address for the entire transfer. This addressing mode is useful when accessing peripheral devices as well as other single address devices such as FIFOs.

In the postincrement by one addressing mode the DMA is accessing consecutive addresses. This addressing mode is useful when accessing data structures in memories, when the data elements are placed in successive memory locations.

In the two-dimensional addressing mode of operation the DMA is accessing data at consecutive addresses for a given number of times (DCOL) and then an offset register is added to the generated address. The entire process is repeated for another given number of times (DCOH). DCOL and DCOH are the two sections of the DCO counter. See Section 8.1.4 for a detailed description of the DCO operation. This addressing mode is useful when accessing two dimensional arrays of data.

Table 8-12. Source Address Generation Mode (D3D = 0)

DAM(2:0)	addressing mode	counter mode	offset select
000	two-dimensional	B	DOR0
001	two-dimensional	B	DOR1
010	two-dimensional	B	DOR2
011	two-dimensional	B	DOR3
100	no update	A	N/A
101	post increment by 1	A	N/A
110	reserved		
111	reserved		

Note: if the source address generation mode specify different counter mode than the destination address generation mode, then the counter mode is B.

Table 8-13. Destination Address Generation Mode (D3D = 0)

DAM(5:3)	addressing mode	counter mode	offset select
000	two-dimensional	B	DOR0
001	two-dimensional	B	DOR1
010	two-dimensional	B	DOR2
011	two-dimensional	B	DOR3
100	no update	A	N/A
101	post increment by 1	A	N/A
110	reserved		
111	reserved		

Note: if the destination address generation mode specify different counter mode than the source address generation mode, then the counter mode is B.

Three dimensional modes (D3D = 1).

In this case DAM bits are separated into three groups: DAM(1:0) that defines the DMA counter mode, DAM2 that is the address mode select and DAM(5:3) that defines the address generation mode. The encoding is defined in Table 8-14, Table 8-15 and Table 8-16. When D3D equals one, either the source addressing mode or the destination addressing mode or both are three-dimensional.

In the three-dimensional address generation mode of operation the DMA is accessing data at consecutive addresses for a given number of times (DCOL) and then an offset register is added to the generated address. This process is repeated for another given number of times (DCOM) after which another offset is added to the generated address. The entire process is repeated for a given number of times (DCOH). DCOL, DCOM and DCOH are the three sections of the DCO counter. See Section 8.1.4 for a detailed description of the DCO operation. This addressing mode is useful when accessing a number of two dimensional arrays of data.

Table 8-14. Counter Mode (D3D = 1)

DAM(1:0)	counter mode
00	mode C
01	mode D
10	mode E
11	reserved

Table 8-15. Address Mode Select (D3D = 1)

DAM2	addressing mode	offset select
0	source: three-dimensional	source: DOR0 : DOR1
	destination: defined by DAM(5:3)	destination: defined by DAM(5:3)
1	source: defined by DAM(5:3)	source: defined by DAM(5:3)
	destination: three-dimensional	destination: DOR2 : DOR3

Table 8-16. Address Generation Mode (D3D = 1)

DAM(5:3)	addressing mode	offset select
000	two-dimensional	DOR0
001	two-dimensional	DOR1
010	two-dimensional	DOR2
011	two-dimensional	DOR3
100	no update	none
101	post increment by 1	none
110	three-dimensional	DOR0 : DOR1
111	three-dimensional	DOR2 : DOR3

The offset select in Tables 8-12, 8-13, 8-15 and 8-16 defines the offset registers that are selected to increment the address register, when DCOL or DCOM:DCOL equals zero. In two-dimensional mode only one offset register is needed to increment the address register when DCOL equals zero. In three dimensional mode, two offset registers are needed, DORi:DORj. When DCOL equals zero and DCOM does not equal zero, then DORi is used to increment the address register. If both DCOL and DCOM equal zero, then DORj is used to increment the address register.

8.1.5.9 DCR DMA Destination Space (DDS0-DDS1) Bits 3, 2

The DMA Destination Space control bits specify the memory space that will be referenced as destination by the DMA.

DDS1	DDS0	DMA Destination Memory Space
0	0	X Memory Space
0	1	Y Memory Space
1	0	P Memory Space
1	1	Reserved

Note: In Cache Mode, a DMA to P Memory Space has some limitations (as described in Chapter 5).

8.1.5.10 DCR DMA Source Space (DSS0-DSS1) Bits 1, 0

The DMA Source Space control bits specify the memory that will be referenced as source by the DMA.

DSS1	DSS0	DMA Source Memory Space
0	0	X Memory Space
0	1	Y Memory Space
1	0	P Memory Space
1	1	Reserved

Note: In Cache Mode, a DMA from P Memory Space has some limitations (as described in Chapter 5).

8.1.6 DMA Status Register (DSTR)

The DMA Status Register (DSTR) is a 24-bit read only register that reflects the status of the DMA operation. Each bit is shown in Figure 8-2 and described in the following paragraphs.

Figure 8-2. DMA Status Register

23	22	21	20	19	18	17	16	15	14	13	12
11	10	9	8	7	6	5	4	3	2	1	0
DCH2	DCH1	DCH0	DACT			DTD5	DTD4	DTD3	DTD2	DTD1	DTD0

 Reserved, read as zero.

8.1.6.1 DSTR DMA Channel Transfer Done Status (DTD)- bits 5-0

The DMA Transfer Done status bits (DTD5-DTD0) are set when the last word during a single block transfer is stored in the destination, stopping channel operation. At the same time, the DE bit in the related DCR register, may be cleared according to the transfer mode as defined by DTM(2:0). The last transfer is defined as the one where the DMA counter reloads to its initial value, or when DE is explicitly cleared by software. If the DIE bit in the related DCR is set, then the assertion of the DTD bit will cause a DMA interrupt request. When the DMA Interrupt is disabled, the core may verify the channel status by polling this bit. DTD bits are set by Hardware Reset. The DTD bit is reset by explicitly setting bit DE at the corresponding DCR register by software.

Note1: Due to pipeline dependencies, after setting DE in a DCR register, the corresponding DTD bit will be affected only after additional two instruction cycles.

Note2: If the DMA channel works in a word transfer mode, than clearing DE will set the corresponding DTD bit only after a trigger that was already captured by the DMA is handled.

8.1.6.2 DSTR reserved bits - bits 23:12 and 7:6

These bits are reserved and are read as zero.

8.1.6.3 DSTR DMA Active state (DACT) - bit 8

The DMA Active state status bit is set if the DMA is in the middle of a transfer. This bit is cleared if all the DMA channels are disabled or wait for DMA requests. This bit should be polled and tested for zero before entering to a low power mode by executing a STOP instruction. The DACT status bit is cleared by Hardware Reset

8.1.6.4 DSTR DMA Active Channel (DCH2-DCH0) - bits 11:9

The DMA Active Channel status bits are the encoding of the current active channel. These bits are cleared at Hardware Reset. Their value is valid only if DACT=1.

Table 8-17. DCH Status bits encoding

DCH(2:0)	Active channel
000	DMA Channel 0
001	DMA Channel 1
010	DMA Channel 2
011	DMA Channel 3
100	DMA Channel 4
101	DMA Channel 5
110	reserved
111	reserved

8.2 DMA Restrictions

The following are some restrictions that apply to the DMA operation:

1. The user should take care when he needs to enter into the STOP processing state. Before executing the STOP instruction, the Dma ACTIVE (DACT) status bit should be polled until it is read as '0'. When the chip enters the STOP state all the DMA triggers that were previously latched are cleared.
2. The core will exit the WAIT processing state when a DMA channel accepts a trigger that is programmed as the selected source trigger. The DMA will prevent the core from entering WAIT processing state if the DMA is active.
3. Only the Transmit/Receive Data registers of the peripheral interfaces may be accessed by the DMA Controller when specifying source or destination in the internal I/O space.
4. If one of the DMA channels is accessing external memory and the access is delayed due to bus arbitration or memory wait, the other DMA channels will also stop, since the DMA mechanism does not distinguish between the different channels.
5. The internal RAM is divided into 256-word banks. If the Core and DMA are accessing different banks they will not interfere one with another, i.e. each will continue operations at its maximum speed. If both Core and DMA are accessing the same bank then the Core will always have priority and the DMA will be delayed until a free slot will be available.
6. The DMA Address Registers (DSR, DDR, DOR) and the DMA Counter

-
- (DCO) should be written only when the channel that uses them is disabled (DE=0 and DTD=1). The operation of the DMA controller cannot be guaranteed if one of these registers is written while the DMA channel that uses them is busy.
7. A change in the request source should be initiated only when the corresponding DMA channel is idle. If the channel is forced to enter the idle state by clearing the Dma Enable (DE) control bit, the corresponding Dma Transfer Done (DTD) status bit should be polled until it is read as '1'.
 8. If a DMA channel is programmed to perform accesses in the word transfer mode, the corresponding DTD status bit will be set only after the current captured request will be serviced by an appropriate transfer. This will assure that the last captured request will not be lost. Note that if this channel's priority is low, the DTD will be set only when it receives the priority to perform its accesses. In order to shorten this time, the channel's priority may be raised before DE is cleared.
 9. While a DMA channel is enabled (DE=1) the user should not modify any of the channel's DCR bits, but for the DE bit itself.
 10. Due to the DSP56300 Core pipeline, after DE bit in DCRx is set, the corresponding DTDx bit in DSTR will be cleared only after two instruction cycles.

9 PLL AND CLOCK GENERATOR

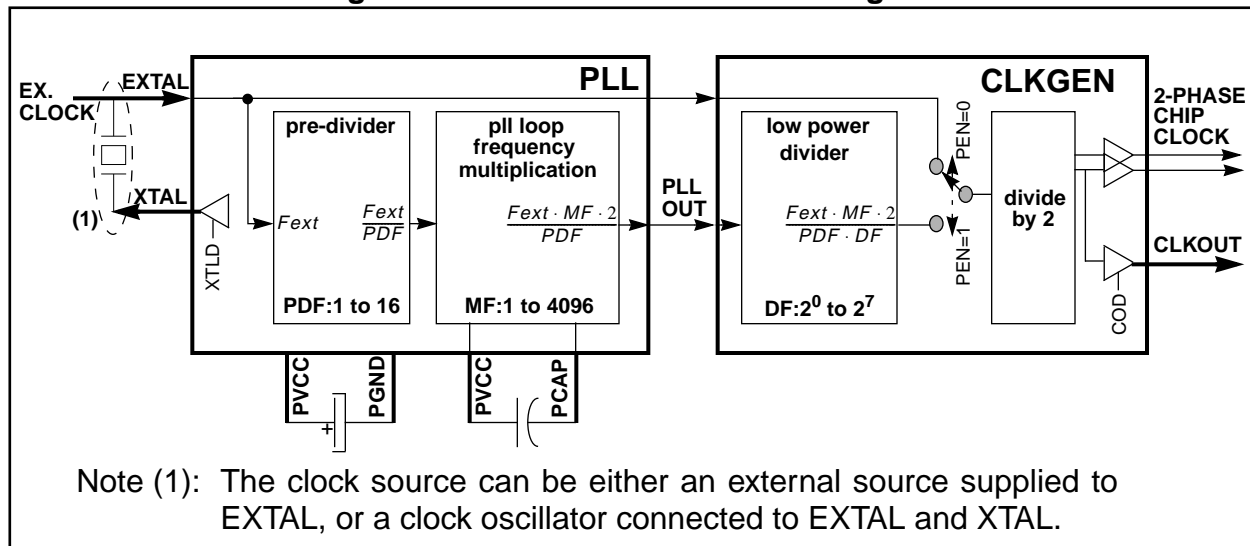
9.1 INTRODUCTION

The DSP56300 Core features a PLL (phase-locked loop) clock oscillator in its central processing module. The PLL allows the processor to operate at a high internal clock frequency using a low frequency clock input, a feature which offers two immediate benefits: Lower frequency clock input reduces 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 clock generation in the DSP56300 Core is composed of two main blocks:

- Phase Locked Loop (PLL) that performs
 - Clock input division
 - Frequency multiplication
 - Skew elimination
- CLOCK GENERATOR (CLKGEN) that performs
 - Low power division
 - Internal & External clock pulse generation

Figure 9-1. PLL & CLOCK Block Diagram



9.1.1 Clock Input Division

The PLL can divide the input frequency by any integer between 1 and 16. The combination of input division and output low-power division (see Section 9.1.4) enables the user to generate almost every frequency value out of the PLL. The division factor may be modified by changing the value of the Pre-Division Factor Bits (PDF - PD3:0) in the PLL control register. The output frequency of the pre-divider is

$$\frac{F_{ext}}{PDF}$$

9.1.2 Frequency Multiplication

The PLL can multiply the input frequency by any integer between 1 and 4096. The multiplication factor may be modified by changing the value of the Multiplication Factor (MF) Bits MF[11:0] in the PLL control register. The output frequency of the PLL ("PLL OUT" in figure 9-1) is

$$\frac{F_{ext} \cdot MF \cdot 2}{PDF}$$

Notice that this is not the chip operating frequency but rather the input to the CLKGEN block.

9.1.3 Skew Elimination

The PLL is capable of eliminating the skew between the external clock entering the chip (EXTAL) and the internal clock phases and CLKOUT pin, making it useful for tighter synchronous timings. Skew elimination is active only when the PLL is enabled and programmed with a multiplication factor less than or equal to 4. When the PLL is disabled, or when the multiplication factor is greater than 4, or when the pre division factor is greater than 1, clock skew may exist.

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

9.1.4 Low Power Divide and Output Stage

The Clock-Generator has a divider connected to the output of the PLL. The output frequency of the PLL may be divided by a factor of 2^n (where $0 \leq n \leq 7$). The division factor may be modified by changing the value of the Division Factor Bits (DF - DF2:0) in the PLL control register. This divider permits reducing or restoring the chip operating frequency without losing the PLL lock.

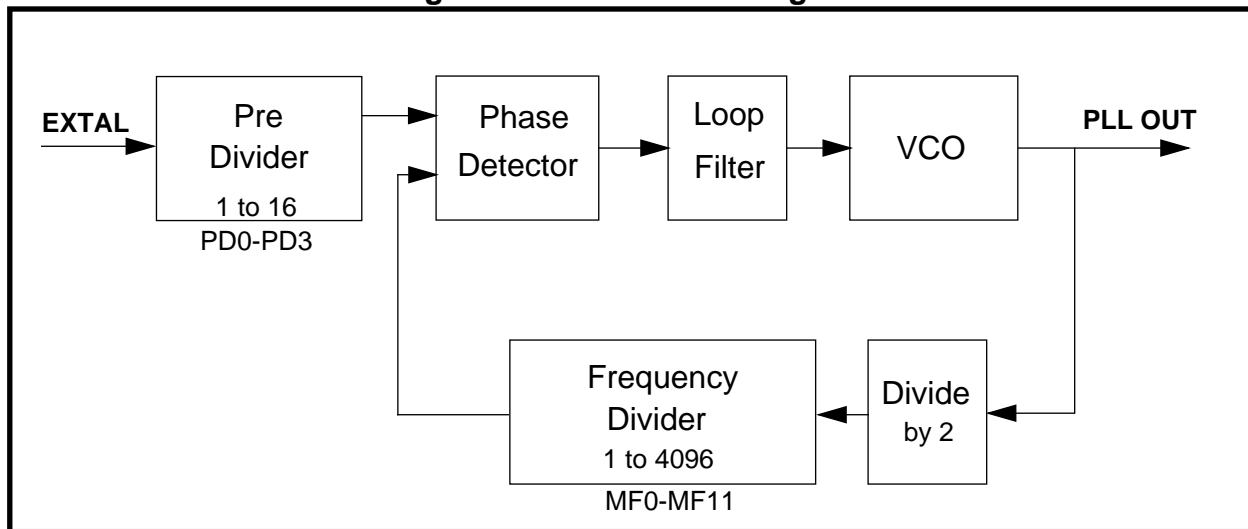
The Output Stage of the Clock-Generator generates the clock signals to the core and the chip peripherals, and drives the CLKOUT pin. The Output Stage divides the frequency by 2. The input source to the Output Stage is selected between:

- EXTAL itself (PEN=0 i.e. PLL disabled), that causes chip frequency to be
- Low Power Divider output (PEN=1 i.e. PLL enabled), that cause chip frequency to be

9.2 PLL BLOCK DIAGRAM

The PLL block diagram is shown in Figure 9-2. The components of the PLL are described in the following sections.

Figure 9-2. PLL Block Diagram



9.2.1 Frequency Pre-Divider

Clock input frequency division is accomplished by means of a frequency divider of the input frequency. The programmable division factor ranges from 1 to 16.

9.2.2 Phase Frequency 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 9.4) and determines the PLL operation. (See the appropriate Technical Data Sheet for more detailed information about a particular device’s capacitor value.)

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.

9.2.3 PLL Control Register (PCTL)

The PLL control register (PCTL) is an X-I/O mapped 24-bit read/write register used to direct the operation of the on-chip PLL. The PCTL control bits are described in the

following sections.

Figure 9-3. PLL Control Register (PCTL)

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
PD3	PD2	PD1	PD0	COD	PEN	PSTP	XTLD	XTLR	DF2	DF1	DF0

9.2.3.1 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 multiplication factor MF can be any integer from 1 to 4096. Table 9-1 shows how to program the MF0-MF11 bits. The VCO will oscillate at a frequency of:

Where PDF is the division factor of the Pre-Divider.

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 multiplication factor bits (MF0-MF11) are set to a predetermined value during hardware reset; the value is implementation dependent and may be found in each DSP56300 based derivative user's manual.

Table 9-1. Multiplication Factor Bits MF0-MF11

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

9.2.3.2 Division Factor Bits (DF2-DF0) - Bits 12-14

The Division Factor Bits DF2-DF0 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 2^7 . Table 9-2 shows the programming of the DF2-DF0 bits. Changing the value of the DF2-DF0 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 DF2-DF0 bits rather than changing the MF0-MF11 bits. For $MF \leq 4$, changing DF2-DF0 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. These bits are cleared (division by one) by hardware reset.

Table 9-2. Division Factor Bits DF0-DF2

DF2-DF0	Division Factor DF
\$0	2^0
\$1	2^1
\$2	2^2
•	•
•	•
•	•
\$7	2^7

9.2.3.3 Crystal Range Bit (XTLR) - Bit 15

The Crystal Range (XTLR) bit controls the on-chip crystal oscillator transconductance. If the external crystal frequency is less than 200kHz (“fork crystal”), this bit should be set in order to decrease the transconductance of the input amplifier, otherwise the internal clocks may not be stable. If the external crystal frequency is greater than 200kHz, this bit should be cleared in order to have the full transconductance, otherwise the crystal oscillator may not function at all. The XTLR bit is set to a predetermined value during hardware reset; the value is implementation dependent and may vary between each DSP56300 based derivative.

9.2.3.4 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 to set XTLD (disabling XTAL) to minimize RFI noise and power dissipation. The XTLD bit is set to a predetermined value during hardware reset; the value is implementation dependent and may vary between each DSP56300 based derivative.

9.2.3.5 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. When PSTP is cleared, the PLL and the on-chip crystal oscillator will be disabled when the chip enters

the STOP processing state. For minimum 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, PSTP should be set. PSTP is cleared by hardware reset.

9.2.3.6 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 is not operating in order to minimize power consumption. The PEN bit may be set or cleared by software any time during the chip operation. During hardware reset this bit receives the value of the PLL's PINIT pin, usually connected to the chip's PINIT pin.

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

PSTP	PEN	Operation during STOP		Recovery Time from STOP	Power Consumption during STOP
		PLL	Oscillator		
0	x	Disabled	Disabled	long	minimal
1	0	Disabled	Enabled	short	lower
1	1	Enabled	Enabled	short	higher

9.2.3.7 Clock Output Disable Bit (COD) - Bit 19

The COD bit controls the output buffer of the clock at the CLKOUT pin. When this bit is set, the CLKOUT pin is held in the high ("1") state. When this bit is cleared, the CLKOUT pin provides a 50% duty cycle clock synchronized to the internal core clock. If the CLKOUT pin is not connected to external circuits, it is recommended to set COD (disabling clock output) to minimize RFI noise and power dissipation. The COD bit is cleared by hardware reset. CLKOUT pin oscillates at all the machine operating states except the STOP processing state.

9.2.3.8 PreDivider Factor Bits (PD0-PD3) - Bits 20-23

The PreDivider Factor Bits PD0-PD3 define the predivision factor PDF that will be applied to the PLL input frequency. The predivision factor PDF can be any integer from 1 to 16. Table 9-1 shows how to program the PD0-PD3 bits. The VCO will oscillate at a frequency of

The predivision 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 PD0-PD3 bits, the PLL will lose the lock condition. After a time delay, the PLL will relock. The pre-divider factor bits (PD0-PD3) are set to a predetermined value during hardware reset; the value is implementation dependent and may be found in each DSP56300 based derivative user's manual.

Table 9-4. Predivision Factor Bits PD0-PD3

PD3-PD0	Predivision Factor PDF
\$0	1
\$1	2
\$2	3
• • •	• • •
\$E	15
\$F	16

9.2.4 Voltage Controlled Oscillator (VCO)

The VCO is capable of oscillating at frequencies from the minimum speed specified in a device's Technical Data Sheet (typically 30 MHz) up to the maximum allowed clock input

frequency.

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

If EXTAL frequency is less than the VCO's minimum working frequency, the user should hold PINIT pin low during hardware reset and then change (by software) MF to the desired value and change PEN to 1.

9.2.5 Divide by 2

The output of the VCO is divided by 2. This results in a constant x2 multiplication of the PLL clock output used to generate the special chip clock phases.

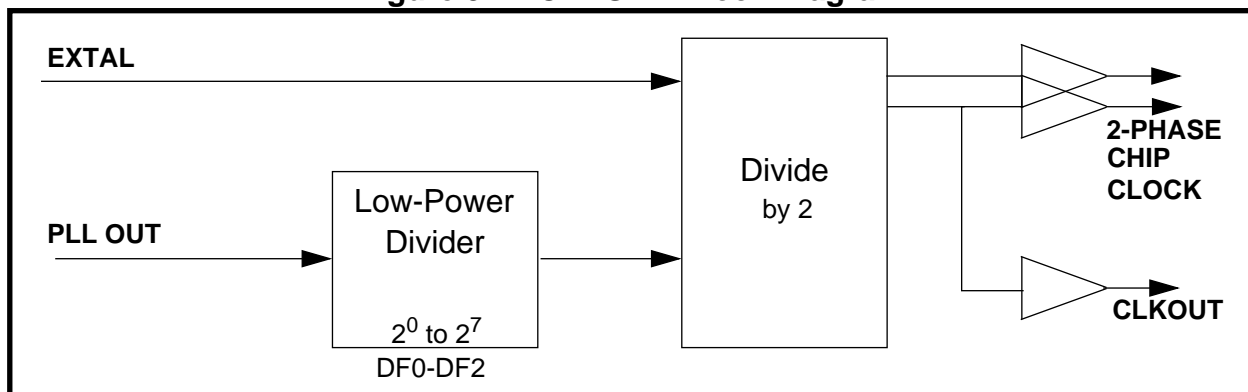
9.2.6 Frequency Divider

The Frequency Divider, connected in the feedback loop of the PLL, is used to multiply the incoming external clock. In the PLL close-loop, the effect of the frequency divider is to multiply the PLL input frequency by its division factor. The programmable division factor ranges from 1 to 4096, resulting in frequency multiplication in the same range.

9.3 CLKGEN BLOCK DIAGRAM

The CLOCK GENERATOR block diagram is shown in Figure 9-4. The components of the CLOCK GENERATOR are described in the following sections.

Figure 9-4. CLKGEN Block Diagram



9.3.1 Low Power Divider (LPD)

The Low Power Divider (LPD) divides the output frequency of the VCO by any power of 2 from 2^0 to 2^7 . Since the LPD is not in the closed loop of the PLL, changes in the divide factor 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.

9.3.2 Divide by 2

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

9.3.3 Operating Frequency

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

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

where MF is the multiplication factor defined by the MF0-MF11 bits, PDF is the predivision factor defined by the PD0-PD3 bits and DF is the division factor defined by the DF0-DF2 bits. F_{CHIP} is the chip operating frequency, and F_{EXT} is the external input frequency to the chip at the EXTAL pin.

9.3.4 Synchronization among EXTAL, CLKOUT, and the Internal Clock

When the PLL is enabled (PEN bit asserted), low clock skew between EXTAL and CLKOUT is guaranteed if $\text{MF} \leq 4$. CLKOUT and the internal chip clock are fully synchronized.

9.4 PLL PINS

Some of the PLL pins need not be implemented. The specific PLL pin configuration for each DSP56300 Core 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.

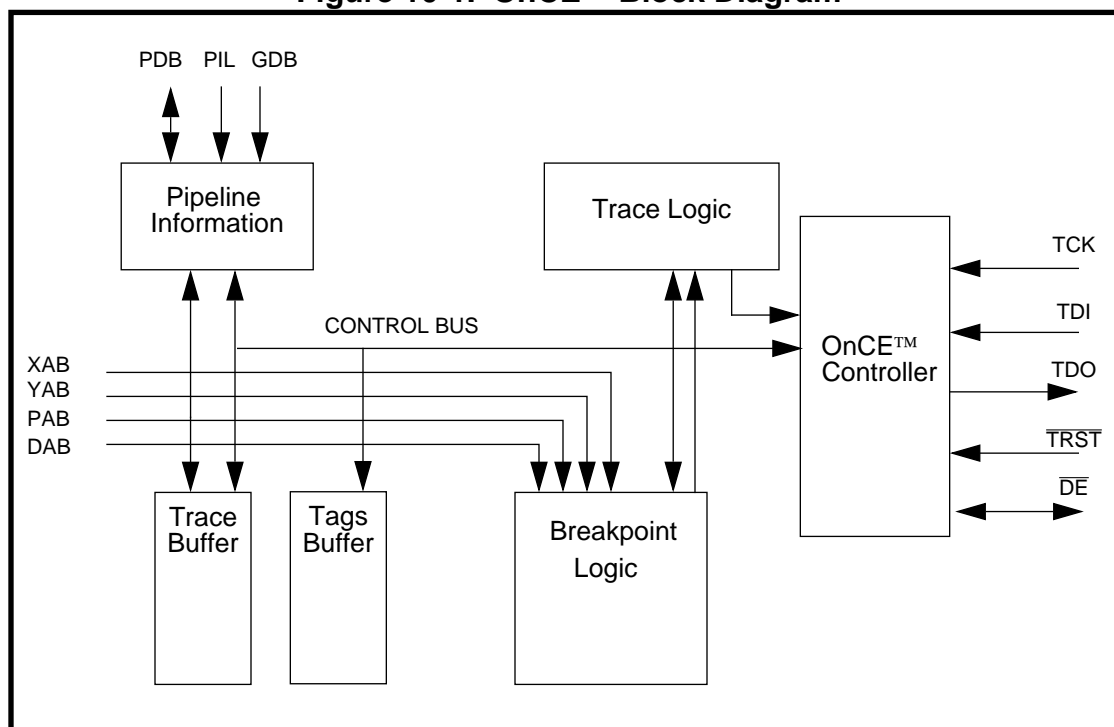
- PGND** GND dedicated to the analog PLL circuits. The pin should be provided with an extremely low impedance path to ground.
- PGND1** GND dedicated for isolating the analog PLL circuits. The pin should be provided with an extremely low impedance path to ground.
- CLVCC** VCC for the CLKOUT output. The voltage should be well regulated and the pin should be provided with an extremely low impedance path to the VCC power rail. This pin doesn't have to be a dedicated one if it can be guaranteed that it is regulated enough.
- CLGND** GND for the CLKOUT output. The pin should be provided with an extremely low impedance path to ground. This pin doesn't have to be a dedicated one if it can be guaranteed that it is regulated enough.
- 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.
- CLKOUT** 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 CLKOUT is derived from, and has half the frequency of, EXTAL. This pin oscillates in all chip processing states except STOP processing state and except a condition when bit COD in the PCTL register is implicitly set. When the chip is in the WAIT processing state, the CLKOUT pin continues to oscillate.
- 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 negated, the PINIT pin is ignored by the PLL and can have a different function in the chip.
- 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. This pin is optional and will not be implemented in all the DSP56300-Core based derivatives.

10 ON-CHIP EMULATOR (OnCE™)

10.1 INTRODUCTION

The DSP56300 Core on-chip emulation (OnCE™) circuitry provides a means of interacting with the DSP56300 Core and its peripherals non-intrusively so that a user may examine registers, memory or on-chip peripherals facilitating hardware/software development on the DSP56300 Core processor. To achieve this, special circuits and dedicated pins on the DSP56300 Core are defined to avoid sacrificing any user-accessible on-chip resource. The OnCE™ resources can be accessed only after executing the JTAG instruction ENABLE_ONCE (these resources are accessible even when the chip is operating in Normal Mode). See Chapter 11 for a description of the JTAG functionality and its relation to the OnCE. Figure 10-1 illustrates the block diagram of the OnCE™.

Figure 10-1. OnCE™ Block Diagram



10.2 ON-CHIP EMULATION (OnCE™) PINS

Since the OnCE controller functionality is accessed through the JTAG port, there are no

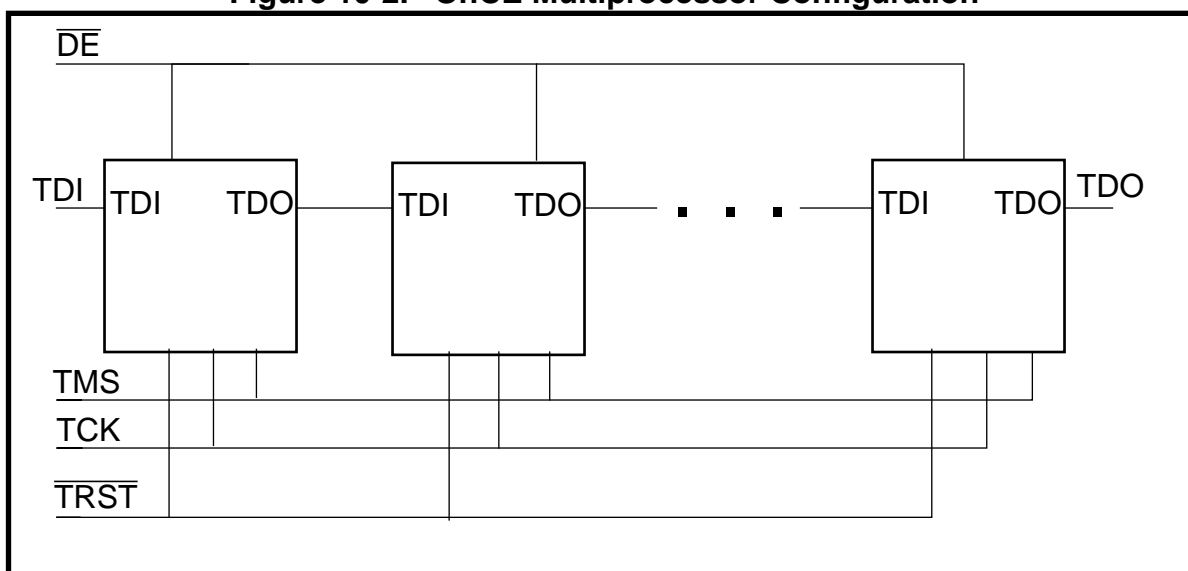
dedicated OnCE pins for clock, data-in and data-out. The JTAG pins TCK, TDI and TDO are used to shift in and out data and/or instructions. See Paragraph 11.2.1 for the description of the JTAG pins. In order to facilitate emulation specific functions, one additional pin, called \overline{DE} , may be used. This pin is described below.

10.2.1 Debug Event (\overline{DE})

The bidirectional open drain debug event pin \overline{DE} provides a fast means of entering the Debug Mode of operation from an external command controller (when input) as well as a fast means of acknowledging the entering the Debug Mode of operation to an external command controller (when output). The assertion of this pin by a command controller causes the DSP56300 Core to finish the current instruction being executed, save the instruction pipeline information, enter the Debug Mode, and wait for commands to be entered from the TDI line. If \overline{DE} was used to enter the Debug Mode then \overline{DE} must be negated after the OnCE™ responds with an acknowledge and before sending the first OnCE™ command. The assertion of this pin by the DSP56300 Core indicates that the DSP has entered the Debug Mode and is waiting for commands to be entered from the TDI line.

The \overline{DE} pin also facilitates multiple processor connections as depicted in Figure 10-2.

Figure 10-2. OnCE Multiprocessor Configuration



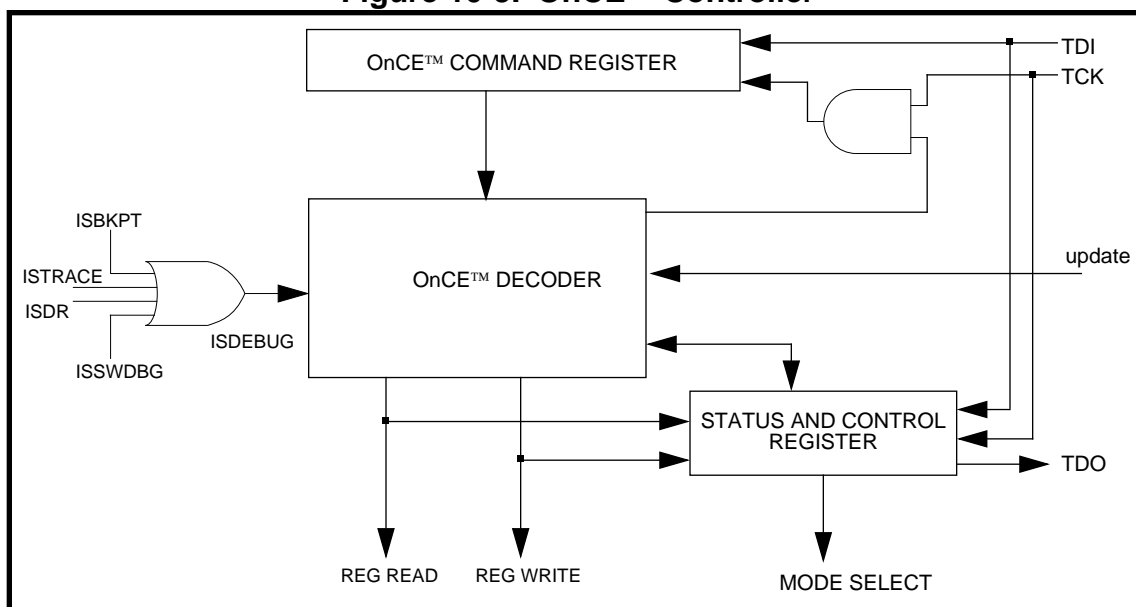
In this way the user is able to stop all the devices in the system when one of the devices has entered the Debug Mode. The user can also stop all the devices synchronously by asserting the \overline{DE} line.

10.3 OnCE™ CONTROLLER

The OnCE™ Controller contains the following blocks: OnCE™ command register, OnCE™

decoder, and the status/control register. Figure 10-3 illustrates a block diagram of the OnCE™ controller.

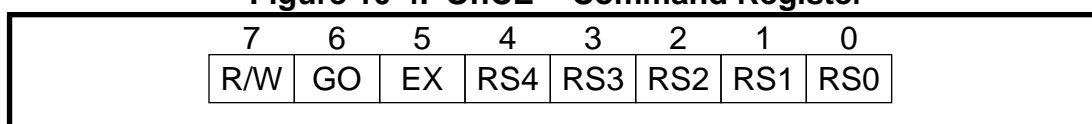
Figure 10-3. OnCE™ Controller



10.3.1 OnCE™ Command Register (OCR)

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

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. See Table 10-1 for the OnCE™ register addresses.

10.3.1.2 Exit Command (EX) Bit 5

If the EX bit is set, leave Debug Mode and resume normal operation. The Exit command is executed only if the Go command is issued, and the operation is write to OPDBR or read/write to “No Register Selected”. Otherwise the EX bit is ignored.

EX	Action
0	remain in Debug Mode
1	leave Debug Mode

Table 10-1. OnCE™ Register Addressing.

RS4-RS0	Register Selected
00000	OnCE™ Status and Control Register (OSCR)
00001	Memory Breakpoint Counter (OMBC)
00010	Breakpoint Control Register (OBCR)
00011	Reserved
00100	Reserved
00101	Memory Limit Register0 (OMLR0)
00110	Memory Limit Register1 (OMLR1)
00111	Reserved
01000	Reserved
01001	GDB Register (OGDBR)
01010	PDB Register (OPDBR)
01011	PIL Register (OPILR)
01100	PDB GO-TO Register (for GO TO command)
01101	Trace Counter (OTC)
01110	Tags Buffer (TAGB)
01111	PAB Register for Fetch (OPABFR)
10000	PAB Register for Decode (OPABDR)
10001	PAB Register for Execute (OPABEX)
10010	Trace Buffer and Increment Pointer
10011	Reserved Address
101xx	Reserved Address
11xx0	Reserved Address
11x0x	Reserved Address
110xx	Reserved Address
11111	No Register Selected

10.3.1.3 Go Command (GO) Bit 6

If the GO bit is set, execute instruction which resides in the PIL register. To execute the instruction, the core leaves the Debug Mode. The core will return to the Debug Mode immediately after executing the instruction if the EX bit is cleared. The core goes on to normal operation if the EX bit is set. The GO command is executed only if the operation is write to OPDBR or read/write to “No Register Selected”. Otherwise the GO bit is ignored.

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.

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

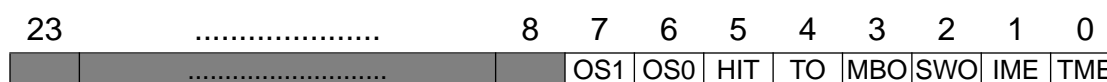
10.3.2 OnCE™ Decoder (ODEC)


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

10.3.3 OnCE™ Status and Control Register (OSCR)

The Status and Control Register is a 24-bit register used to enable the trace mode of operation and to indicate the cause of entering the Debug Mode. The control bits are read/write while the status bits are read only. The OSCR bits are cleared on hardware reset. The OSCR is shown in Figure 10-5.

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



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

10.3.3.1 Trace Mode Enable (TME) Bit 0

The TME control bit, when set, enables the Trace Mode of operation.

10.3.3.2 Interrupt Mode Enable (IME) Bit 1

The IME control bit, when set, will cause the chip to execute a vectored interrupt to the address VBA:\$06 instead of entering the Debug Mode.

10.3.3.3 Software Debug Occurrence (SWO) Bit 2

This read-only status bit is set when the Debug Mode of operation is entered due to the execution of the DEBUG or DEBUGcc instruction with condition true. This bit is cleared when leaving the Debug Mode.

10.3.3.4 Memory Breakpoint Occurrence (MBO) Bit 3

This read-only status bit is set when the Debug Mode of operation is entered due to the occurrence of a memory breakpoint. This bit is cleared when leaving the Debug Mode.

10.3.3.5 Trace Occurrence (TO) Bit 4

This read-only status bit is set when the Debug Mode of operation is entered when the trace counter is zero while trace mode is enabled. This bit is cleared when leaving the Debug Mode.

10.3.3.6 Cache Hit (HIT) Bit 5

This read only status bit is set when a cache hit has occurred when in cache mode and in Debug Mode of operation. When in PRAM mode this bit will read as one.

10.3.3.7 Core Status (OS0,OS1) Bits 6-7

These read only status bits provide core status information. By examining the status bits the user can determine whether the chip has entered the Debug Mode (examining SWO, MBO and TO will identify the cause of entering the Debug Mode). The user can also examine these bits and determine the cause why the chip has not entered the Debug Mode after debug event assertion (\overline{DE}) or as a result of the execution of the JTAG Debug Request instruction (core waiting for the bus, STOP or WAIT instruction etc.). These bits are also reflected in the JTAG instruction shift register which allows the polling of the core status information at the JTAG level. This is useful for the case in which the DSP56300 Core executes the STOP instruction (and therefore there are no clocks) to allow the reading of OSCR. See Table 10-4 for the definition of the OS0-OS1 bits.

Table 10-2. Core Status Bits Description.

OS1	OS0	DESCRIPTION
0	0	DSP56300 Core is executing instructions
0	1	DSP56300 Core is in Wait or STOP
1	0	DSP56300 Core is waiting for bus
1	1	DSP56300 Core is in Debug Mode

10.3.3.8 Reserved Bits 8-23

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

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 approximate address range of where the program may be executing. This significantly increases the programmer's ability to monitor what the program is doing in real-time.

The breakpoint logic, described in Figure 10-6., contains a latch for the addresses, registers that store the upper and lower address limit, address comparators and a breakpoint counter. Address comparators are useful in determining where a program may be getting lost or when data is being written 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 and while in any operating mode. Memory accesses are monitored according to the contents of the OBCR as specified in Section 10.4.6.

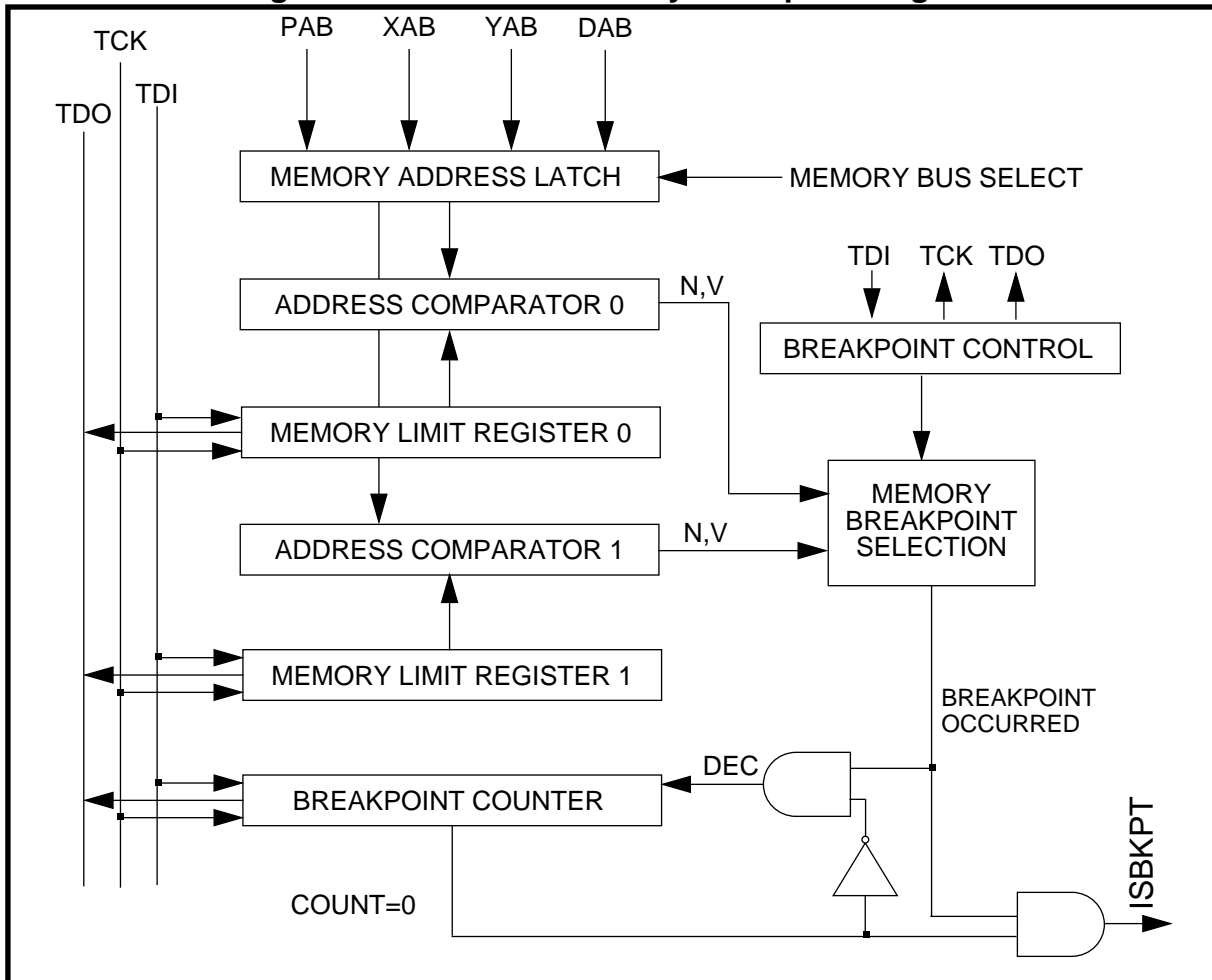
10.4.1 Memory Address Latch (OMAL)

The Memory Address Latch is a 24-bit register that latches the PAB, XAB, YAB or DAB on every instruction cycle according to the MBS1-MBS0 bits in OBCR.

10.4.2 Memory Limit Register 0 (OMLR0)

The Memory Limit Register 0 is a 24-bit register that stores the memory breakpoint limit. OMLR0 can be read or written through the JTAG port. Before enabling breakpoints, OMLR0 must be loaded by the external command controller.

Figure 10-6. OnCE™ Memory Breakpoint Logic 0



10.4.3 Memory Address Comparator 0 (OMAC0)

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

10.4.4 Memory Limit Register 1 (OMLR1)

The Memory Limit Register1 is a 24-bit register that stores the memory breakpoint limit. OMLR1 can be read or written through JTAG port. Before enabling breakpoints, OMLR1 must be loaded by the external command controller.

10.4.5 Memory Address Comparator 1 (OMAC1)

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


10.4.6 Breakpoint Control Register (OBCR)

The Breakpoint Control Register is a 24-bit register used to define the memory breakpoint events. OBCR can be read or written through the JTAG port. All the bits of the OBCR are

cleared on hardware reset. The OBCR is described in Figure 10-7.

Figure 10-7. Breakpoint Control Register

23	22	21	20	19	18	17	16	15	14	13	12
11	10	9	8	7	6	5	4	3	2	1	0
BT1	BT0	CC11	CC10	RW11	RW10	CC01	CC00	RW01	RW00	MBS1	MBS0

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

10.4.6.1 Memory Breakpoint Select (MBS0-MBS1) Bits 0-1

These control bits enable memory breakpoints 0 and 1, allowing them to occur when a memory access is performed on P, X or Y space or when a DMA access is performed. See the following table for the definition of the MBS0-MBS1 bits.

Table 10-3. Memory Breakpoint 0 and 1 Select Table.

MBS1	MBS0	DESCRIPTION
0	0	Breakpoint on DMA access
0	1	Breakpoint on P access
1	0	Breakpoint on X access
1	1	Breakpoint on Y access

10.4.6.2 Breakpoint 0 Read/Write Select (RW00-RW01) Bits 2-3

These control bits define the memory breakpoints 0 to occur when a memory address accesses is performed for read, write or both. See the following table for the definition of the RW00-RW01 bits.

Table 10-4. Breakpoint 0 Read/Write Select Table

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

10.4.6.3 Breakpoint 0 Condition Code Select (CC00-CC01) Bits4-5

These control bits define the condition of the comparison between the current memory address (OMAL) and the memory limit register 0 (OMLR0). See the following table for the definition of the CC00-CC01 bits.

Table 10-5. Breakpoint 0 Condition Select Table

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

10.4.6.4 Breakpoint1 Read/Write Select (RW10-RW11) Bits 6-7

These control bits define memory breakpoints 1 to occur when a memory address accesses is performed for read, write or both. See the following table for the definition of the RW10-RW11 bits.

Table 10-6. Breakpoint 1 Read/Write Select Table

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

10.4.6.5 Breakpoint1 Condition Code Select (CC10-CC11) Bits8-9

These control bits define the condition of the comparison between the current memory address (OMAL) and the memory limit register 1 (OMLR1). See the following table for the definition of the CC10-CC11 bits.

Table 10-7. Breakpoint 1 Condition Select Table

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

10.4.6.6 Breakpoint 0 and 1 Event Select (BT1-BT0) Bits10-11

These control bits define the sequence between breakpoint 0 and 1. If the condition defined by BT1-BT0 is met, then the Breakpoint Counter (OMBC) is decremented. See the following table for the definition of the BT1-BT0 bits.

Table 10-8. Breakpoint 0 and 1 Event Select Table

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

10.4.7 Memory Breakpoint Counter (OMBC)

The Memory Breakpoint Counter is a 24-bit counter which is loaded with a value equal to the number of times minus one that a memory access event should occur before a memory breakpoint is declared. The memory access event is specified by the OBCR register and by the memory 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 or written through the JTAG port. Every time that the limit register is changed, or a different breakpoint event is selected in the OBCR, the breakpoint counter must be written afterwards. This ensures that the OnCE™ breakpoint logic is reset and that no previous events will affect the new breakpoint event selected. The breakpoint counter is cleared by hardware reset.

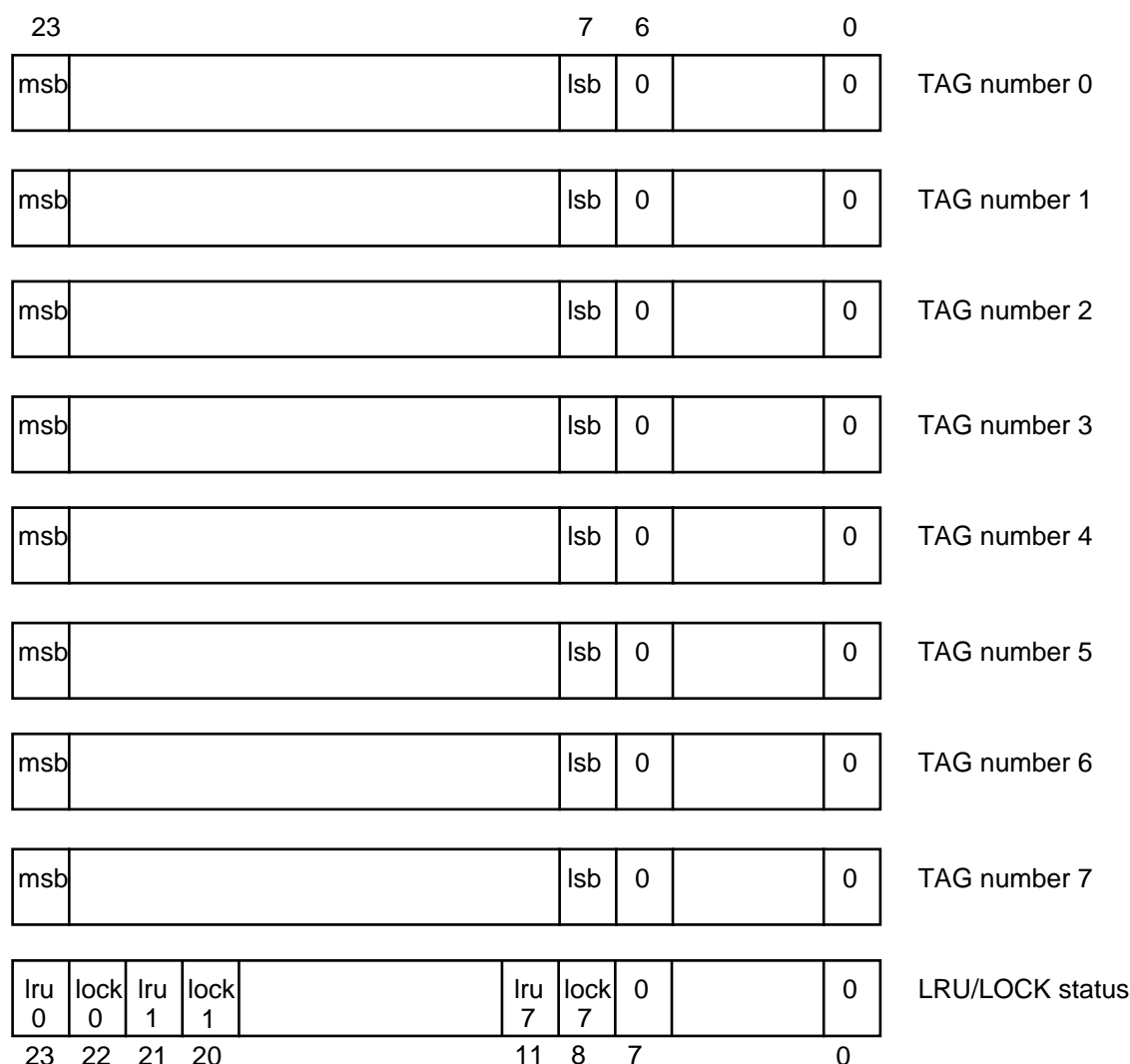
10.5 CACHE SUPPORT

To keep track of the cache contents and status, the eight tags values, tags lock/unlock status and LRU status can be read via the OnCE™. Nine 24 bits registers are implemented as a circular buffer with a 4-bit counter. All these registers have the same address but any access to the tags buffer will cause the counter to increment thus pointing to the next register in the circular buffer. When exiting the Debug Mode the counter is cleared and therefore, when entering Debug Mode again, the first read from the tags buffer address will always start from the first register of the nine (tag number 0) and will continue circularly among these nine registers.

The registers mapping in the circular tags buffer is shown in Figure 10-8.

Determining the “next to be replaced sector”: In any time point at least one LRU bit in the “LRU/LOCK status” register will be set. But it is possible for more than one of the LRU bits to be set simultaneously. This is due to the fact that locked sectors could be “least recently used” although they can not be replaced. Therefore the “next to be replaced sector” is the only sector whose LRU bit is set and lock bit cleared. There is one exception to this rule and this is the case where all the 8 sectors are locked and LRU, in which case there is no “next to be replaced sector” since no sector will be replaced until at least one sector is unlocked.

Figure 10-8. Circular Tags Buffer (TAGB)



10.6 OnCE™ TRACE LOGIC

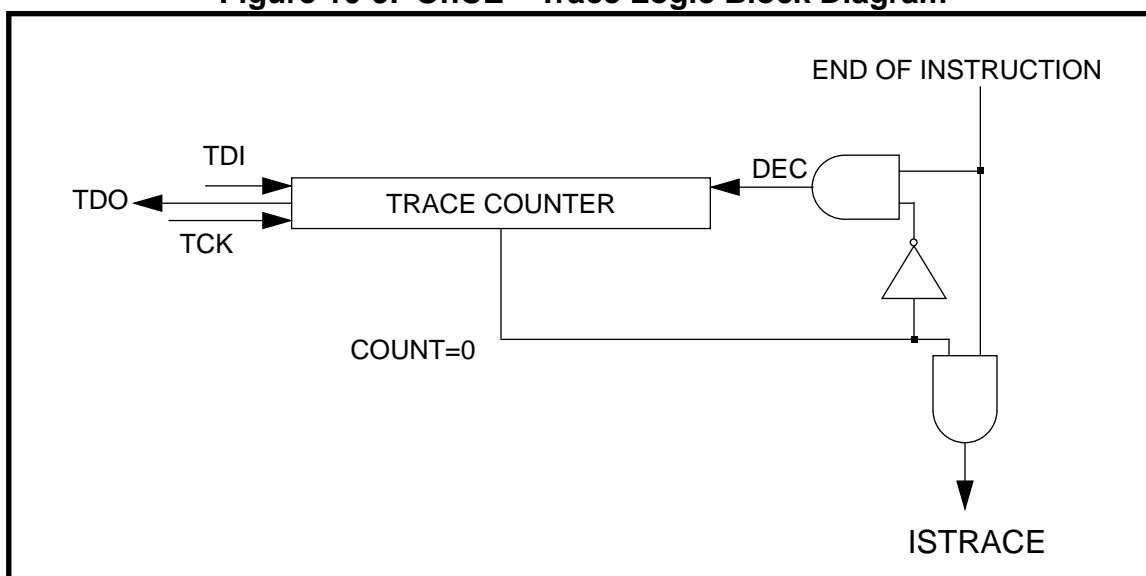
Using the OnCE™ Trace logic, execution of instructions in single or multiple steps is possible. The OnCE™ Trace logic causes the chip to enter the Debug Mode of operation after the execution of one or more instructions and wait for OnCE™ commands from the debug serial port. The OnCE™ Trace logic block diagram is shown in Figure 10-9.

The trace mode has a counter associated with it so that more than one instruction may be executed before returning back to the Debug Mode of operation. The objective of the counter is to allow the user to take multiple instruction steps real-time before entering the Debug Mode. This feature helps the software developer debug sections of code 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 enable the trace mode of operation the counter is loaded with a value, the program counter is set to the start location of the instruction(s) to be executed real-time, the TME bit is set in the OSCR and the DSP56300 Core exits the Debug Mode by executing the appropriate command issued by the external command controller.

Upon exiting the Debug Mode, the counter is decremented after each execution of an instruction. Interrupts are serviceable and all instructions executed (including fast interrupt services and the execution of each repeated instruction) will decrement the trace counter. Upon decrementing to zero, the DSP56300 Core will re-enter the Debug Mode, the trace occurrence bit TO in the OSCR will be set, the Core Status bits OS1 and OS0 will be set to 11 and the \overline{DE} pin will be asserted to indicate that the DSP56300 Core has entered Debug Mode and is requesting service.

Figure 10-9. OnCE™ Trace Logic Block Diagram



10.6.1 Trace Counter (OTC)

The Trace Counter (OTC) is a 24-bit counter that can be read or written through the JTAG port. If N instructions are to be executed before entering the Debug Mode, the Trace Counter should be loaded with N-1. The Trace Counter is cleared by hardware reset.

10.7 METHODS OF ENTERING THE DEBUG MODE

Entering the Debug Mode is acknowledged by the chip by setting the Core Status bits OS1 and OS0 and asserting the \overline{DE} line. This informs the external command controller that the chip has entered the Debug Mode and is waiting for commands. The DSP56300 Core may disable the OnCE™ circuitry if the ROM Security option is implemented. If the ROM Security is implemented, the OnCE™ will remain inactive until a write operation to the OGDBR register is executed by the DSP56300 Core.

10.7.1 External Debug Request During $\overline{\text{RESET}}$

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

10.7.2 External Debug Request During Normal Activity

Holding the $\overline{\text{DE}}$ line asserted during normal chip activity causes the chip to finish the execution of the current instruction and then enter the Debug Mode. After receiving the acknowledge, the external command controller must negate the $\overline{\text{DE}}$ line before sending the first command. 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 instructions that will be aborted by the interrupt processing.

10.7.3 Executing the JTAG `DEBUG_REQUEST` Instruction

Executing the JTAG instruction `DEBUG_REQUEST` will assert an internal debug request signal. Consequently, the chip will finish the execution of the current instruction and will stop after the newly fetched instruction enters the instruction latch. After entering the Debug Mode the Core Status bits `OS1` and `OS0` will be set and the $\overline{\text{DE}}$ line will be asserted thus acknowledging the external command controller that the Debug Mode of operation has been entered.

10.7.4 External Debug Request During `STOP`

Executing the JTAG instruction `DEBUG_REQUEST` (or asserting $\overline{\text{DE}}$) while the chip is in the `STOP` state (i. e., has executed a `STOP` instruction) causes the chip to exit the `STOP` state and enter the Debug Mode. After receiving the acknowledge, the external command controller must negate $\overline{\text{DE}}$ before sending the first command. Note that in this case, the chip completes the execution of the `STOP` instruction and halts after the next instruction enters the instruction latch.

10.7.5 External Debug Request During `WAIT`

Executing the JTAG instruction `DEBUG_REQUEST` (or asserting $\overline{\text{DE}}$) while the chip is in the `WAIT` state (i. e., has executed a `WAIT` instruction) causes the chip to exit the `WAIT` state and enter the Debug Mode. After receiving the acknowledge, the external command controller must negate $\overline{\text{DE}}$ before sending the first command. Note that in this case, the chip completes the execution of the `WAIT` instruction and halts after the next instruction enters the instruction latch.

10.7.6 Software Request During Normal Activity

Upon executing the DSP56300 Core instruction `DEBUG` (or `DEBUGcc` when the specified condition is true), the chip enters the Debug Mode after the instruction following the

DEBUG instruction has entered the instruction latch.

10.7.7 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. Execution of an instruction when the Trace Counter is zero will cause the chip to enter the Debug Mode after completing the execution of the instruction. 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.

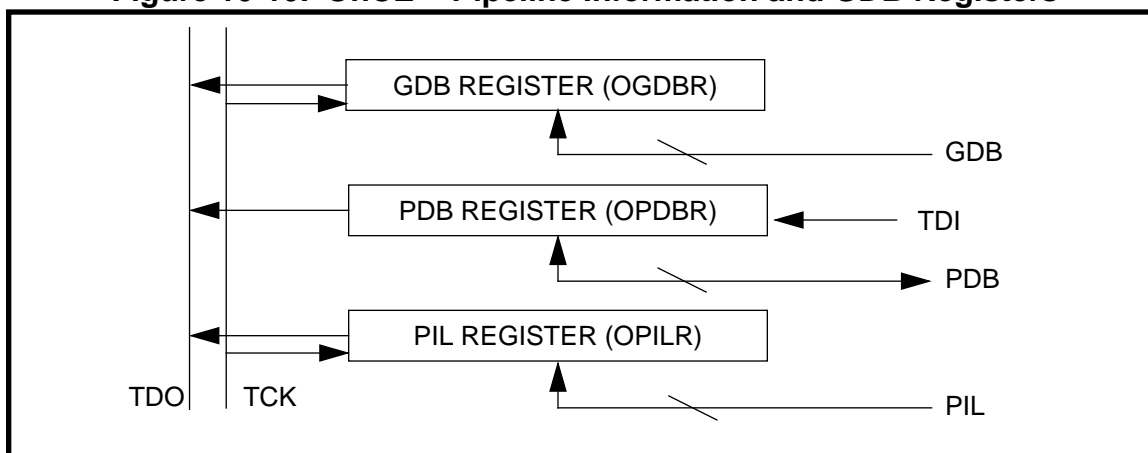
10.7.8 Enabling Memory Breakpoints

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 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.

10.8 PIPELINE INFORMATION AND GDB REGISTER

To restore the pipeline and to resume normal chip activity upon returning from the Debug Mode, a number of on-chip registers store the chip pipeline status. Figure 10-10 shows the block diagram of the Pipeline Information Registers with the exception of the PAB registers which are shown in Figure 10-11.

Figure 10-10. OnCE™ Pipeline Information and GDB Registers



10.8.1 PDB Register (OPDBR)

The PDB Register is a 24-bit latch that stores the value of the Program Data Bus

generated by the last program memory access of the core before the Debug Mode is entered. OPDBR can be read or written through the JTAG port. This register is affected by the operations performed during the Debug Mode and must be restored by the external command controller when returning to Normal Mode.

10.8.2 PIL Register (OPILR)

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

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

10.8.3 GDB Register (OGDBR)

The GDB Register is a 24-bit latch that can only be read through the JTAG port. 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 \$FFFFFFC. 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".

10.9 TRACE BUFFER

To ease debugging activity and keep track of program flow the DSP56300 Core provides a number of on-chip dedicated resources. There are three read-only PAB registers which give pipeline information when the Debug Mode is entered and a Trace Buffer which stores the address of the last instruction that was executed as well as the addresses of the last 12 change of flow instructions.

10.9.1 PAB Register for Fetch (OPABFR)

The PAB Register for Fetch is a 24-bit register that stores the address of the last instruction whose fetch was started before the Debug Mode was entered. OPABFR can only be read through the JTAG port. This register is not affected by the operations performed during the Debug Mode.

10.9.2 PAB Register for Decode (OPABDR)

The PAB Register for Decode is a 24-bit register that stores the address of the instruction

currently on the PDB. This is the instruction whose fetch was completed before the chip has entered the Debug Mode. OPABDR can only be read through the JTAG port. This register is not affected by the operations performed during the Debug Mode.

10.9.3 PAB Register for Execute (OPABEX)

The PAB Register for Execute is a 24-bit register that stores the address of the instruction currently in the Instruction Latch. This is the instruction that would have been decoded and executed if the chip would not have entered the Debug Mode. OPABEX can only be read through the JTAG port. This register is not affected by the operations performed during the Debug Mode.

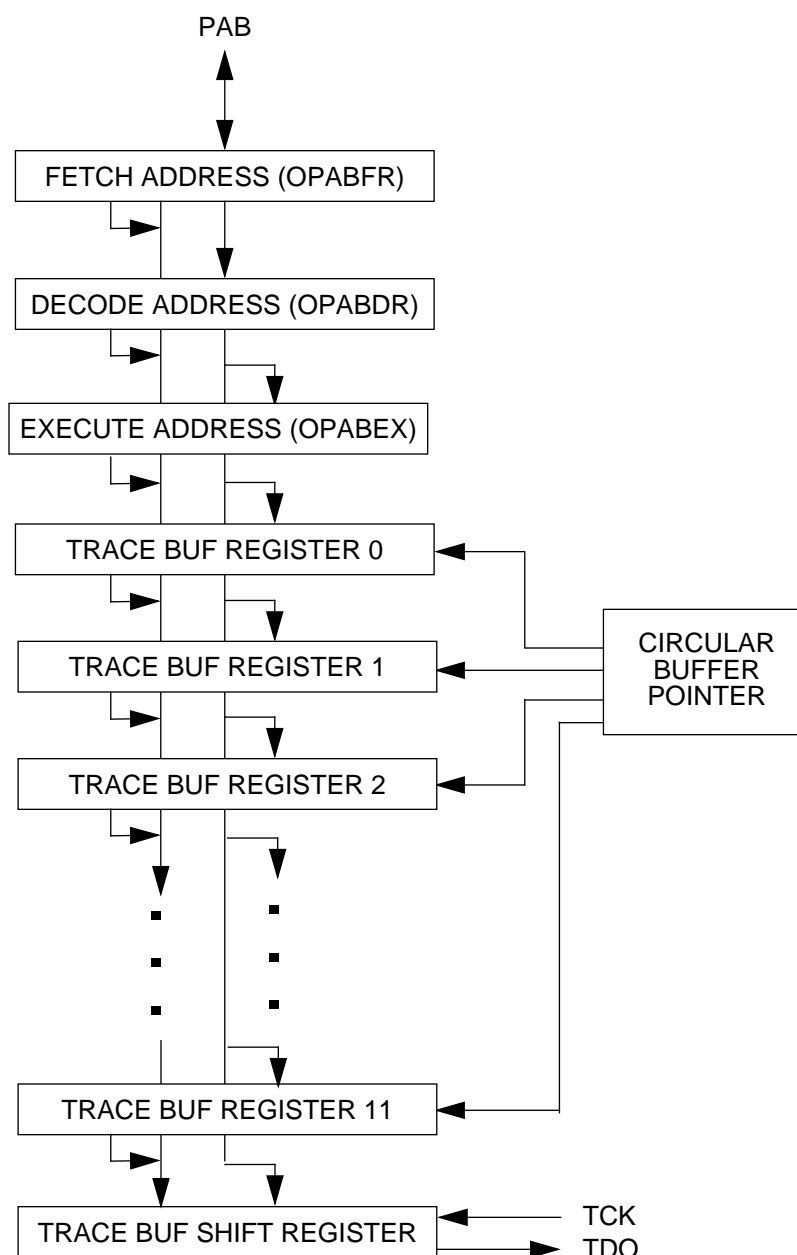
10.9.4 Trace Buffer

The Trace Buffer stores the addresses of the last 12 change of flow instructions that were executed as well as the address of the last executed instruction. The Trace Buffer is implemented as a circular buffer containing 12 25-bit registers and one 4-bit counter. All the registers have the same address but any read access to the Trace Buffer address will cause the counter to increment thus pointing to the next Trace Buffer register. The registers are serially available to the external command controller through their common Trace Buffer address. Figure 10-11 shows the block diagram of the Trace Buffer. The Trace Buffer is not affected by the operations performed during the Debug Mode except for the Trace Buffer pointer increment when reading the Trace Buffer. When entering the Debug Mode, the Trace Buffer counter will be pointing to the Trace Buffer register containing the address of the last executed instructions. The first Trace Buffer read will obtain the oldest address and the following Trace Buffer reads will get the other addresses from the oldest to the newest (the order of execution).

Note 1: To ensure Trace Buffer coherence, a complete set of 12 reads of the Trace Buffer must be performed. This is necessary due to the fact that each read increments the Trace Buffer pointer thus pointing to the next location. After 12 reads the pointer will point to the same location as before starting the read procedure.

Note 2: On any change of flow instruction, the Trace Buffer stores both the address of the change of flow instruction as well as the address of the target of the change of flow instruction. In the case of conditional change of flows, the address of the change of flow instruction is always stored (regardless of the fact that the change of flow is true or false) but if the conditional change of flow is false (i.e. not taken) the address of the target is not stored. In order to facilitate the program trace reconstruction every Trace Buffer location has an additional “invalid bit” (the 25th bit). If a conditional change of flow instruction has a “condition false”, the “invalid bit” will be set thus marking this instruction as “not taken”. Therefore it is imperative to read 25 bits of data when reading the 12 Trace Buffer registers. Since data is read lsb first, the “invalid bit” is the first bit to be read.

Figure 10-11. OnCE™ Trace Buffer



10.10 SERIAL PROTOCOL DESCRIPTION

To permit an efficient means of communication between the external command controller and the DSP56300 Core chip, the following protocol is adopted. Before starting any debugging activity, the external command controller has to wait for an acknowledge on the

\overline{DE} line indicating that the chip has entered the Debug Mode (optionally the external command controller can poll the OS1, OS0 bits in JTAG instruction shift register). 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 least significant bit first. After sending a command, the external command controller should wait for the DSP56300 Core chip to acknowledge execution of the command. The external command controller may send a new command only after the chip has acknowledged execution of the previous command.

10.10.1 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.

10.11 TARGET SITE DEBUG SYSTEM REQUIREMENTS

A typical debug environment consists of a target system where the DSP56300 Core based device resides in the user defined hardware. The JTAG port interfaces to the external command controller over a 8-wire link consisting of the five JTAG wires, one OnCE™ wires, a ground and a reset wire. The reset wire is optional and is only used to reset the DSP56300 Core based device and its associated circuitry.

The external command controller acts as the medium between the DSP56300 Core target system and a host computer. The external command controller circuit acts as a JTAG port driver and host computer command interpreter. The controller issues commands based on the host computer inputs from a user interface program which communicates with the user.

10.12 EXAMPLES OF USING THE OnCE™

Following are some examples of debugging procedures. Note that all the examples assume that the DSP is the only device in the JTAG chain. If there are more than one device in the chain (other DSP's or even other devices), the other devices can be forced to execute the JTAG BYPASS instruction such as their effect in the serial stream will be one bit per additional device. The events "select-DR", "select-IR", "update-DR", "shift-DR"

etc. refer to bringing the JTAG TAP in the corresponding state. Please refer to Chapter 11 for a detailed description of the JTAG protocol.

10.12.1 Checking whether the chip has entered the Debug Mode

There are two methods of verifying that the chip has entered the Debug Mode:

1. Every time the chip enters the Debug Mode, a pulse is generated on the \overline{DE} line. A pulse will also be generated every time the chip acknowledges the execution of an instruction while in Debug Mode. An external command controller can connect the \overline{DE} line to an interrupt pin in order to sense the acknowledge.
2. An external command controller can poll the JTAG instruction shift register for the status bits OS1-OS0. When the chip is in Debug Mode these bits are set to the value 11.

Note: In the following paragraphs, the ACK notation denotes the operation performed by the command controller to check whether the Debug Mode has been entered (either by sensing \overline{DE} or by polling JTAG instruction shift register).

10.12.2 Polling the JTAG instruction shift register

In order to poll the core status bits in the JTAG instruction shift register the following sequence must be performed:

1. Select shift-IR. Passing through capture-IR loads the core status bits into the instruction shift register.
2. Shift in ENABLE_ONCE. While shifting-in the new instruction the captured status information is shifted-out. Pass through update-IR.
3. . Return to Run-Test/Idle.

The external command controller can analyze the information shifted out and detect whether the chip has entered the Debug Mode.

10.12.3 Saving Pipeline Information

The debugging activity is accomplished by means of DSP56300 Core instructions supplied from the external command controller. Therefore the current state of the DSP56300 Core pipeline must be saved prior to starting the debug activity and of course the state must be restored prior to returning to the Normal Mode of operation. Following is the description of the saving procedure (assume that ENABLE_ONCE has been executed and Debug Mode has been entered and verified as described in 10.12.1):

1. Select shift-DR. Shift in the "Read PDB". Pass through update-DR.
2. Select shift-DR. Shift out the 24 bit OPDB register. Pass through update-DR.
3. Select shift-DR. Shift in the "Read PIL". Pass through update-DR.

-
4. Select shift-DR. Shift out the 24 bit OPILR register. Pass through update-DR.

Note that there is no need to verify acknowledge between steps 1 and 2 as well as 3 and 4 because completion is guaranteed by design.

10.12.4 Reading the Trace Buffer

An optional step during debugging activity is reading the information associated with the Trace Buffer in order to enable an external program to reconstruct the full trace of the executed program. Following is the description of the read Trace Buffer procedure (assume that all actions described in 10.12.3 have been executed):

1. Select shift-DR. Shift in the "Read PABFR". Pass through update-DR.
2. Select shift-DR. Shift out the 24 bit OPABFR register. Pass through update-DR.
3. Select shift-DR. Shift in the "Read PABDR". Pass through update-DR.
4. Select shift-DR. Shift out the 24 bit OPABDR register. Pass through update-DR.
5. Select shift-DR. Shift in the "Read PABEX". Pass through update-DR.
6. Select shift-DR. Shift out the 24 bit OPABEX register. Pass through update-DR.
7. Select shift-DR. Shift in the "Read FIFO". Pass through update-DR.
8. Select shift-DR. Shift out the 25 bit FIFO register. Pass through update-DR.
9. Repeat steps 7 and 8 for the entire FIFO (12 times).

Note that the user must read the entire FIFO since each read will increment the FIFO pointer thus pointing to the next FIFO location. At the end of this procedure the FIFO pointer will point back to the beginning of the FIFO.

The information that has been read by the external command controller contains now the address of the newly fetched instruction, the address of the instruction currently on the PDB, the address of the instruction currently on the instruction latch as well as the addresses of the last 12 instructions that have been executed and are change of flow. A user program can now reconstruct the flow of a full trace based on this information and on the original source code of the currently running program.

10.12.5 Displaying a specified register

The DSP56300 must be in Debug Mode and all actions described in 10.12.3 have been executed. The sequence of actions is:

1. Select shift-DR. Shift in the "Write PDB with GO no-EX". Pass through update-DR.
 2. Select shift-DR. Shift in the 24 bit opcode: "MOVE reg, X:OGDB". Pass through update-DR to actually write OPDBR and thus begin executing the MOVE instruction.
-

-
3. Wait for DSP to reenter Debug Mode (wait for \overline{DE} or poll core status).
 4. Select shift-DR and shift in "READ GDB REGISTER". Pass through update-DR (this will select OGDBR as the data register for read).
 5. Select shift-DR. Shift out the OGDBR contents. Pass through update-DR. Wait for next command.

10.12.6 Displaying X memory area starting at address \$xxxxxx

The DSP56300 must be in Debug Mode and all actions described in 10.12.3 have been executed. Since R0 will be used as pointer for the memory, R0 will be first saved. The sequence of actions is:

1. Select shift-DR. Shift in the "Write PDB with GO no-EX". Pass through update-DR.
 2. Select shift-DR. Shift in the 24 bit opcode: "MOVE R0, X:OGDB". Pass through update-DR to actually write OPDBR and thus begin executing the MOVE instruction.
 3. Wait for DSP to reenter Debug Mode (wait for \overline{DE} or poll core status).
 4. Select shift-DR and shift in "READ GDB REGISTER". Pass through update-DR (this will select OGDBR as the data register for read).
 5. Select shift-DR. Shift out the OGDBR contents. Pass through update-DR. R0 is now saved.
 6. Select shift-DR. Shift in the "Write PDB with no-GO no-EX". Pass through update-DR.
 7. Select shift-DR. Shift in the 24 bit opcode: "MOVE #\$xxxxxx,R0". Pass through update-DR to actually write OPDBR.
 8. Select shift-DR. Shift in the "Write PDB with GO no-EX". Pass through update-DR.
 9. Select shift-DR. Shift in the second word of the 24 bit opcode: "MOVE #\$xxxxxx,R0" (the \$xxxxxx field). Pass through update-DR to actually write OPDBR and execute the instruction. R0 is loaded with the base address of the memory block to be read.
 10. Wait for DSP to reenter Debug Mode (wait for \overline{DE} or poll core status).
 11. Select shift-DR. Shift in the "Write PDB with GO no-EX". Pass through update-DR.
 12. Select shift-DR. Shift in the 24 bit opcode: "MOVE X:(R0)+, X:OGDB". Pass through update-DR to actually write OPDBR and thus begin executing the MOVE instruction.
 13. Wait for DSP to reenter Debug Mode (wait for \overline{DE} or poll core status).
 14. Select shift-DR and shift in "READ GDB REGISTER". Pass through update-DR (this will select OGDBR as the data register for read).
 15. Select shift-DR. Shift out the OGDBR contents. Pass through update-DR. The memory contents of address \$xxxxxx has been read.
 16. Select shift-DR. Shift in the "NO SELECT with GO no-EX". Pass through update-DR. This will execute again the same "MOVE X:(R0)+, X:OGDB" instruction.
 17. Repeat from step #14 to complete the reading of the entire block. When
-

finished, restore the original value of R0.

10.12.7 Returning from Debug Mode to Normal Mode to current program

This is the case in which the user has finished examining the current state of the machine, changed some of the registers and wishes to return and continue execution of its program from the point where it stopped. Therefore the user has to restore the pipeline of the machine and enable normal instruction execution. The sequence of actions is:

1. Select shift-DR. Shift in the "Write PDB with no-GO no-EX". Pass through update-DR.
2. Select shift-DR. Shift in the 24 bit of saved PIL (instruction latch value). Pass through update-DR to actually write the Instruction Latch.
3. Select shift-DR. Shift in the "Write PDB with GO and EX". Pass through update-DR.
4. Select shift-DR. Shift in the 24 bit of saved PDB. Pass through update-DR to actually write the PDB. At the same time the internally saved value of the PAB is driven back from the PABFR register onto the PAB, the ODEC releases the chip from Debug Mode and the normal flow of execution is continued.

10.12.8 Returning from Debug Mode to Normal Mode to a new program

This is the case in which the user has finished examining the current state of the machine, changed some of the registers and wishes to start the execution of a new program (the GOTO command). Therefore the user has to force a "change of flow" to the starting address of the new program (\$xxxxxx). The sequence of actions is:

1. Select shift-DR. Shift in the "Write PDB with no-GO no-EX". Pass through update-DR.
2. Select shift-DR. Shift in the 24 bit "\$0AF080" which is the opcode of the JUMP instruction. Pass through update-DR to actually write the Instruction Latch.
3. Select shift-DR. Shift in the "Write PDB-GO-TO with GO and EX". Pass through update-DR.
4. Select shift-DR. Shift in the 24 bit of "\$xxxxxx". Pass through update-DR to actually write the PDB. At this time the ODEC releases the chip from Debug Mode and the execution is started from the address \$xxxxxx.

Note that if the entering of the Debug Mode happened during a DO LOOP, REP instruction or other special cases like interrupt processing, STOP, WAIT, conditional branching etc. it is mandatory that the user will first reset the DSP56300 and only afterwards proceed with the execution of the new program.

10.13 EXAMPLES OF JTAG-OnCE INTERACTION

This paragraph exemplifies the details of the JTAG-OnCE™ interaction by describing the TMS sequencing required in order to achieve the communication depicted in the Paragraph 10.12.

The external command controller can force the DSP56300 into Debug Mode by executing the JTAG instruction `DEBUG_REQUEST`. In order to check that the DSP56300 has entered the Debug Mode the external command controller must poll the status by reading the `OS1,OS0` bits in the JTAG instruction shift register. The TMS sequencing is depicted in Table 10-9.

The sequencing of enabling the OnCE™ is described in Table 10-10.

After executing the JTAG instructions `DEBUG_REQUEST` and `ENABLE_ONCE` and after the core status was polled to verify that the chip is in Debug Mode, the pipeline saving procedure must take place. The TMS sequencing for this procedure is depicted in Table 10-9.

Table 10-9. TMS Sequencing for DEBUG_REQUEST and poll the status

step	TMS	JTAG	OnCE™	Note
a	0	Run-Test/Idle	Idle	
b	1	Select-DR-Scan	Idle	
c	1	Select-IR-Scan	Idle	
d	0	Capture-IR	Idle	status is sampled in shifter
e	0	Shift-IR	Idle	the 4 bits of the JTAG DEBUG_REQUEST (0111)are shifted in while status is shifted-out
			
e	0	Shift-IR	Idle	
f	1	Exit1-IR	Idle	
g	1	Update-IR	Idle	debug req is generated
h	1	Select-DR-Scan	Idle	
i	1	Select-IR-Scan	Idle	
j	0	Capture-IR	Idle	status is sampled in shifter
k	0	Shift-IR	Idle	the 4 bits of the JTAG DEBUG_REQUEST (0111)are shifted in while status is shifted-out
			
k	0	Shift-IR	Idle	
l	1	Exit1-IR	Idle	
m	1	Update-IR	Idle	
n	0	Run-Test/Idle	Idle	This step is repeated enabling an external com- mand controller to poll the status
.....				
n	0	Run-Test/Idle	Idle	

In “step n” the external command controller verifies that the OS1-OS0 have the value 11 indicating that the chip has entered the Debug Mode. If the chip has not yet entered the Debug Mode the external command controller will go to “step b”, “step c” etc. until the Debug Mode is acknowledged.

Table 10-10. TMS Sequencing for ENABLE_ONCE

step	TMS	JTAG	OnCE™	Note
a	1	Test-Logic-Reset	Idle	
b	0	Run-Test/Idle	Idle	
c	1	Select-DR-Scan	Idle	
d	1	Select-IR-Scan	Idle	
e	0	Capture-IR	Idle	capture core status bits
f	0	Shift-IR	Idle	the 4 bits of the JTAG ENABLE_ONCE instruction (0110) are shifted into the JTAG instruction register while status is shifted-out
g	0	Shift-IR	Idle	
h	0	Shift-IR	Idle	
i	0	Shift-IR	Idle	
j	1	Exit1-IR	Idle	
k	1	Update-IR	Idle	OnCE™ is enabled
l	0	Run-Test/Idle	Idle	This step can be repeated enabling an external command controller to poll the status
.....				
1	0	Run-Test/Idle	Idle	

Table 10-11. TMS Sequencing for reading pipeline registers

step	TMS	JTAG	OnCE™	Note
a	0	Run-Test/Idle	Idle	
b	1	Select-DR-Scan	Idle	
c	0	Capture-DR	Idle	
d	0	Shift-DR	Idle	the 8 bits of the OnCE “Read PIL” (10001011)are shifted-in
.....				
d	0	Shift-DR	Idle	
e	1	Exit1-DR	Idle	
f	1	Update-DR	Execute “Read PIL”	PIL value is loaded in shifter
g	1	Select-DR-Scan	Idle	
h	0	Capture-DR	Idle	
i	0	Shift-DR	Idle	the 24 bits of the PIL are shifted-out (24 steps)
.....				
i	0	Shift-DR	Idle	
j	1	Exit1-DR	Idle	
k	1	Update-DR	Idle	
l	1	Select-DR-Scan	Idle	
m	0	Capture-DR	Idle	
n	0	Shift-DR	Idle	the 8 bit of the OnCE “Read PDB” (10001010)are shifted-in
.....				
n	0	Shift-DR	Idle	
o	1	Exit1-DR	Idle	
p	1	Update-DR	Execute “Read PDB”	PDB value is loaded in shifter
q	1	Select-DR-Scan	Idle	
r	0	Capture-DR	Idle	

step	TMS	JTAG	OnCE™	Note
s	0	Shift-DR	Idle	the 24 bits of the PDB are shifted-out (24 steps)
.....				
s	0	Shift-DR	Idle	
t	1	Exit1-DR	Idle	
u	1	Update-DR	Idle	
v	0	Run-Test/Idle	Idle	This step can be repeated enabling an external command controller to analyze the information.
.....				
v	0	Run-Test/Idle	Idle	

During “step v” the external command controller stores the pipeline information and afterwards it can proceed with the debug activities as requested by the user.

11 JTAG (IEEE 1149.1) Test Access Port

11.1 INTRODUCTION

The DSP56300 Core provides a dedicated user-accessible test access port (TAP) that is fully compatible with the IEEE 1149.1 Standard Test Access Port and Boundary Scan Architecture. Problems associated with testing high density circuit boards have led to development of this proposed standard under the sponsorship of the Test Technology Committee of IEEE and the Joint Test Action Group (JTAG). The DSP56300 Core implementation supports circuit-board test strategies based on this standard.

The test logic includes a test access port (TAP) consisting of four dedicated signal pins, a 16-state controller, and three test data registers. A boundary scan register links all device signal pins into a single shift register. The test logic, implemented utilizing static logic design, is independent of the device system logic. The DSP56300 Core implementation provides the following capabilities:

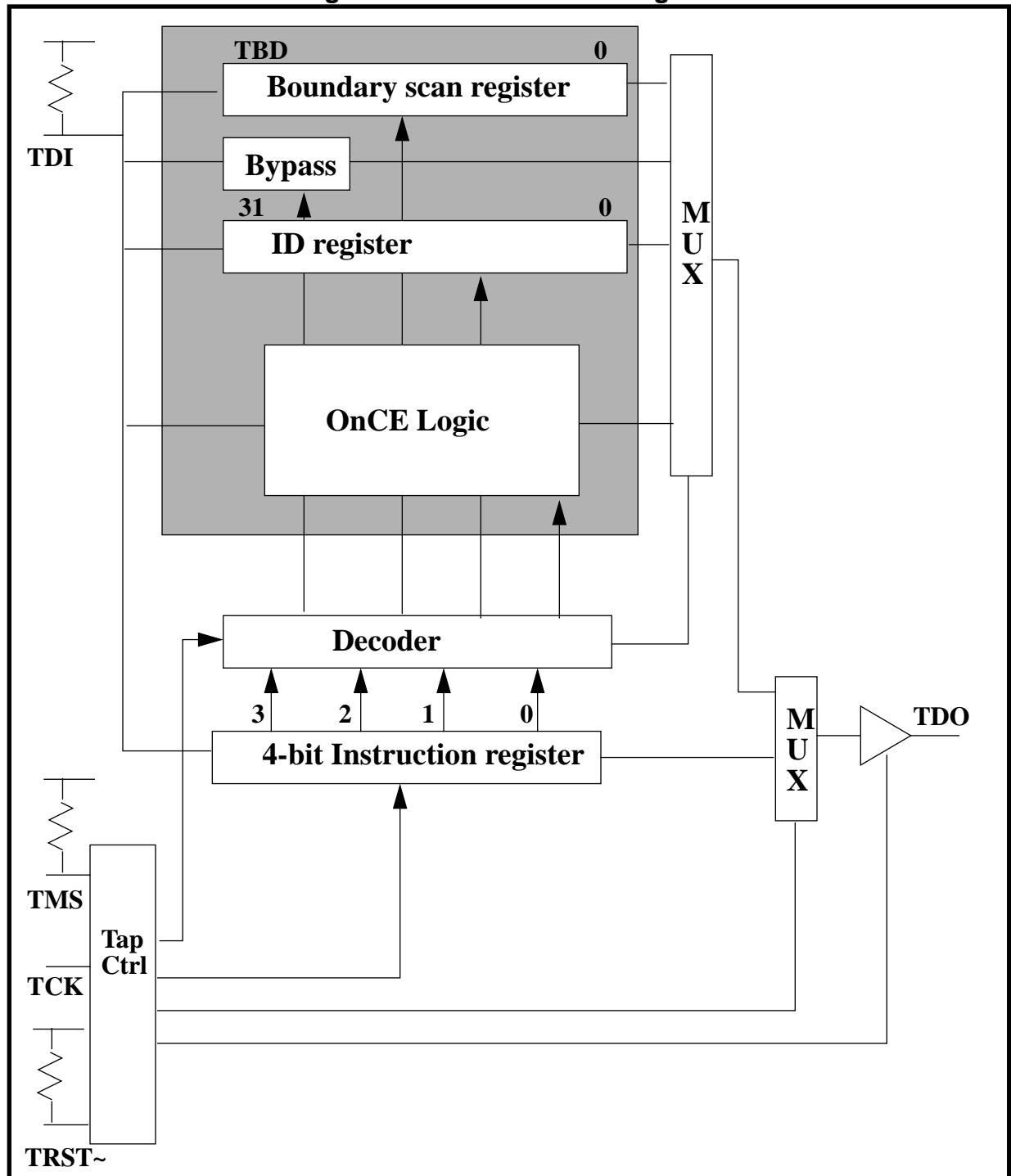
1. Perform boundary scan operations to test circuit-board electrical continuity (EXTEST).
2. Bypass the DSP56300 Core for a given circuit-board test by effectively reducing the boundary scan register to a single cell (BYPASS).
3. Sample the DSP56300 Core based device system pins during operation and transparently shift out the result in the boundary scan register. Preload values to output pins prior to invoking the EXTEST instruction (SAMPLE/PRELOAD).
4. Disable the output drive to pins during circuit-board testing (HIGHZ).
5. Provide a means of accessing the OnCE controller and circuits to control a target system (ENABLE_ONCE).
6. Provide a means of entering the Debug Mode of operation (DEBUG_REQUEST).
7. Query identification information (manufacturer, part number and version) from an DSP56300 Core based device (IDCODE).
8. Force test data onto the outputs of an DSP56300 Core based device while replacing its boundary-scan register in the serial data path with a single bit register (CLAMP).

11.2 OVERVIEW

This section, which includes aspects of the JTAG implementation that are specific to the

DSP56300 Core, is intended to be used with the supporting IEEE 1149.1 document. The discussion includes those items required by the standard to be defined and, in certain cases, provides additional information specific to the DSP56300 Core implementation. For internal details and applications of the standard, refer to the IEEE 1149.1 document. The block diagram of the DSP56300 Core implementation of JTAG is shown in Figure 11-1.

Figure 11-1. JTAG Block Diagram



The DSP56300 Core implementation includes a 4-bit instruction register and three test registers: a 1-bit bypass register, a 32-bit identification register and a TBD-bit boundary scan register. This implementation includes a dedicated TAP and four pins.

11.2.1 JTAG PINS

11.2.1.1 Test Clock (TCK)

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

11.2.1.2 Test Mode Select (TMS)

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

11.2.1.3 Test Data Input (TDI)

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

11.2.1.4 Test Data Output (TDO)

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

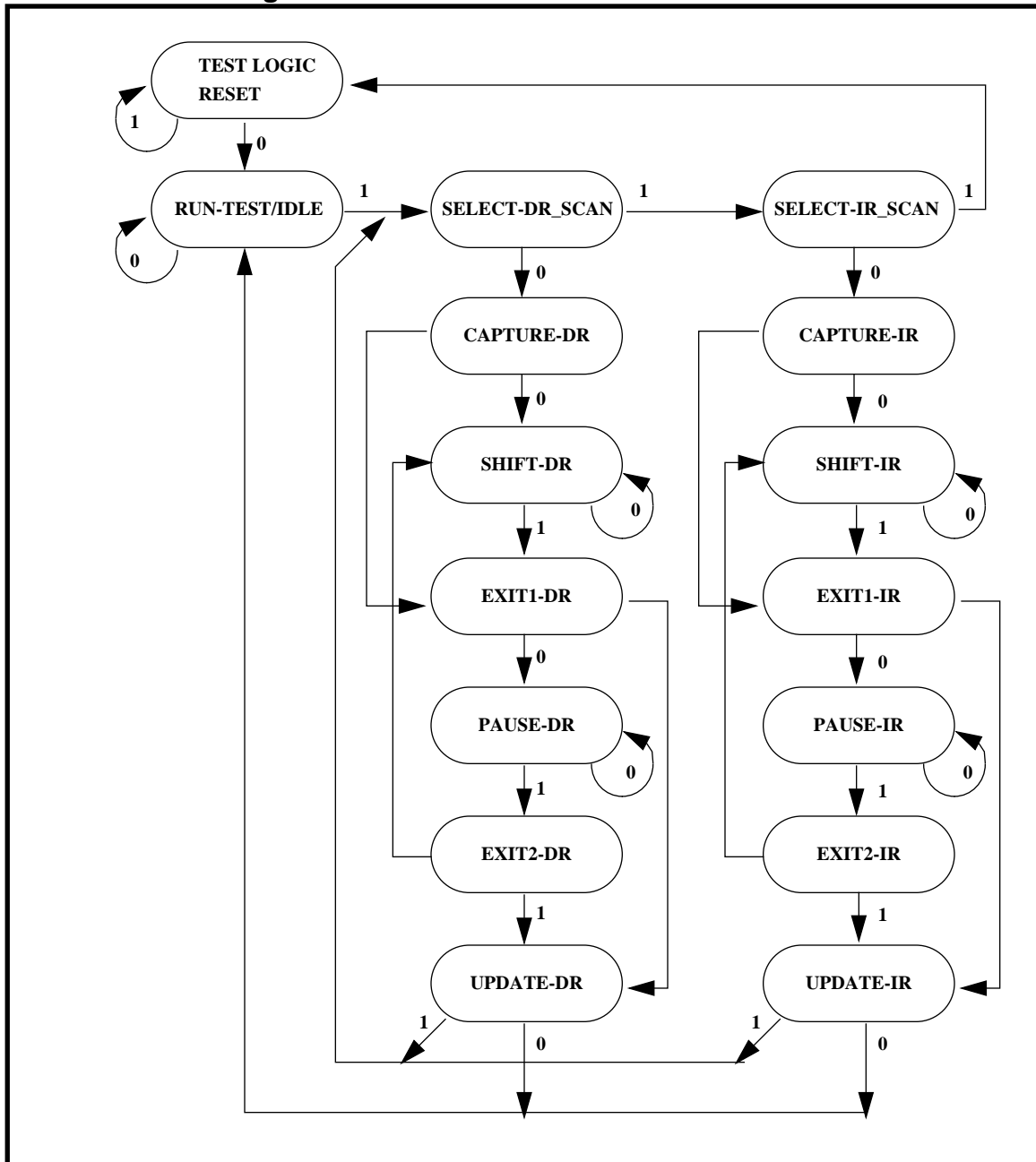
11.2.1.5 Test Reset (TRST~)

The test reset input (TRST~) pin is used to asynchronously initialize the test controller. The TRST~ has an internal pullup resistor.

11.2.2 TAP CONTROLLER

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

Figure 11-2. TAP Controller State Machine



11.2.3 BOUNDARY SCAN REGISTER

The boundary scan register (BSR) in the DSP56300 Core JTAG implementation contains bits for all device signal and clock pins and associated control signals. All DSP56300 Core bidirectional pins have a single register bit in the boundary scan register for pin data, and are controlled by an associated control bit in the boundary scan register.

The boundary scan bit definitions varies according to the specific chip implementation of the DSP56300 core and is described in it's specification document.

11.2.4 INSTRUCTION REGISTER

The DSP56300 Core JTAG implementation includes the three mandatory public instructions (EXTEST, SAMPLE/PRELOAD, and BYPASS), and also supports the optional CLAMP instruction defined by IEEE 1149.1. The public instruction (HI-Z) provides the capability for disabling all device output drivers. The public instruction (ENABLE_ONCE) enables the JTAG port to communicate with the OnCE circuitry. The public instruction (DEBUG_REQUEST) enables the JTAG port to force the DSP56300 Core into the Debug Mode of operation. The DSP56300 Core includes a 4-bit instruction register without parity consisting of a shift register with four parallel outputs. Data is transferred from the shift register to the parallel outputs during the update-IR controller state. The four bits are used to decode the eight unique instructions shown in Table 11-1. All other encodings are reserved for future enhancements and will be decoded as BYPASS.

Table 11-1. JTAG Instructions

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

The parallel output of the instruction register is reset to 0010 in the test-logic-reset controller state which is equivalent to the IDCODE instruction.

During the capture-IR controller state, the parallel inputs to the instruction shift register are loaded with the code 01 in the least significant bits as required by the standard. The two most significant bits are loaded with the values of the core status bits OS1 and OS0 from the OnCE controller. See Chapter 10 ON-CHIP EMULATOR (OnCE™) for a description of the status bits. Figure 11-3 shows the Instruction Register configuration.

Figure 11-3. Instruction Register

3	2	1	0
OS1	OS0	0	1

11.2.4.1 EXTEST

The external test (EXTEST) instruction selects the TBD-bit boundary scan register. EXTEST also asserts internal reset for the DSP56300 Core system logic to force a predictable internal state while performing external boundary scan operations.

By using the TAP, the register is capable of:

1. scanning user-defined values into the output buffers,
2. capturing values presented to input pins
3. controlling the direction of bidirectional pins,
4. controlling the output drive of three-stateable output pins.

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

11.2.4.2 SAMPLE/PRELOAD

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

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

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

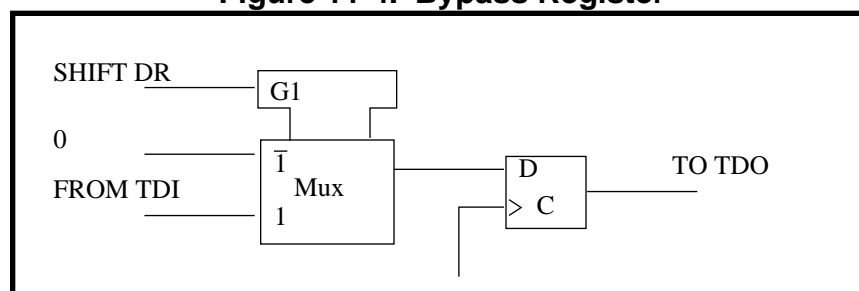
11.2.4.3 BYPASS

The BYPASS instruction selects the single-bit bypass register as shown in Figure 11-4. This creates a shift-register path from TDI to the bypass register and, finally, to TDO, circumventing the TBD-bit boundary scan register. This instruction is used to enhance test efficiency when a component other than the DSP56300 Core based device becomes the device under test.

When the bypass register is selected by the current instruction, the shift-register stage is set to a logic zero on the rising edge of TCK in the capture-DR controller state. Therefore,

the first bit to be shifted out after selecting the bypass register will always be a logic zero.

Figure 11-4. Bypass Register



11.2.4.4 IDCODE

The IDCODE instruction selects the ID register. This instruction is provided as a public instruction to allow the manufacturer, part number and version of a component to be determined through the TAP. Figure 11-3 shows the ID register configuration.

Figure 11-5. Identification Register Configuration

31	28	27	12	11	1	0
Version Information		Customer Part Number		Manufacturer Identity		1

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

Motorola's Manufacturer Identity is 00000001110. The Customer Part Number consists of two parts: Motorola Design Center Number (bits 27:22) and a sequence number (bits 21:12). MSIL Design Center Number is 000110.

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

11.2.4.5 HI-Z

The HI-Z instruction is not included in the IEEE 1149.1 standard. It is provided as a manufacturer's optional public instruction to prevent having to backdrive the output pins during circuit-board testing. When HI-Z is invoked, all output drivers, including the

two-state drivers, are turned off (i.e., high impedance). The instruction selects the bypass register. The HI-Z instruction also asserts internal reset for the DSP56300 Core system logic to force a predictable internal state while performing external boundary scan operations

11.2.4.6 CLAMP

The CLAMP instruction is not included in the IEEE 1149.1 standard. It is provided as a public instruction that selects the 1-bit bypass register as the serial path between TDI and TDO while allowing signals driven from the component pins to be determined from the boundary scan register. During testing of ICs on PCB, it may be necessary to place static guarding values on signals that control operation of logic not involved in the test. The EXTEST instruction could be used for this purpose, but since it selects the boundary-scan register the required guarding signals would be loaded as part of the complete serial data stream shifted in, both at the start of the test and each time a new test pattern is entered. Since the CLAMP instruction allows guarding values to be applied using the boundary-scan register of the appropriate ICs while selecting their bypass registers, it allows much faster testing than does the EXTEST instruction. Data in the boundary scan cell remains unchanged until a new instruction is shifted in or the JTAG state machine is set to its reset state. The CLAMP instruction also asserts internal reset for the DSP56300 Core system logic to force a predictable internal state while performing external boundary scan operations.

11.2.4.7 ENABLE_ONCE

The ENABLE_ONCE instruction is not included in the IEEE 1149.1 standard. It is provided as a public instruction to allow the user to perform system debug functions. When the ENABLE_ONCE instruction is decoded the TDI and TDO pins are connected directly to the OnCE registers. The particular OnCE register connected between TDI and TDO at a given time is selected by the OnCE controller depending on the OnCE instruction being currently executed. All communication with the OnCE controller is done through the Select-DR-Scan path of the JTAG TAP Controller. See Chapter 10 ON-CHIP EMULATOR (OnCE™) for more information.

11.2.4.8 DEBUG_REQUEST

The DEBUG_REQUEST instruction is not included in the IEEE 1149.1 standard. It is provided as a public instruction to allow the user to generate a debug request signal to the DSP56300 Core. When the DEBUG_REQUEST instruction is decoded the TDI and TDO pins are connected to the Instruction Registers. Due to the fact that in the capture-IR state of the TAP the OnCE status bits are captured in the Instruction shift register, the external JTAG controller must continue to shift-in the DEBUG_REQUEST instruction while polling the status bits that are shifted-out until the Debug Mode of operation is entered (acknowledged by the combination 11 on OS1-OS0). After the acknowledgment of the Debug Mode is received, the external JTAG controller must issue the ENABLE_ONCE instruction to allow the user to perform system debug functions. See Chapter 10 ON-CHIP EMULATOR (OnCE™) for more information.

11.3 DSP56300 RESTRICTIONS

The control afforded by the output enable signals using the boundary scan register and the EXTEST instruction requires a compatible circuit-board test environment to avoid device-destructive configurations. The user must avoid situations in which the DSP56300 Core output drivers are enabled into actively driven networks.

There are two constraints related to the JTAG interface. First, the TCK input does not include an internal pullup resistor and should not be left unconnected to preclude mid-level inputs. The second constraint is to ensure that the JTAG test logic is kept transparent to the system logic by forcing TAP into the test-logic-reset controller state, using either of two methods. During power-up, TRST \sim must be externally asserted to force the TAP controller into this state. After power-up is concluded, TMS must be sampled as a logic one for five consecutive TCK rising edges. If TMS either remains unconnected or is connected to VCC, then the TAP controller cannot leave the test-logic-reset state, regardless of the state of TCK.

The DSP56300 Core features a low-power stop mode, which is invoked using an instruction called STOP. The interaction of the JTAG interface with low-power stop mode is as follows:

1. The TAP controller must be in the test-logic-reset state to either enter or remain in the low-power stop mode. Leaving the TAP controller test-logic-reset state negates the ability to achieve low-power, but does not otherwise affect device functionality.
2. The TCK input is not blocked in low-power stop mode. To consume minimal power, the TCK input should be externally connected to VCC or ground.
3. The TMS and TDI pins include on-chip pullup resistors. In low-power stop mode, these two pins should remain either unconnected or connected to VCC to achieve minimal power consumption.

Since during STOP state all DSP56300 Core clocks are disabled, the JTAG interface provides the means of polling the device status (sampled in the capture-IR state). For an DSP56300 derivative that does not include the \overline{DE} pin, the JTAG interface provides the software means of entering the Debug Mode by executing the DEBUG_REQUEST instruction.

12 OPERATING MODES AND MEMORY SPACES

12.1 CHIP OPERATING MODES

The DSP56300 Core mode pins, MODA, MODB, MODC and MODD determine the reset vector address that should point to the start-up procedure when the chip leaves the reset state. The MODA, MODB, MODC and MODD pins are sampled as the chip leaves the reset state. The sampled state of these pins is subject to a mask programmed look-up table that may be used as a filter to disable the user from entering to some of the operating modes. This filtered state is written to MD,MC,MB and MA bits in the chip Operating Mode Register (OMR). When the reset state is exited, the MODA, MODB, MODC and MODD pins become general-purpose interrupt pins, IRQA, IRQB, IRQC and IRQD, respectively. When not in the RESET state, the OMR mode bits (MA, MB, MC and MD) can be changed by software.

Table 12-1 depicts the mode assignments in the DSP56300 core. The reset vector is chosen from three mask programmed addresses: RESET1, RESET2 and RESET3. Each reset vector is mask programmed to one of two different values, according to Table 12-1.

Table 12-1. DSP56300 Core reset vectors

RESET1 possible values	RESET2 possible values	RESET3 possible values
\$000000	\$004000	\$000000
\$C00000	\$008000	\$FF0000

Table 12-2. DSP56300 Core operating modes

MOD{D:A}	Operating mode	Description	Reset Vector
0000	0	Expanded Mode	RESET1
0001-0111	1-7	System Configuration Mode 1-7	RESET3
1000	8	Expanded Mode	RESET2
1001-1111	9-F	System Configuration Mode 8-14	RESET3

12.1.1 Expanded Modes (Modes 0 and 8)

In the Expanded Modes 0 and 8, a hardware reset causes the DSP56300 Core to jump to the mask programmed external program memory location RESET1 or RESET2 respectively, and execute the code fetched from this location.

12.1.2 System Configuration Modes 1-15 (Mode 1-7 and 9-F)

In the System Configuration Modes 1-15, a hardware reset causes the DSP56300 Core to jump to the mask programmed internal program memory (usually ROM) location RESET3, and execute the code fetched from this location.

12.2 DSP56300 CORE MEMORY MAP

The memory space of the DSP56300 Core is partitioned into program memory space (P), X data memory space and Y data memory space. The data memory space is divided into X data memory and to Y data memory in order to work with the two address arithmetic logic units (ALUs) and to feed two operands simultaneously to the data ALU. Each memory space may include internal RAM, internal ROM and can be expanded off-chip under software control. The three independent memory spaces of the DSP56300 Core: X data, Y data, and program, are shown in Figure 12-1.

Figure 12-1. DSP56300 Core Memory Map

PROGRAM		X DATA		Y DATA	
\$FFFFFF	RESERVED FOR INTERNAL P-MEMORY	\$FFFFFF	INTERNAL X-I/O	\$FFFFFF	INTERNAL Y-I/O
		\$FFFF80	INTERNAL X-I/O OR EXTERNAL X-MEMORY	\$FFFF80	or EXTERNAL Y-I/O
		\$FFF000		\$FFF000	INTERNAL Y-I/O OR EXTERNAL Y-MEMORY
\$FF00C0	192-Words BOOTSTRAP ROM		RESERVED FOR INTERNAL X-MEMORY		RESERVED FOR INTERNAL Y-MEMORY
\$FF0000		\$FF0000	EXTERNAL X-MEMORY	\$FF0000	EXTERNAL Y-MEMORY
	EXTERNAL P-MEMORY				
maximum \$00FFFF	INTERNAL ICACHE 1K or 2K	maximum \$00FFFF	INTERNAL X-MEMORY	maximum \$00FFFF	INTERNAL Y-MEMORY
\$000000	INTERNAL P-MEMORY	\$000000		\$000000	

12.2.1 X Data Memory Space

The X data memory space is divided into five parts:

- **Internal X I/O space.** The on-chip peripheral registers (X I/O) occupy the top 128 locations of the X data memory space (\$FFFF80–\$FFFFFF) and can be accessed by MOVE, MOVEP instructions and by bit oriented

instructions (BCHG, BCLR, BSET, BTST, BRCLR, BRSET, BSCLR, BSSET, JCLR, JSET, JSCLR and JSSET). Some of the DSP56300 Core registers are mapped onto the internal X I/O space, as described in Table 12-3.

Table 12-3. Internal X I/O Space Map

Register	BLOCK	Address	Register Name and Description
IPRC	PIC	\$FFFFFFF	INTERRUPT PRIORITY REGISTER CORE
IPRP		\$FFFFFFE	INTERRUPT PRIORITY REGISTER PERIPHERAL
PCTL	PLL	\$FFFFFFD	PLL CONTROL REGISTER
OGDB	OnCE	\$FFFFFFC	ONCE GDB REGISTER
BCR	PORT A	\$FFFFFFB	BUS CONTROL REGISTER
DCR		\$FFFFFFA	DRAM CONTROL REGISTER
AAR0		\$FFFFFF9	ADDRESS ATTRIBUTE REGISTER 0
AAR1		\$FFFFFF8	ADDRESS ATTRIBUTE REGISTER 1
AAR2		\$FFFFFF7	ADDRESS ATTRIBUTE REGISTER 2
AAR3		\$FFFFFF6	ADDRESS ATTRIBUTE REGISTER 3
IDR		\$FFFFFF5	ID REGISTER
DSTR	DMA	\$FFFFFF4	DMA STATUS REGISTER
DOR0		\$FFFFFF3	DMA OFFSET REGISTER 0
DOR1		\$FFFFFF2	DMA OFFSET REGISTER 1
DOR2		\$FFFFFF1	DMA OFFSET REGISTER 2
DOR3		\$FFFFFF0	DMA OFFSET REGISTER 3
DSR0	DMA Channel 0	\$FFFFFFF	DMA SOURCE ADDRESS REGISTER
DDR0		\$FFFFFFE	DMA DESTINATION ADDRESS REGISTER
DCO0		\$FFFFFFD	DMA COUNTER
DCR0		\$FFFFFFC	DMA CONTROL REGISTER

Register	BLOCK	Address	Register Name and Description
DSR1	DMA Channel 1	\$FFFFEB	DMA SOURCE ADDRESS REGISTER
DDR1		\$FFFFEA	DMA DESTINATION ADDRESS REGISTER
DCO1		\$FFFFE9	DMA COUNTER
DCR1		\$FFFFE8	DMA CONTROL REGISTER
DSR2	DMA Channel 2	\$FFFFE7	DMA SOURCE ADDRESS REGISTER
DDR2		\$FFFFE6	DMA DESTINATION ADDRESS REGISTER
DCO2		\$FFFFE5	DMA COUNTER
DCR2		\$FFFFE4	DMA CONTROL REGISTER
DSR3	DMA Channel 3	\$FFFFE3	DMA SOURCE ADDRESS REGISTER
DDR3		\$FFFFE2	DMA DESTINATION ADDRESS REGISTER
DCO3		\$FFFFE1	DMA COUNTER
DCR3		\$FFFFE0	DMA CONTROL REGISTER
DSR4	DMA Channel 4	\$FFFFDF	DMA SOURCE ADDRESS REGISTER
DDR4		\$FFFFDE	DMA DESTINATION ADDRESS REGISTER
DCO4		\$FFFFDD	DMA COUNTER
DCR4		\$FFFFDC	DMA CONTROL REGISTER
DSR5	DMA Channel 5	\$FFFFDB	DMA SOURCE ADDRESS REGISTER
DDR5		\$FFFFDA	DMA DESTINATION ADDRESS REGISTER
DCO5		\$FFFFD9	DMA COUNTER
DCR5		\$FFFFD8	DMA CONTROL REGISTER
Reserved	On-Chip X-I/O mapped Registers	\$FFFFD7	Reserved for On-Chip X-I/O mapped Register
		..	Reserved for On-Chip X-I/O mapped Register
		..	Reserved for On-Chip X-I/O mapped Register
		..	Reserved for On-Chip X-I/O mapped Register
		\$FFFF80	Reserved for On-Chip X- I/O mapped Register

- **Switchable Internal X I/O or External X-I/O Memory.** The X memory

space located at locations \$FFF000-\$FFFF7F can mask configured to be either external X-memory or internal X-I/O for on-chip memory-mapped peripheral registers.

- **Reserved Space for X ROM or RAM.** The X memory space located at locations \$FF0000-\$FFEFFF is reserved for inclusion of X data ROM or RAM modules, 2048 locations each.
The importance of modular organization of the X ROM/RAM is visible in case of DMA access to the internal X memory simultaneous to core access to the same space. DMA and CORE accesses to different banks can be completed at full speed, while accesses to the same bank halt the DMA until a P memory slot will be available.
- **External X Memory.** The X memory space located at locations \$000000-\$FEFFFF is used for expanding to external X memory. The starting address of the external X memory space is mask programmed.
- **Internal X RAM.** The X memory space located at locations \$000000-\$00FFFF is used for internal X RAM modules, 256 locations each. The last address of the internal X memory is mask programmed and is dependent on the amount of X memory modules in the chip.
The importance of modular organization of the X RAM is visible in case of DMA access to the internal X memory simultaneous to core access to the same space. DMA and CORE accesses to different banks can be completed at full speed, while accesses to the same bank halt the DMA until a P memory slot will be available.

12.2.2 Y Data Memory Space

The Y data memory space is divided into five parts:

- **Internal/External Y-I/O space.** The off-chip or on-chip peripheral registers (Y-I/O) occupy the top 128 locations of the Y data memory space (\$FFFF80-\$FFFFFF) and can be accessed by MOVE, MOVEP instructions and by bit oriented instructions (BCHG, BCLR, BSET, BTST, BRCLR, BRSET, BSCLR, BSSET, JCLR, JSET, JSCLR and JSSET).
This space is partitioned into eight equal parts, 16 locations each. Each part can be mask programmed to be either external Y-I/O or internal Y-I/O.
- **Switchable Internal Y-I/O or External Y-I/O Memory.** The Y memory space located at locations \$FFF000-\$FFFF7F can be mask configured to be either external Y-memory or internal Y-I/O for on-chip memory-mapped peripheral registers.
- **Reserved Space for Y ROM or RAM.** The Y memory space located at

locations \$FF0000-\$FFEEFF is reserved for inclusion of Y data ROM or RAM modules, 2048 locations each.

The importance of modular organization of the Y ROM/RAM is visible in case of DMA access to the internal Y memory simultaneous to core access to the same space. DMA and CORE accesses to different banks can be completed at full speed, while accesses to the same bank halt the DMA until a P memory slot will be available.

- **External Y Memory.** The Y memory space located at locations \$000000-\$FEFFFF is used for expanding to external Y memory. The starting address of the external Y memory space is mask programmed.
- **Internal Y RAM.** The Y memory space located at locations \$000000-\$00FFFF is used for internal Y RAM modules, 256 locations each. The last address of the internal Y memory is mask programmed and is dependent on the amount of Y memory modules in the chip.
The importance of modular organization of the Y RAM is visible in case of DMA access to the internal Y memory simultaneous to core access to the same space. DMA and CORE accesses to different banks can be completed at full speed, while accesses to the same bank halt the DMA until a P memory slot will be available.

12.2.3 Program Memory

The Program memory space is divided into four parts:

- **192-Words Bootstrap ROM.** The P memory space located at locations \$FF0000-\$FF00BF is used for the internal Bootstrap ROM. The ROM contains 192 words combining the bootstrap program for the DSP56300-based derivative. The Bootstrap ROM cannot be accessed by DMA.
- **Reserved Space for Program ROM.** The P memory space located at locations \$FF00C0-\$FFFFFF is reserved for inclusion of Program ROM modules, 2048 locations each. Program ROM may be used to contain some operating-system program or other application-specific pre-defined user programs.
The importance of modular organization of the P ROM is visible in case of DMA access to the internal P memory simultaneous to core access to the same space. DMA and CORE accesses to different banks can be completed at full speed, while accesses to the same bank halt the DMA until a P memory slot will be available.
- **External Program Memory.** The Program memory space located at locations \$000000-\$FEFFFF is used for expanding to external Program memory. The starting address of the external Program memory space is

mask programmed and is dependent on the amount of on-chip Program RAM or ICACHE in the chip.

- **Internal Program RAM.** The Program memory space located at locations \$000000-\$00FFFF is used for internal Program RAM modules, 256 locations each. The last address of the internal Program RAM is masked programmed.

The importance of modular organization of the P RAM is visible in case of DMA access to the internal P memory simultaneous to core access to the same space. DMA and CORE accesses to different banks can be completed at full speed, while accesses to the same bank halt the DMA until a P memory slot will be available.

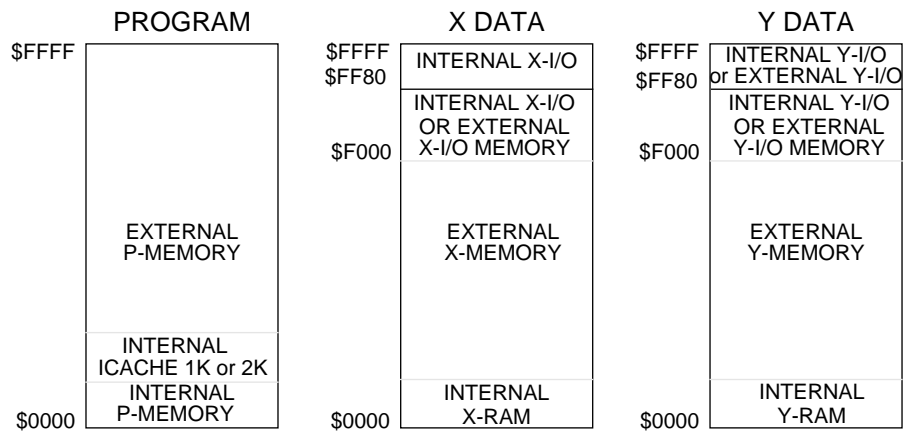
Program RAM provides a method of changing the program dynamically, allowing efficient overlaying of DSP software algorithms.

- **Internal Instruction Cache (ICACHE) RAM.** The Program memory space located at locations \$000000-\$00FFFF is used for internal Instruction Cache RAM modules, 256 locations each. The size of the Icache is mask programmed and can be either 1024 or 2048 words (4 or 8 RAM modules). The starting address of the Icache space is above the internal Program RAM and is also mask programmed. The Icache can be disabled by clearing the CE bit (Cache Enable) in the chip SR. If CE bit is cleared, the Icache ram becomes the high part of the internal program ram. The Instruction Cache is used to minimize the contention with accesses to external program memory space. A complete description of the Instruction Cache is provided in Chapter 5.

12.3 SIXTEEN-BIT COMPATIBILITY MODE

When the SIXTEEN-BIT COMPATIBILITY mode bit (see Figure 6-5 on page 6-10) is set, the memory map is changed to allow easy access to memory mapped I/O, as described in the following figure:

Figure 12-2. DSP56300 Core Memory Map (SC = 1)



For more information about this mode, its effects on the AGU, and restrictions, refer to Section 4.2.

12.4 MEMORY SWITCH MODE

when the MEMORY SWITCH MODE bit (see Figure 6-6 on page 6-17) is set, addresses of internal data memory (X, Y or both) become part of the chip internal program ram. The addresses are in the higher part of the internal ram which resides in the lower part of the data memory, and the amount of addresses is a multiplication of 256 and determined by via programing.

Due to pipelining, a change in the bit takes affect only after the following four instruction cycles. Inserting four NOP instructions after the instruction that changes the value of this bit will guarantee proper operation.

Appendix A INSTRUCTION SET

A-1 INTRODUCTION

The programming model indicates that the DSP56300 Core central processor architecture can be viewed as three functional units operating in parallel: data arithmetic logic unit (ALU), address generation unit (AGU), and program control unit. The goal of the instruction set is to provide the capability to keep each of these units busy each instruction cycle, achieving maximum speed and minimum program size.

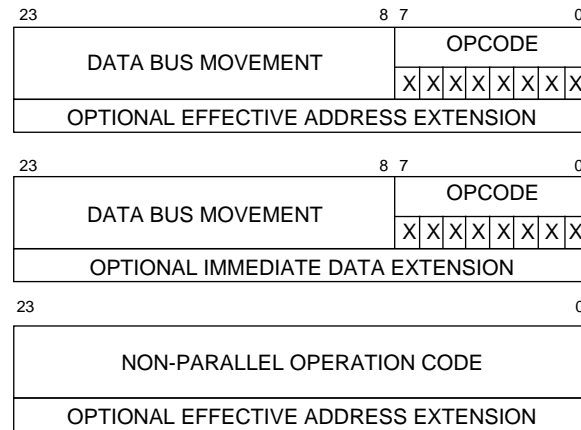
This section introduces the DSP56300 Core instruction set and instruction format. The complete range of instruction capabilities combined with the flexible addressing modes used in this processor provide a very powerful assembly language for implementing digital signal processing (DSP) algorithms. The 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.

A-2 INSTRUCTION FORMATS AND SYNTAX

The DSP56300 Core instructions consist of one or two 24-bit words – an operation word and an optional extension word. This extension word can be either effective address extension word or an immediate data extension word. General formats of the instruction word are shown in Figure A-1. Most instructions specify data movement on the XDB, YDB, and data ALU operations in the same operation word. The DSP56300 Core is designed to perform each of these operations in parallel.

The data bus movement field provides the operand reference type, which selects the type of memory or register reference to be made, the direction of transfer, and the effective address(es) for data movement on the XDB and/or YDB. This field may require additional information to fully specify the operand for certain addressing modes. An extension word following the operation word is used to provide immediate data, absolute address or address displacement, if required. Examples of operations that may include the extension word include move operation such as: `MOVE X:$100,X0`

Figure A-1. General Formats of an Instruction Word



The opcode field of the operation word specifies the data ALU operation or the program control unit operation to be performed.

The operation codes form a very versatile microcontroller unit (MCU) style instruction set, providing highly parallel operations in most programming situations.

The instruction syntax has two formats - Parallel and NonParallel, as shown in Table A-1 and Table A-2. Parallel instruction is organized into five columns: opcode, operands, and two parallel-move fields, each of them is optional, and an optional condition field. The condition field is used to disable the execution of the opcode if the condition is not true, and cannot be used in conjunction with the parallel move fields. Assembly-language source codes for some typical one-word instructions are shown in Table A-1. Because of the multiple bus structure and the parallelism of the DSP56300 Core, 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.). The opcode column indicates the data ALU operation to be performed but may be excluded if only a MOVE operation is needed. The operands column specifies the operands to be used by the opcode. The XDB and YDB columns specify optional data transfers over the XDB and/or YDB and the associated addressing modes. The address space qualifiers (X:, Y:, and L:) indicate which address space is being referenced.

Table A-1. Parallel Instructions Format

	Opcode	Operands	XDB	YDB	Condition
Example 1:	MAC	X0,Y0,A	X:(R0)+,X0	Y:(R4)+,Y0	
Example 2:	MOVE	X:-(R1),X1			
Example 3:	MAC	X1,Y1,B			
Example 4:	MPY	X0,Y0,A			IF eq

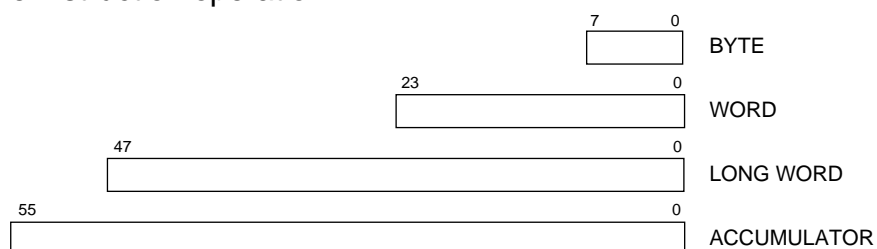
NonParallel instruction is basically organized into two columns: opcode and operands. Assembly-language source codes for some typical one-word instructions are shown in Table A-2. Nonparallel instructions include all the program control, looping and peripherals read/write instructions. They also include some Data ALU instructions that are impossible to be encoded in the opcode field of the Parallel format.

Table A-2. NonParallel Instructions Format

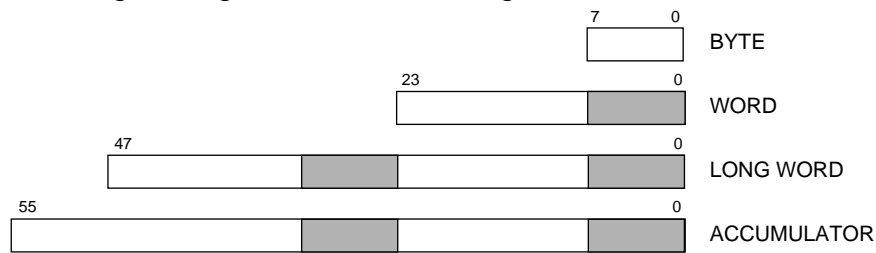
	Opcode	Operands
Example 1:	JEQ	(R5)
Example 2:	MOVEP	#data,X:ipr
Example 3:	RTS	

A-2.1 Operand Sizes

Operand sizes are defined as follows: a byte is 8 bits long, a word is 24 bits long, a long word is 48 bits long, and an accumulator is 56 bits long (see following diagram). The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation.



When in Sixteen Bit Arithmetic mode the operand sizes are as follows: a byte is 8 bits long, a word is 16 bits long, a long word is 32 bits long, and an accumulator is 40 bits long.



A-2.2 Data Organization in Registers

The ten data ALU registers support 8- or 24-bit data operands, and 16-bit data in Sixteen Bit mode. Instructions also support 48- or 56-bit data operands (32- or 40-bit in Sixteen Bit mode) by concatenating groups of specific data ALU registers. The eight address registers in the AGU support 24-bit address or data operands. The eight AGU offset registers support 24-bit offsets or may support 24-bit address or data operands. The eight AGU modifier registers support 24-bit modifiers or may support 24-bit address or data operands. The program counter (PC) supports 24-bit address operands. The status register (SR) and operating mode register (OMR) support 8, 16 or 24-bit data operands. Both the loop counter (LC) and loop address (LA) registers support 24-bit address operands.

A-2.3 Data ALU Registers

The eight main data registers are 24 bits wide. Word operands occupy one register; long-word operands occupy two concatenated registers. The least significant bit (LSB) is the right-most bit (bit 0); whereas, the most significant bit (MSB) is the left-most bit (bit 23 for word operands and bit 47 for long-word operands). When in Sixteen Bit mode, the least significant bit (LSB) is bit 8; bits 24 to 31 are ignored for long-word operands; whereas, the most significant bit (MSB) is the left-most bit.

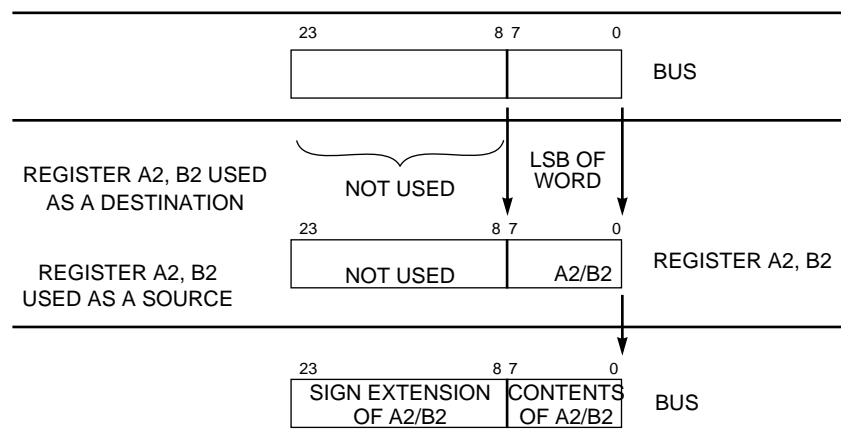
The two accumulator extension registers are eight bits wide. When an accumulator extension register is used as a source operand, it occupies the low-order portion (bits 0–7) of the word; the high-order portion (bits 8–23) is sign extended (see Table A-2). When used as a destination operand, this register 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 for 24 bit mode and bit 8 for Sixteen Bit mode), and the MSB is the left-most bit (bit 55).

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. 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. 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).

Figure A-2. Reading and Writing the ALU Extension Registers



When in Sixteen Bit mode, the move operations associated with Data ALU registers are altered. For further details refer to Section 3.4.1.

A-2.4 AGU Registers

The 24 AGU registers, which are 24 bits wide, may be accessed as word operands for address, address offset, address modifier, and data storage. The notation R_n is used to designate one of the eight address registers, R_0 – R_7 ; the notation N_n is used to designate one of the eight address offset registers, N_0 – N_7 ; and the notation M_n is used to designate one of the eight address modifier registers, M_0 – M_7 .

A-2.5 Program Control Registers

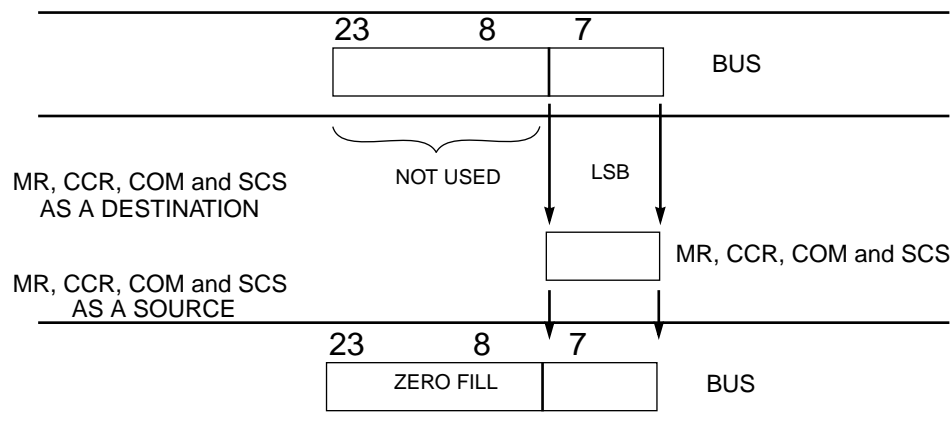
The 24-bit OMR has the chip operating mode register (COM) occupying the low-order eight bits and the extended chip operating mode register (EOM) occupying the middle-order eight bits and the system stack control status register (SCS) occupying the high-order eight bits. The Operating Mode Register (OMR) and the Vector Base Address (VBA) are accessed as word operands; however, not all of their bits are defined. These bits read as zero and should be written with zero for future compatibility. The 24-bit SR has the user condition code register (CCR) occupying the low-order eight bits and the system mode

register (MR) occupying the middle-order eight bits and the extended mode register (EMR) occupying the high-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 A-3). The Loop Counter (LC), Loop Last Address (LA), stack size (SZ), system stack high (SSH), and system stack low (SSL) registers are 24 bits wide and are accessed as word operands. The system stack pointer (SP) is a 24-bit register that is accessed as a word operand. The PC, a special 24-bit-wide program counter register, is generally referenced implicitly as a word operand, but may also be referenced explicitly (by all PC-relative operation codes) also as a word operand.

A-2.6 Data Organization in Memory

The 24-bit program memory can store both 24-bit instruction words and instruction extension words. The 48-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 and byte operands. When in Sixteen Bit Arithmetic mode the X and Y memories can store 16-bit words, that occupy the low-portion of the memory word. 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.

Figure A-3. Reading and Writing Control Registers



A-3 INSTRUCTION GROUPS

The instruction set is divided into the following groups:

- Arithmetic
- Logical
- Bit Manipulation

- Loop
- Move
- Program Control

Each instruction group is described in the following paragraphs.

A-3.1 Arithmetic Instructions

The arithmetic instructions perform all of the arithmetic operations within the data ALU. These instructions may affect all of the CCR bits. Arithmetic instructions are register based (register direct addressing modes used for operands) so that the data ALU operation indicated by the instruction does not use the XDB, the YDB, or the global data bus (GDB). Optional data transfers may be specified with most arithmetic instructions, which allows for parallel data movement over the XDB and YDB or over the GDB during a data ALU operation. This parallel movement allows new data to be prefetched for use in subsequent instructions and allows results calculated in previous instructions to be stored. A ✓ sign in a table cell in the “Parallel Instruction” column indicates that the corresponding instruction is a parallel instruction, while a blank table cell indicates that the instruction is not a parallel instruction. The move operation that can be specified in parallel to the instruction marked is one of the parallel instructions listed in Table A-7. Arithmetic instructions can be executed conditionally, based on the condition codes generated by the previous instructions. Conditional arithmetic instructions don’t allow parallel data movement over the various data busses. Table A-3 lists the arithmetic instructions.

Table A-3. Arithmetic Instructions

Mnemonic	Description	Parallel Instruction
ABS	Absolute Value	✓
ADC	Add Long with Carry	✓
ADD	Add	✓
ADD (imm.)	Add (immediate operand)	
ADDL	Shift Left and Add	✓
ADDR	Shift Right and Add	✓
ASL	Arithmetic Shift Left	✓
ASL (mb.)	Arithmetic Shift Left (multi-bit)	
ASL (mb., imm.)	Arithmetic Shift Left (multi-bit, immediate operand)	
ASR	Arithmetic Shift Right	✓
ASR (mb.)	Arithmetic Shift Right (multi-bit)	

Mnemonic	Description	Parallel Instruction
ASR (mb., imm.)	Arithmetic Shift Right (multi-bit, immediate operand)	
CLR	Clear an Operand	✓
CMP	Compare	✓
CMP (imm.)	Compare (immediate operand)	
CMPM	Compare Magnitude	✓
CMPU	Compare Unsigned	
DEC	Decrement Accumulator	
DIV	Divide Iteration	
DMAC	Double Precision Multiply-Accumulate	
INC	Increment Accumulator	
MAC	Signed Multiply-Accumulate	✓
MAC (su,uu)	Mixed Multiply-Accumulate	
MACI	Signed Multiply-Accumulate (immediate operand)	
MACR	Signed Multiply-Accumulate and Round	✓
MACRI	Signed Multiply-Accumulate and Round (immediate operand)	
MAX	Transfer By Signed Value	✓
MAXM	Transfer By Magnitude	✓
MPY	Signed Multiply	✓
MPY (su,uu)	Mixed Multiply	
MPYI	Signed Multiply (immediate operand)	
MPYR	Signed Multiply and Round	✓
MPYRI	Signed Multiply and Round (immediate operand)	
NEG	Negate Accumulator	✓
NORM	Normalize	
NORMF	Fast Accumulator Normalize	

Mnemonic	Description	Parallel Instruction
RND	Round	✓
SBC	Subtract Long with Carry	✓
SUB	Subtract	✓
SUB (imm.)	Subtract (immediate operand)	
SUBL	Shift Left and Subtract	✓
SUBR	Shift Right and Subtract	✓
Tcc	Transfer Conditionally	
TFR	Transfer Data ALU Register	✓
TST	Test an Operand	✓

A-3.2 Logical Instructions

The logical instructions, which execute in one instruction cycle, perform all of the logical operations within the data ALU (except ANDI and ORI). They may affect all of the CCR bits and, like the arithmetic instructions, are register based. Optional data transfers may be specified with most logical instructions, allowing parallel data movement over the XDB and YDB or over the GDB during a data ALU operation. This parallel movement allows new data to be prefetched for use in subsequent instructions and allows results calculated in previous instructions to be stored. A ✓ sign in a table cell in the “Parallel Instruction” column indicates that the corresponding instruction is a parallel instruction, while a blank table cell indicates that the instruction is not a parallel instruction. The move operation that can be specified in parallel to the instruction marked is one of the parallel instructions listed in Table A-7. Table A-4 lists the logical instructions.

Table A-4. Logical Instructions

Mnemonic	Description	Parallel Instruction
AND	Logical AND	✓
AND (imm.)	Logical AND (immediate operand)	
ANDI	AND Immediate to Control Register	
CLB	Count Leading Bits	
EOR	Logical Exclusive OR	✓
EOR (imm.)	Logical Exclusive OR (immediate operand)	

Mnemonic	Description	Parallel Instruction
EXTRACT	Extract Bit Field	
EXTRACT (imm.)	Extract Bit Field (immediate operand)	
EXTRACTU	Extract Unsigned Bit Field	
EXTRACTU (imm.)	Extract Unsigned Bit Field (immediate operand)	
INSERT	INSERT Bit Field	
INSERT (imm.)	INSERT Bit Field (immediate operand)	
LSL	Logical Shift Left	✓
LSL (mb.)	Logical Shift Left (multi-bit)	
LSL (mb., imm.)	Logical Shift Left (multi-bit, immediate operand)	
LSR	Logical Shift Right	✓
LSR (mb.)	Logical Shift Right (multi-bit)	
LSR (mb.,imm.)	Logical Shift Right (multi-bit, immediate operand)	
MERGE	Merge Two Half Words	
NOT	Logical Complement	✓
OR	Logical Inclusive OR	✓
OR (imm.)	Logical Inclusive OR (immediate operand)	
ORI	OR Immediate to Control Register	
ROL	Rotate Left	✓
ROR	Rotate Right	✓

A-3.3 Bit Manipulation Instructions

The bit manipulation instructions test the state of any single bit in a memory location and then optionally set, clear, or invert the bit. The carry bit of the CCR will contain the result of the bit test. Table A-5 lists the bit manipulation instructions.

Table A-5. Bit Manipulation Instructions

Mnemonic	Description	Parallel Instruction
BCHG	Bit Test and Change	
BCLR	Bit Test and Clear	
BSET	Bit Test and Set	
BTST	Bit Test	

A-3.4 Loop Instructions

The hardware DO loop executes with no overhead cycles – 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. Table A-6 lists the loop instructions.

Table A-6. Loop Instructions

Mnemonic	Description	Parallel Instruction
BRKcc	Conditionally Break the current Hardware Loop	
DO	Start Hardware Loop	
DOR	Start Hardware Loop to PC-Related End-Of-Loop Location	
DO FOREVER	Start Forever Hardware Loop	
DOR FOREVER	Start Forever Hardware Loop to PC-Related End-Of-Loop Location	
ENDDO	Abort and Exit from Hardware Loop	

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

A-3.5 Move Instructions

The move instructions perform data movement over the XDB and YDB or over the GDB.

Move instructions, most of which allow Data ALU opcode in parallel, do not affect the CCR except the limit bit L if limiting is performed when reading a data ALU accumulator register. Table A-7 lists the move instructions.

Table A-7. Move Instructions

Mnemonic	Description	Parallel Instruction
LUA	Load Updated Address	
LRA	Load PC-Relative Address	
MOVE	Move Data Register	✓
MOVEC	Move Control Register	
MOVEM	Move Program Memory	
MOVEP	Move Peripheral Data	
U MOVE	Update Move	✓

A-3.6 Program Control Instructions

The program control instructions include jumps, conditional jumps, and other instructions affecting the PC, SS and the program Cache. Program control instructions may affect the CCR bits as specified in the instruction. Optional data transfers over the XDB and YDB may be specified in some of the program control instructions. Table A-8 lists the program control instructions.

Table A-8. Program Control Instructions

Mnemonic	Description	Parallel Instruction
IFcc.U	Execute Conditionally and Update CCR	
IFcc	Execute Conditionally	
Bcc	Branch Conditionally	
BRA	Branch Always	
BRCLR	Branch if Bit Clear	
BRSET	Branch if Bit Set	
BSc	Branch to Subroutine Conditionally	
BSR	Branch to Subroutine Always	
BSCLR	Branch to Subroutine if Bit Clear	
BSSET	Branch to Subroutine if Bit Set	

Mnemonic	Description	Parallel Instruction
DEBUGcc	Enter into the Debug Mode Conditionally	
DEBUG	Enter into the Debug Mode Always	
Jcc	Jump Conditionally	
JMP	Jump Always	
JCLR	Jump if Bit Clear	
JSET	Jump if Bit Set	
JScC	Jump to Subroutine Conditionally	
JSR	Jump to Subroutine Always	
JSCLR	Jump to Subroutine if Bit Clear	
JSSET	Jump to Subroutine if Bit Set	
NOP	No Operation	
PLOCK	Lock Program Cache Sector	
PUNLOCK	Unlock Program Cache Sector	
PLOCKR	Lock PC-Related Program Cache Sector	
PUNLOCKR	Unlock PC-Related Program Cache Sector	
PFREE	Unlock all Program Cache Locked Sectors	
PFLUSH	Reset Program Cache State	
PFLUSHUN	Reset Program Cache State to all Unlocked Sectors	
REP	Repeat Next Instruction	
RESET	Reset On-Chip Peripheral Devices	
RTI	Return from Interrupt	
RTS	Return from Subroutine	
STOP	Stop Processing (Low-Power Standby)	
TRAPcc	Trap Conditionally	
TRAP	Trap Always	
WAIT	Wait for Interrupt (Low-Power Standby)	

A-4 INSTRUCTION GUIDE

The following information is included in each instruction description:

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. **Condition Codes:** The status register is depicted with the condition code bits which can be affected by the instruction. Not all bits in the status register are used. Those which are reserved are indicated with a gray box covering its area.
5. **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.

A-4.1 NOTATION

Each instruction description contains symbols used to abbreviate certain operands and operations. Table A-9 lists the symbols used and their respective meanings. Depending on the context, registers refer to either the register itself or the contents of the register.

Table A-9. Instruction Description Notation

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)

Data ALU Registers Operands

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)

Program Control Unit Registers Operands

PC	Program Counter Register (24 Bits)
EMR	Extended Mode Register (8 Bits)
MR	Mode Register (8 Bits)
CCR	Condition Code Register (8 Bits)
SR	Status Register = EMR:MR:CCR (24 Bits)
SCS	System Stack Control Status Register (8 Bits)
EOM	Extended Chip Operating Mode Register (8 Bits)
COM	Chip Operating Mode Register (8 Bits)
OMR	Operating Mode Register = SCS:EOM:COM (24 Bits)
SZ	System Stack Size Register (24 Bits)
SC	System Stack Counter Register (5 Bits)
VBA	Vector Base Address (24 Bits, 8 of them are always zero)
LA	Hardware Loop Address Register (24 Bits)
LC	Hardware Loop Counter Register (24 Bits)
SP	System Stack Pointer Register (24 Bits)
SSH	Upper Portion of the Current Top of the Stack (24 Bits)
SSL	Lower Portion of the Current Top of the Stack (24 Bits)
SS	System Stack RAM = SSH: SSL (16 Locations by 32 Bits)

Address Operands

ea	Effective Address
eax	Effective Address for X Bus
eay	Effective Address for Y Bus
xxxx	Absolute or Long Displacement Address (24 Bits)
xxx	Short or Short Displacement Jump Address (12 Bits)
xxx	Short Displacement Jump Address (9 Bits)
aaa	Short Displacement Address (7 Bits Sign Extended)
aa	Absolute Short Address (6 Bits, Zero Extended)
pp	High I/O Short Address (6 Bits, Ones Extended)
qq	Low I/O Short Address (6 Bits)
<...>	Specifies the Contents of the Specified Address
X:	X Memory Reference
Y:	Y Memory Reference
L:	Long Memory Reference = X Concatenated with Y
P:	Program Memory Reference

Miscellaneous Operands

S, Sn	Source Operand Register
D, Dn	Destination Operand Register
D [n]	Bit n of D Destination Operand Register
#n	Immediate Short Data (5 Bits)
#xx	Immediate Short Data (8 Bits)
#xxx	Immediate Short Data (12 Bits)
#xxxxxx	Immediate Data (24 Bits)
r	Rounding Constant
#bbbbbb	Operand Bit Select (5 Bits)

Unary Operands

-	Negation Operator
—	Logical NOT Operator (Overbar)
PUSH	Push Specified Value onto the System Stack (SS) Operator

Unary Operands

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 Operands

+	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

Addressing Mode Operators

<<	I/O Short Addressing Mode Force Operator
<	Short Addressing Mode Force Operator
>	Long Addressing Mode Force Operator
#	Immediate Addressing Mode Operator
#>	Immediate Long Addressing Mode Force Operator
#<	Immediate Short Addressing Mode Force Operator

Mode Register Symbols

LF	Loop Flag Bit Indicating When a DO Loop is in Progress
DM	Double Precision Multiply Bit Indicating if the Chip is in Double Precision Multiply Mode
SB	Sixteen Bit Arithmetic Mode
RM	Rounding Mode

Mode Register Symbols

S1, S0	Scaling Mode Bits Indicating the Current Scaling Mode
I1, I0	Interrupt Mask Bits Indicating the Current Interrupt Priority Level

Condition Code Register (CCR) Symbols

S	Block Floating Point Scaling Bit Indicating Data Growth Detection
L	Limit Bit Indicating Arithmetic Overflow and/or Data Shifting/Limiting
E	Extension Bit Indicating if the Integer Portion of Data ALU result is in Use
U	Unnormalized Bit Indicating if the Data ALU Result is Unnormalized
N	Negative Bit Indicating if Bit 55 of the Data ALU Result is Set
Z	Zero Bit Indicating if the Data ALU Result Equals Zero
V	Overflow Bit Indicating if Arithmetic Overflow has Occurred in Data ALU
C	Carry Bit Indicating if a Carry or Borrow Occurred in Data ALU Result

()	Optional Letter, Operand, or Operation
(...)	Any Arithmetic or Logical Instruction Which Allows Parallel Moves
EXT	Extension Register Portion of an Accumulator (A2 or B2)
LS	Least Significant
LSP	Least Significant Portion of an Accumulator (A0 or B0)
MS	Most Significant
MSP	Most Significant Portion of a n Accumulator (A1 or B1)
S/L	Shifting and/or Limiting on a Data ALU Register
Sign Ext	Sign Extension of a Data ALU Register
Zero	Zeroing of a Data ALU Register

Address ALU Registers Operands

Rn	Address Registers R0 - R7 (24 Bits)
Nn	Address Offset Registers N0 - N7 (24 Bits)
Mn	Address Modifier Registers M0 - M7 (24 Bits)

A-5 CONDITION CODE COMPUTATION

The condition code register (CCR) portion of the status register (SR) consists of eight defined bits.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
CCR							

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 “sticky”** 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 **“sticky” 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 “sticky” bit used in block floating point operations to indicate the need to scale the number in A or B.

The full description of every instruction contains an illustration showing how the instruction affects the various condition codes.

An instruction can affect a condition code according to three different rules:

standard mark	The affect on the condition code
x	Unchanged by the instruction
✓	Changed by the instruction, according to the standard definition of the condition code
●	Changed by the instruction, according to a special definition of the condition code, depicted as part of the instruction full description

The standard definition of the condition code bits follows.

S (Scaling Bit) This bit is computed, according to the logical equations in the table below, when an instruction or a parallel move reads the contents of accumulator A or B to XDB or YDB. It is a “sticky” bit, cleared only by an instruction that specifically clears it, or hardware reset.

S0	S1	Scaling Mode	S bit equation
0	0	No scaling	$S = (A46 \text{ XOR } A45) \text{ OR } (B46 \text{ XOR } B45) \text{ OR } S \text{ (previous)}$
0	1	Scale down	$S = (A47 \text{ XOR } A46) \text{ OR } (B47 \text{ XOR } B46) \text{ OR } S \text{ (previous)}$
1	0	Scale up	$S = (A45 \text{ XOR } A44) \text{ OR } (B45 \text{ XOR } B44) \text{ OR } S \text{ (previous)}$
1	1	Reserved	S undefined

The scaling bit (S) is used to detect data growth, which is required in Block Floating Point FFT operation. The scaling bit will be set if the absolute value in the accumulator, before scaling, was greater or equal to 0.25 and smaller than 0.75. Typically, the bit is tested after each pass of a radix 2 decimation-in-time FFT and, if it is set, the appropriate scaling mode should be activated in the next pass. The Block Floating Point FFT algorithm is described in the Motorola application note APR4/D, “Implementation of Fast Fourier Transforms on Motorola’s DSP56000/DSP56001 and DSP96002 Digital Signal Processors.”

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 while reading the contents of accumulator A or B to XDB or YDB. In Arithmetic Saturation Mode, the limit bit is also set when an arithmetic saturation occurs in the Data ALU result. Not affected otherwise. This bit is “sticky” and must be cleared only by an instruction that specifically clears it, or hardware reset.

E (Extension Bit) Cleared if all the bits of the **signed integer portion** of the Data ALU

result are the **same** – i.e., the bit patterns are either 00. . . 00 or 11. . . 11. Set otherwise.

The signed integer portion is defined by the scaling mode as shown in the following table:

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

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 Data ALU 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	$U = \overline{(\text{Bit } 47 \text{ xor Bit } 46)}$
0	1	Scale Down	$U = \overline{(\text{Bit } 48 \text{ xor Bit } 47)}$
1	0	Scale Up	$U = \overline{(\text{Bit } 46 \text{ xor Bit } 45)}$

The result of calculating the U bit in this fashion is that the definition of positive normalized number, p, is $0.5 \leq p < 1.0$ and the definition of negative normalized number, n, is $-1.0 \leq n < -0.5$.

N (Negative Bit) Set if the MS bit (bit 55 in arithmetic instructions or bit 47 in logical instructions) of the Data ALU result is set. Cleared otherwise.

Z (Zero Bit) Set if the Data ALU result equals zero. Cleared otherwise.

V (Overflow Bit) Set if an arithmetic overflow occurs in the 56-bit Data ALU result (40-bit result in Sixteen Bit mode). Cleared otherwise. This indicates that the result cannot be represented in the 56-bit (40-bit) accumulator; thus, the accumulator has overflowed.
In Arithmetic Saturation Mode, an arithmetic overflow occurs if the Data ALU result is not representable in the accumulator without the extension part, i.e. 48-bit accumulator (32-bit in Sixteen Bit Mode).

C (Carry Bit) Set if a carry is generated out of the MS bit of the Data ALU result of an addition or if a borrow is generated out of the MS bit of the Data ALU result of a subtraction. Cleared otherwise. The carry or borrow is generated out of bit 55 of the Data ALU result. The carry bit is also affected by bit manipulation, rotate, shift and compare instructions. The carry bit is not affected by the Arithmetic Saturation Mode.

A-6 INSTRUCTIONS DESCRIPTIONS

The following section describes each instruction in the DSP56300 Core instruction set in complete detail. Instructions which allow parallel moves include the notation “(parallel move)” in both the **Assembler Syntax** and the **Operation** fields. The MOVE instruction is equivalent to a NOP with parallel moves. Therefore, a detailed description of each parallel move is given with the MOVE instruction details.

Whenever an instruction uses an accumulator as both a destination operand for data ALU operation and as a source for a parallel move operation, the parallel move operation will use the value in the accumulator prior to execution of any data ALU operation.

ABS

ABS

Absolute Value

Operation:

| D | → D (parallel move)

Assembler Syntax:

ABS D (parallel move)

Description: Take the absolute value of the destination operand D and store the result in the destination accumulator.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	×
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
ABS D	DATA BUS MOVE FIELD				0 0 1 0	d 1 1 0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

A-6.2 Add Long with Carry (ADC)

ADC

ADC

Add Long with Carry

Operation:

$S+C+D \rightarrow D$ (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).

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	✓
CCR							

- ✓ This bit is changed according to the standard definition
× This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
ADC S,D	DATA BUS MOVE FIELD				0 0 1 J	d 0 0 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

- {S} J Source register [X,Y] (see Table A-11 on page A-239)
{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

ADD

ADD

Add

Operation:

S+D→D (parallel move)

#xx+D→D

#xxxxxx+D→D

Assembler Syntax:

ADD S,D (parallel move)

ADD #xx,D

ADD #xxxxxx,D

Description: Add the source operand S to the destination operand D and store the result in the destination accumulator. The source can be a register (word - 24 bits, long word - 48 bits or accumulator - 56 bits), short immediate (6 bits) or long immediate (24 bits).

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the 6 bits will be right aligned and the remaining bits will be zeroed to form a 24-bit source operand.

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, B10 or immediate operand are used as source operands and A2 and B2 are not replicas of bit 47.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	✓
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
ADD S,D	DATA BUS MOVE FIELD				0	J J J
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

	23	16 15								8 7								0							
ADD #xx,D	0	0	0	0	0	0	0	1	0	1	i	i	i	i	i	i	1	0	0	0	d	0	0	0	0

	23	16 15	8 7	0
ADD #xxxxxx,D	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 0	1 1 0 0 d 0 0 0	
	IMMEDIATE DATA EXTENSION			

Instruction Fields:

{S}	JJJ	Source register [B/A,X,Y,X0,Y0,X1,Y1] (see Table A-14 on page A-240)
{D}	d	Destination accumulator [A/B] (see Table A-10 on page A-239)
{#xx}	iiii	6-bit Immediate Short Data
{#xxxxxx}		24-bit Immediate Long Data extension word

ADDL

ADDL

Shift Left and Add Accumulators

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	●	✓
CCR							

- V Set if overflow has occurred in A or B result or the MS bit of the destination operand is changed as a result of the instruction's left shift
- ✓ This bit is changed according to the standard definition
- × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
ADDL S,D	DATA BUS MOVE FIELD				0 0 0 1	d 0 1 0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

- {D} **d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- {S} The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B

A-6.5 Shift Right and Add Accumulators (ADDR)

ADDR

ADDR

Shift Right and Add Accumulators

Operation:

$S+D / 2 \rightarrow D$ (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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	✓
CCR							

- ✓ This bit is changed according to the standard definition
× This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
ADDR S,D	DATA BUS MOVE FIELD				0 0 0 0	d 0 1 0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

- {D} **d** Destination accumulator [A,B] (see Table A-10 on page A-239)
{S} The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B

AND

AND

Logical AND

Operation:

S • D[47:24] → D[47:24] (parallel move)

#xx • D[47:24] → D[47:24]

#xxxxxx • D[47:24] → D[47:24]

where • denotes the logical AND operator

Assembler Syntax:

AND S,D (parallel move)

AND #xx,D

AND #xxxxxx,D

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. The source can be a 24-bit register, 6-bit short immediate or 24-bit long immediate. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the 6 bits will be right aligned and the remaining bits will be zeroed to form a 24-bit source operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	x	x	x	●	●	●	x
CCR							

- N Set if bit 47 of the result is set
- Z Set if bits 47-24 of the result are zero
- V Always cleared
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0						
AND S,D	DATA BUS MOVE FIELD				0	1	J	J	d	1	1	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

	23	16 15								8 7								0						
AND #xx,D	0	0	0	0	0	0	0	1	0	1	i	i	i	i	i	i	1	0	0	0	d	1	1	0

	23	16	15	8	7	0												
AND #xxxxxx,D	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0
	IMMEDIATE DATA EXTENSION																	

Instruction Fields:

{S}	JJ	Source input register [X0,X1,Y0,Y1] (see Table A-12 on page A-239)
{D}	d	Destination accumulator [A/B] (see Table A-10 on page A-239)
{#xx}	iiii	6-bit Immediate Short Data
{#xxxxxx}		24-bit Immediate Long Data extension word

ANDI

ANDI

AND Immediate with Control Register

Operation:

#xx • D → D

where • denotes the logical AND operator

Assembler Syntax:

AND(I) #xx,D

Description: Logically AND the 8-bit immediate operand (#xx) with the contents of the destination control register D and store the result in the destination control register. The condition codes are affected only when the condition code register (CCR) is specified as the destination operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
•	•	•	•	•	•	•	•
CCR							

For CCR Operand:

- S Cleared if bit 7 of the immediate operand is cleared
- L Cleared if bit 6 of the immediate operand is cleared
- E Cleared if bit 5 of the immediate operand is cleared
- U Cleared if bit 4 of the immediate operand is cleared
- N Cleared if bit 3 of the immediate operand is cleared
- Z Cleared if bit 2 of the immediate operand is cleared
- V Cleared if bit 1 of the immediate operand is cleared
- C Cleared if bit 0 of the immediate operand is cleared

For MR and OMR Operands: The condition codes are not affected using these operands.

Instruction Formats and opcodes:

	23	16 15								8 7								0					
AND(I) #xx,D	0	0	0	0	0	0	0	0	i	i	i	i	i	i	i	1	0	1	1	1	0	E	E

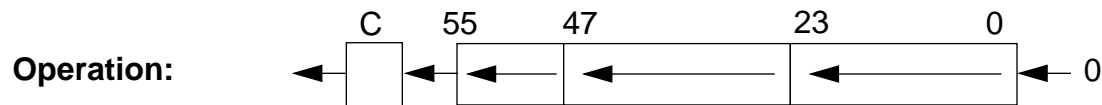
Instruction fields:

{D}	EE	Program Controller register [MR,CCR,COM,EOM] (see Table A-13 on page A-239)
{#xx}	iiiiiii	Immediate Short Data

ASL

ASL

Arithmetic Shift Accumulator Left



Assembler Syntax:

ASL D (parallel move)

ASL #ii,S2,D

ASL S1,S2,D

Description:

Single bit shift:

Arithmetically shift the destination accumulator D one bit to the left and store the result in the destination accumulator. The 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.

Multi-bit shift:

The contents of the source accumulator S2 are shifted left #ii bits. Bits shifted out of position 55 are lost, but for the last bit which is latched in the carry bit C. Zeros are supplied to the vacated positions on the right. The result is placed into destination accumulator D. The number of bits to shift is determined by the 6-bit immediate field in the instruction, or by the 6-bit unsigned integer located in the 6 LSBs of the control register S1. 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.

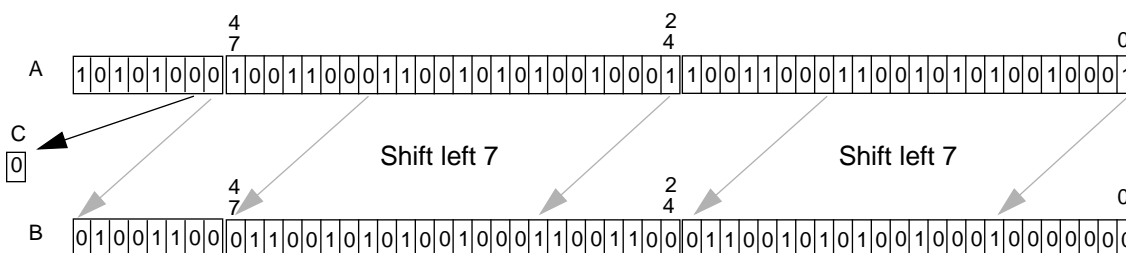
This is a 56 bit operation.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	●	●
CCR							

- V Set if bit 55 is changed any time during the shift operation. Cleared otherwise.
- C Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero.
- × This bit is unchanged by the instruction
- ✓ This bit is changed according to the standard definition

Example: ASL #7,A, B



Instruction Formats and opcodes:

		23													8	7					0														
ASL	D		DATA BUS MOVE FIELD												0 0 1 1 d 0 1 0																				
			OPTIONAL EFFECTIVE ADDRESS EXTENSION																																
		23													16	15													8	7					0
ASL	#ii,S2,D		0 0 0 0 1 1 0 0								0 0 0 1 1 1 0 1								S i i i i i i D																
		23													16	15													8	7					0
ASL	S1,S2,D		0 0 0 0 1 1 0 0								0 0 0 1 1 1 1 0								0 1 0 S s s s D																

Instruction Fields:

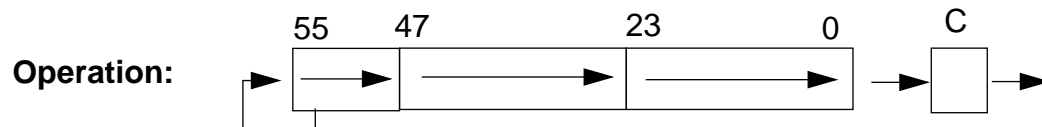
{S2}	S	Source accumulator [A,B] (see Table A-10 on page A-239)
{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S1}	sss	Control register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)
{#ii}	iiiiii	6 bit unsigned integer [0-55] denoting the shift amount

In the **control register** S1: bits 5-0 (LSB) are used as #ii field, and the rest of the register is ignored.

ASR

ASR

Arithmetic Shift Accumulator Right



Assembler Syntax:

ASR D (parallel move)

ASR #ii,S2,D

ASR S1,S2,D

Description:

Single bit shift:

Arithmetically shift the destination operand D one bit to the right and store the result in the destination accumulator. The LS bit of D prior to instruction execution is shifted into the carry bit C, and the MS bit of D is held constant.

Multi-bit shift:

The contents of the source accumulator S2 are shifted right #ii bits. Bits shifted out of position 0 are lost, but for the last bit which is latched in the carry bit. Copies of the MSB are supplied to the vacated positions on the left. The result is placed into destination accumulator D. The number of bits to shift is determined by the 6-bit immediate field in the instruction, or by the 6-bit unsigned integer located in the 6 LSBs of the control register S1. If a zero shift count is specified, the carry bit is cleared.

This is a 56- or 40-bit operation, depending on SA bit value in status register.

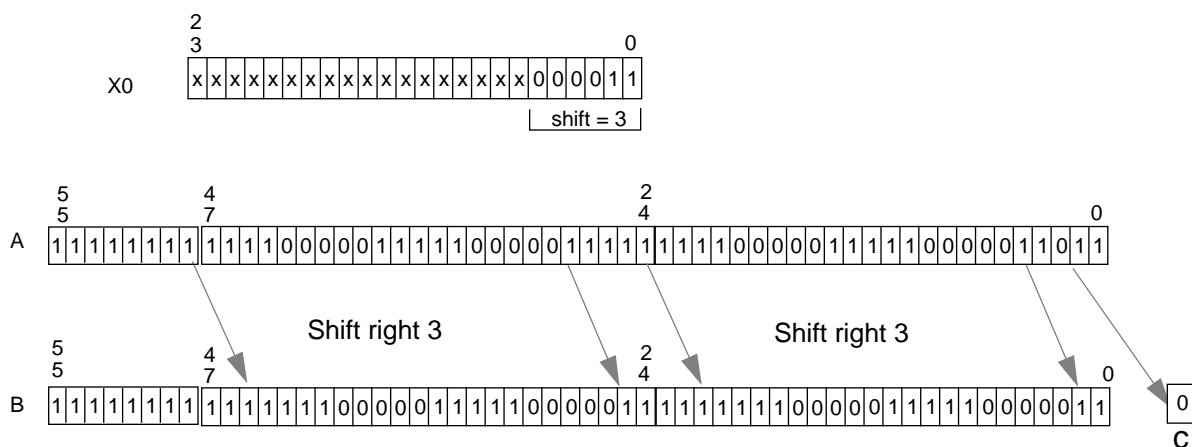
Note: if the number of shifts indicated by the 6 LSBs of the control register or by the immediate field, exceeds the value of 56 (40 in sixteen bit arithmetic mode), then the result would be undefined.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	●	●
CCR							

- V Always cleared
- C Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero
- × This bit is unchanged by the instruction
- ✓ This bit is changed according to the standard definition

Example: ASR X0,A,B



Instruction Formats and opcodes:

ASR	D	23	8	7	0		
		DATA BUS MOVE FIELD					
		0 0 1 0 d 0 1 0					
		OPTIONAL EFFECTIVE ADDRESS EXTENSION					
ASR	#ii,S2,D	23	16	15	8	7	0
		0 0 0 0 1 1 0 0				S i i i i i i D	
ASR	S1,S2,D	23	16	15	8	7	0
		0 0 0 0 1 1 0 0				0 1 1 S s s s D	

Instruction Fields:

{S2}	S	Source accumulator [A,B] (see Table A-10 on page A-239)
{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S1}	sss	Control register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)
{#ii}	iiiiii	6 bit unsigned integer [0-55] denoting the shift amount

In the **control register** S1: bits 5-0 (LSB) are used as #ii field, and the rest of the register is ignored.

Bcc**Bcc****Branch Conditionally****Operation:**

If cc, then PC+xxxx → PC
 else PC+1 → PC

If cc, then PC+xxx → PC
 else PC+1 → PC

If cc, then PC+Rn → PC
 else PC+1 → PC

Assembler Syntax:

Bcc xxxx

Bcc xxx

Bcc Rn

Description: If the specified condition is true, program execution continues at location PC+displacement. If the specified condition is false, the PC is incremented and program execution continues sequentially. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 9-bit data is sign extended to form the PC relative displacement.

The conditions that the term “**cc**” can specify are listed on Table A-42 on page A-250.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23	16	15	8	7	0																		
Bcc	xxxx	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	C	C	C	C
		PC RELATIVE DISPLACEMENT																							
		23	16	15	8	7	0																		
Bcc	xxx	0	0	0	0	0	1	0	1	C	C	C	C	0	1	a	a	a	a	0	a	a	a	a	a
		23	16	15	8	7	0																		
Bcc	Rn	0	0	0	0	1	1	0	1	0	0	0	1	1	R	R	R	0	1	0	0	C	C	C	C

Instruction Fields:

{cc}	CCCC	Condition code (see Table A-43 on page A-251)
{xxxx}		24-bit PC Relative Long Displacement
{xxx}	aaaaaaaa	Signed PC Relative Short Displacement
{Rn}	RRR	Address register [R0-R7]

BCHG

BCHG

Bit Test and Change

Operation: $D[n] \rightarrow C$ $\overline{D[n]} \rightarrow D[n]$ $D[n] \rightarrow C$ $\overline{D[n]} \rightarrow D[n]$ $D[n] \rightarrow C$ $\overline{D[n]} \rightarrow D[n]$ $D[n] \rightarrow C$ $\overline{D[n]} \rightarrow D[n]$ $D[n] \rightarrow C$ $\overline{D[n]} \rightarrow D[n]$ **Assembler Syntax:**

BCHG #n,[XorY]:ea

BCHG #n,[XorY]:aa

BCHG #n,[XorY]:pp

BCHG #n,[XorY]:qq

BCHG #n,D

Description: Test the n^{th} bit of the destination operand D, complement it, and store the result in the destination location. The state of the n^{th} bit is stored in the carry bit C of the condition code register. The bit to be tested is selected by an immediate bit number from 0–23. This instruction performs a read-modify-write operation on the destination location using two destination accesses before releasing the bus. This instruction provides a test-and-change capability which is useful for synchronizing multiple processors using a shared memory. This instruction can use all memory alterable addressing modes.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	●	●	●	●	●	●
CCR							

For destination operand SR:

- C Complemented if bit 0 is specified. Not affected otherwise.
- V Complemented if bit 1 is specified. Not affected otherwise.
- Z Complemented if bit 2 is specified. Not affected otherwise.
- N Complemented if bit 3 is specified. Not affected otherwise.
- U Complemented if bit 4 is specified. Not affected otherwise.
- E Complemented if bit 5 is specified. Not affected otherwise.
- L Complemented if bit 6 is specified. Not affected otherwise.
- S Complemented if bit 7 is specified. Not affected otherwise.

For other destination operands:

- C Set if bit tested is set. Cleared otherwise.
- V Not affected.
- Z Not affected.
- N Not affected.
- U Not affected.
- E Not affected.
- L According to the standard definition.
- S According to the standard definition.

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
- RM Changed if bit 12 is specified. Not affected otherwise
- SB 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: MR status bits are not affected.

Instruction Formats and opcodes:

	23	16 15	8 7	0
BCHG #n,[X or Y]:ea	0 0 0 0 1 0 1 1	0 1 M M M R R R	0 S 0 b b b b b	
	OPTIONAL EFFECTIVE ADDRESS EXTENSION			
	23	16 15	8 7	0
BCHG #n,[X or Y]:aa	0 0 0 0 1 0 1 1	0 0 a a a a a a	0 S 0 b b b b b	
	23	16 15	8 7	0
BCHG #n,[X or Y]:pp	0 0 0 0 1 0 1 1	1 0 p p p p p p	0 S 0 b b b b b	
	23	16 15	8 7	0
BCHG #n,[X or Y]:qq	0 0 0 0 0 0 0 1	0 1 q q q q q q	0 S 0 b b b b b	
	23	16 15	8 7	0
BCHG #n,D	0 0 0 0 1 0 1 1	1 1 D D D D D D	0 1 0 b b b b b	

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMMRRR	Effective Address (see Table A-16 on page A-241)
{X /Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFF80-\$FFFFBF]
{D}	DDDDDD	Destination register [all on-chip registers] (see Table A-22 on page A-243)

BCLR

BCLR

Bit Test and Clear

Operation:

D[n] → C 0 → D[n]

D[n] → C 0 → D[n]

D[n] → C 0 → D[n]

D[n] → C 0 → D[n]

D[n] → C 0 → D[n]

Assembler Syntax:

BCLR #n,[XorY]:ea

BCLR #n,[XorY]:aa

BCLR #n,[XorY]:pp

BCLR #n,[XorY]:qq

BCLR #n,D

Description: Test the n^{th} bit of the destination operand D, clear it and store the result in the destination location. The state of the n^{th} bit is stored in the carry bit C of the condition code register. The bit to be tested is selected by an immediate bit number from 0–23. This instruction performs a read-modify-write operation on the destination location using two destination accesses before releasing the bus. This instruction provides a test-and-clear capability which is useful for synchronizing multiple processors using a shared memory. This instruction can use all memory alterable addressing modes.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
•	•	•	•	•	•	•	•
CCR							

CCR Condition Codes:

For destination operand SR:

- C 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 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 According to the standard definition.
- S According to the standard definition.

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
- RM Changed if bit 12 is specified. Not affected otherwise
- SB 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

Instruction Formats and opcodes:

	23	16 15								8 7								0						
BCLR #n,[X or Y]:ea	0	0	0	0	1	0	1	0	0	1	M	M	M	R	R	R	0	S	0	b	b	b	b	b
	OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

	23	16 15							8 7							0								
BCLR #n,[X or Y]:aa	0	0	0	0	1	0	1	0	0	0	a	a	a	a	a	a	0	S	0	b	b	b	b	b

	23	16 15							8 7							0								
BCLR #n,[X or Y]:pp	0	0	0	0	1	0	1	0	1	0	p	p	p	p	p	p	0	S	0	b	b	b	b	b

	23	16 15							8 7							0								
BCLR #n,[X or Y]:qq	0	0	0	0	0	0	0	1	0	0	q	q	q	q	q	q	0	S	0	b	b	b	b	b

	23	16 15								8 7								0						
BCLR #n,D	0	0	0	0	1	0	1	0	1	1	D	D	D	D	D	D	0	1	0	b	b	b	b	b

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMMRRR	Effective Address (see Table A-16 on page A-241)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{D}	DDDDDD	Destination register [all on-chip registers] (see Table A-22 on page A-243)

BRA

BRA

Branch Always

Operation: $PC + \text{xxxx} \rightarrow PC$ $PC + \text{xxx} \rightarrow PC$ $PC + R_n \rightarrow PC$ **Assembler Syntax:**

BRA xxxx

BRA xxx

BRA R_n**Description:**

Program execution continues at location PC+displacement. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 9-bit data is sign extended to form the PC relative displacement.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23	16 15								8 7								0						
BRA	xxxx	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0
		PC RELATIVE DISPLACEMENT																							

		23	16 15								8 7				0							
BRA	xxx	0 0 0 0 0 1 0 1								0 0 0 0 1 1 a a				a a 0 a a a a a								

		23	16 15								8 7								0						
BRA	Rn	0	0	0	0	1	1	0	1	0	0	0	1	1	R	R	R	1	1	0	0	0	0	0	0

Instruction Fields:

{xxxx}		24-bit PC Relative Long Displacement
{xxx}	aaaaaaaa	Signed PC Relative Short Displacement
{Rn}	RRR	Address register [R0-R7]

BRCLR

BRCLR

Branch if bit Clear

Operation:

If $S\{n\}=0$ then $PC+xxxx \rightarrow PC$
 else $PC+1 \rightarrow PC$

If $S\{n\}=0$ then $PC+xxxx \rightarrow PC$
 else $PC+1 \rightarrow PC$

If $S\{n\}=0$ then $PC+xxxx \rightarrow PC$
 else $PC+1 \rightarrow PC$

If $S\{n\}=0$ then $PC+xxxx \rightarrow PC$
 else $PC+1 \rightarrow PC$

If $S\{n\}=0$ then $PC+xxxx \rightarrow PC$
 else $PC+1 \rightarrow PC$

Assembler Syntax:

BRCLR $\#n,[X \text{ or } Y]:ea,xxxx$

BRCLR $\#n,[X \text{ or } Y],aa,xxxx$

BRCLR $\#n,[X \text{ or } Y]:pp,xxxx$

BRCLR $\#n,[X \text{ or } Y]:qq,xxxx$

BRCLR $\#n,S,xxxx$

Description: The n th bit in the source operand is tested. If the tested bit is cleared, program execution continues at location $PC+\text{displacement}$. If the tested bit is set, the PC is incremented and program execution continues sequentially. However, the address register specified in the effective address field is always updated independently of the condition. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. The 24-bit displacement is contained in the extension word of the instruction. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one. The bit to be tested is selected by an immediate bit number 0-23.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	×	×	×	×	×	×
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16						15	8				7	0												
BRCLR #n,[X or Y]:ea,xxxx	0	0	0	0	1	1	0	0	1	0	M	M	M	R	R	R	0	S	0	b	b	b	b	b	b	
	PC RELATIVE DISPLACEMENT																									

	23	16 15								8 7								0						
BRCLR #n,[X or Y]:aa,xxxx	0	0	0	0	1	1	0	0	1	0	a	a	a	a	a	a	1	S	0	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16	15	8	7	0																			
BRCLR #n,[X or Y]:pp,xxxx	0	0	0	0	1	1	0	0	1	1	p	p	p	p	p	p	0	S	0	b	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																								

	23	16	15	8	7	0																			
BRCLR #n,[X or Y]:qq,xxxx	0	0	0	0	0	1	0	0	1	0	q	q	q	q	q	q	0	S	0	b	b	b	b	b	
	PC RELATIVE DISPLACEMENT																								

	23	16 15								8 7								0							
BRCLR #n,S,xxxx	0	0	0	0	1	1	0	0	1	1	D	D	D	D	D	D	1	0	0	b	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																								

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMMRRR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{xxxx}		24-bit PC relative displacement
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{S}	DDDDDD	Source register [all on-chip registers] (see Table A-22 on page A-243)

A-6.15 Exit Current Do Loop Conditionally (BRKcc)

BRKcc

BRKcc

Exit Current Do Loop Conditionally

Operation:

If cc LA+1→PC; SSL(LF,FV)→SR; SP-1→SP
 SSH→LA; SSL→LC; SP-1→SP
else PC+1→PC

Assembler Syntax:

BRKcc

Description: Exit conditionally the current hardware DO loop before the current loop counter (LC) equals one. It also terminates the DO FOREVER (or DOR FOREVER) loop. If the value of the current DO loop counter (LC) is needed, it must be read before the execution of the BRKcc instruction. Initially, the PC is updated from the LA, the loop flag (LF) and the ForeVer flag (FV) are restored and the remaining portion of the status register (SR) is purged from the system stack. The loop address (LA) and the loop counter (LC) registers are then restored from the system stack.

The conditions that the term “cc” can specify are listed on Table A-43 on page A-251.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23		16	15		8	7		0
BRKcc	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0
	0	0	0	1	C	C	C	C	C

Instruction Fields:

{cc} CCCC Condition code (see Table A-43 on page A-251)

BRSET

BRSET

Branch if bit Set

Operation:

If	S{n}=1	then PC+xxxx	→	PC
		else PC+ 1	→	PC
If	S{n}=1	then PC+xxxx	→	PC
		else PC+ 1	→	PC
If	S{n}=1	then PC+xxxx	→	PC
		else PC+ 1	→	PC
If	S{n}=1	then PC+xxxx	→	PC
		else PC+ 1	→	PC
If	S{n}=1	then PC+xxxx	→	PC
		else PC+ 1	→	PC

Assembler Syntax:

BRSET	#n,[X or Y]:ea,xxxx
BRSET	#n,[X or Y],aa,xxxx
BRSET	#n,[X or Y]:pp,xxxx
BRSET	#n,[X or Y]:qq,xxxx
BRSET	#n,S,xxxx

Description: The nth bit in the source operand is tested. If the tested bit is set, program execution continues at location PC+displacement. If the tested bit is cleared, the PC is incremented and program execution continues sequentially. However, the address register specified in the effective address field is always updated independently of the condition. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. The 24-bit displacement is contained in the extension word of the instruction. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one. The bit to be tested is selected by an immediate bit number 0-23.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	×	×	×	×	×	×
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0																		
BRSET #n,[X or Y]:ea,xxxx	0	0	0	0	1	1	0	0	1	0	M	M	M	R	R	R	0	S	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16 15								8 7								0						
BRSET #n,[X or Y]:aa,xxxx	0	0	0	0	1	1	0	0	1	0	a	a	a	a	a	a	1	S	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16 15								8 7								0						
BRSET #n,[X or Y]:pp,xxxx	0	0	0	0	1	1	0	0	1	1	p	p	p	p	p	p	0	S	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16 15								8 7								0						
BRSET #n,[X or Y]:qq,xxxx	0	0	0	0	0	1	0	0	1	0	q	q	q	q	q	q	0	S	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16 15								8 7								0						
BRSET #n,S,xxxx	0	0	0	0	1	1	0	0	1	1	D	D	D	D	D	D	1	0	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

Instruction Fields:

{#n}	bbbbb	Bit number [0-23]
{ea}	MMRRR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{xxxx}		24-bit PC relative displacement
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{S}	DDDDDD	Source register [all on-chip registers] (see Table A-22 on page A-243)

A-6.17 Branch to Subroutine Conditionally (BScc)

BScc

BScc

Branch to Subroutine Conditionally

Operation:

If cc, then PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC
else PC+1 \rightarrow PC

If cc, then PC \rightarrow SSH; SR \rightarrow SSL; PC+xxx \rightarrow PC
else PC+1 \rightarrow PC

If cc, then PC \rightarrow SSH; SR \rightarrow SSL; PC+Rn \rightarrow PC
else PC+1 \rightarrow PC

Assembler Syntax:

BScc xxxx

BScc xxx

BScc Rn

Description: If the specified condition is true, the address of the instruction immediately following the BScc instruction and the status register are pushed onto the stack. Program execution then continues at location PC+displacement. If the specified condition is false, the PC is incremented and program execution continues sequentially. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 9-bit data is sign extended to form the PC relative displacement.

The conditions that the term “cc” can specify are listed on Table A-42 on page A-250.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23	16 15								8 7								0								
BScC	xxxx	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	C	C	C	C	
		PC RELATIVE DISPLACEMENT																									

		23	16 15								8 7								0					
BScC	xxx	0	0	0	0	0	1	0	1	C	C	C	C	0	0	a	a	a	a	0	a	a	a	a

		23	16 15								8 7								0						
BScC	Rn	0	0	0	0	1	1	0	1	0	0	0	1	1	R	R	R	0	0	0	0	C	C	C	C

Instruction Fields:

{cc}	CCCC	Condition code (see Table A-43 on page A-251)
{xxxx}		24-bit PC Relative Long Displacement
{xxx}	aaaaaaaa	Signed PC Relative Short Displacement
{Rn}	RRR	Address register [R0-R7]

BSCLR

BSCLR

Branch to Subroutine if Bit Clear

Operation:

If $S\{n\}=0$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

If $S\{n\}=0$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

If $S\{n\}=0$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

If $S\{n\}=0$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

If $S\{n\}=0$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

Assembler Syntax:

BSCLR #n,[X or Y]:ea,xxxx

BSCLR #n,[X or Y],aa,xxxx

BSCLR #n,[X or Y]:pp,xxxx

BSCLR #n,[X or Y]:qq,xxxx

BSCLR #n,S,xxxx

Description: The nth bit in the source operand is tested. If the tested bit is cleared, the address of the instruction immediately following the BSCLR instruction and the status register are pushed onto the stack. Program execution then continues at location $PC+\text{displacement}$. If the tested bit is set, the PC is incremented and program execution continues sequentially. However, the address register specified in the effective address field is always updated independently of the condition. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. The 24-bit displacement is contained in the extension word of the instruction. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one; if the condition is true, the push operation will write over the stack level where the SSH value was taken. The bit to be tested is selected by an immediate bit number 0-23.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	×	×	×	×	×	×
CCR							

- ✓ This bit is changed according to the standard definition
× This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16						15	8						7	0								
BSCLR #n,[X or Y]:ea,xxxx	0	0	0	0	1	1	0	1	1	0	M	M	M	R	R	R	0	S	0	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16	15	8	7	0																		
BSCLR #n,[X or Y]:aa,xxxx	0	0	0	0	1	1	0	1	1	0	a	a	a	a	a	a	1	S	0	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16	15	8	7	0																		
BSCLR #n,[X or Y]:qq,xxxx	0	0	0	0	0	1	0	0	1	0	q	q	q	q	q	q	1	S	0	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16	15	8	7	0																		
BSCLR #n,[X or Y]:pp,xxxx	0	0	0	0	1	1	0	1	1	1	p	p	p	p	p	p	0	S	0	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16	15	8	7	0																		
BSCLR #n,S,xxxx	0	0	0	0	1	1	0	1	1	1	D	D	D	D	D	D	1	0	0	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMMRRR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{xxxx}		24-bit Relative Long Displacement
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{S}	DDDDDD	Source register [all on-chip registers] (see Table A-22 on page A-243)

BSET

BSET

Bit Test and Set

Operation:

D[n] → C 1 → D[n]

D[n] → C 1 → D[n]

D[n] → C 1 → D[n]

D[n] → C 1 → D[n]

D[n] → C 1 → D[n]

Assembler Syntax:

BSET #n,[XorY]:ea

BSET #n,[XorY]:aa

BSET #n,[XorY]:pp

BSET #n,[XorY]:qq

BSET #n,D

Description: Test the n^{th} bit of the destination operand D, set it, and store the result in the destination location. The state of the n^{th} bit is stored in the carry bit C of the condition code register. The bit to be tested is selected by an immediate bit number from 0–23. This instruction performs a read-modify-write operation on the destination location using two destination accesses before releasing the bus. This instruction provides a test-and-set capability which is useful for synchronizing multiple processors using a shared memory. This instruction can use all memory alterable addressing modes.

When this instruction performs a bit manipulation/test on either the A or B 56-bit accumulator, it optionally shifts the accumulator value according to scaling mode bits S0 and S1 in the system status register (SR). In the data out of the shifter indicates that the accumulator extension register is in use, the instruction will act on the limited value (limited on the maximum positive or negative saturation constant). In addition the “L” flag in the SR will be set accordingly.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	●	●	●	●	●	●
CCR							

CCR Condition Codes:

For destination operand SR:

- C 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 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 According to the standard definition.
- S According to the standard definition.

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
- RM Changed if bit 12 is specified. Not affected otherwise
- SB 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: MR status bits are not affected.

Instruction Formats and opcodes:

	23	16						15	8						7	0								
BSET #n,[X or Y]:ea	0	0	0	0	1	0	1	0	0	1	M	M	M	R	R	R	0	S	1	b	b	b	b	b
	OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

	23	16 15								8 7								0						
BSET #n,[X or Y]:aa	0	0	0	0	1	0	1	0	0	0	a	a	a	a	a	a	0	S	1	b	b	b	b	b

	23	16 15								8 7								0						
BSET #n,[X or Y]:pp	0	0	0	0	1	0	1	0	1	0	p	p	p	p	p	p	0	S	1	b	b	b	b	b

	23	16						15	8						7	0								
BSET #n,[X or Y]:qq	0	0	0	0	0	0	0	1	0	0	q	q	q	q	q	q	0	S	1	b	b	b	b	b

	23	16 15								8 7								0						
BSET #n,D	0	0	0	0	1	0	1	0	1	1	D	D	D	D	D	D	0	1	1	b	b	b	b	b

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMMRRR	Effective Address (see Table A-16 on page A-241)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{D}	DDDDDD	Destination register [all on-chip registers] (see Table A-22 on page A-243)

BSR

BSR

Branch to Subroutine

Operation:

PC →SSH;SR →SSL;PC+xxxx→PC

PC →SSH;SR →SSL;PC+xxx→PC

PC →SSH;SR →SSL;PC+Rn→PC

Assembler Syntax:

BSR xxxx

BSR xxx

BSR Rn

Description: The address of the instruction immediately following the BSR instruction and the status register are pushed onto the stack. Program execution then continues at location PC+displacement. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. Short Displacement, Long Displacement and Address Register PC Relative addressing modes may be used. The Short Displacement 9-bit data is sign extended to form the PC relative displacement.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23	16 15								8 7								0						
BSR	xxxx	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
		PC RELATIVE DISPLACEMENT																							

		23	16 15								8 7				0							
BSR	xxx	0 0 0 0 0 1 0 1								0 0 0 0 1 0 a a				a a 0 a a a a a								

		23	16 15								8 7								0						
BSR	Rn	0 0 0 0 1 1 0 1								0 0 0 1 1 R R R								1 0 0 0 0 0 0 0							

Instruction Fields:

{xxxx}		24-bit PC Relative Long Displacement
{xxx}	aaaaaaa	Signed PC Relative Short Displacement
{Rn}	RRR	Address register [R0-R7]

BSSET

BSSET

Branch to Subroutine if Bit Set

Operation:

If $S\{n\}=1$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

If $S\{n\}=1$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

If $S\{n\}=1$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

If $S\{n\}=1$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

If $S\{n\}=1$ then $PC \rightarrow SSH; SR \rightarrow SSL; PC+xxxx \rightarrow PC$
else $PC+1 \rightarrow PC$

Assembler Syntax:

BSSET #n,[X or Y]:ea,xxxx

BSSET #n,[X or Y],aa,xxxx

BSSET #n,[X or Y]:pp,xxxx

BSSET #n,[X or Y]:qq,xxxx

BSSET #n,S,xxxx

Description: The nth bit in the source operand is tested. If the tested bit is set, the address of the instruction immediately following the BSSET instruction and the status register are pushed onto the stack. Program execution then continues at location PC+displacement. If the tested bit is cleared, the PC is incremented and program execution continues sequentially. However, the address register specified in the effective address field is always updated independently of the condition. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. The 24-bit displacement is contained in the extension word of the instruction. All memory alterable addressing modes may be used to reference the source operand. Absolute Short, I/O Short and Register Direct addressing modes may also be used. Note that if the specified source operand S is the SSH, the stack pointer register will be decremented by one; if the condition is true, the push operation will write over the stack level where the SSH value was taken. The bit to be tested is selected by an immediate bit number 0-23.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	×	×	×	×	×	×
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16						15	8						7	0								
BSSET #n,[X or Y]:ea,xxxx	0	0	0	0	1	1	0	1	1	0	M	M	M	R	R	R	0	S	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16	15	8	7	0																		
BSSET #n,[X or Y]:aa,xxxx	0	0	0	0	1	1	0	1	1	0	a	a	a	a	a	a	1	S	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16	15	8	7	0																	
BSSET #n,[X or Y]:pp,xxxx	0	0	0	0	1	1	0	1	1	p	p	p	p	p	p	0	S	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																						

	23	16	15	8	7	0																		
BSSET #n,[X or Y]:qq,xxxx	0	0	0	0	0	1	0	0	1	0	q	q	q	q	q	q	1	S	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

	23	16	15	8	7	0																		
BSSET #n,S,xxxx	0	0	0	0	1	1	0	1	1	1	D	D	D	D	D	D	1	0	1	b	b	b	b	b
	PC RELATIVE DISPLACEMENT																							

Instruction Fields:

{#n}	bbbbb	Bit number [0-23]
{ea}	MMRRR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{xxxx}		24-bit Relative Long Displacement
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{S}	DDDDDD	Source register [all on-chip registers] (see Table A-22 on page A-243)

BTST

BTST

Bit Test

Operation:

D[n] → C

D[n] → C

D[n] → C

D[n] → C

D[n] → C

Assembler Syntax:

BTST #n,[XorY]:ea

BTST #n,[XorY]:aa

BTST #n,[XorY]:pp

BTST #n,[XorY]:qq

BTST #n,D

Description: Test the n^{th} bit of the destination operand D. The state of the n^{th} bit is stored in the carry bit C of the condition code register. The bit to be tested is selected by an immediate bit number from 0–23. This instruction is useful for performing serial to parallel conversion when used with the appropriate rotate instructions. This instruction can use all memory alterable addressing modes.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	●
CCR							

- C Set if bit tested is set. Cleared otherwise.
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

SP — Stack Pointer:

For destination operand SSH: SP — Decrement by 1.

For other destination operands: Not affected

Instruction Formats and opcodes:

	23	16 15										8 7					0									
BTST #n,[X or Y]:ea	0	0	0	0	1	0	1	1	0	1	M	M	M	R	R	R	0	S	1	b	b	b	b	b	b	
	OPTIONAL EFFECTIVE ADDRESS EXTENSION																									

	23	16 15						8 7						0										
BTST #n,[X or Y]:aa	0	0	0	0	1	0	1	1	0	0	a	a	a	a	a	a	0	S	1	b	b	b	b	b

	23	16 15								8 7								0						
BTST #n,[X or Y]:pp	0	0	0	0	1	0	1	1	1	0	p	p	p	p	p	p	0	S	1	b	b	b	b	b

	23	16 15							8 7							0								
BTST #n,[X or Y]:qq	0	0	0	0	0	0	0	1	0	1	q	q	q	q	q	q	0	S	1	b	b	b	b	b

	23	16 15								8 7								0						
BTST #n,D	0	0	0	0	1	0	1	1	1	1	D	D	D	D	D	D	0	1	1	b	b	b	b	b

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMMRRR	Effective Address (see Table A-18 on page A-241)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{D}	DDDDDD	Destination register [all on-chip registers] (see Table A-22 on page A-243)

CLB**CLB****Count Leading Bits****Operation:**

If S[55]=0 then

9 - (Number of consecutive leading zeros in S[55:0]) → D[47:24]

else

9 - (Number of consecutive leading ones in S[55:0]) → D[47:24]

Assembler Syntax:

CLB S,D

Description: Count leading zeros or ones according to bit 55 of the source accumulator. Scan bits 55-0 of the source accumulator starting from bit 55. The MSP of the destination accumulator is loaded with 9 minus the number of consecutive leading ones or zeros found. The result is a signed integer in MSP whose range of possible values is from +8 to -47. This is a 56-bit operation. The LSP of the destination accumulator D is filled with zeros. The EXP of the destination accumulator D is sign extended.

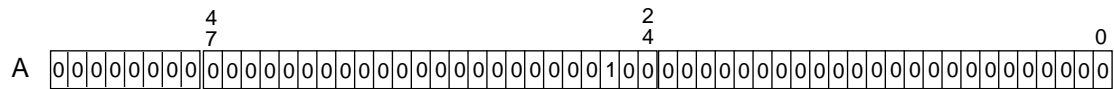
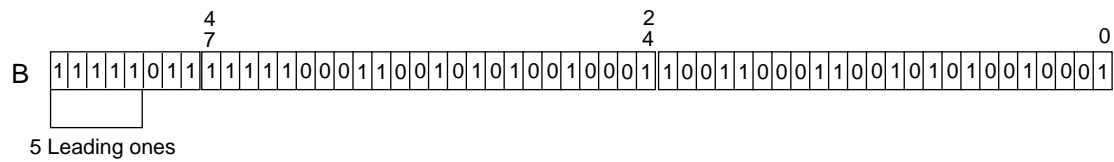
Notes:

- 1) If the source accumulator is all zeros then the result will be zero.
- 2) When in sixteen bit arithmetic mode, the count ignores the unused 8 least significant bits of the MSP and LSP of the source accumulator. Therefore the result is a signed integer whose range of possible values is from +8 to -31.
- 3) This instruction may be used in conjunction with NORMF instruction, to specify the shift direction and amount needed for normalization.

Condition Codes:

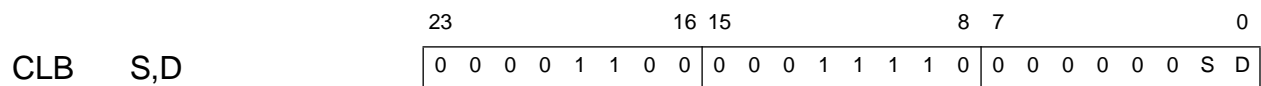
7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	●	●	●	x
CCR							

- N Set if bit 47 of the result is set. Cleared otherwise
- Z Set if bits 47-24 of the result are zero.
- V Always cleared
- x This bit is unchanged by the instruction



Result in A is $9 - 5 = 4$

Instruction Formats and opcode:



Instruction Fields:

{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S}	S	Source accumulator [A,B] (see Table A-10 on page A-239)

CLR

CLR

Clear Accumulator

Operation:

0 → D (parallel move)

Assembler Syntax:

CLR D (parallel move)

Description: Clear the destination accumulator. This is a 56-bit clear instruction.**Condition Codes:**

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	●	●	●	●	●	×
CCR							

- E Always cleared
- U Always set
- N Always cleared
- Z Always set
- V Always cleared
- ✓ This bit is changed according to the standard definition
- × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
CLR D	DATA BUS MOVE FIELD				0 0 0 1	d 0 1 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

CMP

CMP

Compare

Operation:

S2 – S1 (parallel move)

S2 – #xx

S2 – #xxxxxx

Assembler Syntax:

CMP S1, S2 (parallel move)

CMP #xx, S2

CMP #xxxxxx, S2

Description: Subtract the source one operand from the source two accumulator, S2, and update the condition code register. The result of the subtraction operation is not stored.

The source one operand can be a register (word - 24 bits or accumulator - 56 bits), short immediate (6 bits) or long immediate (24 bits). When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the 6 bits will be right aligned and the remaining bits will be zeroed to form a 24-bit source operand.

Note: This instruction subtracts 56-bit operands. When a word is specified as the source one operand, 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	✓
CCR							

- ✓ This bit is changed according to the standard definition
- × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0					
CMP S1, S2	DATA BUS MOVE FIELD			0	J	J	J	d	1	0	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION										

	23	16							15	8							7	0						
CMP #xx, S2	0	0	0	0	0	0	0	1	0	1	i	i	i	i	i	i	1	0	0	0	d	1	0	1

	23	16	15	8	7	0																		
CMP #xxxxxx,S2	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0	d	1	0	1
	IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S1}	JJJ	Source one register [B/A,X0,Y0,X1,Y1] (see Table A-24 on page A-243)
{S2}	d	Source two accumulator [A/B] (see Table A-10 on page A-239)
{#xx}	iiiiii	6-bit Immediate Short Data
{#xxxxxx}		24-bit Immediate Long Data extension word

CMPM

CMPM

Compare Magnitude

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	✓
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0						
CMPM S1, S2	DATA BUS MOVE FIELD				0	J	J	J	d	1	1	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

Instruction Fields:

{S1}	JJJ	Source one register [B/A,X0,Y0,X1,Y1] (see Table A-24 on page A-243)
{S2}	d	Source two accumulator [A,B] (see Table A-10 on page A-239)

A-6.27 Compare Unsigned (CMPU)

CMPU

CMPU

Compare Unsigned

Operation:

S2 – S1

Assembler Syntax:

CMPU S1, S2

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 a 24 or 48-bit unsigned operand from a 48-bit unsigned operand. When a 24-bit word is specified as S1 it is aligned to the left and zero filled to form a valid 48-bit operand. If an accumulator is specified as an operand, the value in the EXP does not affect the operation.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	✓	●	●	✓
CCR							

- V Always cleared
- Z Set if bits 47-0 of the result are zero
- x This bit is unchanged by the instruction
- ✓ This bit is changed according to the standard definition

Instruction Formats and opcodes:

	23	16 15				8 7				0													
CMPU S1, S2	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	g	g	g	d

Instruction Fields:

{S1} **ggg** Source one register [A,B,X0,Y0,X1,Y1] (see Table A-15 on page A-240)
{S2} **d** Source two accumulator [A,B] (see Table A-10 on page A-239)

DEBUG

DEBUG

Enter Debug Mode

Operation:

Assembler Syntax:

Enter the debug mode

DEBUG

Description: Enter the debug mode and wait for OnCE commands.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23		16	15		8	7		0
DEBUG	0	0	0	0	0	0	0	0	0

Instruction Fields: None

A-6.29 Enter Debug Mode Conditionally (DEBUGcc)

DEBUGcc

DEBUGcc

Enter Debug Mode Conditionally

Operation:

If cc, then enter the debug mode

Assembler Syntax:

DEBUGcc

Description: If the specified condition is true, enter the debug mode and wait for OnCE commands. If the specified condition is false, continue with the next instruction.

The conditions that the term “cc” can specify are listed on Table A-42 on page A-250.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23		16	15		8	7		0
DEBUGcc	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	1	0
							0	0	0
							C	C	C
							C	C	C

Instruction Fields:

{cc} CCCC Condition code (see Table A-43 on page A-251)

DEC

DEC

Decrement by One

Operation:

Assembler Syntax:

D - 1 → D

DEC D

Description: Decrement by one the specified operand and store the result in the destination accumulator. One is subtracted from the LSB of D.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	✓
CCR							

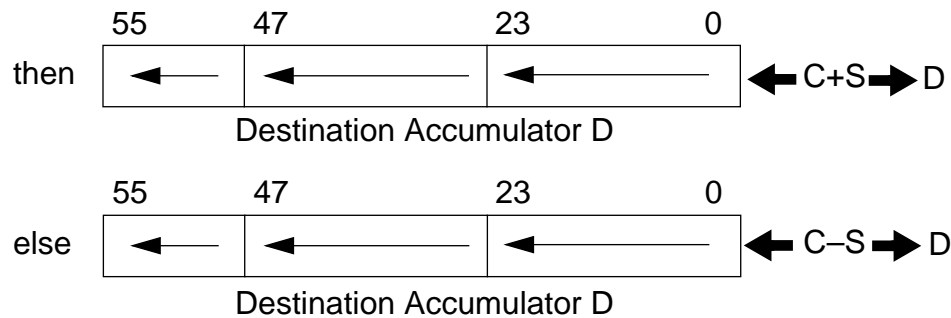
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16 15								8 7								0							
DEC D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	d

Instruction Fields:

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

DIV**DIV****Divide Iteration****Operation:** If $D[55] \oplus S[23] = 1$,where \oplus denotes the logical exclusive OR operator**Assembler Syntax:** DIV S,D**Description:**

Divide the destination operand D by the source operand S and store the result in the destination accumulator D. **The 48-bit dividend must be a positive fraction which has been sign extended to 56-bits and is stored in the full 56-bit destination accumulator D. The 24-bit divisor is a signed fraction and is stored in the source operand S.** Each DIV iteration calculates one quotient bit using a nonrestoring fractional division algorithm (see description on the next page). After the execution of the first DIV instruction, the destination operand holds both the partial remainder and the formed quotient. The partial remainder occupies the high-order portion of the destination accumulator D and is a signed fraction. The formed quotient occupies the low-order portion of the destination accumulator D (A0 or B0) and is a positive fraction. One bit of the formed quotient is shifted into the LS bit of the destination accumulator at the start of each DIV iteration. The formed quotient is the true quotient if the true quotient is positive. If the true quotient is negative, the formed quotient must be negated. **Valid results are obtained only when $|D| < |S|$ and the operands are interpreted as fractions.** Note that this condition ensures that the magnitude of the quotient is less than one (i.e., is fractional) and precludes division by zero.

The DIV instruction calculates one quotient bit based on the divisor and the previous

partial remainder. To produce an N-bit quotient, the DIV instruction is executed N times where N is the number of bits of precision desired in the quotient, $1 \leq N \leq 24$. 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.

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:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	●	x	x	x	x	●	●
CCR							

- 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.
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16 15								8 7								0							
DIV S,D	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	J	J	d	0	0	0

Instruction Fields:

- {S} JJ Source input register [X0,X1,Y0,Y1] (see Table A-12 on page A-239)
- {D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

DMAC

DMAC

Double (Multi) Precision Multiply Accumulate with Right Shift

Operation:

$[D \gg 24] \pm S1 * S2 \rightarrow D$
(S1 signed, S2 signed)

$[D \gg 24] \pm S1 * S2 \rightarrow D$
(S1 signed, S2 unsigned)

$[D \gg 24] \pm S1 * S2 \rightarrow D$
(S1 unsigned, S2 unsigned)

Assembler Syntax:

DMACss $(\pm)S1, S2, D$ (no parallel move)

DMACsu $(\pm)S2, S1, D$ (no parallel move)

DMACuu $(\pm)S2, S1, D$ (no parallel move)

Description: Multiply the two 24-bit source operands S1 and S2 and add/subtract the product to/from the specified 56-bit destination accumulator D, which has been previously shifted 24 bits to the right. The multiplication can be performed on signed numbers (ss), unsigned numbers (uu), or mixed (unsigned * signed, (su)). The “-” sign option is used to negate the specified product prior to accumulation. The default sign option is “+”. This instruction is optimized for multiprecision multiplication support.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
 x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16				15	8				7	0									
DMAC (\pm)S1,S2,D	0	0	0	0	0	0	0	1	0	0	1	0	s	1	s	d	k	Q	Q	Q	Q

Instruction Fields:

{S1,S2}	QQQQ	Source registers S1,S2 [all combinations of X0,X1,Y0 and Y1] (see Table A-30 on page A-245)
{D}	d	Destination accumulator [A,B] (see Table A-10 on page A-239)
{<u>++</u>}	k	Sign [+,-] (see Table A-29 on page A-244)
{ss,su,uu}	ss	[ss,su,uu] (see Table A-39 on page A-248)

DO**DO****Start Hardware Loop****Operation:**

SP+1 → SP; LA → SSH; LC → SSL; [X or Y]:ea → LC
 SP+1 → SP; PC → SSH; SR → SSL; expr - 1 → LA
 1 → LF

SP+1 → SP; LA → SSH; LC → SSL; [Xor Y]:aa → LC
 SP+1 → SP; PC → SSH; SR → SSL; expr - 1 → LA
 1 → LF

SP+1 → SP; LA → SSH; LC → SSL; #xxx → LC
 SP+1 → SP; PC → SSH; SR → SSL; expr - 1 → LA
 1 → LF

SP+1 → SP; LA → SSH; LC → SSL; S → LC
 SP+1 → SP; PC → SSH; SR → SSL; expr - 1 → LA
 1 → LF

End of Loop:

SSL(LF) → SR; SP - 1 → SP
 SSH → LA; SSL → LC; SP - 1 → SP

Assembler Syntax:

DO [Xor Y]:ea,expr

DO [Xor Y]:aa,expr

DO #xxx,expr

DO S,expr

Description: Begin a hardware DO loop that is to be repeated the number of times specified in the instruction's source operand and whose range of execution is terminated by the destination operand (previously shown as "expr"). No overhead other than the execution of this DO instruction is required to set up this loop. DO loops can be nested and the loop count can be passed as a parameter.

During the first instruction cycle, the current contents of the loop address (LA) and the loop counter (LC) registers are pushed onto the system stack. The DO instruction's source operand is then loaded into the loop counter (LC) register. The LC register contains the remaining number of times the DO loop will be executed and can be accessed from inside the DO loop subject to certain restrictions. If LC initial value is zero and the 16-bit compatibility mode bit (bit 13, SC, in the Chip Status Register) is cleared, the DO loop is not executed. If LC initial value is zero but SC is set, the DO loop will be 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 12 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 24 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.

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.

During the "end-of-loop" processing, the loop flag (LF) from the lower portion (SSL) of SP is written into the status register (SR), the contents of the loop address (LA) register are restored from the upper portion (SSH) of (SP-1), the contents of the loop counter (LC) are restored from the lower portion (SSL) of (SP-1) and the stack pointer (SP) is decremented by two. Instruction fetches now continue at the address of the instruction following the last instruction in the DO loop. Note that LF is the only bit in the status register (SR) that is restored after a hardware DO loop has been exited.

Note: The loop flag (LF) is cleared by a hardware reset.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	x	x	x	x	x	x
CCR							

- S Set if the instruction sends A/B accumulator contents to XDB or YDB.
- L Set if data limiting occurred [see note 2]
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

DO [X or Y]:ea, expr	<div> <div>231615870</div> <div>0000011001MMMRRR0S00000000</div> <div>ABSOLUTE ADDRESS EXTENSION WORD</div> </div>
DO [X or Y]:aa, expr	<div> <div>231615870</div> <div>0000011000aaaaaa0S00000000</div> <div>ABSOLUTE ADDRESS EXTENSION WORD</div> </div>
DO #xxx, expr	<div> <div>231615870</div> <div>00000110iiiiiiii1000hhhh</div> <div>ABSOLUTE ADDRESS EXTENSION WORD</div> </div>
DO S, expr	<div> <div>231615870</div> <div>0000011011DDDDDD00000000</div> <div>ABSOLUTE ADDRESS EXTENSION WORD</div> </div>

Instruction Fields:

{ea}	MMMR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{expr}		24-bit Absolute Address in 24-bit extension word
{aa}	aaaaaa	Absolute Address [0-63]
{#xxx}	hhhhiiiiiii	Immediate Short Data [0-4095]
{S}	DDDDDD	Source register [all on-chip registers, except SSH] (see Table A-22 on page A-243)

Note:

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.
--------	-----------	---

DO FOREVER DO FOREVER

Start Infinite Loop

Operation:

SP+1 → SP; LA → SSH; LC → SSL
SP+1 → SP; PC → SSH; SR → SSL; expr - 1 → LA
1 → LF; 1 → FV

Assembler Syntax:

DO FOREVER,expr

Description: Begin a hardware DO loop that is to be repeated for ever and whose range of execution is terminated by the destination operand (shown above as “expr”). No overhead other than the execution of this DO FOREVER instruction is required to set up this loop. DO FOREVER loops can be nested. 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 loop counter (LC) register is pushed onto the stack but is not updated by this instruction.

During the second instruction cycle, the current contents of the Program Counter (PC) register and the Status Register (SR) are pushed onto the system stack. Stacking the LA, LC, PC, and SR registers permits nesting DO FOREVER loops. The DO FOREVER instruction's destination operand (shown as “expr”) is then loaded into the Loop Address (LA) register. This 24-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 FOREVER instruction (i.e., the first actual instruction in the DO FOREVER 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) and the ForeVer flag are set. This results in the PC being repeatedly compared with LA to determine if the last instruction in the loop has been fetched. If LA equals PC, the last instruction in the loop has been fetched and SSH is loaded into the PC to fetch the first instruction in the loop again. The loop counter (LC) register is then decremented by one without being tested. This register can be used by the programmer to count the number of loops already executed.

When executing a DO FOREVER loop, the instructions are actually fetched each time through the loop. Therefore, a DO FOREVER loop can be interrupted. DO FOREVER loops can also be nested. When DO FOREVER loops are nested, the end of loop addresses must also be nested and are not allowed to be equal. The assembler generates an error message when DO FOREVER loops are improperly nested.

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.

The loop counter (LC) register is never tested by the DO FOREVER instruction and the only way of terminating the loop process is to use either the ENDDO or BRKcc instructions. LC is decremented every time PC=LA so that it can be used by the programmer to keep track of the number of times the DO FOREVER loop has been executed. If the programmer wants to initialize LC to a particular value before the DO FOREVER, care should be taken to save it before if the DO loop is nested. If so, LC should also be restored immediately after exiting the nested DO FOREVER loop.

Condition Codes:

	7	6	5	4	3	2	1	0
	S	L	E	U	N	Z	V	C
	X	X	X	X	X	X	X	X
	CCR							

- × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16 15								8 7								0						
DO FOREVER	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	
	ABSOLUTE ADDRESS EXTENSION WORD																							

Instruction Fields: None.

DOR

DOR

Start PC relative Hardware Loop

Operation:

SP+1 → SP; LA → SSH; LC → SSL; [X or Y]:ea → LC
 SP+1 → SP; PC → SSH; SR → SSL; PC+xxxx → LA
 1 → LF

SP+1 → SP; LA → SSH; LC → SSL; [X or Y]:ea → LC
 SP+1 → SP; PC → SSH; SR → SSL; PC+xxxx → LA
 1 → LF

SP+1 → SP; LA → SSH; LC → SSL; #xxx → LC
 SP+1 → SP; PC → SSH; SR → SSL; PC+xxxx → LA
 1 → LF

SP+1 → SP; LA → SSH; LC → SSL; S → LC
 SP+1 → SP; PC → SSH; SR → SSL; PC+xxxx → LA
 1 → LF

Assembler Syntax:

DOR [Xor Y]:ea,label

DOR [Xor Y]:aa,label

DOR #xxx,label

DOR S,label

Description:

This instruction initiates the beginning of a PC relative hardware program loop. The current loop address (LA) and loop counter (LC) values are pushed onto the system stack. With proper system stack management, this allows unlimited nested hardware DO loops. The PC and SR are pushed onto the system stack. The PC is added to the 24-bit address displacement extension word and the resulting address is loaded into the loop address register (LA). The effective address specifies the address of the loop count which is loaded into the loop counter (LC). The DO loop is executed LC times. If LC initial value is zero and the 16-bit compatibility mode bit (bit 13, SC, in the Chip Status Register) is cleared, the DO loop is not executed. If LC initial value is zero but SC is set, the DO loop will be executed 65,536 times. All address register indirect addressing modes (less Long Displacement) may be used. Register Direct addressing mode may also be used. If immediate short data is specified, the LC is loaded with the zero extended 12-bit immediate data.

During hardware loop operation, each instruction is fetched each time through the program loop. Therefore, instructions being executed in a hardware loop are interruptible and may be nested. The value of the PC pushed onto the system stack is the location of

the first instruction after the DOR instruction. This value is read from the top of the system stack to return to the start of the program loop. When DOR instructions are nested, the end of loop addresses must also be nested and are not allowed to be equal.

The assembler calculates the end of loop address LA (PC relative address extension word xxxx) by evaluating the end of loop expression and subtracting one. Thus the end of loop expression in the source code represents the “next address” after the end of the loop. If a simple end of loop address label is used, it should be placed after the last instruction in the loop.

Since the end of loop comparison is at fetch time and ahead of the end of loop execution, instructions which change program flow or change the system stack may not be used near the end of the loop without some restrictions. Proper hardware 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 destination register. Also, SSH cannot be specified as a source register in the DOR instruction itself. The assembler will generate a warning if the restricted instructions are found within their restricted boundaries.

Implementation Notes:

DOR SP,xxxx The actual value that will be loaded in the LC is the value of the SP before the DOR instruction incremented by one.

DOR SSL,xxxx The LC will be loaded with its previous value that was saved in the stack by the DOR instruction itself.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	x	x	x	x	x	x
CCR							

- S Set if the instruction sends A/B accumulator contents to XDB or YDB.
- L Set if data limiting occurred
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16 15	8 7	0
DOR [X or Y]:ea,label	0 0 0 0 0 1 1 0	0 1 M M M R R R	0 S 0 1 0 0 0 0	
	PC RELATIVE DISPLACEMENT			

	23	16 15	8 7	0
DOR [X or Y]:aa,label	0 0 0 0 0 1 1 0	0 0 a a a a a a	0 S 0 1 0 0 0 0	
	PC RELATIVE DISPLACEMENT			

	23	16 15	8 7	0
DOR #xxx, label	0 0 0 0 0 1 1 0	i i i i i i i i	1 0 0 1 h h h h	
	PC RELATIVE DISPLACEMENT			

	23	16 15	8 7	0
DOR S, label	0 0 0 0 0 1 1 0	1 1 D D D D D D	0 0 0 1 0 0 0 0	
	PC RELATIVE DISPLACEMENT			

Instruction Fields:

{ea}	MMMR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{label}		24-bit Address Displacement in 24-bit extension word
{aa}	aaaaaa	Absolute Address [0-63]
{#xxx}	hhhhiiiiiii	Immediate Short Data [0-4095]
{S}	DDDDDD	Source register [all on-chip registers except SSH] (see Table A-22 on page A-243)

DOR FOREVER DOR FOREVER

Start PC Relative Infinite Loop

Operation:

SP+1 → SP; LA → SSH; LC → SSL
SP+1 → SP; PC → SSH; SR → SSL; PC+xxxx → LA
1 → LF; 1 → FV

Assembler Syntax:

DOR FOREVER,label

Description: Begin a hardware DO loop that is to be repeated for ever and whose range of execution is terminated by the destination operand (shown above as label). No overhead other than the execution of this DOR FOREVER instruction is required to set up this loop. DOR FOREVER loops can be nested. 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 loop counter (LC) register is pushed onto the stack but is not updated by this instruction.

During the second instruction cycle, the current contents of the Program Counter (PC) register and the Status Register (SR) are pushed onto the system stack. Stacking the LA, LC, PC, and SR registers permits nesting DOR FOREVER loops. The DOR FOREVER instruction's destination operand (shown as label) is then loaded into the Loop Address (LA) register after having been added to the PC. This 24-bit operand is located in the instruction's 24-bit relative 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 DOR FOREVER instruction (i.e., the first actual instruction in the DOR FOREVER 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) and the ForeVer flag are set. This results in the PC being repeatedly compared with LA to determine if the last instruction in the loop has been fetched. If LA equals PC, the last instruction in the loop has been fetched and SSH is read (i.e copied but not pulled) into the PC to fetch the first instruction in the loop again. The loop counter (LC) register is then decremented by one without being tested. This register can be used by the programer to count the number of loops already executed.

When executing a DOR FOREVER loop, the instructions are actually fetched each time through the loop. Therefore, a DOR FOREVER loop can be interrupted. DOR FOREVER loops can also be nested. When DOR FOREVER loops are nested, the end of loop

addresses must also be nested and are not allowed to be equal. The assembler generates an error message when DOR FOREVER loops are improperly nested.

Note: The assembler calculates the end of loop address LA (PC relative address extension word xxxx) by evaluating the end of loop expression and subtracting one. Thus the end of loop expression in the source code represents the “next address” after the end of the loop. If a simple end of loop address label is used, it should be placed after the last instruction in the loop.

The loop counter (LC) register is never tested by the DOR FOREVER instruction and the only way of terminating the loop process is to use either the ENDDO or BRKcc instructions. LC is decremented every time PC=LA so that it can be used by the programmer to keep track of the number of times the DOR FOREVER loop has been executed. If the programmer wants to initialize LC to a particular value before the DOR FOREVER, care should be taken to save it before if the DO loop is nested. If so, LC should also be restored immediately after exiting the nested DOR FOREVER loop.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16 15								8 7								0						
DOR FOREVER	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	
	PC RELATIVE DISPLACEMENT																							

Instruction Fields: None.

ENDDO

ENDDO

End Current DO Loop

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23							16	15								8	7							0
ENDDO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0

Instruction Fields: None

EOR

EOR

Logical Exclusive OR

Operation:

$S \oplus D[47:24] \rightarrow D[47:24]$ (parallel move)

$\#xx \oplus D[47:24] \rightarrow D[47:24]$

$\#xxxxxx \oplus D[47:24] \rightarrow D[47:24]$

Assembler Syntax:

EOR S,D (parallel move)

EOR #xx,D

EOR #xxxxxx,D

where \oplus denotes the logical XOR 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. The source can be a 24-bit register, 6-bit short immediate or 24-bit long immediate. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the 6 bits will be right aligned and the remaining bits will be zeroed to form a 24-bit source operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	×	×	●	●	●	×
CCR							

- N Set if bit 47 of the result is set
 - Z Set if bits 47-24 of the result are zero
 - V Always cleared
 - ✓ This bit is changed according to the standard definition
 - ×
- This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
EOR S,D	DATA BUS MOVE FIELD				0 1 J J	d 0 1 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

	23	16	15	8	7	0
EOR #xx,D	0 0 0 0 0 0 0 1	0 1 i i i i i i	1 0 0 0 d 0 1 1			

	23	16	15	8	7	0										
EOR #xxxxxx,D	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1
	IMMEDIATE DATA EXTENSION															

Instruction Fields:

{S}	JJ	Source register [X0,X1,Y0,Y1] (see Table A-12 on page A-239)
{D}	d	Destination accumulator [A/B] (see Table A-10 on page A-239)
{#xx}	iiii	6-bit Immediate Short Data
{#xxxxxx}		24-bit Immediate Long Data extension word

EXTRACT

EXTRACT

Extract Bit Field

Operation:

Offset = S1[5:0]
Width = S1[17:12]

$S2[(\text{offset} + \text{width} - 1) : \text{offset}] \rightarrow D[(\text{width} - 1) : 0]$
 $S2[\text{offset} + \text{width} - 1] \rightarrow D[55 : \text{width}]$ (sign extension)

Offset = #CO[5:0]
Width = #CO[17:12]

$S2[(\text{offset} + \text{width} - 1) : \text{offset}] \rightarrow D[(\text{width} - 1) : 0]$
 $S2[\text{offset} + \text{width} - 1] \rightarrow D[55 : \text{width}]$ (sign extension)

Assembler Syntax:

EXTRACT S1,S2,D

EXTRACT #CO,S2,D

Description: Extract a bit-field from source accumulator S2. The bit-field width is specified by bits 17-12 in S1 register or in immediate control word #CO. The offset from the least significant bit is specified by bits 5-0 in S1 register or in immediate control word #CO. The extracted field is placed in the destination accumulator D, aligned to the right. The construction of the control register can be done by using the MERGE instruction.

This is a 56 bit operation. Bits outside the field are filled with sign extension according to the most significant bit of the extracted bit field.

Notes:

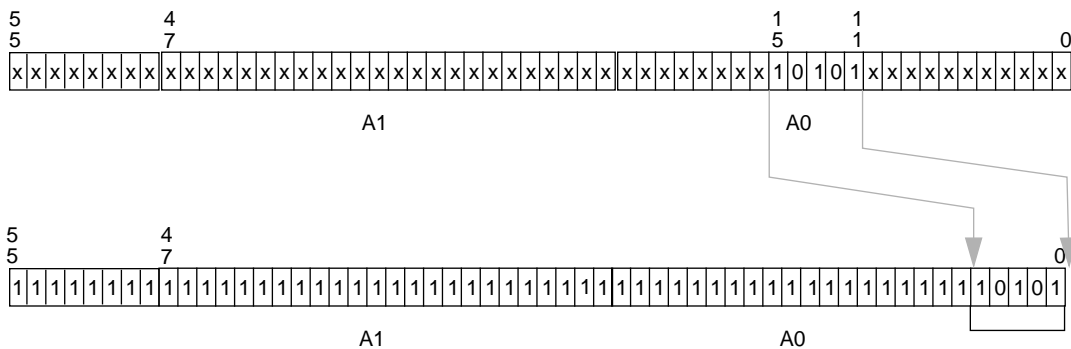
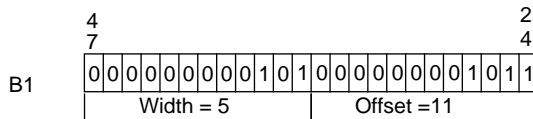
- 1) In 16 bit arithmetic mode, the offset field is located in bits 13-8 of the control register and the width field is located in bits 21-16 of the control register. These fields corresponds to the definition of the fields in the MERGE instruction.
- 2) In 16 bit arithmetic mode, when the width value is zero, then the result will be undefined.
- 3) If offset + width exceeds the value of 56, the result will be undefined.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	✓	✓	✓	✓	●	●
CCR							

- V Always cleared
- C Always cleared
- x This bit is unchanged by the instruction
- ✓ This bit is changed according to the standard definition

Example: EXTRACT B1,A,A



Instruction Formats and opcodes:

EXTRACT	S1,S2,D	23	16 15								8 7								0						
		0	0	0	0	1	1	0	0	0	0	0	1	1	0	1	0	0	0	0	s	S	S	S	D
EXTRACT	#CO,S2,D	23	16 15								8 7								0						
		0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	s	0	0	0	D
		CONTROL WORD EXTENSION																							

Instruction Fields:

{S2}	s	Source accumulator [A,B] (see Table A-10 on page A-239)
{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S1}	SSS	Control register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)
{#CO}		Control word extension.

EXTRACTU EXTRACTU

Extract Unsigned Bit Field

Operation:

Offset = S1[5:0]
Width = S1[17:12]

S2[(offset+width-1):offset] → D[(width-1):0]
zero → D[55:width]

Offset = #CO[5:0]
Width = #CO[17:12]

S2[(offset+width-1):offset] → D[(width-1):0]
zero → D[55:width]

Assembler Syntax:

EXTRACTU S1,S2,D

EXTRACTU #CO,S2,D

Description: Extract an unsigned bit-field from source accumulator S2. The bit-field width is specified by bits 17-12 in S1 register or in immediate control word #CO. The offset from the least significant bit is specified by bits 5-0 in S1 register or in immediate control word #CO. The extracted field is placed in the destination accumulator D, aligned to the right. The construction of the control register can be done by using the MERGE instruction.

This is a 56 bits operation. Bits outside the field are filled with zeros.

Notes:

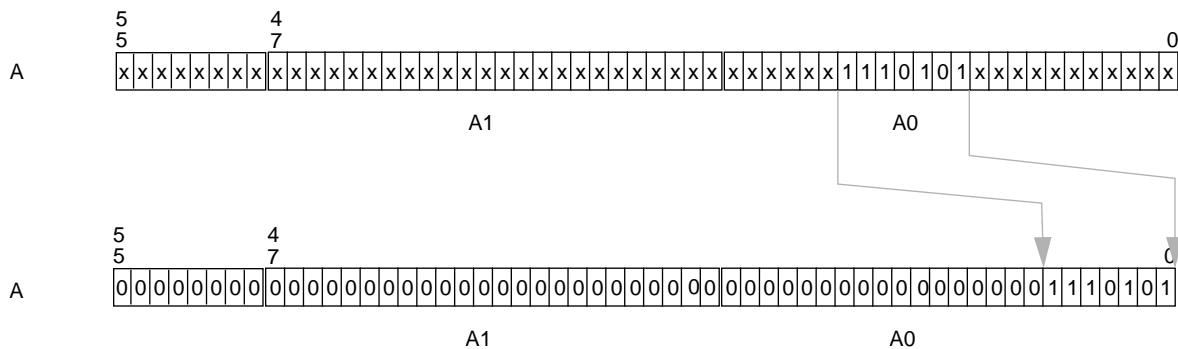
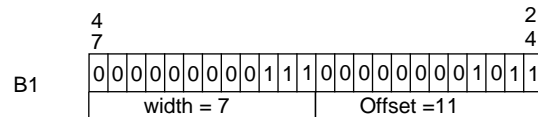
1) in 16 bit arithmetic mode, the offset field is located in bits 13-8 of the control register and the width field is located in bits 21-16 of the control register. These fields corresponds to the definition of the fields in the MERGE instruction.

2) If offset + width exceeds the value of 56, the result will be undefined.

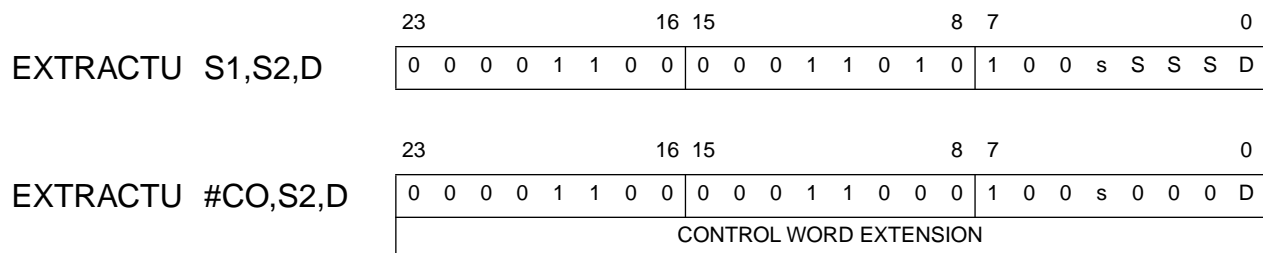
7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
X	X	✓	✓	✓	✓	●	●
CCR							

- V Always cleared
- C Always cleared
- × This bit is unchanged by the instruction
- ✓ This bit is changed according to the standard definition

Example :EXTRACTU B1,A,A



Instruction Formats and opcodes:



Instruction Fields:

{S2}	s	Source accumulator [A,B] (see Table A-10 on page A-239)
{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S1}	SSS	Control register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)
{#CO}		Control word extension.

A-6.41 Execute Conditionally without CCR Update (IFcc)

IFcc

IFcc

Execute Conditionally without CCR Update

Operation:

If cc, then opcode operation

Assembler Syntax:

Opcode-Operands IFcc

Description: If the specified condition is true, execute and store result of the specified Data ALU operation. If the specified condition is false, no destination is altered. The CCR is never updated with the condition codes generated by the Data ALU operation.

The instructions that can conditionally be executed by using IFcc are the arithmetic and logical instructions that are considered as “parallel” instructions. See Table A-3 and Table A-4 for a list of those instructions.

The conditions that the term “cc” may specify are listed on Table A-42 on page A-250

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23		16	15		8	7		0
IFcc	0	0	1	0	0	0	0	0	0
	0	0	1	0	C	C	C	C	INSTRUCTION OPCODE

Instruction Fields:

{cc} CCCC Condition code (see Table A-43 on page A-251)

A-6.42 Execute Conditionally with CCR Update (IFcc.U)

IFcc.U

IFcc.U

Execute Conditionally with CCR Update

Operation:

If cc, then opcode operation

Assembler Syntax:

Opcode-Operands IFcc

If the specified condition is true, execute and store result of the specified Data ALU operation and update the CCR with the status information generated by the Data ALU operation. If the specified condition is false, no destination is altered and the CCR is not affected.

The instructions that can conditionally be executed by using IFcc.U are the arithmetic and logical instructions that are considered as “parallel” instructions. See Table A-3 and Table A-4 for a list of those instructions.

The conditions that the term “cc” may specify are listed on Table A-42 on page A-250

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	●	●	●	●	●	●
CCR							

- If the specified condition is true changed according to the instruction. Not changed otherwise.

Instruction Formats and opcodes:

	23	16				15	8				7	0																	
IF _{CC.U}	0	0	1	0	0	0	0	0	0	0	0	1	1	C	C	C	C	INSTRUCTION OPCODE											

Instruction Fields:

{cc} **CCCC** Condition code (see Table A-43 on page A-251)

A-6.43 Illegal Instruction Interrupt (ILLEGAL)

ILLEGAL

ILLEGAL

Illegal Instruction Interrupt

Operation:

Begin Illegal Instruction exception processing

Assembler Syntax:

Opcode-Operands IFcc

Description: The ILLEGAL instruction is executed as if it were a NOP instruction. Normal instruction execution is suspended and illegal instruction exception processing is initiated. The interrupt vector address is located at address P:\$3E. The interrupt priority level (I1, I0) is set to 3 in the status register if a long interrupt service routine is used. The purpose of the ILLEGAL instruction is to force the DSP into an illegal instruction exception for test purposes. Exiting an illegal instruction is a fatal error. A long exception routine should be used to indicate this condition and cause the system to be restarted.

If the ILLEGAL instruction is in a DO loop at LA and the instruction at LA-1 is being interrupted, then LC 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0																	
ILLEGAL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

Instruction Fields: None

A-6.44 Increment by One (INC)

INC

INC

Increment by One

Operation: $D + 1 \rightarrow D$ **Assembler Syntax:**

INC D

Description: Increment by one the specified operand and store the result in the destination accumulator. One is added from the LSB of D.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	✓
CCR							

✓ This bit is changed according to the standard definition

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16 15								8 7								0						
INC D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	d

Instruction Fields:

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

INSERT

INSERT

Insert Bit Field

Operation:

Offset = S1[5:0]
Width = S1[17:12]

$S2[(width-1):0] \rightarrow D[(offset+width-1):offset]$

Offset = #CO[5:0]
Width = #CO[17:12]

$S2[(width-1):0] \rightarrow D[(offset+width-1):offset]$

Assembler Syntax:

INSERT S1,S2,D

INSERT #CO,S2,D

Description: Insert a bit-field into the destination accumulator D. The bit-field whose width is specified by bits 17-12 in S1 register, begins at the least significant bit of the S2 register. This bit-field is inserted in the destination accumulator D, with an offset according to bits 5-0 in S1 register. S1 operand can be an immediate control word #CO. Width specified by S1 should not exceed value of 24. The construction of the control register can be done by using the MERGE instruction.

This is a 56 bit operation. Any bits outside the field remain unchanged.

Notes:

1) In 16 bit arithmetic mode, the offset field is located in bits 13-8 of the control register and the width field is located in bits 21-16 of the control register. These fields corresponds to the definition of the fields in the MERGE instruction. Width specified by S1 should not exceed value of 16.

2) In 16 bit arithmetic mode, the offset value, located in the offset field, should be the needed offset pre-incremented by the user by a bias of 16.

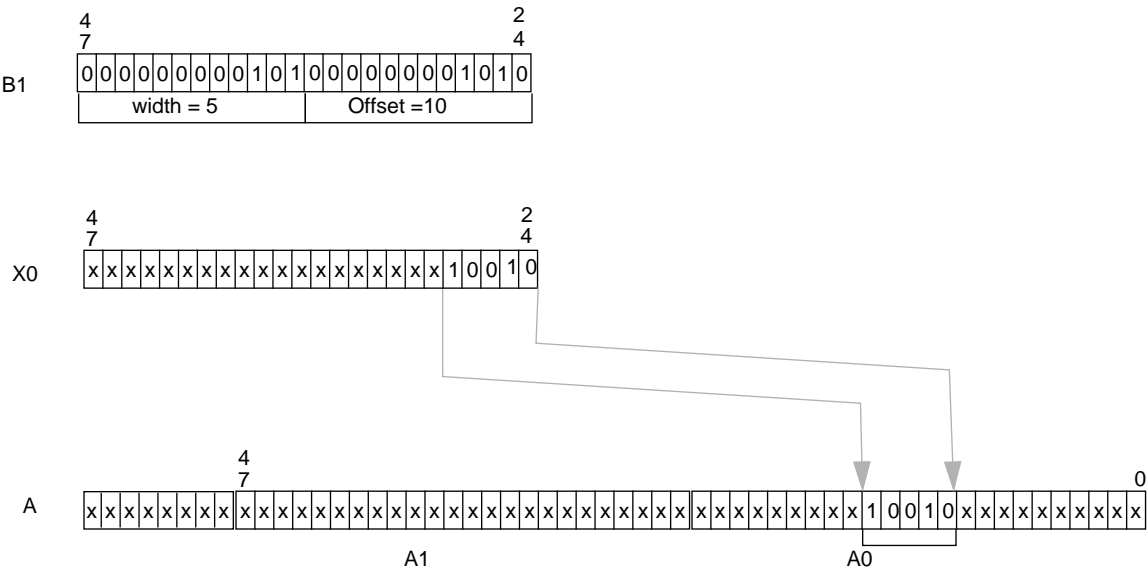
2) If offset + width exceeds the value of 56, the result will be undefined.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
X	X	✓	✓	✓	✓	●	●
CCR							

- V Always cleared
- C Always cleared
- x This bit is unchanged by the instruction
- ✓ This bit is changed according to the standard definition

Example: INSERT B1,X0,A



Instruction Formats and opcodes:

		23	16 15								8 7				0																										
INSERT	S1,S2,D	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>								0	0	0	0	1	1	0	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>				0	0	0	1	1	0	1	1	<table border="1"><tr><td>0</td><td>q</td><td>q</td><td>q</td><td>S</td><td>S</td><td>S</td><td>D</td></tr></table>				0	q	q	q	S	S	S	D
0	0	0	0	1	1	0	0																																		
0	0	0	1	1	0	1	1																																		
0	q	q	q	S	S	S	D																																		
		23	16 15								8 7				0																										
INSERT	#CO,S2,D	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>								0	0	0	0	1	1	0	0	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>				0	0	0	1	1	0	0	1	<table border="1"><tr><td>0</td><td>q</td><td>q</td><td>q</td><td>0</td><td>0</td><td>0</td><td>D</td></tr></table>				0	q	q	q	0	0	0	D
0	0	0	0	1	1	0	0																																		
0	0	0	1	1	0	0	1																																		
0	q	q	q	0	0	0	D																																		
		<table border="1"><tr><td colspan="16">CONTROL WORD EXTENSION</td></tr></table>																CONTROL WORD EXTENSION																							
CONTROL WORD EXTENSION																																									

Instruction Fields:

{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S1}	SSS	Control register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)
{S2}	qqq	Source register [X0,X1,Y0,Y1,A0,B0] (see Table A-15 on page A-240)
{#CO}		Control word extension.

Jcc

Jcc

Jump Conditionally

Operation:

If cc, then 0xxx → PC
 else PC+1 → PC

If cc, then ea → PC
 else PC+1 → PC

Assembler Syntax:

Jcc xxx

Jcc ea

Description: Jump to the location in program memory given by the instruction's effective address if the specified condition is true. If the specified condition is false, the program counter (PC) is incremented and the effective address is ignored. However, the address register specified in the effective address field is always updated independently of the specified condition. All memory alterable addressing modes 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 conditions that the term “cc” can specify are listed on Table A-42 on page A-250.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23	16 15								8 7								0					
Jcc	xxx	0	0	0	0	1	1	1	0	C	C	C	C	a	a	a	a	a	a	a	a	a	a	a

		23	16 15								8 7								0								
Jcc	ea	0	0	0	0	1	0	1	0	1	1	M	M	M	R	R	R	1	0	1	0	C	C	C	C		
		OPTIONAL EFFECTIVE ADDRESS EXTENSION																									

Instruction Fields:

{cc}	CCCC	Condition code (see Table A-43 on page A-251)
{xxx}	aaaaaaaaaaaa	Short Jump Address
{ea}	MMRRRR	Effective Address (see Table A-18 on page A-241)

A-6.47 Jump if Bit Clear (JCLR)

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

Assembler Syntax:

JCLR #n,[X or Y]:ea,xxxx

JCLR #n,[X or Y],aa,xxxx

JCLR #n,[X or Y]:pp,xxxx

JCLR #n,[X or Y]:qq,xxxx

JCLR #n,S,xxxx

Description: Jump to the 24-bit absolute address in program memory specified in the instruction's 24-bit extension word if the n^{th} bit of the source operand S is clear. The bit to be tested is selected by an immediate bit number from 0–23. If the specified memory bit is not clear, the program counter (PC) is incremented and the absolute address in the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the n^{th} bit. All address register indirect addressing modes may be used to reference the source operand S. Absolute Short and I/O Short addressing modes may also be used.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
x This bit is unchanged by the instruction

Instruction Formats and opcodes:

JCLR	#n,[X or Y]:ea,xxxx	<div> <div>231615870</div> <div>00001010001MMMMRRR1S0bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>
JCLR	#n,[X or Y]:aa,xxxx	<div> <div>231615870</div> <div>00001010000aaaaaa1S0bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>
JCLR	#n,[X or Y]:pp,xxxx	<div> <div>231615870</div> <div>00001010010pppppp1S0bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>
JCLR	#n,[X or Y]:qq,xxxx	<div> <div>231615870</div> <div>0000000110qqqqqq1S0bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>
JCLR	#n,S,xxxx	<div> <div>231615870</div> <div>00001010011DDDDDD000bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMRRRR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{xxxx}		24-bit absolute Address extension word
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{S}	DDDDDD	Source register [all on-chip registers] (see Table A-22 on page A-243)

A-6.48 Jump (JMP)

JMP

JMP

Jump

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23								16 15								8 7								0							
JMP	ea	0	0	0	0	1	0	1	0	1	1	M	M	M	R	R	R	1	0	0	0	0	0	0	0	0							
		OPTIONAL EFFECTIVE ADDRESS EXTENSION																															

		23								16 15				8 7				0							
JMP	xxx	0	0	0	0	1	1	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a	a	a

Instruction Fields:

{xxx} aaaaaaaaaaaa Short Jump Address
{ea} MMMRRR Effective Address (see Table A-18 on page A-241)

A-6.49 Jump to Subroutine Conditionally (JScC)

JScC

JScC

Jump to Subroutine Conditionally

Operation:

If *cc*, then $SP+1 \rightarrow SP$; $PC \rightarrow SSH$; $SR \rightarrow SSL$; $0xxx \rightarrow PC$ JScC *xxx*
else $PC+1 \rightarrow PC$

Assembler Syntax:

If *cc*, then $SP+1 \rightarrow SP$; $PC \rightarrow SSH$; $SR \rightarrow SSL$; $ea \rightarrow PC$ JScC *ea*
else $PC+1 \rightarrow PC$

Description: Jump to the subroutine whose location in program memory is given by the instruction's effective address if the specified condition is true. If the specified condition is true, the address of the instruction immediately following the JScC instruction (PC) and the 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 conditions that the term “**cc**” can specify are listed on Table A-42 on page A-250.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23	16 15								8 7								0					
JScC	xxx	0	0	0	0	1	1	1	1	C	C	C	C	a	a	a	a	a	a	a	a	a	a	a

		23	16 15								8 7								0						
JScC	ea	0	0	0	0	1	0	1	1	1	1	M	M	M	R	R	R	1	0	1	0	C	C	C	C
		OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

Instruction Fields:

{cc}	CCCC	Condition code (see Table A-43 on page A-251)
{xxx}	aaaaaaaaaaaa	Short Jump Address
{ea}	MMMRRR	Effective Address (see Table A-18 on page A-241)

JSCLR

JSCLR

Jump to Subroutine if Bit Clear

Operation:

```
If S{n}=0 then SP+1→SP;PC →SSH;SR →SSL;
                ;xxxx →PC
                else PC+1→PC
```

```
If S{n}=0 then SP+1→SP;PC →SSH;SR →SSL;
                ;xxxx →PC
                else PC+1→PC
```

```
If S{n}=0 then SP+1→SP;PC →SSH;SR →SSL;
                ;xxxx →PC
                else PC+1→PC
```

```
If S{n}=0 then SP+1→SP;PC →SSH;SR →SSL;
                ;xxxx →PC
                else PC+1→PC
```

```
If S{n}=0 then SP+1→SP;PC →SSH;SR →SSL;
                ;xxxx →PC
                else PC+1→PC
```

Assembler Syntax:

```
JSCLR    #n,[X or Y]:ea,xxxx
```

```
JSCLR    #n,[X or Y],aa,xxxx
```

```
JSCLR    #n,[X or Y]:pp,xxxx
```

```
JSCLR    #n,[X or Y]:qq,xxxx
```

```
JSCLR    #n,S,xxxx
```

Description: Jump to the subroutine at the 24-bit absolute address in program memory specified in the instruction's 24-bit extension word if the n^{th} bit of the source operand S is clear. The bit to be tested is selected by an immediate bit number from 0–23. If the n^{th} bit of the source operand S is clear, the address of the instruction immediately following the JSCLR instruction (PC) and the system status register (SR) are pushed onto the system stack. Program execution then continues at the specified absolute address in the instruction's 24-bit extension word. If the 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 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	×	×	×	×	×	×
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0																		
JSCLR #n,[X or Y]:ea,xxxx	0	0	0	0	1	0	1	1	0	1	M	M	M	R	R	R	1	S	0	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

	23	16	15	8	7	0																		
JSCLR #n,[X or Y]:aa,xxxx	0	0	0	0	1	0	1	1	0	0	a	a	a	a	a	a	1	S	0	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

	23	16	15	8	7	0																		
JSCLR #n,[X or Y]:pp,xxxx	0	0	0	0	1	0	1	1	1	0	p	p	p	p	p	p	1	S	0	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

	23	16	15	8	7	0																		
JSCLR #n,[X or Y]:qq,xxxx	0	0	0	0	0	0	0	1	1	1	q	q	q	q	q	q	1	S	0	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

	23	16	15	8	7	0																		
JSCLR #n,S,xxxx	0	0	0	0	1	0	1	1	1	1	D	D	D	D	D	D	0	0	0	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMMRRR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{xxxx}		24-bit absolute Address extension word
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{S}	DDDDDD	Source register [all on-chip registers] (see Table A-22 on page A-243)

JSET

JSET

Jump if Bit Set

Operation:

If $S\{n\}=1$ then xxxx \rightarrow PC
 else PC+ 1 \rightarrow PC

If $S\{n\}=1$ then xxxx \rightarrow PC
 else PC+ 1 \rightarrow PC

If $S\{n\}=1$ then xxxx \rightarrow PC
 else PC+ 1 \rightarrow PC

If $S\{n\}=1$ then xxxx \rightarrow PC
 else PC+ 1 \rightarrow PC

If $S\{n\}=1$ then xxxx \rightarrow PC
 else PC+ 1 \rightarrow PC

Assembler Syntax:

JSET #n,[X or Y]:ea,xxxx

JSET #n,[X or Y],aa,xxxx

JSET #n,[X or Y]:pp,xxxx

JSET #n,[X or Y]:qq,xxxx

JSET #n,S,xxxx

Description: Jump to the 24-bit absolute address in program memory specified in the instruction's 24-bit extension word if the n^{th} bit of the source operand S is set. The bit to be tested is selected by an immediate bit number from 0–23. If the specified memory bit is not set, the program counter (PC) is incremented, and the absolute address in the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the n^{th} bit. All address register indirect addressing modes may be used to reference the source operand S. Absolute short and I/O short addressing modes may also be used.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
 x This bit is unchanged by the instruction

Instruction Formats and opcodes:

JSET	#n,[X or Y]:ea,xxxx	<div> <div>231615870</div> <div>00001010001MMMMRRR1S1bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>
JSET	#n,[X or Y]:aa,xxxx	<div> <div>231615870</div> <div>00001010000aaaaaaa1S1bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>
JSET	#n,[X or Y]:pp,xxxx	<div> <div>231615870</div> <div>00001010010ppppppp1S1bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>
JSET	#n,[X or Y]:qq,xxxx	<div> <div>231615870</div> <div>0000000110qqqqqqq1S1bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>
JSET	#n,S,xxxx	<div> <div>231615870</div> <div>00001010011DDDDDD001bbbbbb</div> <div>ABSOLUTE ADDRESS EXTENSION</div> </div>

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMRRR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{xxxx}		24-bit Absolute Address in extension word
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFF80-\$FFFFBF]
{S}	DDDDDD	Source register [all on-chip registers] (see Table A-22 on page A-243)

JSR

JSR

Jump to Subroutine

Operation:

SP+1→SP; PC→SSH; SR→SSL; 0xxx→PC

SP+1→SP; PC→SSH; SR→SSL; ea→PC

Assembler Syntax:

JSR xxx

JSR ea

Description: Jump to the subroutine whose location in program memory is given by the instruction's effective address. The address of the instruction immediately following the JSR instruction (PC) and the system status register (SR) is pushed onto the system stack. Program execution then continues at the specified effective address in program memory. All memory alterable addressing modes 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23	16 15								8 7								0						
JSR	ea	0 0 0 0 1 0 1 1								1 1 M M M R R R								1 0 0 0 0 0 0 0							
		OPTIONAL EFFECTIVE ADDRESS EXTENSION																							
		23	16 15								8 7								0						
JSR	xxx	0 0 0 0 1 1 0 1								0 0 0 0 a a a a								a a a a a a a a							

Instruction Fields:

{xxx}	aaaaaaaaaaaa	Short Jump Address
{ea}	MMRRRR	Effective Address (see Table A-18 on page A-241)

JSSET

JSSET

Jump to Subroutine if Bit Set

Operation:

```

If  S{n}=1  then  SP+1→SP;PC →SSH;SR →SSL;
                    ;xxx →PC
else PC+1→PC

```

Assembler Syntax:

JSSET #n,[X or Y]:ea,xxxx

```

If  S{n}=1  then  SP+1→SP;PC →SSH;SR →SSL;
                    ;xxxx →PC
else  PC+1→PC

```

JSSET #n,[X or Y],aa,xxxx

```

If  S{n}=1  then  SP+1→SP;PC →SSH;SR →SSL;
                    ;xxxx →PC
else PC+1→PC

```

JSSET #n,[X or Y]:pp,xxxx

```

If  S{n}=1  then  SP+1→SP;PC →SSH;SR →SSL;
                    ;xxxx →PC
else  PC+1→PC

```

JSSET #n,[X or Y]:qq,xxxx

```

If  S{n}=1  then  SP+1→SP;PC →SSH;SR →SSL;
                    ;xxxx →PC
else PC+1→PC

```

JSSET #n,S,xxx

Description: Jump to the subroutine at the 24-bit absolute address in program memory specified in the instruction's 24-bit extension word if the n^{th} bit of the source operand S is set. The bit to be tested is selected by an immediate bit number from 0–23. If the n^{th} bit of the source operand S is set, the address of the instruction immediately following the JSSET instruction (PC) and the system status register (SR) are pushed onto the system stack. Program execution then continues at the specified absolute address in the instruction's 24-bit extension word. If the specified memory bit is not set, the program counter (PC) is incremented, and the extension word is ignored. However, the address register specified in the effective address field is always updated independently of the state of the n^{th} bit. All address register indirect addressing modes may be used to reference the source operand S. Absolute short and I/O short addressing modes may also be used.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	×	×	×	×	×	×
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0																		
JSSET #n,[X or Y]:ea,xxxx	0	0	0	0	1	0	1	1	0	1	M	M	M	R	R	R	1	S	1	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

	23	16	15	8	7	0																		
JSSET #n,[X or Y]:aa,xxxx	0	0	0	0	1	0	1	1	0	0	a	a	a	a	a	a	1	S	1	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

	23	16	15	8	7	0																		
JSSET #n,[X or Y]:pp,xxxx	0	0	0	0	1	0	1	1	1	0	p	p	p	p	p	p	1	S	1	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

	23	16 15								8 7								0						
JSSET #n,[X or Y]:qq,xxxx	0	0	0	0	0	0	0	1	1	1	q	q	q	q	q	q	1	S	1	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

	23	16 15								8 7								0						
JSSET #n,S,xxxx	0	0	0	0	1	0	1	1	1	1	D	D	D	D	D	D	0	0	1	b	b	b	b	b
	ABSOLUTE ADDRESS EXTENSION																							

Instruction Fields:

{#n}	bbbbbb	Bit number [0-23]
{ea}	MMMRRR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{xxxx}		24-bit PC absolute Address extension word
{aa}	aaaaaa	Absolute Address [0-63]
{pp}	pppppp	I/O Short Address [64 addresses: \$FFFC0-\$FFFFFF]
{qq}	qqqqqq	I/O Short Address [64 addresses: \$FFFF80-\$FFFFBF]
{S}	DDDDDD	Source register [all on-chip registers] (see Table A-22 on page A-243)

A-6.54 Load PC Relative Address (LRA)

LRA

LRA

Load PC Relative Address

Operation: $PC + R_n \rightarrow D$ $PC + \text{xxxx} \rightarrow D$ **Assembler Syntax:**LRA R_n, D LRA xxxx, D

Description: The PC is added to the specified displacement and the result is stored in destination D. The displacement is a 2's complement 24-bit integer that represents the relative distance from the current PC to the destination PC. Long Displacement and Address Register PC Relative addressing modes may be used. Note that if D is SSH, the SP will be preincremented by one.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

LRA	Rn,D	23	16 15								8 7								0						
		0	0	0	0	0	1	0	0	1	1	0	0	0	R	R	R	0	0	0	d	d	d	d	d
LRA	xxxx,D	23	16 15								8 7								0						
		0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	d	d	d	d	d
		LONG DISPLACEMENT																							

Instruction Fields:

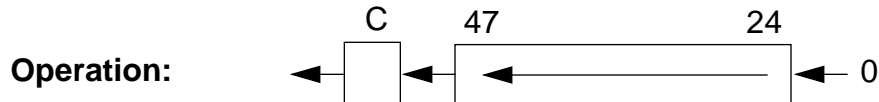
{Rn}	RRR	Address register [R0-R7]
{D}	dddd	Destination address register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0-R7,N0-N7] (see Table A-31 on page A-245)
{xxxx}		24-bit PC Long Displacement

A-6.55 Logical Shift Left (LSL)

LSL

LSL

Logical Shift Left



Assembler Syntax:

LSL D (parallel move)

LSL #ii,D

LSL S,D

Description:

Single-bit shift:

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.

Multi-bit shift:

The contents of bits 47-24 of the destination accumulator D are shifted left #ii bits. Bits shifted out of position 47 are lost, but for the last bit which is latched in the carry bit. Zeros are supplied to the vacated positions on the right. The result is placed into bits 47-24 of the destination accumulator D. The number of bits to shift is determined by the 5-bit immediate field in the instruction, or by the unsigned integer located in the control register S. If a zero shift count is specified, the carry bit is cleared.

This is a 24 bit operation. The remaining bits of the destination accumulator are not affected.

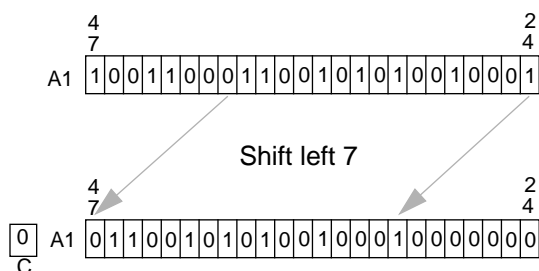
Note: The number of shifts should not exceed the value of 24.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	●	●	●	●
CCR							

- N Set if bit 47 of the result is set
 - Z Set if bits 47-24 of the result are zero
 - V Always cleared
 - C Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero.
- x This bit is unchanged by the instruction

Example: LSL #7, A



Instruction Formats and opcodes:

LSL	D	23	8	7	0										
		DATA BUS MOVE FIELD						0	0	1	1	D	0	1	1
		OPTIONAL EFFECTIVE ADDRESS EXTENSION													

LSL	#ii,D	23	16	15	8	7	0																		
		0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	1	0	i	i	i	i	i	D

LSL	S,D	23	16	15	8	7	0																		
		0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	s	s	s	D

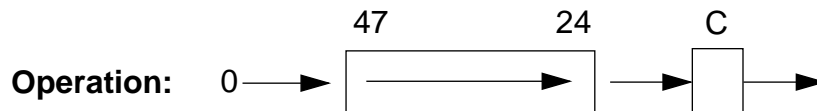
Instruction Fields:

{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S}	sss	Control register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)
{#ii}	iiii	5bit unsigned integer [0-23] denoting the shift amount

LSR

LSR

Logical Shift Right



Assembler Syntax:

LSR D (parallel move)

LSR #ii,D

LSR S,D

Description:

Single-bit shift:

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.

Multi-bit shift:

The contents of bits 47-24 of the destination accumulator D are shifted right #ii bits. Bits shifted out of position 24 are lost, but for the last bit which is latched in the carry bit. Zeros are supplied to the vacated positions on the left. The result is placed into bits 47-24 of the destination accumulator D. The number of bits to shift is determined by the 5-bit immediate field in the instruction, or by the unsigned integer located in the control register S. If a zero shift count is specified, the carry bit is cleared.

This is a 24 bit operation. The remaining bits of the destination register are not affected.

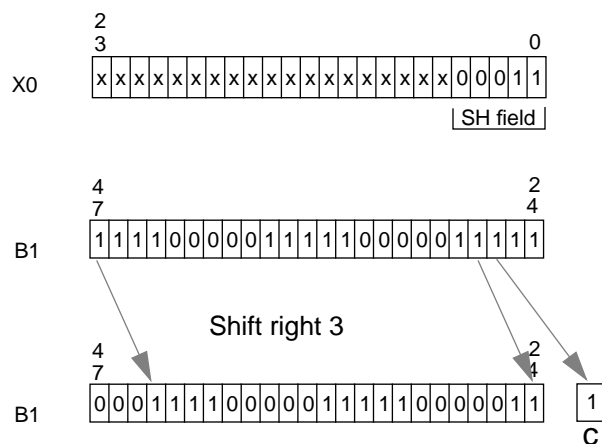
Note: The number of shifts should not exceed the value of 24.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	●	●	●	●
CCR							

- N Set if bit 47 of the result is set
- Z Set if bits 47-24 of the result are zero
- V Always cleared
- C Set if the last bit shifted out of the operand is set. Cleared otherwise. Cleared for a shift count of zero
- x This bit is unchanged by the instruction

Example: LSR X0,B



Instruction Formats and opcodes:

		23		8	7		0					
LSR	D	DATA BUS MOVE FIELD			0	0	1	0	D	0	1	1
		OPTIONAL EFFECTIVE ADDRESS EXTENSION										

		23		16	15		8	7		0																																							
LSR	#ii,D	<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>i</td><td>i</td><td>i</td><td>i</td><td>i</td><td>D</td></tr></table>																								0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	1	1	i	i	i	i	i	D
0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	1	1	i	i	i	i	i	D																										

		23	16 15							8 7							0					
LSR	S,D	0 0 0 0 1 1 0 0							0 0 0 1 1 1 1 0							0 0 1 1 s s s D						

Instruction Fields:

{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S}	sss	Control register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)
{#ii}	iiii	5 bit unsigned integer [0-23] denoting the shift amount

A-6.57 Load Updated Address (LUA)

LUA

LUA

Load Updated address

Operation:

ea→D (No update performed)

Rn+aa→D

ea→D (No update performed)

Rn+aa→D

Assembler Syntax:

LUA ea,D

LUA (Rn+aa),D

LEA ea,D

LEA (Rn+aa),D

Description: Load the updated address into the destination address register D. The source address register and the update mode used to compute the updated address are specified by the effective address (ea). **Note that the source address register specified in the effective address is not updated. This is the only case where an address register is not updated although stated otherwise in the effective address mode bits.** Only the following addressing modes may be used: Post+N, Post-N, Post+1, Post-1.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

		23	16 15								8 7								0						
LUA/ LEA	ea,D	0	0	0	0	0	1	0	0	0	1	0	M	M	R	R	R	0	0	0	d	d	d	d	d

		23	16 15								8 7								0						
LUA/ LEA	(Rn+aa),D	0	0	0	0	0	1	0	0	0	0	a	a	a	R	R	R	a	a	a	a	d	d	d	d

Note: LEA is a synonym for LUA. The simulator on-line disassembly will translate the opcodes into LUA.

Instruction Fields:

{ea}	MMRRR	Effective address (see Table A-20 on page A-242)
{D}	dddd	Destination address register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0-R7,N0-N7] (see Table A-31 on page A-245)
{D}	dddd	Destination address register [R0-R7,N0-N7] (see Table A-25 on page A-244)
{aa}	aaaaaa	7-bit sign extended short displacement address
{Rn}	RRR	Source address register [R0-R7]

Note: **RRR** refers to a **source** address register (R0-R7), while **dddd/ddddd** refer to a **destination** address register R0-R7 or N0-N7.

MAC

MAC

Signed Multiply Accumulate

Operation:

$D \pm S1 * S2 \rightarrow D$ (parallel move)

$D \pm S1 * S2 \rightarrow D$ (parallel move)

$D \pm (S1 * 2^{-n}) \rightarrow D$ (**no** parallel move)

Assembler Syntax:

MAC $(\pm)S1, S2, D$ (parallel move)

MAC $(\pm)S2, S1, D$ (parallel move)

MAC $(\pm)S, \#n, D$ (**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:

MAC X1, Y0, A MAC X1, Y0, B

MAC X0, Y1, A MAC X0, Y1, B

MAC Y1, X1, A MAC Y1, X1, B

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes 1:

	23	16	15	8	7	0						
MAC (\pm)S1,S2,D	DATA BUS MOVE FIELD				1	Q	Q	Q	d	k	1	0
MAC (\pm)S2,S1,D	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

Instruction Fields:

- {S1,S2} QQQ** Source registers S1,S2
[X0*X0,Y0*Y0,X1*X0,Y1*Y0,X0*Y1,Y0*X0,X1*Y0,Y1*X1]
(see Table A-26 on page A-244)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)

Instruction Formats and opcode 2:

	23	16 15						8 7						0											
MAC	(±)S,#n,D	0	0	0	0	0	0	0	1	0	0	0	s	s	s	s	s	1	1	Q	Q	d	k	1	0

Instruction Fields:

- {S} QQ** Source register [Y1,X0,Y0,X1]] (see Table A-27 on page A-244)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)
- {#n} sssss** Immediate operand (see Table A-32 on page A-246)

MACI

MACI

Signed Multiply-Accumulate with Immediate Operand

Operation: $D \pm \#xxxxxx * S \rightarrow D$ **Assembler Syntax:**MACI (\pm)#xxxxxx,S,D

Description: Multiply the two signed 24-bit source operands #xxxxxx and S 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 “+”.

condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcode:

		23	16								15	8								7	0							
MACI	(±)#xxxxxx,S,D	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	q	q	d	k	1	0		
		IMMEDIATE DATA EXTENSION																										

Instruction Fields:

{S}	qq	Source register [X0,Y0,X1,Y1] (see Table A-28 on page A-244)
{D}	d	Destination accumulator [A,B] (see Table A-10 on page A-239)
{ \pm }	k	Sign [+,-] (see Table A-29 on page A-244)
#xxxxxx		24-bit Immediate Long Data extension word

A-6.60 Mixed Multiply-Accumulate (MAC su/uu)

MAC(su,uu) MAC(su,uu)

Mixed Multiply Accumulate

Operation:

$D \pm S1 * S2 \rightarrow D$ (S1 unsigned, S2 unsigned) MACuu (\pm)S1,S2,D (no parallel move)

$D \pm S1 * S2 \rightarrow D$ (S1 signed, S2 unsigned) MACsu (\pm)S2,S1,D (no parallel move)

Description: Multiply the two 24-bit source operands S1 and S2 and add/subtract the product to/from the specified 56-bit destination accumulator D. One or two of the source operands can be unsigned. The “–” sign option is used to negate the specified product prior to accumulation. The default sign option is “+”.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

MAC _{su} (\pm)S1,S2,D	23	16				15				8				7				0							
MAC _{uu} (\pm)S1,S2,D	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	1	s	d	k	Q	Q	Q	Q

Instruction Fields:

- {S1,S2} QQQQ** Source registers S1,S2 [all combinations of X0,X1,Y0 and Y1]
(see Table A-30 on page A-245)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)
- {s}** [ss,us] (see Table A-40 on page A-249)

MACR

MACR

Signed Multiply Accumulate and Round

Operation:

$D \pm S1 * S2 + r \rightarrow D$ (parallel move)

$D \pm S1 * S2 + r \rightarrow D$ (parallel move)

$D \pm (S1 * 2^{-n}) + r \rightarrow D$ (**no** parallel move)

Assembler Syntax:

MACR $(\pm)S1, S2, D$ (parallel move)

MACR $(\pm)S2, S1, D$ (parallel move)

MACR $(\pm)S, \#n, D$ (**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 either convergent or two's complement rounding. The rounded result is stored in the destination accumulator D.

The “–” sign option negates the specified product prior to accumulation. The default sign option is “+”.

The contribution of the LS bits of the result is rounded into the upper portion of the destination accumulator. Once rounding has been completed, the LS bits of the destination accumulator D are loaded with zeros to maintain an unbiased accumulator value which may be reused by the next instruction. The upper portion of the accumulator contains the rounded result which may be read out to the data buses. Refer to the RND instruction for more complete information on the rounding process.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes 1:

	23	16	15	8	7	0						
MACR (\pm)S1,S2,D	DATA BUS MOVE FIELD				1	Q	Q	Q	d	k	1	1
MACR (\pm)S2,S1,D	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

Instruction Fields:

- {S1,S2} QQQ** Source registers S1,S2
[X0*X0,Y0*Y0,X1*X0,Y1*Y0,X0*Y1,Y0*X0,X1*Y0,Y1*X1]
(see Table A-26 on page A-244)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)

Instruction Formats and opcode 2:

	23	16						15	8						7	0									
MACR	(±)	S	#	n	D																				
	0	0	0	0	0	0	0	1	0	0	0	s	s	s	s	s	1	1	Q	Q	d	k	1	1	

Instruction Fields:

- {S} QQ** Source register [Y1,X0,Y0,X1]] (see Table A-27 on page A-244)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)
- {#n} sssss** Immediate operand (see Table A-32 on page A-246)

MACRI

MACRI

Signed Multiply-Accumulate and Round with Immediate Operand

Operation:

$D \pm \#xxxxxx * S \rightarrow D$

Assembler Syntax:

MACRI $(\pm)\#xxxxxx, S, D$

Description: Multiply the two signed 24-bit source operands $\#xxxxxx$ and S , add/subtract the product to/from the specified 56-bit destination accumulator D , and then round the result using either convergent or two's complement rounding. The rounded result is stored in the destination accumulator D .

The “–” sign option negates the specified product prior to accumulation. The default sign option is “+”.

The contribution of the LS bits of the result is rounded into the upper portion of the destination accumulator. Once rounding has been completed, the LS bits of the destination accumulator D are loaded with zeros to maintain an unbiased accumulator value which may be reused by the next instruction. The upper portion of the accumulator contains the rounded result which may be read out to the data buses. Refer to the RND instruction for more complete information on the rounding process.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16	15	8	7	0																		
MACRI (±)#xxxxxx,S,D	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	q	q	d	k	1	1
	IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	qq	Source register [X0,Y0,X1,Y1] (see Table A-28 on page A-244)
{D}	d	Destination accumulator [A,B] (see Table A-10 on page A-239)
{±}	k	Sign [+,-] (see Table A-29 on page A-244)
#xxxxxx		24-bit Immediate Long Data extension word

A-6.63 Transfer by Signed Value (MAX)

MAX

MAX

Transfer by Signed Value

Operation:

If $B - A \leq 0$ then $A \rightarrow B$

Assembler Syntax:

MAX A,B (parallel move)

Description: Subtract the signed value of the source accumulator from the signed value of the destination accumulator. If the difference is negative or zero (i.e. $A \geq B$) then transfer the source accumulator to destination accumulator, otherwise do not change destination accumulator.

This is a 56 bit operation.

Note: The Carry condition code signifies that a transfer has been performed.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	●
CCR							

- C Cleared if the conditional transfer was performed. Set otherwise.
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0						
MAX A, B	DATA BUS MOVE FIELD				0	0	0	1	1	1	0	1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

A-6.64 Transfer by Magnitude (MAXM)

MAXM

MAXM

Transfer by Magnitude

Operation:

If $|B| - |A| \leq 0$ then $A \rightarrow B$

Assembler Syntax:

MAXM A,B (parallel move)

Description: Subtract the absolute value (magnitude) of the source accumulator from the absolute value of the destination accumulator. If the difference is negative or zero (i.e. $|A| \geq |B|$) then transfer the source accumulator to destination accumulator, otherwise do not change destination accumulator.

This is a 56 bit operation.

Note: The Carry condition code signifies that a transfer has been performed.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	●
CCR							

- C Cleared if the conditional transfer was performed. Set otherwise.
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
MAXM A, B	DATA BUS MOVE FIELD				0 0 0 1	0 1 0 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

MERGE

MERGE

Merge Two Half Words

Operation:

{S[11:0],D[35:24]} → D[47:24]

Assembler Syntax:

MERGE S,D

Description: The contents of bits 11-0 of the source register are concatenated to the contents of bits 35-24 of the destination accumulator. The result is stored in the destination accumulator. This instruction is a 24-bit operation. The remaining bits of the destination accumulator D are not affected.

Notes:

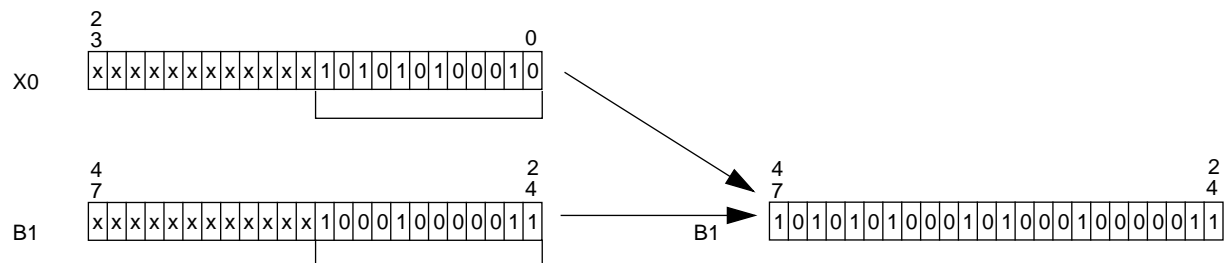
- 1) This instruction may be used in conjunction with EXTRACT or INSERT instructions to concatenate width and offset fields into a control word.
- 2) In 16 bit arithmetic mode the contents of bits 15-8 of the source register are concatenated to the contents of bits 39-32 of the destination accumulator. The result is placed in bits 47-32 of the destination accumulator.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	●	●	●	x
CCR							

- N Set if bit 47 of the result is set
- Z Set if bits 47-24 of the result are zero
- V Always cleared

Example: MERGE X0,B



Instruction Formats and opcodes:

			23					16	15					8	7									0		
MERGE	S,D		0	0	0	0	1	1	0	0	0	0	0	1	1	0	1	1	1	0	0	0	S	S	S	D

Instruction Fields:

- {D} **D** Destination accumulator [A,B] (see Table A-10 on page A-239)
- {S} **SSS** Source register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)

A-6.66 Move Data (MOVE)

MOVE

MOVE

Move Data

Operation:

S→D

Assembler Syntax:

MOVE S,D

Description: Move the contents of the specified data source S to the specified destination D. This instruction is equivalent to a data ALU NOP with a parallel data move.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0				
MOVE S,D	DATA BUS MOVE FIELD			0	0	0	0	0	0	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION									

Instruction Fields:

See **Parallel Move Descriptions** for data bus move field encoding.

Parallel Move Descriptions: Thirty of the sixty-two instructions allow an optional parallel data bus movement over the X and/or Y data bus. This allows a data ALU operation to be executed in parallel with up to two data bus moves during the instruction cycle. Ten types of parallel moves are permitted, including register to register moves, register to memory moves, and memory to register moves. However, not all addressing modes are allowed for each type of memory reference. The following section contains detailed descriptions about each type of parallel move operation.

A-6.67 NO Parallel Data Move

No Parallel Data Move

Operation:

(.....)

Assembler Syntax:

(.....)

where (.....) refers to any arithmetic or logical instruction which allows parallel moves.

Description: Many instructions in the 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23		16	15		8	7		0
(.....)	0	0	1	0	0	0	0	0	INSTRUCTION OP CODE

Instruction Format:

(defined by instruction)

A-6.68 Immediate Short Data Move (I)

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.

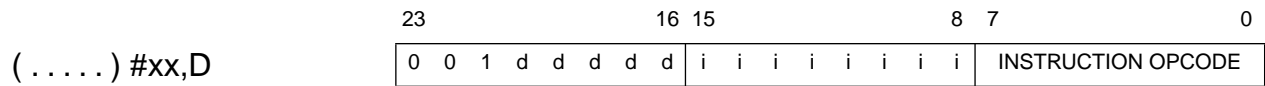
If the arithmetic or logical opcode-operand portion of the instruction specifies a given destination accumulator, that same accumulator or portion of that accumulator may not be specified as a destination D in the parallel data bus move operation. Thus, if the opcode-operand portion of the instruction 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.**

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:



Instruction Fields:

{#xx} **iiiiiii** 8-bit Immediate Short Data
{D} **dddddd** Destination register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0-R7,N0-N7] (see Table A-31 on page A-245)

R

R

Register to Register Data Move

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.**

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.

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	X	X	X	X	X	X
CCR							

- ✓ This bit is changed according to the standard definition
- ✗ This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16 15	8 7	0
(.....) S,D	0 0 1 0 0 0 e e	e e e d d d d d	INSTRUCTION OP CODE	

Instruction Fields:

- | | | |
|------------|---------------|--|
| {S} | eeeeee | Source register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0-R7,N0-N7]
(see Table A-31 on page A-245) |
| {D} | dddddd | Destination register [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0-R7,N0-N7]
(see Table A-31 on page A-245) |

A-6.70 Address Register Update (U)

U

U

Address Register Update

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23		16	15		8	7		0
(.) ea	0	0	1	0	0	0	0	0	0
	0	1	0	M	M	R	R	R	INSTRUCTION OP CODE

Instruction Fields:

{ea} **MMRRR** Effective Address (see Table A-20 on page A-242)

X:

X:

X Memory Data Move

Operation:

(.); X:ea→D

(.); X:aa→D

(.); S→X:ea

(.); S→X:aa

X:(Rn+xxx)→D

X:(Rn+xxxx)→D

D→X:(Rn+xxx)

D→X:(Rn+xxxx)

Assembler Syntax:

(.) X:ea,D

(.) X:aa,D

(.) S,X:ea

(.) S,X:aa

MOVE X:(Rn+xxx),D

MOVE X:(Rn+xxxx),D

MOVE D,X:(Rn+xxx)

MOVE D,X:(Rn+xxxx)

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.**

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

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 Formats and opcodes 1:

(.....)X:ea,D	23	16 15	8 7	0
(.....)S,X:ea	0 1 d d 0 d d d	W 1 M M M R R R	INSTRUCTION OP CODE	
(.....)#xxxxxx,D	OPTIONAL EFFECTIVE ADDRESS EXTENSION			

(.)X:aa,D	23	16 15	8 7	0
(.)S,X:aa	0 1 d d 0 d d d	W 0 a a a a a a	INSTRUCTION OP CODE	

Instruction Fields:

{ea}	MMMR	Effective Address (see Table A-16 on page A-241)
	W	Read S / Write D bit (see Table A-33 on page A-246)
{S,D}	ddddd	Source/Destination registers [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0-R7,N0-N7] (see Table A-31 on page A-245)
{aa}	aaaaaa	6-bit Absolute Short Address

Instruction Formats and opcodes 2:

	23	16 15	8 7	0
MOVE X:(Rn+xxxx),D	0 0 0 0 1 0 1 0	0 1 1 1 0 R R R	1 W D D D D D D	
MOVE S,X:(Rn+xxxx)	Rn RELATIVE DISPLACEMENT			

	23	16 15	8 7	0
MOVE X:(Rn+xxx),D	0 0 0 0 0 0 1 a	a a a a a R R R	1 a 0 W D D D D	
MOVE S,X:(Rn+xxx)				

Instruction Fields:

	W	Read S / Write D bit (see Table A-33 on page A-246)
{xxx}	aaaaaaa	7-bit sign extended Short Displacement Address
{Rn}	RRR	Address register (R0-R7)
{D}	DDDD	Source/Destination registers [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B] (see Table A-34 on page A-246)
{S,D}	DDDDDD	Source/Destination registers [all on-chip registers] (see Table A-22 on page A-243)

X:R

X:R

X Memory and Register Data Move

Operation:

Assembler Syntax:

Class I

(.); X:ea→D1; S2→D2

(.) X:ea,D1 S2,D2

(.); S1→X:ea; S2→D2

(.) S1,X:ea S2,D2

(.); #xxxxxx→D1; S2→D2

(.) #xxxxxx,D1 S2,D2

Class II

(.); A→X:ea; X0→A

(.) A,X:ea X0,A

(.); B→X:ea; X0→B

(.) B,X:ea X0,B

where (.) refers to any arithmetic or logical instruction 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Class I Instruction Formats and opcodes:

(.....) X:ea,D1 S2,D2	23	16	15	8	7	0											
(.....) S1,X:ea S2, D2	0	0	0	1	f	f	d	F	W	0	M	M	M	R	R	R	INSTRUCTION OPCODE
(.....) #xxxxxx,D1 S2,D2	OPTIONAL EFFECTIVE ADDRESS EXTENSION																

Instruction Fields:

{ea}	MMMR	RRR	Effective Address (see Table A-15 on page A-240)
	W		Read S1 / Write D1 bit (see Table A-33 on page A-246)
{S1,D1}	ff		S1/D1 register [X0,X1,A,B] (see Table A-35 on page A-247)
{S2}	d		S2 accumulator [A,B] (see Table A-10 on page A-239)
{D2}	F		D2 input register [Y0,Y1] (see Table A-35 on page A-247)

Class II Instruction Formats and opcodes:

	23	16	15	8	7	0											
(.)A→X:ea X0→A	0	0	0	0	1	0	0	d	0	0	M	M	M	R	R	R	INSTRUCTION OPCODE
(.)B→X:ea X0→B	OPTIONAL EFFECTIVE ADDRESS EXTENSION																

Instruction Fields:

{ea}	MMMR	RRR	Effective Address (see Table A-19 on page A-242)
	d		Move opcode (see Table A-37 on page A-247)

A-6.73 Y Memory Data Move (Y:)

Y:

Y:

Y Memory Data Move

Operation:

(.); Y:ea→D

(.); Y:aa→D

(.); S→Y:ea

(.); S→Y:aa

Y:(Rn+xxx)→D

Y:(Rn+xxxx)→D

D→Y:(Rn+xxx)

D→Y:(Rn+xxxx)

Assembler Syntax:

(.) Y:ea,D

(.) Y:aa,D

(.) S,Y:ea

(.) S,Y:aa

MOVE Y:(Rn+xxx),D

MOVE Y:(Rn+xxxx),D

MOVE D,Y:(Rn+xxx)

MOVE D,Y:(Rn+xxxx)

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.**

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

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 Formats and opcodes 1:

(.)Y:ea,D	23	16 15	8 7	0
(.)S,Y:ea	0 1 d d 1 d d d	W 1 M M M R R R	INSTRUCTION OP CODE	
(.)#xxxxxx,D	OPTIONAL EFFECTIVE ADDRESS EXTENSION			

(.)Y:aa,D	23	16 15	8 7	0
(.)S,Y:aa	0 1 d d 1 d d d	W 0 a a a a a a	INSTRUCTION OP CODE	

Instruction Fields:

{ea}	MMMR	Effective Address (see Table A-15 on page A-240)
	RRR	
	W	Read S / Write D bit (see Table A-33 on page A-246)
{S,D}	ddddd	Source/Destination registers
		[X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B,R0-R7,N0-N7]
		(see Table A-31 on page A-245)
{aa}	aaaaaa	Absolute Short Address

Instruction Formats and opcodes 2:

	23	16 15	8 7	0								
MOVE Y:(Rn+xxxx),D	0 0 0 0 1 0 1 1	0 1 1 1 0 R R R	1 W D D D D D D									
MOVE D,Y:(Rn+xxxx)	Rn RELATIVE DISPLACEMENT											

	23	16 15	8 7	0
MOVE Y:(Rn+xxx),D	0 0 0 0 0 0 1 a	a a a a a R R R	1 a 1 W D D D D	
MOVE D,Y:(Rn+xxx)				

Instruction Fields:

	W	Read S / Write D bit (see Table A-33 on page A-246)
{xxx}	aaaaaaa	7-bit sign extended Short Displacement Address
{Rn}	RRR	Address register (R0-R7)
{D}	DDDD	Source/Destination registers [X0,X1,Y0,Y1,A0,B0,A2,B2,A1,B1,A,B] (see Table A-34 on page A-246)
{S,D}	DDDDDD	Source/Destination registers [all on-chip registers] (see Table A-22 on page A-243)

R:Y

R:Y

Register and Y Memory Data Move

Operation:

Assembler Syntax:

Class I

(.); S1→D1; Y:ea→D2

(.) S1,D1 Y:ea,D2

(.); S1→D1; S2→Y:ea

(.) S1,D1 S2,Y:ea

(.); S1→D1; #xxxxxx→D2

(.) S1,D1 #xxxxxx,D2

Class II

(.); Y0 →A; A→Y:ea

(.) Y0,A A,Y:ea

(.); Y0→B; B→Y:ea

(.) Y0,B B,Y:ea

where (.) refers to any arithmetic or logical instruction 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Class I Instruction Formats and opcodes:

(.)S1,D1	Y:ea,D2	23	16	15	8	7	0											
(.)S1,D1	S2,Y:ea	0	0	0	1	d	e	f	f	W	1	M	M	M	R	R	R	INSTRUCTION OPCODE
(.)S1,D1	#xxxxxxx,D2	OPTIONAL EFFECTIVE ADDRESS EXTENSION																

Instruction Fields :

{ea}	MMMR	Effective Address (see Table A-15 on page A-240)
	RR	
	W	Read S2 / Write D2 bit (see Table A-33 on page A-246)
{S1}	d	S1 accumulator [A,B] (see Table A-10 on page A-239)
{D1}	e	D1 input register [X0,X1] (see Table A-36 on page A-247)
{S2,D2}	ff	S2/D2 register [Y0,Y1,A,B] (see Table A-36 on page A-247)

Class II Instruction Formats and opcodes:

	23	16	15	8	7	0											
(.)Y0 → A A → Y:ea	0	0	0	0	1	0	0	d	1	0	M	M	M	R	R	R	INSTRUCTION OPCODE
(.)Y0 → B B → Y:ea	OPTIONAL EFFECTIVE ADDRESS EXTENSION																

Instruction Fields:

MMMRRR ea=6-bit Effective Address (see Table A-19 on page A-242)
d Move opcode (see Table A-37 on page A-247)

A-6.75 Long Memory Data Move (L:)

L:

L:

Long Memory Data Move

Operation:

(.); X:ea → D1; Y:ea → D2

(.); X:aa → D1; Y:aa → D2

(.); S1 → X:ea; S2 → Y:ea

(.); S1 → X:aa; S2 → Y:aa

Assembler Syntax:

(.) L:ea,D

(.) L:aa,D

(.) S,L:ea

(.) 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

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 Formats and opcodes:

	23	16	15	8	7	0											
(.....)L:ea,D	0	1	0	0	L	0	L	L	W	1	M	M	M	R	R	R	INSTRUCTION OPCODE
(.....)S,L:ea	OPTIONAL EFFECTIVE ADDRESS EXTENSION																

	23	16	15	8	7	0											
(.....)L:aa,D	0	1	0	0	L	0	L	L	W	0	a	a	a	a	a	a	INSTRUCTION OPCODE
(.....)S,L:aa																	

Instruction Fields:

- {ea} **MMMR** Effective Address (see Table A-18 on page A-241)
- W** Read S / Write D bit (see Table A-33 on page A-246)
- {L} **LLL** Two data ALU registers (see Table A-23 on page A-243)
- {aa} **aaaaaa** Absolute Short Address

X: Y:

X: Y:

XY Memory Data Move

Operation:

(.); X:<eax> → D1; Y:<eay> → D2

(.); X:<eax> → D1; S2 → Y:<eay>

(.); S1 → X:<eax>; Y:<eay> → D2

(.); S1 → X:<eax>; S2 → Y:<eay>

Assembler Syntax:

(.) X:<eax>,D1 Y:<eay>,D2

(.) X:<eax>,D1 S2,Y:<eay>

(.) S1,X:<eax> Y:<eay>,D2

(.) 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 instruction specifies an access to an internal X-I/O and internal Y-I/O modules (reflected by the address of the X memory space and of the Y memory space), then only the access to the internal X-I/O module will be executed. The access to the Y-I/O module will be discarded.

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

(.)X:<eax>,D1 Y:<eay>,D2

(.)X:<eax>,D1 S2,Y:<eay>

(.)S1,X:<eax> Y:<eay>,D2

(.)S1,X:<eax> S2,Y:<eay>

23	16	15	8	7	0
1	w	m	m	e	e
f	f	W	r	r	M
		M	R	R	R
INSTRUCTION OP CODE					

Instruction Fields :

{<eax>} **MMRRR** 5-bit X Effective Address (R0–R3 or R4–R7)

{<eay>} **mmrr** 4-bit Y Effective Address (R4–R7 or R0–R3)

{S1,D1} **ee** S1/D1 register [X0,X1,A,B]

{S2,D2} **ff** S2/D2 register [Y0,Y1,A,B]

MMRRR,mmrr,ee,ff: see Table A-38 on page A-248

W X move Operation Control (See Table A-33 on page A-246)

w Y move Operation Control (See Table A-33 on page A-246)

MOVEC

MOVEC

Move Control Register

Operation:

[X or Y]:ea→D1

[X or Y]:aa→D1

S1→[X or Y]:ea

S1→[X or Y]:aa

S1→D2

S2→D1

#xxxx→D1

#xx→D1

Assembler Syntax:

MOVE(C) [Xor Y]:ea,D1

MOVE(C) [Xor Y]:aa,D1

MOVE(C) S1,[X or Y]:ea

MOVE(C) S1,[X or Y]:aa

MOVE(C) S1,D2

MOVE(C) S2,D1

MOVE(C) #xxxx,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. All 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	●	●	●	●	●	●
CCR							

For D1 or D2=SR operand :

- S 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 Set if data growth been detected
- L Set if data limiting has occurred during the move

Instruction Formats and opcodes:

MOVE(C) [X or Y]:ea,D1	23	16	15	8	7	0																		
MOVE(C) S1,[X or Y]:ea	0	0	0	0	0	1	0	1	W	1	M	M	M	R	R	R	O	S	1	d	d	d	d	d
MOVE(C) #xxxx,D1	OPTIONAL EFFECTIVE ADDRESS EXTENSION																							

MOVE(C) [X or Y]:aa,D1	23	16 15								8 7								0						
MOVE(C) S1,[X or Y]:aa	0	0	0	0	0	1	0	1	W	0	a	a	a	a	a	a	0	S	1	d	d	d	d	d

MOVE(C) S1,D2	23	16 15								8 7								0						
MOVE(C) S2,D1	0	0	0	0	0	1	0	0	W	1	e	e	e	e	e	e	1	0	1	d	d	d	d	d

	23	16 15								8 7								0					
MOVE(C) #xx,D1	0	0	0	0	0	1	0	1	i	i	i	i	i	i	i	1	0	1	d	d	d	d	d

Instruction Fields:

{ea}	MMMR	Effective Address (see Table A-15 on page A-240)
	W	Read S / Write D bit (see Table A-33 on page A-246)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{S1,D1}	ddddd	Program Controller register [M0-M7,EP,VBA,SZ,SR,OMR,SP,SSH,SSL,LA,LC] (see Table A-41 on page A-249)
{aa}	aaaaaa	aa=6-bit Absolute Short Address
{S2,D2}	eeeeee	S2/D2 register [all on-chip registers] (see Table A-22 on page A-243)
{#xx}	iiiiiii	#xx=8-bit Immediate Short Data

MOVEM

MOVEM

Move Program Memory

Operation:

S→P:ea

S→P:aa

P:ea→D

P:aa→D

Assembler Syntax:

MOVE(M) S,P:ea

MOVE(M) S,P:aa

MOVE(M) P:ea,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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	●	●	●	●	●	●
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 Set if data growth been selected
- L Set if data limiting has occurred during the move

Instruction Formats and opcodes:

	23	16	15	8	7	0																	
MOVE(M) S,P:ea	0	0	0	0	0	1	1	1	W	1	M	M	M	R	R	R	1	0	d	d	d	d	d
MOVE(M) P:ea,D	OPTIONAL EFFECTIVE ADDRESS EXTENSION																						

MOVE(M) S,P:aa	23	16 15							8 7							0								
MOVE(M) P:aa,D	0	0	0	0	0	1	1	1	W	0	a	a	a	a	a	a	0	0	d	d	d	d	d	d

Instruction Fields:

{ea}	MMMR	RRR	Effective Address (see Table A-18 on page A-241)
	W		Read S / Write D bit (see Table A-33 on page A-246)
{ S,D}	dddddd		Source/Destination register [all on-chip registers] (see Table A-22 on page A-243)
{aa}	aaaaaa		Absolute Short Address

MOVEP

MOVEP

Move Peripheral Data

Operation:

[X or Y]:pp → D

[X or Y]:qq → D

[X or Y]:pp → [X or Y]:ea

[X or Y]:qq → [X or Y]:ea

[X or Y]:pp → P:ea

[X or Y]:qq → P:ea

S → [X or Y]:pp

S → [X or Y]:qq

[X or Y]:ea → [X or Y]:pp

[X or Y]:ea → [X or Y]:qq

P:ea → [X or Y]:pp

P:ea → [X or Y]:qq

Assembler Syntax:

MOVEP [X or Y]:pp,D

MOVEP [X or Y]:qq,D

MOVEP [X or Y]:pp,[X or Y]:ea

MOVEP [X or Y]:qq,[X or Y]:ea

MOVEP [X or Y]:pp,P:ea

MOVEP [X or Y]:qq,P:ea

MOVEP S,[X or Y]:pp

MOVEP S,[X or Y]:qq

MOVEP [X or Y]:ea,[X or Y]:pp

MOVEP [X or Y]:ea,[X or Y]:qq

MOVEP P:ea,[X or Y]:pp

MOVEP P:ea,[X or Y]:qq

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. ALL the I/O space (\$FFFF80-\$FFFFFF) can be accessed, except for the P: reference opcode.

If the system stack register SSH is specified as a source operand, the system stack pointer (SP) is 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	●	●	●	●	●	●
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 Set if data growth been selected
- L Set if data limiting has occurred during the move

Instruction Formats and opcodes:

X: or Y: Reference (high I/O address)

	23	16 15	8 7	0
MOVEP [X or Y]:pp,[X or Y]:ea	0 0 0 0 1 0 0 s	W 1 M M M R R R	1 S p p p p p p	
MOVEP [X or Y]:ea,[X or Y]:pp	OPTIONAL EFFECTIVE ADDRESS EXTENSION			

X: or Y: Reference (low I/O address)

	23	16 15	8 7	0
MOVEP X:qq,[X or Y]:ea	0 0 0 0 0 1 1 1	W 1 M M M R R R	0 S q q q q q q	
MOVEP [X or Y]:ea,X:qq	OPTIONAL EFFECTIVE ADDRESS EXTENSION			

X: or Y: Reference (low I/O address)

	23	16 15	8 7	0
MOVEP Y:qq,[X or Y]:ea	0 0 0 0 0 1 1 1	W 0 M M M R R R	1 S q q q q q q	
MOVEP [X or Y]:ea,Y:qq	OPTIONAL EFFECTIVE ADDRESS EXTENSION			

P: Reference (high I/O address)

MOVEP P:ea,[X or Y]:pp

MOVEP [X or Y]:pp,P:ea

16 15								8 7								0							
0	0	0	0	1	0	0	s	W	1	M	M	M	R	R	R	0	1	p	p	p	p	p	p

P: Reference (low I/O address)

MOVEP P:ea,[X or Y]:qq

MOVEP [X or Y]:qq,P:ea

16 15								8 7								0							
0	0	0	0	0	0	0	0	1	W	M	M	M	R	R	R	0	S	q	q	q	q	q	q

Register Reference (high I/O address)

MOVEP S,[X or Y]:pp

MOVEP [X or Y]:pp,D

23				16 15				8 7				0										
0	0	0	0	1	0	0	s	W	1	d	d	d	d	d	0	0	p	p	p	p	p	p

Register Reference: (low I/O address)

MOVEP S,X:qq

MOVEP X:qq,D

23				16 15				8 7				0											
0	0	0	0	0	1	0	0	W	1	d	d	d	d	d	d	1	q	0	q	q	q	q	q

Register Reference: (low I/O address)

MOVEP S,Y:qq

MOVEP Y:qq,D

				23	16 15				8 7				0									
0	0	0	0	0	1	0	0	W	1	d	d	d	d	d	0	q	1	q	q	q	q	q

Instruction Fields:

{ea}	MMMR	RRR	Effective Address (see Table A-16 on page A-241)
{pp}	pppppp		I/O Short Address [64 addresses: \$FFFC0-\$FFFFFF]
{qq}	qqqqqq		I/O Short Address [64 addresses: \$FFF80-\$FFFFBF]
{X/Y}	S		Memory space [X,Y] (see Table A-17 on page A-241)
{X/Y}	s		Peripheral space [X,Y] (see Table A-17 on page A-241)
	W		Read/write-peripheral (see Table A-33 on page A-246)
{S,D}	dddddd		Source/Destination register [all on-chip registers] (see Table A-22 on page A-243)

MPY**MPY****Signed Multiply****Operation:** $\pm S1 * S2 \rightarrow D$ (parallel move) $\pm S1 * S2 \rightarrow D$ (parallel move) $\pm(S1 * 2^{-n}) \rightarrow D$ (**no** parallel move)**Assembler Syntax:**MPY $(\pm)S1, S2, D$ (parallel move)MPY $(\pm)S2, S1, D$ (parallel move)MPY $(\pm)S, \#n, D$ (**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 store the resulting product in 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:

MPY Y0, X0, A

MPY Y0, X0, B

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
 x This bit is unchanged by the instruction

Instruction Formats and opcodes 1:

	23	16	15	8	7	0						
MPY (±)S1,S2,D	DATA BUS MOVE FIELD				1	Q	Q	Q	d	k	0	0
MPY (±)S2,S1,D	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

Instruction Fields:

- {S1,S2} QQQ** Source registers S1,S2
[X0*X0,Y0*Y0,X1*X0,Y1*Y0,X0*Y1,Y0*X0,X1*Y0,Y1*X1]
(see Table A-26 on page A-244)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm +/-} k** Sign [+,-] (see Table A-29 on page A-244)

Instruction Formats and opcode 2:

	23	16					15	8					7	0											
MPY	(±)S,#n,D	0	0	0	0	0	0	0	1	0	0	0	s	s	s	s	s	1	1	Q	Q	d	k	0	0

Instruction Fields:

- {S} QQ** Source register [Y1,X0,Y0,X1]] (see Table A-27 on page A-244)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)
- {#n} sssss** Immediate operand (see Table A-32 on page A-246)

A-6.81 Mixed Multiply (MPY su/uu)

MPY(su,uu) MPY(su,uu)

Mixed Multiply

Operation:

$\pm S1 * S2 \rightarrow D$ (S1 unsigned, S2 unsigned)

$\pm S1 * S2 \rightarrow D$ (S1 signed, S2 unsigned)

Assembler Syntax:

MPYuu (\pm)S1,S2,D (no parallel move)

MPYsu (\pm)S2,S1,D (no parallel move)

Description: Multiply the two 24-bit source operands S1 and S2 and store the resulting product in the specified 56-bit destination accumulator D. One or two of the source operands can be unsigned. The “–” sign option is used to negate the specified product prior to accumulation. The default sign option is “+”.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes :

MPYsu (\pm)S1,S2,D	23	16 15	8 7	0
MPYuu (\pm)S1,S2,D	0 0 0 0 0 0 0 1	0 0 1 0 0 1 1 1	1 s d k Q Q Q Q	

Instruction Fields:

- {S1,S2} QQQQ** Source registers S1,S2 [all combinations of X0,X1,Y0 and Y1] (see Table A-30 on page A-245)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)
- {s}** [ss,us] (see Table A-40 on page A-249)

A-6.82 Signed Multiply with Immediate Operand (MPYI)

MPYI

MPYI

Signed Multiply with Immediate Operand

Operation: $\pm\#\text{xxxxxx} * S \rightarrow D$ **Assembler Syntax:**MPYI $(\pm)\#\text{xxxxxx}, S, D$

Description: Multiply the immediate 24-bit source operand #xxxxxx with the 24-bit register source operand S and store the resulting product in 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 “+”.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
x This bit is unchanged by the instruction

Instruction Formats and opcode:

		23	16							15	8							7	0						
MPYI	(±)#xxxxxx,S,D	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	q	q	d	k	0	0
		IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	qq	Source register [X0,Y0,X1,Y1] (see Table A-28 on page A-244)
{D}	d	Destination accumulator [A,B] (see Table A-10 on page A-239)
{ \pm }	k	Sign [+,-] (see Table A-29 on page A-244)
#xxxxxx		24-bit Immediate Long Data extension word

MPYR

MPYR

Signed Multiply and Round

Operation: $\pm S1 * S2 + r \rightarrow D$ (parallel move) $\pm S1 * S2 + r \rightarrow D$ (parallel move) $\pm (S1 * 2^{-n}) + r \rightarrow D$ (**no** parallel move)**Assembler Syntax:**MPYR $(\pm)S1, S2, D$ (parallel move)MPYR $(\pm)S2, S1, D$ (parallel move)MPYR $(\pm)S, \#n, D$ (**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 either convergent or two's complement 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. Once the rounding has been completed, the LS bits of the destination accumulator D are loaded with zeros to maintain an unbiased accumulator value which may be reused by the next instruction. The upper portion of the accumulator contains the rounded result which may be read out to the data buses. Refer to the RND instruction for more complete information on the rounding process.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	×
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes 1:

	23	16	15	8	7	0						
MPYR (\pm)S1,S2,D	DATA BUS MOVE FIELD				1	Q	Q	Q	d	k	0	1
MPYR (\pm)S2,S1,D	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

Instruction Fields 1:

- {S1,S2} QQQ** Source registers S1,S2
 $[X0*X0,Y0*Y0,X1*X0,Y1*Y0,X0*Y1,Y0*X0,X1*Y0,Y1*X1]$
 (see Table A-26 on page A-244)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)

Instruction Formats and opcode 2:

	23	16						15	8						7	0																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																															
MPYR	(±)	S	#	n	,	D																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																									

Instruction Fields 2:

- {S} QQ** Source register [Y1,X0,Y0,X1]] (see Table A-27 on page A-244)
- {D} d** Destination accumulator [A,B] (see Table A-10 on page A-239)
- { \pm } k** Sign [+,-] (see Table A-29 on page A-244)
- {#n} sssss** Immediate operand (see Table A-32 on page A-246)

MPYRI

MPYRI

Signed Multiply and Round with Immediate Operand

Operation:

$\pm\#\text{xxxxxx} * S + r \rightarrow D$

Assembler Syntax:

MPYRI $(\pm)\#\text{xxxxxx}, S, D$

Description: Multiply the two signed 24-bit source operands $\#\text{xxxxxx}$ and S , round the result using either convergent or two's complement 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. Once the rounding has been completed, the LS bits of the destination accumulator D are loaded with zeros to maintain an unbiased accumulator value which may be reused by the next instruction. The upper portion of the accumulator contains the rounded result which may be read out to the data buses. Refer to the RND instruction for more complete information on the rounding process.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16	15	8	7	0																		
MPYRI (±)#xxxxxx,S,D	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	1	1	q	q	d	k	0	1	
	IMMEDIATE DATA EXTENSION																							

Instruction Fields:

{S}	qq	Source register [X0,Y0,X1,Y1] (see Table A-28 on page A-244)
{D}	d	Destination accumulator [A,B] (see Table A-10 on page A-239)
{±}	k	Sign [+,-] (see Table A-29 on page A-244)
#xxxxxx		24-bit Immediate Long Data extension word

NEG**NEG****Negate Accumulator****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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	x
CCR							

✓ This bit is changed according to the standard definition

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23		16	15		8	7		0
NEG										
	D	DATA BUS MOVE FIELD					0	0	1	1
		OPTIONAL EFFECTIVE ADDRESS EXTENSION								
							d	1	1	0

Instruction Fields:

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

A-6.86 No Operation (NOP)

NOP

NOP

No Operation

Operation:

PC+1→PC

Assembler Syntax:

NOP

Description: Increment the program counter (PC). Pending pipeline actions, if any, are completed. Execution continues with the instruction following the NOP.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23		16	15		8	7		0
NOP	0	0	0	0	0	0	0	0	0

Instruction Fields : None

NORM

NORM

Norm Accumulator Iteration

Operation:

If $\bar{E} \bullet U \bullet \bar{Z}=1$, then ASL D and $R_{n-1} \rightarrow R_n$
 else if $E=1$, then ASR D and $R_{n+1} \rightarrow R_n$
 else NOP

Assembler Syntax:

NORM R_n, D

where \bar{E} denotes the logical complement of E, and
 where \bullet denotes the logical AND operator

Description: Perform one normalization iteration on the specified destination operand D, update the specified address register R_n 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	●	x
CCR							

- Set if bit 55 is changed as a result of a left shift
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16 15							8 7							0								
NORM Rn,D	0	0	0	0	0	0	0	1	1	1	0	1	1	R	R	R	0	0	0	1	d	1	0	1

Instruction Fields:

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)
{Rn} RRR Address register [R0-R7]

NORMF

NORMF

Fast Accumulator Normalization

Operation:

If S[23]=0 then ASR S,D
else ASL -S,D

Assembler Syntax:

NORMF S,D

Description: Arithmetically shift the destination accumulator either left or right as specified by the source operand sign and value. If the source operand is negative then the accumulator is left shifted, and if the source operand is positive then it is right shifted. The source accumulator value should be between +56 to -55 (or +40 to -39 in sixteen bit mode). This instruction can be used to normalize the specified accumulator D, by arithmetically shifting it either left or right so as to bring the leading one or zero to bit location 46. The number of needed shifts is specified by the source operand. This number could be calculated by a previous CLB instruction. For normalization the source accumulator value should be between +8 to -47 (or +8 to -31 in sixteen bit mode).

This is a 56 bit operation.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	✓	✓	✓	✓	✓	●	x
CCR							

- V Set if bit 55 is changed any time during the shift operation. Cleared otherwise.
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction.

Example:

CLB A,B ;Count leading bits.

NORMF B1,A ;Normalize A.

If the base exponent is stored in R1 it can be updated by the following commands.

MOVE B1,N1 ;Update N1 with shift amount

MOVE (R1)+N1 ;Increment or decrement exponent

	Before execution	After execution
CLB A,B	A: \$20:000000:000000	B: \$00:000007:000000
NORMF B1,A	A: \$20:000000:000000	A: \$00:400000:000000

Explanation of example: Prior to execution, the 56-bit A accumulator contains the value \$20:000000:000000. The CLB instruction updates the B accumulator to the number of needed shifts, 7 in this example. The NORMF instruction performs 7 shifts to the right on A accumulator, and normalization of A is achieved. The exponent register is updated according to the number of shifts.

Instruction Formats and opcode

		23	16	15	8	7	0																	
NORMF	S,D	0	0	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	s	s	s	D

Instruction Fields:

{S}	sss	Source register [X0,X1,Y0,Y1,A1,B1] (see Table A-15 on page A-240)
{D}	D	Destination accumulator [A,B] (see Table A-10 on page A-239)

A-6.89 Logical Complement (NOT)

NOT

NOT

Logical Compliment

Operation:

$\overline{D[47:24]} \rightarrow D[47:24]$ (parallel move)

where “ $\overline{\hspace{0.5em}}$ ” 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	●	●	●	x
CCR							

- N Set if bit 47 of the result is set
- Z Set if bits 47–24 of the result are zero
- V Always cleared
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23	16	15	8	7	0						
NOT	D	DATA BUS MOVE FIELD				0	0	0	1	d	1	1	1
		OPTIONAL EFFECTIVE ADDRESS EXTENSION											

Instruction Fields:

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

OR**OR****Logical Inclusive OR****Operation:**

S+D[47:24] → D[47:24] (parallel move)

#xx+D[47:24] → D[47:24]

#xxxxxx+D[47:24] → D[47:24]

Assembler Syntax:

OR S,D (parallel move)

OR #xx,D

OR #xxxxxx,D

where + denotes the logical inclusive OR operator

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. The source can be a 24-bit register, 6-bit short immediate or 24-bit long immediate. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the 6 bits will be right aligned and the remaining bits will be zeroed to form a 24-bit source operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	●	●	●	x
CCR							

- N Set if bit 47 of the result is set
- Z Set if bits 47–24 of the result are zero
- V Always cleared
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0						
OR S,D	DATA BUS MOVE FIELD				0	1	J	J	d	0	1	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

	23	16 15							8 7							0								
OR #xx,D	0	0	0	0	0	0	0	1	0	1	i	i	i	i	i	i	1	0	0	0	d	0	1	0

	23	16 15								8 7								0							
OR #xxxxxx,D	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	0	0	d	0	1	0
	IMMEDIATE DATA EXTENSION																								

Instruction Fields:

{S}	JJ	Source input register [X0,X1,Y0,Y1] (see Table A-12 on page A-239)
{D}	d	Destination accumulator [A/B] (see Table A-10 on page A-239)
{#xx}	iiii	6-bit Immediate Short Data
{#xxxxxx}		24-bit Immediate Long Data extension word

ORI

ORI

OR Immediate with Control register

Operation:

#xx+D → D

Assembler Syntax:

OR(I) #xx,D

where + denotes the logical inclusive OR operator

Description: Logically OR the 8-bit immediate operand (#xx) with the contents of the destination control register D and store the result in the destination control register. The condition codes are affected only when the condition code register is specified as the destination operand.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	●	●	●	●	●	●
CCR							

For CCR Operand:

- S 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

For MR and OMR Operands: The condition codes are not affected using these operands.

Instruction Formats and opcodes:

	23	16 15							8 7							0								
OR(l) #xx,D	0	0	0	0	0	0	0	0	i	i	i	i	i	i	i	i	1	1	1	1	1	0	E	E

Instruction fields:

{D}	EE	Program Controller register [MR,CCR,COM,EOM] (see Table A-13 on page A-239)
{#xx}	iiiiiii	Immediate Short Data

PFLUSH

PFLUSH

Program Cache Flush

Operation:

Flush instruction cache

Assembler Syntax:

PFLUSH

Description: Flush the whole instruction cache, unlock all cache sectors, set the LRU stack and tag registers to their default values.

The PFLUSH instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16 15								8 7								0			
PFLUSH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Instruction Fields: None

PFLUSHUN

PFLUSHUN

Program Cache Flush Unlocked Sectors

Operation:

Flush Unlocked instruction cache sectors

Assembler Syntax:

PFLUSHUN

Description: Flush the instruction cache sectors which are unlocked, set the LRU stack to its default value and set the unlocked tag registers to their default values.

The PFLUSHUN instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16 15								8 7								0																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																
PFLUSHUN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields: None

A-6.94 Program-Cache Global Unlock (PFREE)

PFREE

PFREE

Program Cache Global Unlock

Operation:

Unlock all locked sectors

Assembler Syntax:

PFREE

Description: Unlock all the locked cache sectors in the instruction cache.

The PFREE instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16	15	8	7	0											
PFREE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Instruction Fields: None

PLOCKR

Lock Instruction Cache Relative Sector

A-6.96 Unlock instruction Cache Sector (PUNLOCK)

PUNLOCK

PUNLOCK

Unlock Instruction Cache Sector

Operation:

Unlock sector by effective address

Assembler Syntax:

PUNLOCK ea

Description: Unlock the cache sector to which the specified effective address belongs. If the specified effective address does not belong to any cache sector, and is therefore definitely unlocked, nevertheless, load the least recently used cache sector tag with the 17 most significant bits of the specified address. Update the LRU stack accordingly. All memory alterable addressing modes may be used for the effective address, but not a short absolute address.

The PUNLOCK instruction is enabled only in Cache Mode. In PRAM Mode it will cause an illegal instruction trap.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16 15							8 7							0								
PUNLOCK ea	0	0	0	0	1	0	1	0	1	1	M	M	M	R	R	R	1	0	0	0	0	0	0	1
	ADDRESS EXTENSION WORD																							

Instruction Fields:

{ea} **MMMR**RR Effective Address (see Table A-18 on page A-241)

PUNLOCKR PUNLOCKR

Unlock Instruction Cache Relative Sector

REP

REP

Repeat Next Instruction

Operation:

LC → TEMP; [X or y]:ea → LC
Repeat next instruction until LC=1
TEMP → LC

LC → TEMP; [X or Y]:aa → LC
Repeat next instruction until LC=1
TEMP → LC

LC → TEMP; S → LC
Repeat next instruction until LC=1
TEMP → LC

LC → TEMP; #xxx → LC
Repeat next instruction until LC=1
TEMP → LC

Assembler Syntax:

REP [X or Y]:ea

REP [X or Y]:aa

REP S

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 24-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 24-bit value that is to be loaded into the loop counter (LC).

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	×	×	×	×	×	×
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0																		
REP [X or Y]:ea	0	0	0	0	0	1	1	0	0	1	M	M	M	R	R	R	0	S	1	0	0	0	0	0

	23	16	15	8	7	0																		
REP [X or Y]:aa	0	0	0	0	0	1	1	0	0	0	a	a	a	a	a	a	0	S	1	0	0	0	0	0

	23	16	15	8	7	0																		
REP #xxx	0	0	0	0	0	1	1	0	i	i	i	i	i	i	i	i	1	0	1	0	h	h	h	h

	23	16	15	8	7	0																		
REP S	0	0	0	0	0	1	1	0	1	1	d	d	d	d	d	d	0	0	1	0	0	0	0	0

Instruction Fields:

{ea}	MMMR	Effective Address (see Table A-19 on page A-242)
{X/Y}	S	Memory Space [X,Y] (see Table A-17 on page A-241)
{aa}	aaaaaa	Absolute Short Address
{#xxx}	hhhhiiiiiii	Immediate Short Data
{S}	dddddd	Source register [all on-chip registers] (see Table A-22 on page A-243)

RESET

RESET

Reset On-Chip Peripherals Devices

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 stack error, NMI, illegal instruction, Trap, Debug request and hardware reset interrupts.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16 15								8 7								0					
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0

Instruction Fields: None

RND**RND****Round Accumulator****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 destination accumulator (A or B). The contribution of the LS bits of the operand is rounded into the upper portion of the operand by adding a rounding constant to the LS bits of the operand. The upper portion of the destination accumulator contains the rounded result. The boundary between the lower portion and the upper portion is determined by the scaling mode bits S0 and S1 in the status register (SR).

Two types of rounding can be used: convergent rounding (also called round to nearest (even)) or two's complement rounding. The type of rounding is selected by the rounding mode bit (RM) in the MR portion of the status register.

In both these rounding modes a rounding constant is first added to the unrounded result. The value of the rounding constant added is determined by the scaling mode bits S0 and S1 in the status register (SR). A "1" is positioned in the rounding constant aligned with the most significant bit of the current LS portion, i.e. the rounding constant weight is actually equal to half the weight of the upper's portion least significant bit.

The following table shows the rounding position and rounding constant as determined by the scaling mode bits:

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

Secondly, if convergent rounding is used, the result of this addition is tested and if all the bits of the result to the right of, and including, the rounding position are cleared, then the bit to the left of the rounding position is cleared in the result. This ensures that the result will not be biased.

Thirdly, in both rounding modes, the least significant bits of the result are cleared. The number of least significant bits cleared is determined by the scaling mode bits in the status register. All bits to the right of, and including, the rounding position are cleared in the result.

In Sixteen Bit Arithmetic mode the 40-bit value (in the 56-bit destination operand D) is rounded and stored in the destination accumulator (A or B). This implies that the boundary between the lower portion and upper portion is in a different position than in 24 bit mode. The following table shows the rounding position and rounding constant in sixteen bit arithmetic mode, as determined by the scaling mode bits:

S1	S0	Scaling Mode	Rounding Position	Rounding Constant				
				55 - 33	32	23	22	21 - 8
0	0	No Scaling	31	0...0	0	1	0	0...0
0	1	Scale Down	32	0...0	1	0	0	0...0
1	0	Scale Up	30	0...0	0	0	1	0...0

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	x
CCR							

- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

RND	D	23	16 15				8 7				0						
		DATA BUS MOVE FIELD								0	0	0	1	d	0	0	1
		OPTIONAL EFFECTIVE ADDRESS EXTENSION															

Instruction Fields:

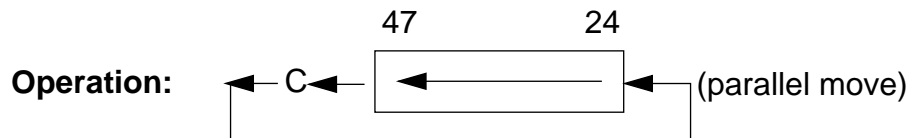
{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

A-6.101 Rotate Left (ROL)

ROL

ROL

Rotate Left



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. The carry bit receives the previous value of bit 47 of the operand. The previous value of the carry bit is shifted into bit 24 of the operand. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	●	●	●	●
CCR							

- N Set if bit 47 of the result is set
- Z Set if bits 47–24 of the result are zero
- V Always cleared
- C Set if bit 47 of the destination operand is set, cleared otherwise.
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

23	16	15	8	7	0
DATA BUS MOVE FIELD				0 0 1 1	d 1 1 1
OPTIONAL EFFECTIVE ADDRESS EXTENSION					

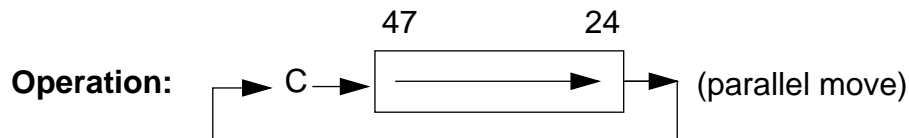
Instruction Fields:

{D} **d** Destination accumulator [A,B] (see Table A-10 on page A-239)

ROR

ROR

Rotate Right

**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. The carry bit receives the previous value of bit 24 of the operand. The previous value of the carry bit is shifted into bit 47 of the operand. This instruction is a 24-bit operation. The remaining bits of the destination operand D are not affected.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	●	●	●	●
CCR							

- N Set if bit 47 of the result is set
- Z Set if bits 47–24 of the result are zero
- V Always cleared
- C Set if bit 24 of the destination operand is set, cleared otherwise.
- ✓ This bit is changed according to the standard definition
- x This bit is unchanged by the instruction

Instruction Formats and opcodes:

		23		16	15		8	7		0					
ROR	D	DATA BUS MOVE FIELD						0	0	1	0	d	1	1	1
		OPTIONAL EFFECTIVE ADDRESS EXTENSION													

Instruction Fields:

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

RTI

RTI

Return from Interrupt

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
●	●	●	●	●	●	●	●
CCR							

- All All the Status Register bits are set according to the value pulled from the stack

Instruction Formats and opcode:

	23		16	15		8	7		0
RTI	0	0	0	0	0	0	0	0	0

Instruction Fields: None

RTS

RTS

A-6.105 Subtract Long with Carry (SBC)

SBC

SBC

Subtract Long with Carry

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) are 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).

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	✓
CCR							

✓ This bit is changed according to the standard definition

× This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
SBC S,D	DATA BUS MOVE FIELD				0 0 1 J	d 1 0 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

{S} J Source register [X,Y] (see Table A-11 on page A-239)

{D} d Destination accumulator [A,B] (see Table A-10 on page A-239)

STOP

STOP

Stop Instruction Processing

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 RESET, DE or $\overline{\text{IRQA}}$ pin is asserted or the Debug Request JTAG command is detected. The clock oscillator is gated off internally. The STOP processing state is a low-power standby state.

During the STOP state, the destination port is in an idle state with the control signals held inactive, the data pins are high impedance, and the address pins are unchanged from the previous instruction.

If the exit from the STOP state was caused by a low level on the $\overline{\text{RESET}}$ pin, then the processor will enter the reset processing state.

If the exit from the STOP state was caused by a low level on the $\overline{\text{IRQA}}$ pin, then the processor will service the highest priority pending interrupt and will not service the $\overline{\text{IRQA}}$ interrupt unless it is highest priority. If no interrupt is pending, the processor will resume program execution at the instruction following the STOP instruction that caused the entry into the STOP state. Program execution (interrupt or normal flow) will resume after an internal delay counter counts:

- If the Stop Delay (SD, OMR[6]) bit is cleared - 131,070 clock cycles
- If the Stop Delay (SD, OMR[6]) bit is set - 24 clock cycles
- If the STOP Processing State (PSTP, PCTL[17]) is set - 8.5 clock cycles

During the clock stabilization count delay, all peripherals and external interrupts are cleared and re-enabled/arbitrated at the end of the count interval. If the $\overline{\text{IRQA}}$ pin is asserted when the STOP instruction is executed, the clock will not be gated off, and only the internal delay counter will be started.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16 15								8 7								0					
STOP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1

Instruction Fields: None

SUB

SUB

Subtract

Operation:

D-S → D (parallel move)

D-#xx → D

D-#xxxxxx → D

Assembler Syntax:

SUB S, D (parallel move)

SUB #xx, D

SUB #xxxxxx, D

Description: Subtract the source operand from the destination operand D and store the result in the destination operand D. The source can be a register (word - 24 bits, long word - 48 bits or accumulator - 56 bits), short immediate (6 bits) or long immediate (24 bits).

When using 6-bit immediate data, the data is interpreted as an unsigned integer. That is, the 6 bits will be right aligned and the remaining bits will be zeroed to form a 24-bit source operand.

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	✓
CCR							

- ✓ This bit is changed according to the standard definition
 x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0						
SUB S,D	DATA BUS MOVE FIELD				0	J	J	J	d	1	0	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

	23	16 15						8 7						0										
SUB #xx,D	0 0 0 0 0 0 0 1								0 1 i i i i i i								1 0 0 0 d 1 0 0							

	23	16	15	8	7	0																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													
SUB #xxxxxx,D	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields:

{S}	JJJ	Source register [B/A,X,Y,X0,Y0,X1,Y1] (see Table A-14 on page A-240)
{D}	d	Destination accumulator [A/B] (see Table A-10 on page A-239)
{#xx}	iiii	6-bit Immediate Short Data
{#xxxxxx}		24-bit Immediate Long Data extension word

A-6.108 Shift Left and Subtract Accumulators (SUBL)

SUBL

SUBL

Shift Left and Subtract Accumulators

Operation:

$2 * D - S \rightarrow D$ (parallel move)

Assembler Syntax:

SUBL S,D (parallel move)

Description: Subtract the source operand S from two times the destination operand D and store the result in the destination accumulator. The destination operand D is arithmetically shifted one bit to the left, and a 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	●	✓
CCR							

- V Set if overflow has occurred in the result or if the MS bit of the destination operand is changed as a result of the instruction's left shift
- ✓ This bit is changed according to the standard definition
- × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
SUBL S,D	DATA BUS MOVE FIELD				0 0 0 1	d 1 1 0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

{D}	d	Destination accumulator [A,B] (see Table A-10 on page A-239)
{S}		The source accumulator is B if the destination accumulator (selected by the d bit in the opcode) is A, or A if the destination accumulator is B

SUBR

SUBR

Shift Right and Subtract Accumulators

Operation:D/2 \rightarrow S \rightarrow 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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	✓	✓
CCR							

- ✓ This bit is changed according to the standard definition
 × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0						
SUBR S,D	DATA BUS MOVE FIELD				0	0	0	0	d	1	1	0
	OPTIONAL EFFECTIVE ADDRESS EXTENSION											

Instruction Fields:

- {D} **d** Destination accumulator [A,B] (see Table A-10 on page A-239)
 {S} The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B

Tcc**Tcc****Transfer Conditionally****Operation:**

If cc, then S1 → D1

If cc, then S1 → D1 and S2 → D2

If cc, then S2 → D2

Assembler Syntax:

Tcc S1,D1

Tcc S1,D1 S2,D2

Tcc S2,D2

Description: Transfer data from the specified source register S1 to the specified destination accumulator D1 if the specified condition is true. If a second source register S2 and a second destination register D2 are also specified, transfer data from address register S2 to address register D2 if the specified condition is true. If the specified condition is false, a NOP is executed.

The conditions that the term “**cc**” can specify are listed on Table A-42 on page A-250.

When used after the CMP or CMPM instructions, the Tcc instruction can perform many useful functions such as a “maximum value,” “minimum value,” “maximum absolute value,” or “minimum absolute value” function. The desired value is stored in the destination accumulator D1. If address register S2 is used as an address pointer into an array of data, the address of the desired value is stored in the address register D2. The Tcc instruction may be used after any instruction and allows efficient searching and sorting algorithms.

The Tcc instruction uses the internal data ALU paths and internal address ALU paths. The Tcc instruction does not affect the condition code bits.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

× This bit is unchanged by the instruction

Tcc	S1,D1	<div> <div>231615870</div> <div>00000010CCCCC00000JJJd000</div> </div>
Tcc	S1,D1 S2,D2	<div> <div>231615870</div> <div>00000011CCCC0tttJJJdTTT</div> </div>
Tcc	S2,D2	<div> <div>231615870</div> <div>00000010CCCC1ttt000000TTT</div> </div>

{cc}	CCCC	Condition code (see Table A-43 on page A-251)
{S1}	JJJ	Source register [B/A,X0,Y0,X1,Y1] (see Table A-24 on page A-243)
{D1}	d	Destination accumulator [A/B] (see Table A-10 on page A-239)
{S2}	ttt	Source address register [R0-R7]
{D2}	TTT	Destination Address register [R0-R7]

TFR

TFR

Transfer Data ALU Register

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	x	x	x	x	x	x
CCR							

- ✓ This bit is changed according to the standard definition
 x This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
TFR S,D	DATA BUS MOVE FIELD				0 J J J	d 0 0 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

- {S} **JJJ** Source register [B/A,X0,Y0,X1,Y1] (see Table A-24 on page A-243)
 {D} **d** Destination accumulator [A/B] (see Table A-10 on page A-239)

TRAP

TRAP

A-6.114 Test Accumulator (TST)

TST

TST

Test Accumulator

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.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
✓	✓	✓	✓	✓	✓	●	×
CCR							

- V Always cleared
- ✓ This bit is changed according to the standard definition
- × This bit is unchanged by the instruction

Instruction Formats and opcodes:

	23	16	15	8	7	0
TST S	DATA BUS MOVE FIELD				0 0 0 0	d 0 1 1
	OPTIONAL EFFECTIVE ADDRESS EXTENSION					

Instruction Fields:

{S} d Source accumulator [A,B] (see Table A-10 on page A-239)

A-6.115 Wait for interrupt (WAIT)

WAIT

WAIT

Wait for Interrupt or DMA request

Operation:

Disable clocks to the processor core and enter the WAIT processing state

Assembler Syntax:

WAIT

Description: Enter the low-power standby WAIT processing state. The internal clocks to the processor core and memories are gated off, and all activity in the processor is suspended until an unmasked interrupt occurs or an enabled DMA channel receives a request. 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. If WAIT is executed when the DMA is active, the effect will be the same as if the processor never entered the WAIT state. When an unmasked interrupt or external (hardware) processor RESET occurs, the processor leaves the WAIT state and begins exception processing of the unmasked interrupt or RESET condition. The processor will exit from the WAIT state also when a Debug Request (\overline{DE}) pin is asserted or when a Debug Request JTAG command is detected.

Condition Codes:

7	6	5	4	3	2	1	0
S	L	E	U	N	Z	V	C
x	x	x	x	x	x	x	x
CCR							

x This bit is unchanged by the instruction

Instruction Formats and opcode:

	23	16 15								8 7								0						
WAIT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0

Instruction Fields: None

A-7 INSTRUCTION PARTIAL ENCODING

This section gives the encodings for (1) various groupings of registers used in the instruction encodings, (2) condition code combinations, (3) addressing, and (4) addressing modes. The symbols used in decoding the various fields of an instruction are identical to those used in the Opcode section of the individual instruction descriptions.

A-7.1 Partial Encodings for Use in Instruction Encoding

Table A-10. Destination Accumulator Encoding

D/	d/S/D
A	0
B	1

Table A-11. Data ALU Operands Encoding

S	J
X	0
Y	1

Table A-12. Data ALU Source Operands Encoding

S	JJ
X0	00
Y0	01
X1	10
Y1	11

Table A-13. Program Control Unit Register Encoding

Register	EE
MR	00
CCR	01
COM	10
EOM	11

Table A-14. Data ALU Operands Encoding

S	J J J
B/A*	001
X	010
Y	011
X0	100
Y0	101
X1	110
Y1	111

* The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B.

Table A-15. Data ALU operands encoding

SSS/sss	S,D	qqq	S,D	ggg	S,D
000	reserved	000	reserved	000	B/A*
001	reserved	001	reserved	001	reserved
010	A1	010	A0	010	reserved
011	B1	011	B0	011	reserved
100	X0	100	X0	100	X0
101	Y0	101	Y0	101	Y0
110	X1	110	X1	110	X1
111	Y1	111	Y1	111	Y1

* The selected accumulator is B if the source two accumulator (selected by the **d** bit in the opcode) is A, or A if the source two accumulator is B.

Table A-16. Effective Addressing Mode Encoding #1

Effective Addressing Mode	MMMR
(Rn)-Nn	0 0 0 r r r
(Rn)+Nn	0 0 1 r r r
(Rn)-	0 1 0 r r r
(Rn)+	0 1 1 r r r
(Rn)	1 0 0 r r r
(Rn+Nn)	1 0 1 r r r
-(Rn)	1 1 1 r r r
Absolute address	1 1 0 0 0 0
Immediate data	1 1 0 1 0 0

“rrr” refers to an address register R0-R7

Table A-17. Memory/Peripheral Space

Space	S
X Memory	0
Y Memory	1

Table A-18. Effective Addressing Mode Encoding #2

Effective Addressing Mode	MMMR
(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

“rrr” refers to an address register R0-R7

Table A-19. Effective Addressing Mode Encoding #3

Effective Addressing Mode	MMRRR
(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

“rrr” refers to an address register R0-R7

Table A-20. Effective Addressing Mode Encoding #4

Effective Addressing Mode	MMRRR
(Rn)-Nn	0 0 r r r
(Rn)+Nn	0 1 r r r
(Rn)-	1 0 r r r
(Rn)+	1 1 r r r

“rrr” refers to an address register R0-R7

Table A-21. Triple-Bit Register Encoding

Code	1DD	DDD	TTT	NNN	FFF	EEE	VVV	GGG
000	-	A0	R0	N0	M0	-	VBA	SZ
001	-	B0	R1	N1	M1	-	SC	SR
010	-	A2	R2	N2	M2	EP	-	OMR
011	-	B2	R3	N3	M3	-	-	SP
100	X0	A1	R4	N4	M4	-	-	SSH
101	X1	B1	R5	N5	M5	-	-	SSL
110	Y0	A	R6	N6	M6	-	-	LA
111	Y1	B	R7	N7	M7	-	-	LC

Table A-22. Six-Bit Encoding For all On-Chip Registers

Destination Register	D D D D D D / d d d d d d
4 registers in Data ALU	0001DD
8 accumulators in Data ALU	001DDD
8 address registers in AGU	010TTT
8 address offset registers in AGU	011NNN
8 address modifier registers in AGU	100FFF
1 address register in AGU	101EEE
2 program controller register	110VVV
8 program controller registers	111GGG

See Table A-21 for the specific encodings.

Table A-23. Long Move Register Encoding

S	S1	S2	S S/L	D	D1	D2	D Sign Ext	D Zero	LLL
A10	A1	A0	no	A10	A1	A0	no	no	0 0 0
B10	B1	B0	no	B10	B1	B0	no	no	0 0 1
X	X1	X0	no	X	X1	X0	no	no	0 1 0
Y	Y1	Y0	no	Y	Y1	Y0	no	no	0 1 1
A	A1	A0	yes	A	A1	A0	A2	no	1 0 0
B	B1	B0	yes	B	B1	B0	B2	no	1 0 1
AB	A	B	yes	AB	A	B	A2,B2	A0,B0	1 1 0
BA	B	A	yes	BA	B	A	B2,A2	B0,A0	1 1 1

Table A-24. Data ALU Source Registers Encoding

S	J J J
B/A*	000
X0	100
Y0	101
X1	110
Y1	111

* The source accumulator is B if the destination accumulator (selected by the **d** bit in the opcode) is A, or A if the destination accumulator is B.

Table A-25. AGU Address and Offset Registers Encoding

Dest. Addr. Reg. D	dddd
R0-R7	onnn
N0-N7	1nnn

Table A-26. Data ALU Multiply Operands Encoding #1

S1*S2	Q Q Q	S1*S2	Q Q Q
X0,X0	0 0 0	X0,Y1	1 0 0
Y0,Y0	0 0 1	Y0,X0	1 0 1
X1,X0	0 1 0	X1,Y0	1 1 0
Y1,Y0	0 1 1	Y1,X1	1 1 1

Note: Only the indicated S1*S2 combinations are valid.X1*X1 and Y1*Y1 are not valid.

Table A-27. Data ALU Multiply Operands Encoding #2

S	Q Q
Y1	00
X0	01
Y0	10
X1	11

Table A-28. Data ALU Multiply Operands Encoding #3

S	qq
X0	00
Y0	01
X1	10
Y1	11

Table A-29. Data ALU Multiply Sign Encoding

Sig n	k
+	0
-	1

Table A-30. Data ALU Multiply Operands Encoding #3

S1*S2	Q Q Q Q	S1*S2	Q Q Q Q
X0,X0	0 0 0 0	X0,Y1	0 1 0 0
Y0,Y0	0 0 0 1	Y0,X0	0 1 0 1
X1,X0	0 0 1 0	X1,Y0	0 1 1 0
Y1,Y0	0 0 1 1	Y1,X1	0 1 1 1
X1,X1	1 0 0 0	Y1,X0	1 1 0 0
Y1,Y1	1 0 0 1	X0,Y0	1 1 0 1
X0,X1	1 0 1 0	Y0,X1	1 1 1 0
Y0,Y1	1 0 1 1	X1,Y1	1 1 1 1

Table A-31. 5-Bit Register Encoding #1

D/S	dddd / eeee	D/S	dddd / eeee
X0	00100	B2	01011
X1	00101	A1	01100
Y0	00110	B1	01101
Y1	00111	A	01110
A0	01000	B	01111
B0	01001	R0-R7	10 r r r
A2	01010	N0-N7	11 n n n

“rrr”=Rn number, “nnn”=Nn number

Table A-32. Immediate Data ALU Operand Encoding

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	0000000000000000000001000
21	10101	00000000000000000000000100
22	10110	00000000000000000000000010
23	10111	00000000000000000000000001

Table A-33. Write Control Encoding

Operation	W
Read Register or Peripheral	0
Write Register or Peripheral	1

Table A-34. ALU Registers Encoding

Destination Register	D D D D
4 registers in Data ALU	01DD
8 accumulators in Data ALU	1DDD

See Table A-21 for the specific encodings.

Table A-35. X:R Operand Registers Encoding

S1, D1	f f	D2	F
X0	00	Y0	0
X1	01	Y1	1
A	10		
B	11		

Table A-36. R:Y Operand Registers Encoding

D1	e	S2, D2	f f
X0	0	Y0	00
X1	1	Y1	01
		A	10
		B	11

Table A-37. Single-Bit Special Register Encoding Tables

d	X:R Class II Opcode	R:Y Class II Opcode
0	A → X:<ea> , X0 → A	Y0 → A , A → Y:<ea>
1	B → X:<ea> , X0 → B	Y0 → B , B → Y:<ea>

Table A-38. X:Y: Move Operands Encoding Tables

X Effective Addressing Mode	MMRRR
(Rn)+Nn	01sss
(Rn)-	10sss
(Rn)+	11sss
(Rn)	00sss

where “sss” refers to an address register R0-R7

Y Effective Addressing Mode	mmrr
(Rn)+Nn	01tt
(Rn)-	10tt
(Rn)+	11tt
(Rn)	00tt

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

S1,D1	e e	S2,D2	f f
X0	00	Y0	00
X1	01	Y1	01
A	10	A	10
B	11	B	11

Table A-39. Signed/Unsigned partial encoding #1

ss/su/uu	ss
ss	00
su	10
uu	11
reserved	01

Table A-40. Signed/Unsigned partial encoding #2

su/uu	s
su	0
uu	1

Table A-41. 5-Bit Register Encoding

S1,D1	ddddd
M0-M7	00nnn
EP	01010
VBA	10000
SC	10001
SZ	11000
SR	11001
OMR	11010
SP	11011
SSH	11100
SSL	11101
LA	11110
LC	11111

where “nnn”=Mn number (M0-M7)

Table A-42. Condition Codes Computation Equations

	"cc" Mnemonic	Condition
CC(HS)	carry clear (higher or same)	$C=0$
CS(LO)	carry set (lower)	$C=1$
EC	extension clear	$E=0$
EQ	equal	$Z=1$
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	$Z+(U \bullet E)=1$
PL	plus	$N=0$
NN	not normalized	$Z+(U \bullet E)=0$

where

\bar{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

Table A-43. Condition Codes Encoding

Mnemonic	CCCC	Mnemonic	CCCC
CC(HS)	0000	CS(LO)	1000
GE	0001	LT	1001
NE	0010	EQ	1010
PL	0011	MI	1011
NN	0100	NR	1100
EC	0101	ES	1101
LC	0110	LS	1110
GT	0111	LE	1111

The condition code computation equations are listed on Table A-42

A-7.2 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.

A-7.2.1 Multiply Instruction Encoding

The 8-bit operation code for multiply instructions allowing parallel moves has different fields than the nonmultiply instruction's operation code.

The 8-bit operation code=**1QQQ dkkk** where

QQQ=selects the inputs to the multiplier (see Table A-26)

kkk = three unencoded bits k2, k1, k0

d = destination accumulator

d = 0 → A

d = 1 → B

Table A-44. Operation Code K0-2 Decode

Code	k2	k1	k0
0	positive	mpy only	don't round
1	negative	mpy and acc	round

Appendix B INSTRUCTION EXECUTION TIMING

B-1 INTRODUCTION

This section describes the various aspects of execution timing analysis for each instruction mnemonic and for various instruction sequences. The section consists of the following tables and information:

1. Tables showing how to calculate DSP56300 Core instruction timing for each instruction mnemonic (instruction timing)
2. Tables showing the number of instruction program words for each instruction mnemonic (instruction program words).
3. Description of various sequences that cause timing delays and stalls in the execution (instruction sequence delays).
4. Description of various instruction sequences that are forbidden and will cause undefined operation (instruction sequence restrictions).

B-2 INSTRUCTION TIMING

The number of oscillator clock cycles per instruction depends on many factors, including the number of words per instruction, the addressing mode, whether the instruction fetch pipeline is full or not, the number of external bus accesses, cache hit/miss/burst, and the number of wait states inserted in each external access.

The timing table is based on the following assumptions:

1. All instruction cycles are counted in **clock cycles**.
2. The instruction fetch pipeline is **full**.

The following terms are used inside the table:

1. **T** - clock cycles for the normal case:
 - All the instructions are fetched from the Instruction Cache (hit) or from the internal program memory.
 - All accesses to data memory are to the internal X and/or Y internal ROMs or RAMs.
 - The previous instructions access internal data memory only.
 - No interlocks with previous instructions.
 - The stack extension mode is disabled.
 - Addressing mode is the Post-Update mode (post increment, post

- decrement and post offset by N) or the No-Update mode.
2. **+ pru** - **PRe** Update - clocks cycles added for using the pre-update addressing modes (pre decrement & offset by N addressing modes).
 3. **+ lab** - **L**ong **AB**solute - clock cycles added for using the long absolute address mode.
 4. **+ lim** - **L**ong **IM**mediate - clock cycles added for using the long immediate data addressing mode.

Note: A '-' sign under one or more of the columns **pru**, **lab** or **lim** indicates that this column is not applicable to the corresponding instruction.

Table B-1. Instruction Timing, Word Count and encoding

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
ADD	ADD #iiii,D	2	-	-	-
	ADD #iii,D	1	-	-	-
AND	AND #iiii,D	2	-	-	-
	AND #iii,D	1	-	-	-
ANDI	ANDI EE	3	-	-	-
ASL	ASL #ii,S,D	1	-	-	-
	ASL sss,S,D	1	-	-	-
ASR	ASR sss,S,D	1	-	-	-
	ASR #ii,S,D	1	-	-	-
Bcc	Bcc (PC+Rn)	4	-	-	-
	Bcc (PC+aaaa)	5	-	-	-
	Bcc (PC+aa)	4	-	-	-
BCHG	BCHG #bbbb,S:<aa>	2	-	-	-
	BCHG #bbbb,S:<ea>	2	1	1	-
	BCHG #bbbb,S:<pp>	2	-	-	-
	BCHG #bbbb,S:<qq>	2	-	-	-
	BCHG #bbbb,DDDDDD	2	-	-	-

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
BCLR	BCLR #bbbbbb,S:<pp>	2	-	-	-
	BCLR #bbbbbb,S:<ea>	2	1	1	-
	BCLR #bbbbbb,S:<aa>	2	-	-	-
	BCLR #bbbbbb,S:<qq>	2	-	-	-
	BCLR #bbbbbb,DDDDDD	2	-	-	-
BRA	BRA (PC+Rn)	4	-	-	-
	BRA (PC+aaaa)	5	-	-	-
	BRA (PC+aa)	4	-	-	-
BRCLR	BRCLR #bbbbbb,S:<pp>,(PC+aaaa)	5	-	-	-
	BRCLR #bbbbbb,S:<qq>,(PC+aaaa)	5	-	-	-
	BRCLR #bbbbbb,S:<ea>,(PC+aaaa)	5	1	-	-
	BRCLR #bbbbbb,S:<aa>,(PC+aaaa)	5	-	-	-
	BRCLR #bbbbbb,DDDDDD,(PC+aaaa)	5	-	-	-
BRKcc	BRKcc	5	-	-	-
BRSET	BRSET #bbbbbb,S:<pp>,(PC+aaaa)	5	-	-	-
	BRSET #bbbbbb,S:<ea>,(PC+aaaa)	5	1	-	-
	BRSET #bbbbbb,S:<aa>,(PC+aaaa)	5	-	-	-
	BRSET #bbbbbb,DDDDDD,(PC+aaaa)	5	-	-	-
	BRSET #bbbbbb,S:<qq>,(PC+aaaa)	5	-	-	-
BScc	BScc (PC+aaaa)	5	-	-	-
	BScc (PC+Rn)	4	-	-	-
	BScc (PC+aa)	4	-	-	-

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
BSCLR	BSCLR #bbbbbb,S:<ea>,(PC+aaaa)	5	1	-	-
	BSCLR #bbbbbb,S:<aa>,(PC+aaaa)	5	-	-	-
	BSCLR #bbbbbb,S:<pp>,(PC+aaaa)	5	-	-	-
	BSCLR #bbbbbb,DDDDDD,(PC+aaaa)	5	-	-	-
	BSCLR #bbbbbb,S:<qq>,(PC+aaaa)	5	-	-	-
BSET	BSET #bbbbbb,S:<pp>	2	-	-	-
	BSET #bbbbbb,S:<ea>	2	1	1	-
	BSET #bbbbbb,S:<aa>	2	-	-	-
	BSET #bbbbbb,DDDDDD	2	-	-	-
	BSET #bbbbbb,S:<qq>	2	-	-	-
BSR	BSR (PC+Rn)	4	-	-	-
	BSR (PC+aaaa)	5	-	-	-
	BSR (PC+aa)	4	-	-	-
BSSET	BSSET #bbbbbb,S:<pp>,(PC+aaaa)	5	-	-	-
	BSSET #bbbbbb,S:<ea>,(PC+aaaa)	5	1	-	-
	BSSET #bbbbbb,S:<aa>,(PC+aaaa)	5	-	-	-
	BSSET #bbbbbb,DDDDDD,(PC+aaaa)	5	-	-	-
	BSSET #bbbbbb,S:<qq>,(PC+aaaa)	5	-	-	-
BTST	BTST #bbbbbb,S:<pp>	2	-	-	-
	BTST #bbbbbb,S:<ea>	2	1	1	-
	BTST #bbbbbb,S:<aa>	2	-	-	-
	BTST #bbbbbb,DDDDDD	2	-	-	-
	BTST #bbbbbb,S:<qq>	2	-	-	-
CLB	CLB S,D	1	-	-	-
CMP	CMP #iiiiii,D	2	-	-	-
	CMP #iii,D	1	-	-	-

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
CMPU	CMPU ggg,D	1	-	-	-
DEBUG/DEBUGcc	DEBUG	1	-	-	-
	DEBUGcc	5	-	-	-
DEC	DEC	1	-	-	-
DIV	DIV	1	-	-	-
DMAC	DMAC S1,S2,D (ss,su,uu)	1	-	-	-
DO	DO #xxx,aaaa	5	-	-	-
	DO DDDDDD,aaaa	5	-	-	-
	DO S:<ea>,aaaa	5	1	-	-
	DO S:<aa>,aaaa	5	-	-	-
DO FOREVER	DO FOREVER,(aaaa)	4	-	-	-
DOR	DOR #xxx,(PC+aaaa)	5	-	-	-
	DOR DDDDDD,(PC+aaaa)	5	-	-	-
	DOR S:<ea>,(PC+aaaa)	5	1	-	-
	DOR S:<aa>,(PC+aaaa)	5	-	-	-
DOR FOREVER	DOR FOREVER,(PC+aaaa)	4	-	-	-
ENDDO	ENDDO	1	-	-	-
EOR	EOR #iiii,D	2	-	-	-
	EOR #iii,D	1	-	-	-
EXTRACT	EXTRACT SSS,s,D	1	-	-	-
	EXTRACT #iii,s,D	2	-	-	-
EXTRACTU	EXTRACTU SSS,s,D	1	-	-	-
	EXTRACTU #iii,s,D	2	-	-	-
IFcc	IFcc(.U)	1	-	-	-
ILLEGAL	ILLEGAL	5	-	-	-
INC	INC	1	-	-	-

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
INSERT	INSERT SSS,qqq,D	1	-	-	-
	INSERT #iii,qqq,D	2	-	-	-
Jcc	Jcc aa	4	-	-	-
	Jcc ea	4	0	0	-
JCLR	JCLR #bbbbbb,S:<ea>,aaaa	4	1	-	-
	JCLR #bbbbbb,S:<pp>,aaaa	4	-	-	-
	JCLR #bbbbbb,S:<aa>,aaaa	4	-	-	-
	JCLR #bbbbbb,DDDDDD,aaaa	4	-	-	-
	JCLR #bbbbbb,S:<qq>,aaaa	4	-	-	-
JMP	JMP aa	3	-	-	-
	JMP ea	3	1	1	-
JScC	JScC aa	4	-	-	-
	JScC ea	4	0	0	-
JSCLR	JSCLR #bbbbbb,S:<pp>,aaaa	4	-	-	-
	JSCLR #bbbbbb,S:<ea>,aaaa	4	1	-	-
	JSCLR #bbbbbb,S:<aa>,aaaa	4	-	-	-
	JSCLR #bbbbbb,DDDDDD,aaaa	4	-	-	-
	JSCLR #bbbbbb,S:<qq>,aaaa	4	-	-	-
JSET	JSET #bbbbbb,S:<pp>,aaaa	4	-	-	-
	JSET #bbbbbb,S:<ea>,aaaa	4	1	-	-
	JSET #bbbbbb,S:<aa>,aaaa	4	-	-	-
	JSET #bbbbbb,DDDDDD,aaaa	4	-	-	-
	JSET #bbbbbb,S:<qq>,aaaa	4	-	-	-
JSR	JSR aa	3	-	-	-
	JSR ea	3	1	1	-

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
JSSET	JSSET #bbbbbb,S:<pp>,aaaa	4	-	-	-
	JSSET #bbbbbb,S:<ea>,aaaa	4	1	-	-
	JSSET #bbbbbb,S:<aa>,aaaa	4	-	-	-
	JSSET #bbbbbb,DDDDDD,aaaa	4	-	-	-
	JSSET #bbbbbb,S:<qq>,aaaa	4	-	-	-
LSL	LSL sss,D	1	-	-	-
	LSL #ii,D	1	-	-	-
LSR	LSR #ii,D	1	-	-	-
	LSR sss,D	1	-	-	-
LRA	LRA (PC+Rn)->0DDDDDD	3	-	-	-
	LRA (PC+aaaa)->0DDDDDD	3	-	-	-
LUA, LEA	LUA ea->0DDDDDD	3	-	-	-
	LUA (Rn+aa)->01DDDD	3	-	-	-
MACI	MACI +/- #iiiiii,QQ,D	2	-	-	-
MAC	MAC +/- 2**s,QQ,d	1	-	-	-
	MAC S1,S2,D (su,uu)	1	-	-	-
MAX	MAX A,B	1	-	-	-
MAXM	MAXM A,B	1	-	-	-
MACRI	MACRI +/- #iiiiii,QQ,D	2	-	-	-
MACR	MACR +/- 2**s,QQ,d	1	-	-	-
MERGE	MERGE SSS,D	1	-	-	-

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
MOVE	No parallel data Move (DALU)	1	-	-	-
	MOVE #xx --> DDDDD	1	-	-	-
	MOVE ddddd --> DDDDD	1	-	-	-
	U move	1	-	-	-
	MOVE S:<ea>,DDDDD	1	1	1	1
	MOVE S:<aa>,DDDDD	1	-	-	-
	MOVE S:<Rn+aa>,DDDD	2	-	-	-
	MOVE S:<Rn+aaaa>,DDDDDD	3	-	-	-
	MOVE d -> X Y:<ea>,YY	1	1	1	1
	MOVE X:<ea>,XX & d -> Y	1	1	1	1
	MOVE A -> X:<ea> X0 A	1	1	-	-
	MOVE B -> X:<ea> X0 B	1	1	-	-
	MOVE Y0 -> A A Y:<ea>	1	1	-	-
	MOVE Y0 -> B B Y:<ea>	1	1	-	-
	MOVE L:<ea>,LLL	1	1	1	-
	MOVE L:<aa>,LLL	1	-	-	-
	MOVE X:<ea>,XX & Y:<ea>,YY	1	-	-	-
MOVEC	MOVEC #xx -> 1DDDDD	1	-	-	-
	MOVEC S:<ea>,1DDDDD	1	1	1	1
	MOVEC S:<aa>,1DDDDD	1	-	-	-
	MOVEC DDDDDD,1dddd	1	-	-	-
MOVEM	MOVEM P:<ea>,DDDDDD	6	1	1	-
	MOVEM P:<aa>,DDDDDD	6	-	-	-

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
MOVEP	MOVEP S:<pp>,s:<ea>	2	1	1	0
	MOVEP S:<pp>,P:<ea>	6	1	1	-
	MOVEP S:<pp>,DDDDDD	1	-	-	-
	MOVEP X:<qq>,s:<ea>	2	1	1	0
	MOVEP Y:<qq>,s:<ea>	2	1	1	0
	MOVEP X:<qq>,DDDDDD	1	-	-	-
	MOVEP Y:<qq>,DDDDDD	1	-	-	-
	MOVEP S:<qq>,P:<ea>	6	1	1	-
MPY	MPY S1,S2,D (su,uu)	1	-	-	-
	MPY +/- 2**s,QQ,d	1	-	-	-
MPYI	MPYI +/- #iiiiii,QQ,D	2	-	-	-
MPYR	MPYR +/- 2**s,QQ,d	1	-	-	-
MPYRI	MPYRI +/- #iiiiii,QQ,D	2	-	-	-
NOP	NOP	1	-	-	-
NORM	NORM	5	-	-	-
NORMF	NORMF SSS,D	1	-	-	-
OR	OR #iiiiii,D	2	-	-	-
	OR #iii,D	1	-	-	-
ORI	ORI EE	3	-	-	-
PFLUSH	PFLUSH	1	-	-	-
PFLUSHUN	PFLUSHUN	1	-	-	-
PFREE	PFREE	1	-	-	-
PLOCK	PLOCK <ea>	2	1	1	-
PLOCKR	PLOCKR (PC+aaaa)	4	-	-	-
PUNLOCK	PUNLOCK <ea>	2	1	1	-
PUNLOCKR	PUNLOCKR (PC+aaaa)	4	-	-	-

Instruction Mnemonic	Instruction Format	T	+ p r u	+ l a b	+ l i m
REP	REP #xxx	5	-	-	-
	REP DDDDDD	5	-	-	-
	REP S:<ea>	5	1	-	-
	REP S:<aa>	5	-	-	-
RESET	RESET	7	-	-	-
RTI/RTS	RTI	3	-	-	-
	RTS	3	-	-	-
STOP	STOP	10	-	-	-
SUB	SUB #iiii,D	2	-	-	-
	SUB #iii,D	1	-	-	-
Tcc	Tcc JJJ -> D ttt TTT	1	-	-	-
	Tcc JJJ -> D	1	-	-	-
	Tcc ttt -> TTT	1	-	-	-
TRAP/TRAPcc	TRAP	9	-	-	-
	TRAPcc	9	-	-	-
WAIT	WAIT	10	-	-	-

B-3 INSTRUCTION SEQUENCE DELAYS

Due to the pipeline nature of the DSP56300 Core, there are certain instruction sequences that cause a delay in the execution of instructions involved in that sequences. Most of these sequences are caused by a source-destination conflict or by the need to access the external bus.

There are six types of sequence delays:

1. External Bus Wait States.
2. External Bus Contention.
3. Instruction fetch delays.
4. Data ALU Interlock.
5. Address Generation Interlock.
6. Stack Extension delays.
7. Pipeline interlocks.

B-3.1 External Bus Wait States

An External Bus Wait State is caused by an instruction accessing the external bus for data read or write. In this case, the execution time of the instruction is increased by the number of clock cycles equal to the number of wait states that is programmed for that external data access. The exact number of wait states depends on the type of memory accessed, as described in Chapter 2 of this document.

B-3.2 External Bus Contention

An External Bus Contention is caused by an attempt to simultaneously access the external bus with more than one source (REFRESH request from the internal DRAM controller, X memory space, Y memory space, P memory space or a DMA channel). In this case, the execution time of the instructions that reside in the pipeline at that time is lengthen by a number of clock cycles that is equal to the number of simultaneous requests minus 1. For every request, additional wait states will be added according to the memory speed, as described in Section B-3.1 above. If one of these requests is a REFRESH request, than this request will be the first to receive mastership over the external bus. If one of these requests is the DMA, then the following cases should be distinguished:

1. The DMA has higher priority than the CORE. In this case, the DMA will receive full control over the external bus and will hold that control for all its transfers, provided that they are all external. After the DMA finished its transfers, the bus will be given to the memory space in the order of P (first), X (second) and Y (last).
2. The DMA has a priority equal to the CORE. In this case, the bus will be given to the memory space in the order of P (first), X (second), Y (third) and DMA (last).
3. The DMA has lower priority than the CORE. In this case, the DMA will wait

for a free external bus slot and the bus will be given to the memory space in the order of P (first), X (second) and Y (last).

B-3.3 Instruction Fetch delays

An external Instruction Fetch is caused by one of the following two cases:

- Instruction Cache is disabled and a fetch to an external address is initiated. In this case, an external fetch will be initiated.
- Instruction Cache is enabled and a program fetch to an instruction that does not exist in the instruction cache is initiated. This produces a miss indication from the instruction cache control unit, and an external fetch will be initiated.

In both cases, if the external memory is an SSRAM (Synchronous Static RAM), one cycle delay is inserted after the external access. The effective number of stall states in the pipeline will be the number specified in the Bus Control Register (BCR) + 1. If the external memory is either SRAM or DRAM, this one cycle delay will not be inserted.

During the operation of the Instruction Cache Controller, the following special cases should be distinguished:

1. When two identical locations are fetched one after another, and the first one is detected as miss, the second one will also be detected as miss although it was written to the cache memory. The number of wait states added will be the same as the general miss case.
2. When the Burst Mode is enabled, then upon detection of miss, up to 4 fetch requests will be initiated by the core. The exact number of fetch requests depends on the two least significant bits of the address of the initiating fetch that was detected as miss -

2 List Significant bits	Number of generated fetches	Number of clock cycles added
11	1	0, as if Burst is Disabled
10	2	2
01	3	3
00	4	4

All these requests will be considered as one for the detection of contention states.

B-3.4 Data ALU Interlock

A Data ALU Interlock may be caused by one of the following sequences:

B-3.4.1 Arithmetic Stall

This interlock is caused by an instruction that uses one of the data ALU accumulators or accumulator-parts (A0, A1, A2, B0, B1, B2) as a source register to the move portion of that instruction, while the preceding instruction was an arithmetic instruction (i.e. an instruction that uses the internal Data-ALU data paths) that used the same accumulator as its destination. The execution of the initiating instruction will be delayed by **one** clock cycle.

B-3.4.2 Transfer Stall

This interlock is caused by an instruction that uses one of the data ALU registers (A0, A1, A2, B0, B1 or B2) or accumulators (A or B) as a source register to the move portion of that instruction, while the preceding instruction used the corresponding accumulator (A or B) or one of the data ALU registers (A0, A1, A2, B0, B1 or B2) that comprise this accumulator as its destination register in the move portion of that instruction. The execution of the initiating instruction will be delayed by **one** instruction cycle.

B-3.4.3 Status Stall

This interlock is caused by an instruction that reads the contents of the Status Register (SR) for either move operation or bit testing, while the **preceding or the second preceding** instruction was an arithmetic instruction (i.e. an instruction that uses the internal Data-ALU data paths). The execution of the initiating instruction will be delayed by **two or one** (respectively) instruction cycles.

B-3.5 Address Registers Interlocks

B-3.5.1 Conditional Transfer Interlock

This interlock is caused by a Transfer On-Condition (Tcc) instruction followed by an instruction that explicitly specifies one of the address generation registers: R0..R7 as its source operand. The execution of the second instruction will be delayed by **one** instruction cycle.

B-3.5.2 Address Generation Interlock

An Address Generation Interlock is caused by a move portion of an instruction that uses one of the AGU registers R0-R7 for address generation or for address calculation, while one of the three preceding instruction cycles used one of the register-set (Ri, Ni or Mi) members as a destination register in its move portion. For example, consider the following code:

```

I1 MOVE #$addr,R0
I2 NOP
I3 NOP
I4 NOP
I5 MOVE #$offset,N0
I6 MOVE X:(R0)+,Y1

```

In this example, the instruction I6 will cause an Address Generation interlock because it used R0 as the source for address generation on the X Address Bus while the preceding instruction, I5, used N0 as its destination.

Three types of Address Generation Interlock exist - type0, type1 and type2, depending on the distance, in term of clock cycles, between the instruction causing the interlock and the preceding instruction that used the AGU register as a destination. The following figure describes an example to each of the types:

Type0 Interlock	Type1 Interlock	Type2 Interlock
I1 MOVE #\$addr,R0	I1 MOVE #\$addr,R0	I1 MOVE #\$addr,R0
I2 MOVE X:(R0)+,Y1	I2 CLR A	I2 CLR A
	I3 MOVE X:(R0)+,Y1	I3 INC B
		I4 MOVE X:(R0)+,Y1

When an Address Generation Interlock of **Type0** is detected (during the decoding of I2 in the example), **three** nop clock cycles will be automatically inserted before the execution of the instruction starts. When an Address Generation Interlock of **Type1** is detected (during the decoding of I3 in the example), **two** nop clock cycles will be automatically inserted before the execution of the instruction starts. When an Address Generation Interlock of **Type2** is detected (during the decoding of I4 in the example), **one** nop clock cycle will be automatically inserted before the execution of the instruction starts.

Note that only **clock cycles** are counted to determine when interlock cycles should be inserted. Whenever an instruction using one of the AGU registers as an Address Generation enters the decoding stage of the DSP56300 Core, the distance from that instruction to the preceding instruction that used the register as destination is measured in term of clock cycles to determine the existence and type of Address Generation Interlock. Once an Address Generation Interlock is detected, the appropriate number of nop clock cycles is inserted. The following instructions take these additional cycles into account for the

detection of a possible new Address Generation Interlock. The following example demonstrates this feature.

```
I1 MOVE # $addr, R0
I2 CLR A
I3 MOVE X: (R0)+, Y1
I4 MOVE X: (R0)+, Y0
```

In this example, a type1 Address Generation Interlock is detected during the decoding phase of I3 and two nop cycles are inserted before the execution of that instruction. During the decoding of I4, no Address Generation Interlock is detected - no nop cycles are inserted! If, however, I3 would have been an instruction that does not use R0, a type2 Address Generation Interlock would have been detected during the decoding phase of I4 and one nop cycle would have been inserted before the execution of that instruction.

B-3.6 Stack Extension Delays

Some instructions access the System Stack as part of their normal activity. If, however, the stack is full, or if it is empty, the special stack extension mechanism is engaged and the access will be completed only after an access to data memory is automatically performed. This will delay the decoding and the execution phases of that instruction. A stack-full or stack-empty states are defined by the contents of the SC (Stack Counter) register. When the stack counter equals 14, it means that the on-chip hardware stack has 14 words (a stack word is a 48-bit long word combined from the low and the high portions of the stack) inside. The stack is declared as stack-full, and any additional push operation will activate the stack extension mechanism. When the stack counter equals 2, it means that the on-chip hardware stack has only 2 words inside. The stack is declared as stack-empty, and any additional pop operation will activate the stack extension mechanism.

The following instructions/cases causes an access to the system stack and may engage the stack extension mechanism:

SUBcc This denotes all the conditional and unconditional 'Jump to Subroutine' instructions e.g. JSR, JSSET, BRCLR etc. These instructions perform a stack PUSH operation that stores the PC and the SR on top of the stack, for the use of the 'Return from Subroutine' instruction that will terminate the subroutine execution.

RET This denotes the two 'Return from Subroutine' instructions RTS and RTI. These instructions perform a stack POP operations that pulls the PC and (optionally) the SR out from the top of stack in order to return back to the calling procedure and to restore the status bits and loop flag state.

END-OF-DO This is a condition achieved by the internal hardware inside the Program Control Unit. This hardware detects the case where a fetch from the last address of a loop is initiated when the Loop Counter equals 1. This condition defines the end of

the loop, thus performs a stack POP operation. This POP operation restores the loop flag, purges the top of stack (PC:SR) and pulls LA and LC from the new top of stack.

LOOP This denotes all the hardware-loop initiating instructions e.g. DO, DOR with all their options. These instructions perform a stack double-PUSH operation that first stores the previous values of LA and LC on top of the stack. Then the DO instruction stores the contents of SR and PC on the new top of stack. This PC value is used every loop iteration in order to go back to the top of loop location and start fetch from there. **DO** performs two accesses to the stack instead of the normal single access done by most stack operations.

ENDDO This is a special instruction that forces an end-of-do condition during a hardware loop. Like **END-OF-DO**, **ENDDO** performs two accesses to the stack instead of the normal single access done by most stack operations.

SSHWR This denotes all the explicit stack PUSH instructions that uses SSH as their destination, e.g. the instruction MOVE R0,SSH.

SSHRD This denotes all the explicit stack POP instructions that uses SSH as their source, e.g. the instruction MOVE SSH,Y1.

The following table describes how many clock cycles are added in the various instructions/cases described above:

Table B-2. Stack Extension Delays

CASE	Stack Full Condition (+ clock cycles)	Stack Empty Condition (+ clock cycles)
SUBcc	2	-
RET	-	3
END-OF-DO	-	5
DO	4	-
ENDDO	-	5
SSHWR	2	-
SSHRD	-	3

B-3.7 Program Flow-Control delays

During the execution of flow-control instructions, some boundary cases exist and introduce interlocks to the program flow. These interlocks lengthen the decoding phase of

the instructions thus delays the execution of them.

Legend:

- I1 - An address of an instruction, where I2, I3, I4 are used to indicate the next instructions in the program flow.
- MOVE - any type of MOVE, MOVEM, MOVEP, MOVEC, BSET, BCHG, BCLR, BTST.
- (LA) - the last address of a DO LOOP.
- (LA-i) - the address of an instruction word located at LA-i.
- CR - Control Register, every one of the registers LA, LC, SR, SP, SC, SSH, SSL, OMR.

Note: The sequences described in this section represent very unusual operations which probably would never be used. The detection of these cases and hence the generation of interlocks is done in order to maintain an object code compatibility between the DSP56300 Core and the 56k Family of Digital Signal Processors.

B-3.7.1 MOVE to CR

Whenever I1 is a MOVE to CR and it is located at (LA-3), (LA-4) or (LA-5) then the decoding phase of I3 is delayed by 3 clock cycles. The decoding of the instruction following (LA) will be also delayed by an additional 1 clock cycle.

B-3.7.2 MOVE from CR

Whenever I1 is a MOVE from CR and it is located at (LA-2) then the decoding phase of the instruction following the instruction at (LA) will be delayed by 1 clock cycle.

B-3.7.3 MOVE to SP/SC

Whenever I1 is a MOVE to SP or to SC then the decoding phase of I3 will be delayed by up to 3 clock cycles.

B-3.7.4 MOVE to LA register

Whenever I1 is a MOVE to the LA register and the preceding instruction was a MOVE to SR then the decoding phase of I3 will be delayed by 3 clock cycles.

B-3.7.5 MOVE to SR

Whenever I1 is a MOVE to SR then the decoding phase of I2 will be delayed by 1 clock cycle.

B-3.7.6 MOVE to SSH/SSL

Whenever I1 is a MOVE to SSH or to SSL and I3 is any one of the instructions DO, DOR, RTI, RTS, ENDDO or BRKcc then the decoding phase of I3 will be delayed by 3 clock

cycles.

B-3.7.7 JMP to (LA) or to (LA-1)

Whenever I1 is any type of JMP with the target address equals to (LA) or to (LA-1) then the decoding phase of the instruction following the instruction at (LA) will be delayed by 2 or 1 clock cycles respectively.

B-3.7.8 RTI to (LA) or to (LA-1)

Whenever I1 is an RTI instruction whose return address is (LA) or (LA-1) then the decoding phase of the instruction following the instruction at (LA) will be delayed by 2 or 1 clock cycles respectively.

B-3.7.9 MOVE from SSH

Whenever I1 is a MOVE from SSH and it is located at (LA-2) then the decoding phase of the instruction following the instruction at (LA) will be delayed by 1 clock cycle.

B-3.7.10 Conditional Instructions

Whenever I1 is a conditional change of flow instruction e.g. Jcc and the condition is false then the decoding phase of I2 will be delayed by 1 clock cycle.

B-3.7.11 Interrupt Abort

Whenever I1 is an instruction which its decoding phase is longer than 1 cycle then it may be aborted by the interrupt control unit. In this case, 1 clock cycle “hole” will be inserted to the pipeline after which the instruction at the interrupt vector will be decoded.

B-3.7.12 Degenerated DO loop

Whenever I1 is a DO loop but the loop contains only one instruction then the decoding phase of I1 is lengthen by 1 clock cycle.

B-3.7.13 Annulled REP and DO

If the repeat count of a REP or DO instruction is 0 then the decoding phase of the REP or the DO instruction is lengthen by 1 or 3 clock cycles respectively.

Note: Annulled REP or DO can be executed only when the Sixteen-Bit compatibility mode in the Status Register (SR[13]) is cleared. When this bit is set, a annulled REP will execute $2^{**}16$ times.

B-4 INSTRUCTION SEQUENCE RESTRICTIONS

Due to the pipelined nature of the DSP56300 Core 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.

Legend:

- MOVE - any type of MOVE, MOVEM, MOVEP, MOVEC.
- LA - the last address of a DO LOOP
- Two-words <inst> - a double-word instruction in which the 2nd word is used as an immediate data or absolute address
- Single-word <inst> - an instruction with an addressing mode that does not need a 2nd word extension

B-4.1 Restrictions Near the End of DO Loops

Proper DO loop operation is not guaranteed if an instruction sequence similar to one of the sequences described below is used.

B-4.1.1 At LA-3

The following instructions should not start at address LA-3:

- MOVE to {LA}
- BCHG, BSET, BCLR on {LA}
- Two-words MOVE to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- Two-words MOVE from SSH
- Two-words PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.1.2 At LA-2

The following instructions should not start at address LA-2:

- DO, DOR, DOFOREVER
- MOVE to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE from SSH
- BCHG, BSET, BCLR on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- REP on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BTST on SSH
- JCLR, JSET, JSCLR, JSSET, BRCLR, BRSET, BSCLR, BSSET on SSH
- Two-words MOVE from {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}

-
- ANDI, ORI on MR
 - BRKcc
 - PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.1.3 At LA-1

The following instructions should not start at address LA-1:

- DO, DOR, DOFOREVER
- MOVE to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE from {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- REP on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BCHG, BSET, BCLR, BTST on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- JCLR, JSET, JSCLR, JSSET on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BRCLR, BRSET, BSCLR, BSSET on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- ANDI, ORI on MR
- BRKcc
- ENDDO
- PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.1.4 At LA

The following instructions should not start at address LA:

- Any Two-word instruction
- MOVE to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE from SSH
- BCHG, BSET, BCLR on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BTST on SSH
- ANDI, ORI on MR
- BRKcc
- JMP, JSR, BRA, BSR, Jcc, JScc, Bcc, BScc
- REP
- RESET, STOP, WAIT
- RTI, RTS
- ENDDO
- PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.2 General DO Restrictions

A DO loop should be initialized and aborted by using only the following instructions: DO, DOR, ENDDO and BREAKcc. The LF and the FV bits in the Status Register (SR) should not be explicitly changed by using one of the MOVE, BCHG, BSET, BCLR, ANDI or ORI instructions.

Proper DO loop operation is not guaranteed if an instruction sequence similar to one of

the sequences described below is used.

- SSH can not be used as the source for the Loop-Count for a DO or DOR instruction
- The following instructions should not appear immediately before a DO, DOR or DOFOREVER:
 - MOVE from SSH
 - BTST on SSH
 - BCHG, BCLR, BSET, MOVE to/on {LA, LC, SR, SP, SC, SSH, SSL}
 - JSR, JScc, JSSET, JSCLR to LA whenever LF is set
 - BSR, BScc, BSSET, BSCLR, to LA whenever LF is set

B-4.3 ENDDO Restrictions

The instructions in the following list should not appear immediately before an ENDDO instruction:

- ANDI, ORI on MR
- MOVE from SSH
- BTST on SSH
- BCHG, BCLR, BSET, MOVE on/to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}

B-4.4 BRKcc Restrictions

The instructions in the following list should not appear immediately before a BRKcc instruction:

- Every arithmetic instruction
- IFcc, Tcc
- BCHG, BCLR, BSET, MOVE on/to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}

B-4.5 RTI and RTS Restrictions

The instructions in the following list should not appear immediately before an RTI instruction:

- MOVE, BCHG, BCLR, BSET on {SR, SSH, SSL, SP, SC}
- MOVE, BTST from/on SSH
- ANDI, ORI on {MR, CCR}

The instructions in the following list should not appear immediately before an RTS instruction:

- MOVE, BCHG, BCLR, BSET on {SR, SSH, SSL, SP, SC}
- MOVE, BTST from/on SSH

B-4.6 SP, SC and SSH/SSL Manipulation Restrictions

The instructions in the following list #a should not appear immediately before any of the instructions from the following list #b.

List #a:

- MOVE to {SP,SC}

-
- BCHG, BSET, BCLR on {SP,SC}

List #b:

- MOVE from {SSH,SSL}
- BTST, BCHG, BSET, BCLR on {SSH,SSL}
- JSET, JCLR, JSSET, JSCLR, BRSET, BRCLR, BSSET, BSCLR on {SSH,SSL}

B-4.7 Fast Interrupt Routines

The following instructions may not be used in a fast interrupt routine:

- DO, DOR, DOFOREVER, REP
- ENDDO, BRKcc,
- RTI, RTS
- STOP, WAIT
- TRAP, TRAPcc
- ANDI, ORI on {MR, CCR}
- MOVE from SSH
- BTST on SSH
- MOVE to {LA, LC, SR, SP, SC, SSH, SSL}
- BCHG, BSET, BCLR on {LA, LC, SR, SP, SC, SSH, SSL}
- PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.8 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 a REP instruction (can not be repeated):

- REP, DO, DOR, DOFOREVER
- ENDDO, BRKcc
- JMP, Jcc, JCLR, JSET
- JSR, JScC, JSCLR, JSSET
- BRA, Bcc, BRSET, BRCLR
- BSR, BScC, BSSET, BSCLR
- RTS, RTI
- TRAP, TRAPcc
- WAIT, STOP
- PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.9 Stack Extension Restrictions

The following instructions, related to the operation of the on-chip hardware stack extension, may not be used whenever the stack extension is enabled:

- MOVE to EP
- BCHG, BSET, BCLR on EP

-
- MOVE to SC with a value greater than 15

B-4.10 Instruction Cache General Restrictions

The following instructions may not be used whenever the Instruction Cache is disabled. Using these instructions when the Instruction Cache is disabled will generate an illegal interrupt.

- PLOCK, PLOCKR
- PUNLOCK, PUNLOCKR
- PFREE, PFLUSH, PFLUSHUN

B-5 PERIPHERAL PIPELINE RESTRICTIONS

The DSP56300 Core is based on a highly optimized pipeline engine. Despite the relatively deep pipeline (seven stages) the latency effects normally associated with long pipelines have been kept to a minimum such as, in fact, these effects are transparent to the user. Design techniques, such as forwarding and interlocking, alleviate the need for the user to have a thorough knowledge of the machine's pipeline in order to avoid data dependencies. This knowledge becomes relevant only when further optimization of the code is pursued. Therefore the pipeline is hidden from the user for the vast majority of the application development stage.

There is, however, an aspect of the machine's pipeline that is exposed to the user and this is the area of peripheral activity. This section describes the cases in which the user must take precautions in order to achieve the desired functionality.

B-5.1 Polling a peripheral device for write

When writing data to a peripheral device there is a two cycle pipeline delay until any status bits affected by this operation are updated. For example, the operates a peripheral port using the polling technique. The user will look for the data empty flag set. After this status bit is set, the user will write new data to the transmit data register. If the user attempts to read the status bit within the next two cycles, due to the pipeline delays associated with the peripheral operations, the user will mistakenly read the flag as set. Therefore the user will assume that the transmit data register is empty and will write a new data word that will in fact overwrite the previously written data. In order to achieve the correct functionality the user must wait (at least) two cycles before attempting to read the status register following a write to the transmit data register. Following is an example of the correct sequence for transmit operations:

```

send
    movep    x:(r0)+,x:STX          ; send new data
    nop                      ; pipeline delay
    nop                      ; pipeline delay
poll
    jclr     #TDE,x:SCSR,poll      ; wait for data empty
    jmp      send                 ; go to send data

```

B-5.2 Writing to a read-only register

Writing to a read-only register is an operation that basically has no effect but if a read operation from the same register is attempted within the following two cycles, the value of the read data will be the value of the data that was written instead of the unchanged data of the read-only register. In order to ensure that the correct data is read after the write operation, the user should wait (at least) two cycles before performing the read.

Appendix C BENCHMARK PROGRAMS

C-1 INTRODUCTION

The following benchmarks illustrate the source code syntax and programming techniques for the DSP56300 Core. The assembly language source is organized into 6 columns as shown below.

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment
FIR	MAC	X0,Y0,A	X:(R0)+,X0	Y:(R4)+,Y0	;Do each tap

The Label column is used for program entry points and end of loop indication. The Opcode column indicates the Data ALU, Address ALU or Program Controller operation to be performed. The Operands column specifies the operands to be used by the opcode. The X Bus Data specifies an optional data transfer over the X Bus and the addressing mode to be used. The Y Bus Data specifies an optional data transfer over the Y Bus and the addressing mode to be used. The Comment column is used for documentation purposes and does not affect the assembled code. The Opcode column must always be included in the source code.

C-2 SET OF BENCHMARKS

C-2.1 Real Multiply

$$c = a \times b$$

				Prog wrds	Clock Cycles
move		x:(r0),x0	y:(r4),y0	1	1
mpyr	x0,y0,a			1	1
move		a,x:(r1)		1	2 i'lock
Totals				3	4

C-2.2 N Real Multiplies

$$c(i) = a(i) \times b(i) \quad i = 1, 2, \dots, N$$

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r4		b(i)
r1	c(i)	

					Prog wrds	Clock Cycles
	move	#AADDR,r0				
	move	#BADDR,r4				
	move	#CADDR,r1				
	move		x:(r0)+,x0	y:(r4)+,y0	1	1
	mpyr	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	1	1
	do	#N-1,end			2	5
	mpyr	x0,y0,a	a,x:(r1)+	y:(r4)+,y0	1	1
	move		x:(r0)+,x0		1	1
end						
	move		a,x:(r1)+		1	1
				Totals	7	2N+8

C-2.3 Real Update

$$d = c + a \times b$$

					Prog wrds	Clock Cycles
	move	#AADDR,r0				
	move	#BADDR,r4				
	move	#CADDR,r1				
	move	#DADDR,r2				
	move		x:(r0),x0	y:(r4),y0	1	1
	move		x:(r1),a		1	1
	macr	x0,y0,a			1	1
	move		a,x:(r2)		1	2 i'lock
				Totals	4	5

C-2.4 N Real Updates

$$d(i) = c(i) + a(i) \times b(i) \quad i = 1, 2, \dots, N$$

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r4		b(i)
r1	c(i)	
r5		d(i)

			Prog wrds	Clock Cycles
move	#AADDR,r0			
move	#BADDR,r4			
move	#CADDR,r1			
move	#DADDR,r5			
move	x:(r0)+,x0	y:(r4)+,y0 ;	1	1
move	x:(r1)+,a	;	1	1
move	x:(r1)+,b	;	1	1
do	#N/2,end	;	2	5
macr	x0,y0,a	x:(r0)+,x1 y:(r4)+,y1 ;	1	1
macr	x1,y1,b	x:(r0)+,x0 y:(r4)+,y0 ;	1	1
move	x:(r1)+,a	a,y:(r5)+ ;	1	1
move	x:(r1)+,b	b,y:(r5)+ ;	1	1
end				
		Totals	9	2N+8

C-2.5 Real Correlation Or Convolution (FIR Filter)

$$c(n) = \sum_{i=0}^{N-1} [a(i) \times b(n-i)]$$

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r4		b(i)

					Prog wrds	Clock Cycles
move	#AADDR,r0					
move	#BADDR,r4			;		
move	#N-1,m4			;		
move	m4,m0			;		
movep	y:input,y:(r4)			;	1	2
clr	a	x:(r0)+,x0	y:(r4)-,y0	;	1	1
rep	#N-1			;	1	5
mac	x0,y0,a	x:(r0)+,x0	y:(r4)-,y0	;	1	1
macr	x0,y0,a		(r4)+	;	1	1
movep	a,y:output			;	1	2 i'lock
				Totals	6	N+14

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r1	b(i)	

					Prog wrds	Clock Cycles
move	#AADDR,r0					
move	#BADDR,r1			;		
move	#N-1,m1			;		
move	m1,m0			;		
movep	y:input,x:(r1)			;	1	2
clr	a	x:(r0)+,x1		;	1	1
do	#N-1,end			;	2	5
move		x:(r1)-,x0		;	1	1
mac	x0,x1,a	x:(r0)+,x1		;	1	1
end				;		
move		x:(r1)-,x0		;	1	1
macr	x0,x1,a	(r1)+		;	1	1
movep	a,y:output			;	1	2 i'lock
				Totals	9	2N+10

C-2.6 Real * Complex Correlation Or Convolution (FIR Filter)

$$cr(n) = jci(n) = \sum_{i=0}^{N-1} [(ar(i) + jai(i)) \times b(n-i)]$$

$$cr(n) = \sum_{i=0}^{N-1} ar(i) \times b(n-i) \quad ci(n) = \sum_{i=0}^{N-1} ai(i) \times b(n-i)$$

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r4	b(i)	
r1	cr(n)	ci(n)

				Prog wrds	Clock Cycles
move	#AADDR,r0		;		
move	#BADDR,r4		;		
move	#CADDR,r1		;		
move	#N-1,m4		;		
move	m4,m0		;		
movep	y:input,x:(r4)		;	1	2
clr	a	x:(r0),x0	;	1	1
clr	b	x:(r4)-,x1	y:(r0)+,y0 ;	1	1
do	#N-1,end		;	2	5
mac	x0,x1,a	x:(r0),x0	;	1	1
mac	y0,x1,b	x:(r4)-,x1	y:(r0)+,y0 ;	1	1
end					
macr	x0,x1,a		;	1	1
macr	y0,x1,b	(r4)+	;	1	1
move		a,x:(r1)	;	1	1
move			b,y:(r1) ;	1	1
			Totals	11	2N+11

C-2.7 Complex Multiply

$$cr + jci = (ar + jai) \times (br + jbi)$$

$$cr = ar \times br - ai \times bi \quad ci = ar \times bi + ai \times br$$

Memory map:

pointer	X mem	Y mem
r0	ar	ai
r4	br	bi
r1	cr	ci

				Prog wrds	Clock Cycles
move	#AADDR,r0				
move	#BADDR,r4				
move	#CADDR,r1				
move		x:(r0),x1	y:(r4),y0	1	1
mpy	y0,x1,b	x:(r4),x0	y:(r0),y1	1	1
macr	x0,y1,b			1	1
mpy	x0,x1,a			1	1
macr	-y0,y1,a		b,y:(r1)	1	1
move		a,x:(r1)		1	2 i'lock
Totals				6	7

C-2.8 N Complex Multiplies

$$\begin{aligned}
 cr(i) + jci(i) &= (ar(i) + jai(i)) \times (br(i) + jbi(i)) & i = 1, 2, \dots, N \\
 cr(i) &= ar(i) \times br(i) - ai(i) \times bi(i) \\
 ci(i) &= ar(i) \times bi(i) + ai(i) \times br(i)
 \end{aligned}$$

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r5	cr(i)	ci(i)

				Prog wrds	Clock Cycles
	move	#AADDR,r0	;		
	move	#BADDR,r4	;		
	move	#CADDR-1,r5	;		
	move	x:(r0),x1	y:(r4),y0 ;	1	1
	move	x:(r5),a	;	1	1
	do	#N,end	;	2	5
	mpy	y0,x1,b	x:(r4)+,x0 y:(r0)+,y1 ;	1	1
	macr	x0,y1,b	a,x:(r5)+ ;	1	1
	mpy	-y0,y1,a	y:(r4),y0 ;	1	1
	macr	x0,x1,a	x:(r0),x1 b,y:(r5) ;	1	1
end					
	move	a,x:(r5)	;	1	2 i'lock
		Totals		9	4N+9

C-2.9 Complex Update

$$dr + jdi = (cr + jci) + (ar + jai) \times (br + jbi)$$

$$dr = cr + ar \times br - ai \times bi \quad di = ci + ar \times bi + ai \times br$$

Memory map:

pointer	X mem	Y mem
r0	ar	ai
r4	br	bi
r1	cr	ci
r2	dr	di

				Prog wrds	Clock Cycles
	move	#AADDR,r0			
	move	#BADDR,r4			
	move	#CADDR,r1			
	move	#DADDR,r2			
	move		y:(r1),b ;	1	1
	move	x:(r0),x1	y:(r4),y0 ;	1	1
	mac	y0,x1,b	x:(r4),x0 y:(r0),y1 ;	1	1
	macr	x0,y1,b	x:(r1),a ;	1	1
	mac	x0,x1,a	;	1	1
	macr	-y0,y1,a	b,y:(r2) ;	1	1
	move	a,x:(r2)	;	1	2 i'lock
		Totals		7	8

C-2.10 N Complex Updates

$$dr(i) + jdi(i) = (cr(i) + jci(i)) + (ar(i) + jai(i)) \times (br(i) + jbi(i))$$

$$dr(i) = cr(i) + ar(i) \times br(i) - ai(i) \times bi(i)$$

$$di(i) = ci(i) + ar(i) \times bi(i) + ai(i) \times br(i)$$

$$i = 1, 2, \dots, N$$

Memory map:

pointer	X mem	Y mem
r0	ar(i) ; ai(i)	
r4		br(i) ; bi(i)
r1	cr(i) ; ci(i)	
r5		dr(i) ; di(i)

				Prog wrds	Clock Cycles
	move	#AADDR,r0	;		
	move	#BADDR,r4	;		
	move	#CADDR,r1	;		
	move	#DADDR-1,r5	;		
	move	x:(r0)+,x1	y:(r4)+,y0 ;	1	1
	move	x:(r1)+,b	y:(r5),a ;	1	1
	do	#N,end	;25 ;	2	5
	mac	y0,x1,b	x:(r0)+,x0 y:(r4)+,y1 ;	1	1
	macr	-x0,y1,b	x:(r1)+,a a,y:(r5)+ ;	1	1
	mac	x0,y0,a	x:(r1)+,b b,y:(r5)+ ;	1	2 i'lock
	macr	x1,y1,a	x:(r0)+,x1 y:(r4)+,y0 ;	1	1
end					
	move		a,y:(r5)+ ;	1	2 i'lock
			Totals	9	5N+9

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r1	cr(i)	ci(i)
r5	dr(i)	di(i)

				Prog wrds	Clock Cycles
move	#AADDR,r0		;		
move	#BADDR,r4		;		
move	#CADDR,r1		;		
move	#DADDR-1,r5		;		
move		x:(r5),a	;	1	1
move		x:(r0),x1	y:(r4),y0	1	1
move		x:(r4)+,x0	y:(r1),b	1	1
do	#N,end		;	2	5
mac	y0,x1,b	a,x:(r5)+	y:(r0)+,y1	1	1
macr	x0,y1,b	x:(r1)+,a		1	1
mac	-y0,y1,a	y:(r4),y0		1	1
macr	x0,x1,a	x:(r0),x1	b,y:(r5)	1	1
move		x:(r4)+,x0	y:(r1),b	1	1
end					
move		a,x:(r5)	;	1	1
		Totals		11	5N+9

C-2.11 Complex Correlation Or Convolution (FIR Filter)

$$cr(n) + jci(n) = \sum_{i=0}^{N-1} [(ar(i) + jai(i)) \times (br(n-i) + jbi(n-i))]$$

$$cr(n) = \sum_{i=0}^{N-1} [ar(i) \times br(n-i) - ai(i) \times bi(n-i)]$$

$$ci(n) = \sum_{i=0}^{N-1} [ar(i) \times bi(n-i) + ai(i) \times br(n-i)]$$

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r1	cr(i)	ci(i)

				Prog wrds	Clock Cycles
	move	#AADDR,r0	;		
	move	#BADDR,r4	;		
	move	#CADDR,r1			
	move	#N-1,m4			
	move	#m4,m0			
	movep	y:input,x:(r4)		1	2
	movep	y:input,y:(r4)		1	2
	clr	a	;	1	1
	clr	b	x:(r0),x1 y:(r4),y0 ;	1	1
	do	#N-1,end	;	2	5
	mac	y0,x1,b	x:(r4)-,x0 y:(r0)+,y1 ;	1	1
	mac	x0,y1,b	;	1	1
	mac	x0,x1,a	;	1	1
	mac	-y0,y1,a	x:(r0),x1 y:(r4),y0 ;	1	1
end					
	mac	y0,x1,b	x:(r4),x0 y:(r0)+,y1 ;	1	1
	macr	x0,y1,b	;	1	1
	mac	x0,x1,a	;	1	1
	macr	-y0,y1,a	;	1	1
	move		b,y:(r1) ;	1	1
	move		a,x:(r1) ;	1	1
			Totals	16	4N+13

C-2.12 Nth Order Power Series (Real)

$$c = \sum_{i=0}^{N-1} [a(i) \times b^i]$$

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r4		b
r1	c	

				Prog wrds	Clock Cycles
	move	#AADDR,r0	;		
	move	#BADDR,r4			
	move	#CADDR,r1			
	move	x:(r0)+,a	;	1	1
	move		y:(r4),x0	1	1
	mpyr	x0,x0,b	x:(r0)+,y0	1	1
	move		b,y1	1	2 i'lock
	do	#N-1,end	;	2	5
	mac	y0,x0,a	x:(r0)+,y0	1	1
	mpyr	x0,y1,b	b,x0	1	1
end					
	macr	y0,x0,a	;	1	1
	move		a,x:(r1)	1	2 i'lock
			Totals	10	2N+11

C-2.13 2nd Order Real Biquad IIR Filter

$$w(n)/2 = x(n)/2 - (a1)/2 \times w(n-1) - (a2)/2 \times w(n-2)$$

$$y(n)/2 = w(n)/2 + (b1)/2 \times w(n-1) + (b2)/2 \times w(n-2)$$

Memory map:

pointer	X mem	Y mem
r0	w(n-2), w(n-1)	
r4		a2/2, a1/2, b2/2, b1/2

					Prog wrds	Clock Cycles
ori	#\$08,mr			;		
move	#AADDR,r0			;		
move	#BADDR,r4			;		
move	#1,m0					
move	#3,m4					
movep	y:input,a			;	1	1
rnd	a	x:(r0)+,x0	y:(r4)+,y0	;	1	1
mac	-y0,x0,a	x:(r0)-,x1	y:(r4)+,y0	;	1	1
mac	-y0,x1,a	x1,x:(r0)+	y:(r4)+,y0	;	1	1
mac	y0,x0,a	a,x:(r0)	y:(r4),y0	;	1	2 i'lock
macr	y0,x1,a			;	1	1
movep	a,y:output			;	1	2 i'lock
				Totals	7	9

C-2.14 N Cascaded Real Biquad IIR Filter

$$w(n)/2 = x(n)/2 - (a1)/2 \times w(n-1) - (a2)/2 \times w(n-2)$$

$$y(n)/2 = w(n)/2 + (b1)/2 \times w(n-1) + (b2)/2 \times w(n-2)$$

Memory map:

pointer	X mem	Y mem
r0	w(n-2)1, w(n-1)1, w(n-2)2, ...	
r4		(a2/2)1, (a1/2)1, (b2/2)1, (b1/2)1, (a2/2)2, ...

				Prog wrds	Clock Cycles
ori	#\$08,mr		;		
move	#AADDR,r0		;		
move	#BADDR,r4		;		
move	\$(2N-1),m0		;		
move	\$(4N-1),m4		;		
move		x:(r0)+,x0	y:(r4)+,y0	;	1
movep	y:input,a			;	1
do	\$(N),end			;	2
mac	-y0,x0,a	x:(r0)-,x1	y:(r4)+,y0	;	1
mac	-y0,x1,a	x1,x:(r0)+	y:(r4)+,y0	;	1
mac	y0,x0,a	a,x:(r0)+	y:(r4)+,y0	;	1
mac	y0,x1,a	x:(r0)+,x0	y:(r4)+,y0	;	1
end					
rnd	a			;	1
movep	a,y:output			;	1
				Totals	10
					5N+10

C-2.15 N Radix-2 FFT Butterflies (DIT, in-place algorithm)

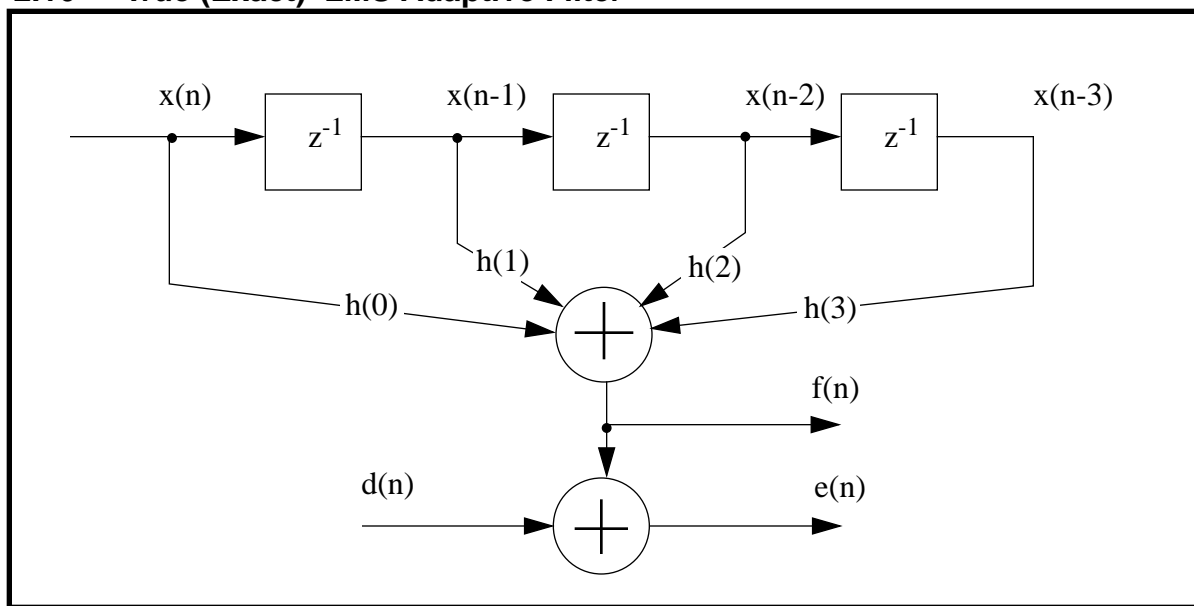
$$\begin{aligned}
 ar' &= ar + cr \times br - ci \times bi & br' &= ar - cr \times br + ci \times bi = 2 \times ar - ar' \\
 ai' &= ai + ci \times br + cr \times bi & bi' &= ai - ci \times br - cr \times bi = 2 \times ai - ai'
 \end{aligned}$$

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r1	br(i)	bi(i)
r6	cr(i)	ci(i)
r4	ar'(i)	ai'(i)
r5	br'(i)	bi'(i)

				Prog wrds	Clock Cycles
move	#AADDR,r0		;		
move	#BADDR,r1		;		
move	#CADDR,r6		;		
move	#ATADDR,r4		;		
move	#BTADDR-1,r5		;		
move		x:(r1),x1	y:(r6),y0	1	1
move		x:(r5),a	y:(r0),b	1	1
do	#N,end		;	2	5
mac	y0,x1,b	x:(r6)+n,x0	y:(r1)+,y1	1	1
macr	x0,y1,b	a,x:(r5)+	y:(r0),a	1	1
subl	b,a		;	1	1
move		x:(r0),b	b,y:(r4)	1	1
mac	x0,x1,b	x:(r0)+,a	a,y:(r5)	1	1
macr	-y0,y1,b	x:(r1),x1	y:(r6),y0	1	1
subl	b,a	b,x:(r4)+	y:(r0),b	1	2 i'lock
end					
move		a,x:(r5)+	;	1	2 i'lock
			Totals	12	8N+9

C-2.16 True (Exact) LMS Adaptive Filter



Notation and symbols:

$x(n)$	-	Input sample at time n .
$d(n)$	-	Desired signal at time n .
$f(n)$	-	FIR filter output at time n .
$H(n)$	-	Filter coefficient vector at time n . $H=\{h_0, h_1, h_2, h_3\}$
$X(n)$	-	Filter state variable vector at time N , $X=\{x(n), x(n-1), x(n-2), x(n-3)\}$.
u	-	Adaptation gain.
NTAPS	-	Number of coefficient taps in the filter. For this example, $ntaps=4$.

System equations:

True LMS Algorithm	Delayed LMS Algorithm
$e(n)=d(n)-H(n)X(n)$	$e(n)=d(n)-H(n)X(n)$
$H(n+1)=H(n)+uX(n)e(n)$	$H(n+1)=H(n)+uX(n-1)e(n-1)$

LMS Algorithm:

True LMS Algorithm	Delayed LMS Algorithm
Get input sample	Get input sample
Save input sample	Save input sample
Do FIR	Do FIR
Get $d(n)$, find $e(n)$	Update coefficients
Update coefficients	Get $d(n)$, find $e(n)$
Output $f(n)$	Output $f(n)$
Shift vector X	Shift vector X

Memory map:

pointer	X mem	Y mem
r0	$x(n), x(n-1), x(n-2), x(n-3)$	
r4, r5		$h(0), h(1), h(2), h(3)$

					Prog wrds	Clock Cycles
	move	#-2,n0		;		
	move	n0,n4				
	move	#NTAPS-1,m0		;		
	move	m0,m4		;		
	move	m0,m5		;		
	move	#AADDR+NTAPS-1,r0		;		
	move	#BADDR,r4		;		
	move	r4,r5		;		
_getsmp						
	movep	y:input,x0		;get input sample	1	1
	clr	a	x0,x:(r0)+	y:(r4)+,y0 ;save	1	1
				;X(n), get h0		
	rep	#NTAPS-1		;do fir	1	5
				;do taps		
	mac	x0,y0,b	x:(r0)+,x0	y:(r4)+,y0 ;	1	1
				;last tap		
	macr	x0,y0,b		;	1	1
;Get d(n), subtract fir output, multiply by "u",						
;put the result in y1.						
;This section is application dependent.						
	move	x:(r0)+,x0	y:(r4)+,a		1	1
	movep	b,y:output		;output fir if desired	1	1
	move		y:(r4)+,b		1	1
	do	#NTAPS/2,cup		;	2	5
	macr	x0,x1,a	x:(r0)+,x0	y:(r4)+,y0 ;	1	1
	macr	x0,x1,b	x:(r0)+,x0	y:(r4)+,y1 ;	1	1
	tfr	y0,a		a,y:(r5)+	1	1
	tfr	y0,b		b,y:(r5)+	1	1
cup						
	move		x:(r0)+n0,x0	y:(r4)+n4,y0 ;	1	1
;continue looping (jmp _getsmp)						
Total					15	3N+16

C-2.17 Delayed LMS Adaptive Filter

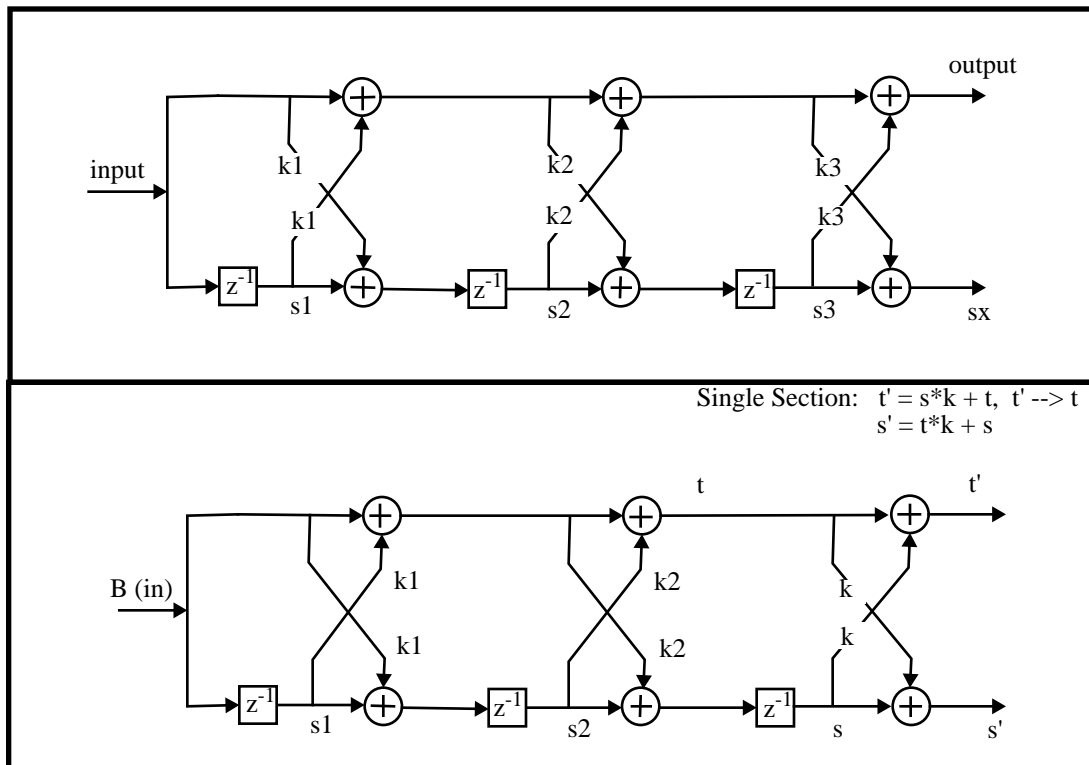
- error signal is in y1
- FIR sum in a = a + h(k)old*x(n-k)
- h(k)new in b = h(k)old + error*x(n-k-1)

Memory map:

pointer	X mem	Y mem
r0	x(n), x(n-1), x(n-2), x(n-3), x(n-4)	
r5, r4		dummy, h(0), h(1), h(2), h(3)

			Prog wrds	Clock Cycles
move	#STATE,r0	;start of X		
move	#2,n0	;used for pointer update		
move	#NTAPS,m0	;number of filter taps		
move	#COEF+1,r4	;start of H		
move	m0,m4	;number of filter taps		
move	#COEF,r5	;start of H-1		
move	m4,m5	;number of filter taps		
movep	y:input,a	;get input sample	1	1
move	a,x:(r0)	;save input sample	1	1
clr	a	x:(r0)+,x0 ;x0<-x(n)	1	1
move		x:(r0)+,x1 y:(r4)+,y0 ;x1<-x(n-1); y0<-h(0)	1	1
do	#TAPS/2,lms	;	2	5
	;a<-h(0)*x(n) b<-h(0) Y<-dummy			
mac	x0,y0,a	y0,b b,y:(r5)+	1	2 i'lock
	;b<-H(0)=h(0)+e*x(n-1), x0<-x(n-2), y0<-h(1)			
macr	x1,y1,b	x:(r0)+,x0 y:(r4)+,y0 ;	1	1
	;a<-a+h(1)*x(n-1); b<-h(1); Y(0)<-H(0)			
mac	x1,y0,a	y0,b b,y:(r5)+ ;	1	2 i'lock
	;b<-H(1)=h(1)+e*x(n-2); x1<-x(n-3); y0<-h(2)			
macr	x0,y1,b	x:(r0)+,x1 y:(r4)+,y0 ;	1	1
lms				
movep	a,y:output		1	1
move	b,y:(r5)+	;Y<-last coef	1	1
move	(r0)-n0	;update pointer	1	1
Totals			13	3N+12

C-2.18 FIR Lattice Filter

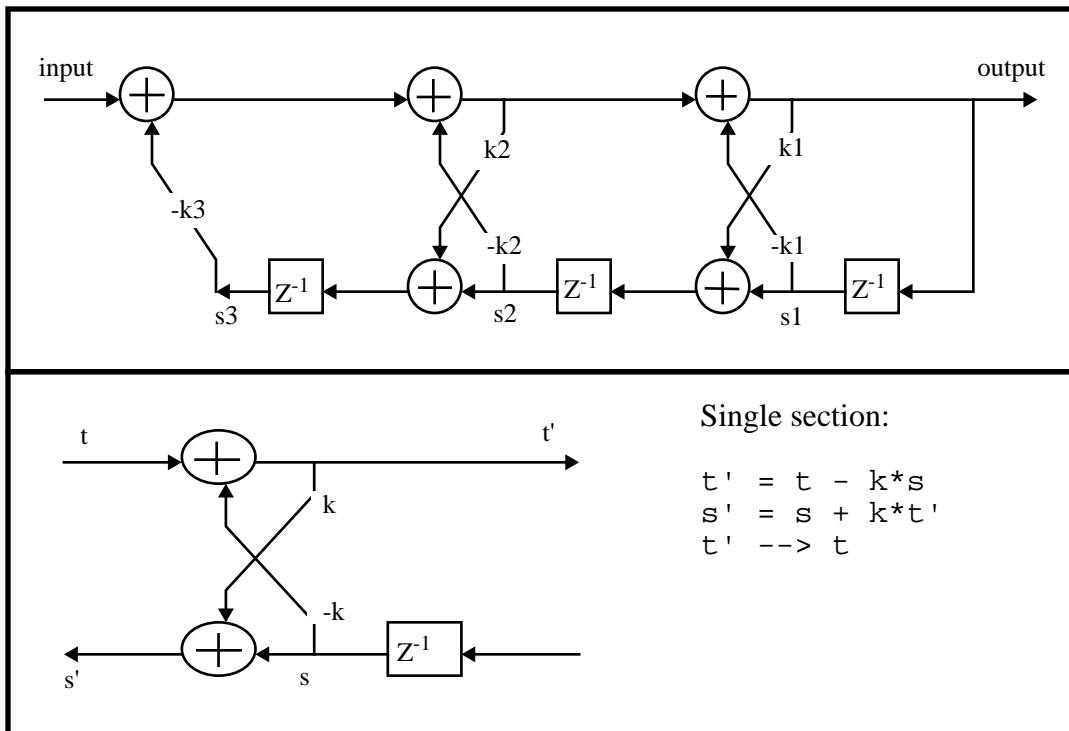


Memory map:

pointer	X mem	Y mem
r0	s1, s2, s3, sx	
r4		k1, k2, k3

				Prog wrds	Clock cycles
	move	#S,r0	;point to s		
	move	#N,m0	;N=number of k coefficients		
	move	#K,r4	;point to k coefficients		
	move	#N-1,m4	;mod for k's		
	movep	y:datin,b	;get input	1	1
	move	b,a	;save first state	1	1
	move	x:(r0),x0	y:(r4)+,y0 ;get s, get k	1	1
	do	#N,_elat	;	2	5
	macr	x0,y0,b	b,y1 ;s*k+t,copy t for mul	1	1
	tfr	x0,a	a,x:(r0)+ ;save s', copy next s	1	1
	macr	y1,y0,a	x:(r0),x0 y:(r4)+,y0 ;t*k+s, get s, get k	1	1
_elat					
	move	a,x:(r0)+	y:(r4)-,y0 ;adj r4,dummy load	1	1
	movep	b,y:datout	;output sample	1	1
			Totals	10	3N+10

C-2.19 All Pole IIR Lattice Filter

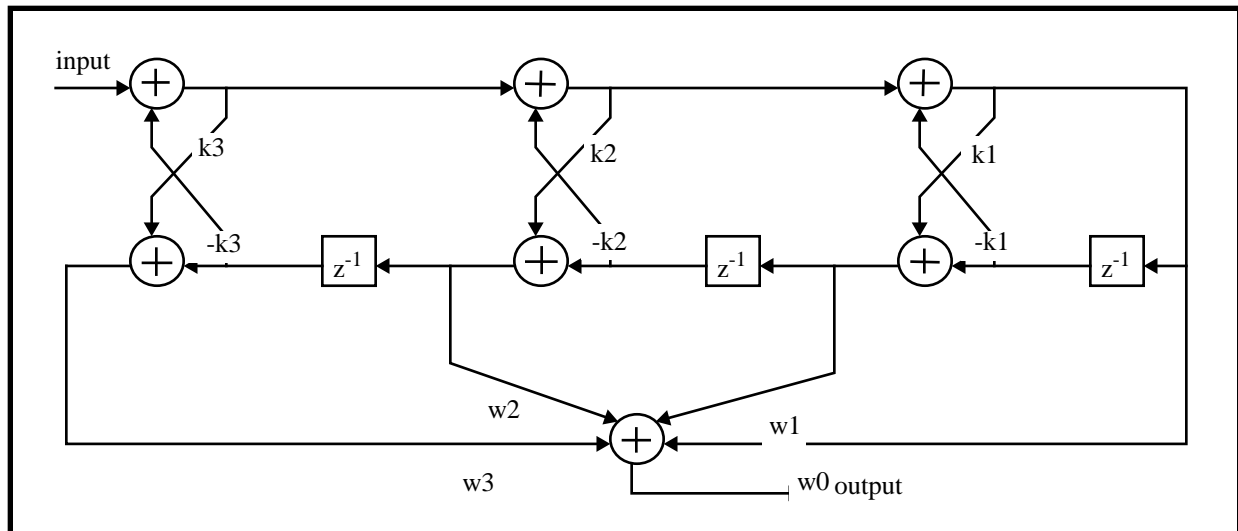


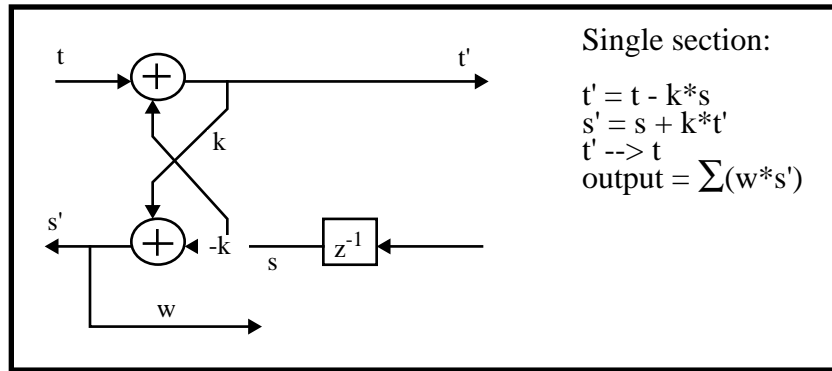
Memory map:

pointer	X mem	Y mem
r0	k3, k2, k1	
r4		s3, s2, s1

					Prog wrds	Clock Cycles
move	#k+N-1,r0			;point to k		
move	#N-1,m0			;number of k's-1		
move	#STATE,r4			;point to filter states		
move	m0,m4			;mod for states		
move	#1,n4			;		
movep	y:datin,a		y:(r4)+,b	;get input	1	1
move		x:(r0)-,x0	y:(r4)+,y0	;get s, get k	1	1
macr	-x0,y0,a	x:(r0)-,x0	y:(r4),y0	;s*k+t	1	1
do	#N-1,_endlat			;do sections	2	5
macr	-x0,y0,a		y:(r4)+,y1	;	1	1
tfr	y1,b	a,x1	b,y:(r4)	;	1	2 i'lock
macr	x1,x0,b	x:(r0)-,x0	y:(r4),y0		1	1
_endlat						
movep	a,y:datout				1	1
move		x:(r0)+,x0	y:(r4)+,r0	;output sample	1	1
move	b,y:(r4)+			;save s'	1	1
;save last s', update r4						
move		a,y:(r4)			1	1
				Totals	12	4N+8

C-2.20 General Lattice Filter



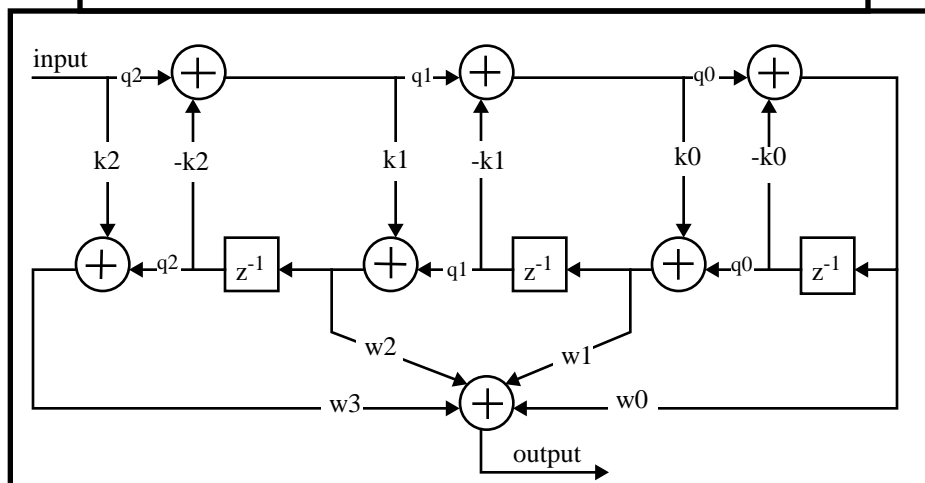
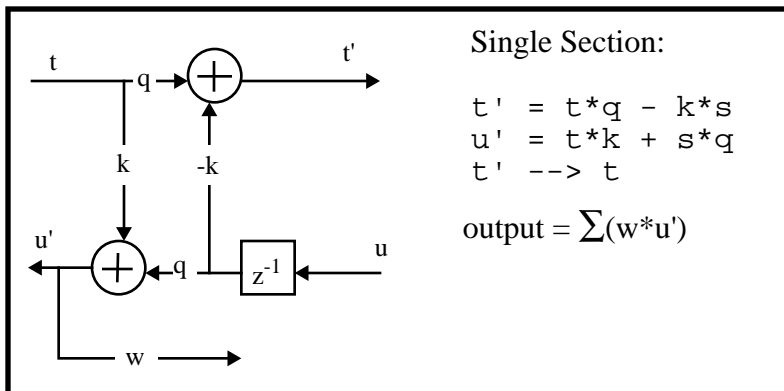


Memory map:

pointer	X mem	Y mem
r0	k3, k2, k1, w3, w2, w1, w0	
r4		s4, s3, s2, s1

				Prog wrds	Clock Cycles
	move	#K,r0	;point to coefficients		
	move	#2*N,m0	;mod 2*(# of k's)+1		
	move	#STATE,r4	;point to filter states		
	move	#-2,n4			
	move	#N,m4	;mod on filter states		
	movep	y:datin,a	;get input	1	1
	move	x:(r0)+,x0	y:(r4)-,y0	1	1
	do	#N,_endlat		2	5
	macr	-x0,y0,a		1	1
	tfr	y0,b	a,x1 b,y:(r4)+n4	1	2 i'lock
	macr	x1,x0,b	x:(r0)+,x0 y:(r4)-,y0	1	1
_endlat					
	move		b,y:(r4)+ ;save s'	1	2 i'lock
	clr	a	a,y:(r4)+ ;save last s', update r4	1	1
	move		y:(r4)+,y0	1	1
	rep	#N		1	5
	mac	x0,y0,a	x:(r0)+,x0 y:(r4)+,y0 ;s*w+out, get s, get w	1	1
	macr	x0,y0,a	;last mac	1	1
	movep	a,y:datout	;output sample	1	2 i'lock
			Totals	14	5N+19

C-2.21 Normalized Lattice Filter



Memory map:

pointer	X mem	Y mem
r0	q2, k2, q1, k1, q0, k0, w3, w2, w1, w0	
r4		sx, s2, s1, s0

				Prog wrds	Clock Cycles
move	#COEF,r0		;point to coefficients		
move	#3*N,m0		;mod on coefficients		
move	#STATE+1,r4		;point to state variables		
move	#N,m4		;mod on filter states		
movep	y:datin,y0		;get input sample	1	1
move		x:(r0)+,x1	;get q in the table	1	1
do	#N,_elat			2	5
mpy	x1,y0,a	x:(r0)+,x0	y:(r4),y1 ;q*t,get k,get s	1	1
macr	-x0,y1,a		b,y:(r4)+ ;q*t-k*s,save new s	1	1

	mpy	x0,y0,b			;k*t	1	1
	macr	x1,y1,b	x:(r0)+,x1	a,y0	;k*t+q*s,get next q,set t'	1	1
_elat							
	move	b,y:(r4)+			;save second last state	1	2 i'lock
	move	a,y:(r4)+			;save last state	1	1
	clr	a		y:(r4)+,y0	;clear a, get first state	1	1
	rep	#N				1	5
	mac	x1,y0,a	x:(r0)+,x1	y:(r4)+,y0	;fir taps	1	1
	macr	x1,y0,a	(r4)+		;round, adj pointer	1	1
	movep	a,y:datout			;output sample	1	2 i'lock
					Total	15	5N+19

C-2.22 [1x3][3x3] Matrix Multiplication

						Prog wrds	Clock Cycles
_init							
	move	#MAT_A,r0			;point to A matrix		
	move	#MAT_B,r4			;point to B matrix		
	move	#MAT_X,r1			;output X matrix		
	move	#2,m0			;mod 3		
	move	#8,m4			;mod 9		
	move	m0,m1			;mod 3		
_start							
	move	x:(r0)+,x0	y:(r4)+,y0			1	1
	mpy	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0		1	1
	mac	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0		1	1
	macr	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0		1	1
	mpy	x0,y0,b	x:(r0)+,x0	y:(r4)+,y0		1	1
	move			a,y:(r1)+		1	1
	mac	x0,y0,b	x:(r0)+,x0	y:(r4)+,y0		1	1
	macr	x0,y0,b	x:(r0)+,x0	y:(r4)+,y0		1	1
	mpy	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0		1	1
	move			b,y:(r1)+		1	1
	mac	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0		1	1
	macr	x0,y0,a				1	1
	move			a,y:(r1)+		1	2 i'lock
_end							
					Totals	13	14

C-2.23 N Point 3x3 2-D FIR Convolution

The two dimensional FIR uses a [3x3] coefficient mask:

c(1,1) c(1,2) c(1,3)

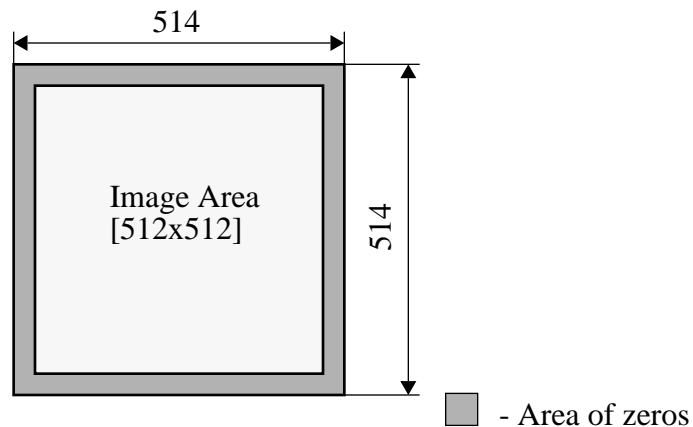
c(2,1) c(2,2) c(2,3)

c(3,1) c(3,2) c(3,3)

stored in Y memory in the order:

c(1,1), c(1,2), c(1,3), c(2,1), c(2,2), c(2,3), c(3,1), c(3,2), c(3,3).

The image is an array of 512x512 pixels. To provide boundary conditions for the FIR filtering, the image is surrounded by a set of zeros such that the image is actually stored as a 514x514 array. i.e.



The image (with boundary) is stored in row major storage. The first element of the array image(.) is image(1,1) followed by image(1,2). The last element of the first row is image(1,514) followed by the beginning of the next column image(2,1). These are stored sequentially in the array "im" in X memory:

Image(1,1) maps to index 0, image(1,514) maps to index 513;

Image(2,1) maps to index 514 (row major storage).

Although many other implementations are possible, this is a realistic type of image environment where the actual size of the image may not be an exact power of 2. Other possibilities include storing a 512x512 image but computing only a 511x511 result, computing a 512x512 result without boundary conditions but throwing away the pixels on the border, etc.

Memory map:

r0 --> image(n,m)
 image(n,m+1)
 image(n,m+2)

r1	-->	image(n+514,m) image(n+514,m+1) image(n+514,m+2)
r2	-->	image(n+2*514,m) image(n+2*514,m+2) image(n+2*514,m+3)
r4	-->	FIR coefficients
r5	-->	output image

				Prog wrds	Clock Cycles
move	#MASK,r4		;point to coeffi- cients		
move	#8,m4		;mod 9		
move	#IMAGE,r0		;top boundary		
move	#IMAGE+514,r1		;left of first pixel		
;left of first pixel 2nd row					
move	#IMAGE+2*514,r2		;		
;adjust. for end of row					
move	#2,n1		;		
move	n1,n2		;		
move	#IMAGEOUT,r5		;output image		
;first element, c(1,1)					
move		x:(r0)+,x0 y:(r4)+,y0	;	1	1
do	#512,row		;	2	5
do	#512,col		;	2	5
mpy	x0,y0,a	x:(r0)+,x0 y:(r4)+,y0	;c(1,2)	1	1
mac	x0,y0,a	x:(r0)-,x0 y:(r4)+,y0	;c(1,3)	1	1
mac	x0,y0,a	x:(r1)+,x0 y:(r4)+,y0	;c(2,1)	1	1
mac	x0,y0,a	x:(r1)+,x0 y:(r4)+,y0	;c(2,2)	1	1
mac	x0,y0,a	x:(r1)-,x0 y:(r4)+,y0	;c(2,3)	1	1
mac	x0,y0,a	x:(r2)+,x0 y:(r4)+,y0	;c(3,1)	1	1
mac	x0,y0,a	x:(r2)+,x0 y:(r4)+,y0	;c(3,2)	1	1
mac	x0,y0,a	x:(r2)-,x0 y:(r4)+,y0	;c(3,3)	1	1
;preload, get c(1,1)					
macr	x0,y0,a	x:(r0)+,x0 y:(r4)+,y0	;	1	1
;output image sample					
move		a,y:(r5)+	;	1	2 i'lock
col					
; adjust pointers for frame boundary					
;adj r0,r5 w/dummy loads					

move	x:(r0)+,x0	y:(r5)+,y1	;	1	1
;adj r1,r5 w/dummy loads					
move	x:(r1)+n1,x	y:(r5)+,y1	;	1	1
0					
;adj r2 (dummy load y1), preload x0 for next pass					
move	x:(r0)+,x0		;	1	1
move		y:(r2)+n2,y1	;	1	1
row					
Total				19 (prog. words)	11N ² +8N+7 (clock cycles)

C-2.24 Parsing data stream

This routine implements parsing of data stream for MPEG audio.

The data stream, composed by concatenated words of variable length, is allocated in consecutive memory words. The words lengths reside in another memory buffer.

The routine extracts words from data stream according to their length.

Two consecutive words are read from the stream buffer and are concatenated in the accumulator. Using bit offset and the specified length, a field of variable length can be extracted. The decision whether to load a new memory word into the accumulator from the stream is determined when bit offset overflow to the LSP of the accumulator.

The following describes the pointers and registers used by the routine:

- r0 - pointer to the buffer in X memory containing the variable length stream.
- r5 - pointer to buffer in Y memory where the length of each field is stored.
- r4 - pointer to a location that stores the "bits offset", number of bits left to be consumed. 48 initially.
- r3 - pointer to a location storing the constant 24.
- r1 - used as temporary storage (no need to initialize).
- y1 - stores the length of the field to be extracted.
- x0 - stores 24.

Memory map:

pointer	X mem	Y mem
r0	stream buffer	
r5		length buffer
r4		"bits offset"
r3	'24'	

init_	;this is the initialization code				
	move	#stream_buffer,r0			
	move	#length_buffer,r5			
	move	#bits_offset,r4			
	move	#boundary,r3			
	move	#>48,b			
	move	#>24,x0			
		x0,x:(r3)	b,y:(r4)		
				Prog wrds	Clock Cycles
Get_bits	;bring length of next field and '24'				
	move	x:(r3),x0	y:(r5)+,y1	1	1
	;bring word for parsing and "bits offset"				
	move	x:(r0)+,a	y:(r4),b	1	1
	;bring next word for parsing, point back to first word				
	move	x:(r0)-,a0		1	1
	;calculate new "bits offset", r1 points to current word				
	sub	y1,b	r0,r1	1	1
	;save "bits offset" in x1				
	move	b,x1		1	2
	;merge width and offset				
	merge	y1,b		1	1
	;extract the field according to b, place it in a				
	extract	b1,a,a		1	1
	;restore "bits offset", r0 points to next word				
	tfr	x1,b	(r0)+	1	1
	;compare "bits offset" to 24, extracted word to a1				
	cmp	x0,b	a0,a	1	1
	;if "bits offset" is less or equal 24 another word is needed - update "bits offset" and point to next word				
	add	x0,b	ifl	1	1
	tgt		r1,r0	1	1
	;save "bits field" in memory				
	move		b1,y:(r4)	1	1
Totals				12	13

C-2.25 Creating data stream

This routine implements creation of data stream for MPEG audio.

Words of variable length are concatenated and stored in consecutive memory words.

The words for generating the stream are allocated in a memory buffer, and are aligned to

the right. The words lengths reside in another memory buffer.

The word and its length are loaded for insertion. A word is read from the stream buffer into the accumulator. Using a bit offset and the specified length, a field of variable length is inserted into the accumulator. The accumulator is stored back containing the new concatenated field. The decision whether to read a new word from the stream is determined when bit offset overflow to the LSP of the accumulator.

The following describes the pointers and registers used by the routine:

- r0 - pointer to a buffer in X memory, containing the variable length codes.
The code is right aligned at each location.
- r2 - pointer to a buffer in X memory containing the stream generated.
- r4 - pointer to a buffer in Y memory where the actual length of each field is stored.
- r3 - pointer to a location that stores the "bits offset", number of bits left to be consumed. 48 initially.
- r5 - pointer to a location storing the constant 24.
- r1 - used as temporary storage (no need to initialize).
- x0 - stores the current word to be inserted
- y1 - stores the length of the code brought in x0.
- y0 - stores 24.

Memory map:

pointer	X mem	Y mem
r0	data buffer	
r2	stream buffer	
r4		length buffer
r3		"bits offset"
r5		24

```
init_           ;this is the initialization code
               move    #data_buffer,r0
```

	move	#stream_buffer,r2			
	move	#length_buffer,r4			
	move	#bits_offset,r3			
	move	#boundary,r5			
	move	#>48,b			
	move	#>24,y0			
	move	b,x:(r3)	y0,y:(r5)		
				Prog wrds	Clock Cycles
Put_bits					
		;bring code and its length			
	move	x:(r0)+,x0	y:(r4)+,y1	1	1
		;bring "bits offset" and '24'			
	move	x:(r3),b	y:(r5),y0	1	1
		;calculate new "bits offset", bring current word from stream buffer			
	sub	y1,b	x:(r2),a	1	1
		;save "bits offset" in x1			
	move	b,x1		1	2
		;merge width and offset			
	merge	y1,b		1	1
		;insert the field according to b, place it in a			
	insert	b1,x0,a		1	1
		;restore "bits offset", r1 points to current word			
	tfr	x1,b	r2,r1	1	1
		;compare "bits offset " to 24, send new word to stream buffer			
	cmp	y0,b	a1,x:(r2)+	1	1
		;send a0 to next location in stream buffer in case of crossing boundary			
	move	a0,x:(r2)		1	2
		;if "bits offset" is less or equal 24 then update "bits offset " and point to the next word in stream buffer			
	add	y0,b	ifle	1	1
	tgt		r1,r2	1	1
		;save "bits offset" in memory			
	move	b1,y:(r4)		1	1
			Totals	12	14

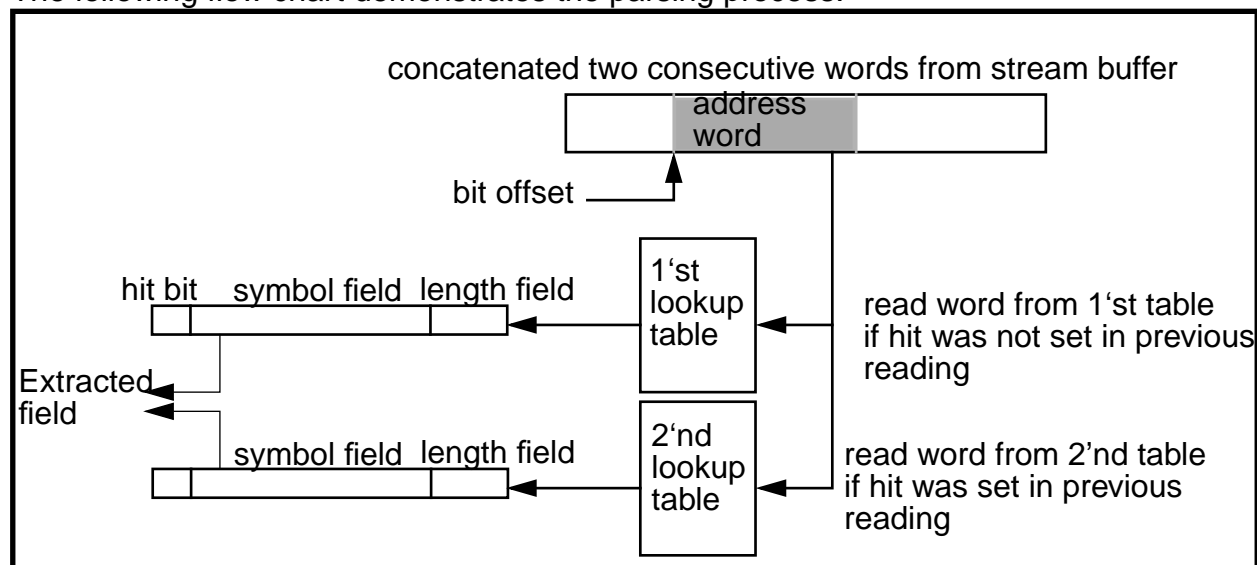
C-2.26 Parsing Hoffman code data stream

This routine implements the parsing of Hoffman code data stream.

The routine extracts a bit field from the stream. Two consecutive words are brought to the accumulator from the stream buffer. An address word is extracted using a bit offset and a field length. The field length is determined by the number of bits needed by the address of the two Hoffman code lookup tables. A word is loaded from the first lookup table. If the hit bit in the word is not set then a field of variable length is extracted. The length of the extracted field is specified in the length field in the word. The bit offset is updated according to the length of the extracted word.

If the hit bit in the word is set then a new address word is read from the stream. A word is brought from the second lookup table. The bit field is extracted according to the same guidelines.

The following flow chart demonstrates the parsing process:



Thek following describes the pointers and registers used by the routine:

- r0 - pointer to the buffer in X memory containing the stream.
- r1 - used as temporary storage (no need to initialize).
- r3 - pointer to buffer in Y memory where the extracted fields are stored.
- r5 - pointer to a location that stores the "bits offset", number of bits left to be consumed. 48 initially.
- r2 - pointer to the right table.
- r6 - pointer to the first lookup table.
- r7 - pointer to the second lookup table.
- r4 - pointer to constants.

Memory map:

pointer	X mem	Y mem
r0	stream buffer	
r3	extracted data buffer	

pointer	X mem	Y mem
r5		"bits offset"
r4		#no.1 address bus length
		#no.2 mask word for length field
		#no.3 merged width and offset
		'24'
r6	first lookup table	
r7	second lookup table	

init_ ;this is the initialization code

```

move #stream_buffer,r0
move #data_buffer,r3
move #bits_offset,r5
move #constants,r4
move #first_table,r2
move #first_table,r6
move #second_table,r7
;move constants to memory
move #>48,b
move b,y:(r5)
move #>3,n4
move #n0_1,y1
move y1,y:(r4)+
move #n0_2,y1
move y1,y:(r4)+
move #n0_3,y1
move y1,y:(r4)+
move #>24,y1
move y1,y:(r4)-n4

```

		Prog wrds	Clock Cycles
Get_bits			
	;bring word from stream, and "bits-offset"		
move	x:(r0)+,a y:(r5)+,b	1	1
	;bring next word from stream, and address length		
move	y:(r4)+,y0	1	1

move	x:(r0)-,a0	1	1
	;calculate new "bits offset", and save old one in x1		
sub	y0,b b,x1	1	1
	;merge width and offset		
merge	y0,b	1	1
	;extract the field according to b, place it in a		
extract	b1,a,a	1	1
	;move address to n2		
move	a0,n2	1	1
	;bring mask for length field in tookup table words		
move	y:(r4)+,y1	1	1
	;bring the merged offset and length for extactionf		
move	y:(r4)+,x0	1	1
	;r1 points to current address for extracted field		
move	r3,r1	1	1
	;bring word from lookup table		
move	x:(r2+n2),a	1	1
	;extract the field according to x0, place it in b		
extract	x0,a,b	1	1
	;test if hit bit is set, r2 points s first lookup table		
tst	a r6,r2	1	1
	; if hit bit is set, r2 points second lookup table, a holds address length		
tmi	y0,a r7,r2	1	1
	;restore "bit offset" , send extracted field to memory		
tfr	x1,b b0,x:(r3)+	1	1
	; if hit bit is set, restore r3		
tmi	r1,r3	1	1
	;mask length field , save pointer to current stream word		
and	y1,a r0,r1	1	1
	;calculate new "bits offset", y1 holds '24'		
sub	a,b y:(r4)-n4,y1	1	1
	;compare "bits offset " to 24, update steam pointer		
cmp	y1,b (r0)+	1	1
	;if "bits offset" is less or equal 24 another word is needed - update "bits offset " and point to next word		
add	y1,b ifle	1	1
tgt	r1,r0	1	1
	;save "bits field" in memory		
move	b1,y:(r5)	1	1
Totals		22	22

C-3 BENCHMARK OVERVIEW

Benchmark	Program Length in Words	Program Length in Clock Cycles	Sample Rate or Execution Time for 50MHz Clock Cycle	Sample Rate or Execution Time for 60MHz Clock Cycle
Real Multiply on page 3	3	4	80 ns	67 ns
N Real Multiplies on page 3	7	2N+8	40N+160ns	33.3N+133 ns
Real Update on page 4	4	5	100 ns	83 ns
N Real Updates on page 4	9	2N+8	40N+160 ns	33.3N+133.6ns
Real Correlation Or Convolution (FIR Filter) on page 5	6	N+14	50/(N+14) MHz	60/(N+14) MHz
Real * Complex Correlation Or Convolution (FIR Filter) on page 7	9	2N+10	25/(N+5) MHz	30/(N+5) MHz
Complex Multiply on page 7	6	7	140 ns	117 ns
N Complex Multiplies on page 8	9	5N+9	80N+180 ns	66.7N+150.3ns
Complex Update on page 9	7	8	160 ns	133 ns
N Complex Updates on page 10	9	4N+9	80N+180 ns	66.7N+150.3ns
Complex Correlation Or Convolution (FIR Filter) on page 11	16	4N+13	25/(2N+5.5) MHz	30/(2N+5.5) MHz
Nth Order Power Series (Real) on page 12	10	2N+11	40N+220 ns	33.3N+183.7ns
2nd Order Real Biquad IIR Filter on page 13	7	9	180 ns	150.3 ns
N Cascaded Real Biquad IIR Filter on page 14	10	5N+10	10/(N+2) MHz	12/(N+2) MHz
N Radix-2 FFT Butterflies (DIT, in-place algorithm) on page 15	12	8N+9	160N+180 ns	133.6N+150.3 ns
True (Exact) LMS Adaptive Filter on page 16	15	3N+16	50/(3N+17) MHz	60/(3N+17) MHz
Delayed LMS Adaptive Filter on page 18	13	3N+12	50/(3N+12) MHz	60/(3N+12) MHz

Benchmark	Program Length in Words	Program Length in Clock Cycles	Sample Rate or Execution Time for 50MHz Clock Cycle	Sample Rate or Execution Time for 60MHz Clock Cycle
FIR Lattice Filter on page 20	10	$3N+10$	$50/(3N+10)$ MHz	$60/(3N+10)$ MHz
All Pole IIR Lattice Filter on page 21	12	$4N+8$	$25/(2N+4)$ MHz	$30/(2N+4)$ MHz
General Lattice Filter on page 22	14	$5N+19$	$50/(5N+19)$ MHz	$60/(5N+19)$ MHz
Normalized Lattice Filter on page 24	15	$5N+19$	$50/(5N+19)$ MHz	$60/(5N+19)$ MHz
[1x3][3x3] Matrix Multiplication on page 25	13	14	280 ns	233.8 ns
N Point 3x3 2-D FIR Convolution on page 25	19	$11N^2+8N+7$	$50/(11N^2+8N+7)$ MHz	$60/(11N^2+8N+7)$ MHz

NOTICE OF CHANGES FROM THE FIRST PRINTING

Pages C-36 and C-37 have been changed in this printing to reflect improved clock rate specifications.

How to reach us:


USA / EUROPE: Motorola Literature Distribution;
P.O. Box 20912; Phoenix, Arizona 85036. 1-800-441-2447

MFAX: RMFAX0@email.sps.mot.com – TOUCHTONE (602) 244-6609
INTERNET: <http://Design-NET.com>

JAPAN: Nippon Motorola Ltd.; Tatsumi-SPD-JLDC, Toshikatsu Otsuki,
6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan.
03-3521-8315

HONG KONG: Motorola Semiconductors H.K. Ltd.;
8B Tai Ping Industrial Park, 51 Ting Kok Road, Tai Po, N.T., Hong Kong.
852-26629298

Order this document by DSP56300FM/AD

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

OnCE™ is a trade mark of Motorola, Inc.

© Motorola Inc., 1995

Appendix C BENCHMARK PROGRAMS

C-1 INTRODUCTION

The following benchmarks illustrate the source code syntax and programming techniques for the DSP56300 Core. The assembly language source is organized into 6 columns as shown below.

Label	Opcode	Operands	X Bus Data	Y Bus Data	Comment
FIR	MAC	X0,Y0,A	X:(R0)+,X0	Y:(R4)+,Y0	;Do each tap

The Label column is used for program entry points and end of loop indication. The Opcode column indicates the Data ALU, Address ALU or Program Controller operation to be performed. The Operands column specifies the operands to be used by the opcode. The X Bus Data specifies an optional data transfer over the X Bus and the addressing mode to be used. The Y Bus Data specifies an optional data transfer over the Y Bus and the addressing mode to be used. The Comment column is used for documentation purposes and does not affect the assembled code. The Opcode column must always be included in the source code.

C-2 SET OF BENCHMARKS

C-2.1 Real Multiply

$$c = a \times b$$

				Prog wrds	Clock Cycles
move		x:(r0),x0	y:(r4),y0	1	1
mpyr	x0,y0,a			1	1
move		a,x:(r1)		1	2 i'lock
Totals				3	4

C-2.2 N Real Multiplies

$$c(i) = a(i) \times b(i) \quad i = 1, 2, \dots, N$$

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r4		b(i)
r1	c(i)	

						Prog wrds	Clock Cycles
end	move	#AADDR,r0					
	move	#BADDR,r4					
	move	#CADDR,r1					
	move		x:(r0)+,x0	y:(r4)+,y0	;	1	1
	mpyr	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	;	1	1
	do	#N-1,end			;	2	5
	mpyr	x0,y0,a	a,x:(r1)+	y:(r4)+,y0	;	1	1
	move		x:(r0)+,x0		;	1	1
					;		
	move		a,x:(r1)+		;	1	1
Totals						7	2N+8

C-2.3 Real Update

$$d = c + a \times b$$

					Prog wrds	Clock Cycles
move	#AADDR,r0					
move	#BADDR,r4					
move	#CADDR,r1					
move	#DADDR,r2					
move		x:(r0),x0	y:(r4),y0	;	1	1
move		x:(r1),a		;	1	1
macr	x0,y0,a			;	1	1
move		a,x:(r2)		;	1	2 i'lock
Totals					4	5

C-2.4 N Real Updates

$$d(i) = c(i) + a(i) \times b(i) \quad i = 1, 2, \dots, N$$

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r4		b(i)
r1	c(i)	
r5		d(i)

			Prog wrds	Clock Cycles
move	#AADDR,r0			
move	#BADDR,r4			
move	#CADDR,r1			
move	#DADDR,r5			
move	x:(r0)+,x0	y:(r4)+,y0 ;	1	1
move	x:(r1)+,a	;	1	1
move	x:(r1)+,b	;	1	1
do	#N/2,end	;	2	5
macr	x0,y0,a	x:(r0)+,x1 y:(r4)+,y1 ;	1	1
macr	x1,y1,b	x:(r0)+,x0 y:(r4)+,y0 ;	1	1
move	x:(r1)+,a	a,y:(r5)+ ;	1	1
move	x:(r1)+,b	b,y:(r5)+ ;	1	1
end				
		Totals	9	2N+8

C-2.5 Real Correlation Or Convolution (FIR Filter)

$$c(n) = \sum_{i=0}^{N-1} [a(i) \times b(n-i)]$$

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r4		b(i)

				Prog wrds	Clock Cycles
move	#AADDR,r0				
move	#BADDR,r4		;		
move	#N-1,m4		;		
move	m4,m0		;		
movep	y:input,y:(r4)		;	1	2
clr	a	x:(r0)+,x0	y:(r4)-,y0	;	1
rep	#N-1			;	5
mac	x0,y0,a	x:(r0)+,x0	y:(r4)-,y0	;	1
macr	x0,y0,a		(r4)+	;	1
movep	a,y:output			;	2 i'lock
			Totals	6	N+14

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r1	b(i)	

				Prog wrds	Clock Cycles
move	#AADDR,r0				
move	#BADDR,r1		;		
move	#N-1,m1		;		
move	m1,m0		;		
movep	y:input,x:(r1)			1	2
clr	a	x:(r0)+,x1		;	1
do	#N-1,end			;	5
move		x:(r1)-,x0		;	1
mac	x0,x1,a	x:(r0)+,x1		;	1
end			;		
move		x:(r1)-,x0		;	1
macr	x0,x1,a	(r1)+		;	1
movep	a,y:output			;	2 i'lock
			Totals	9	2N+10

C-2.6 Real * Complex Correlation Or Convolution (FIR Filter)

$$cr(n) = jci(n) = \sum_{i=0}^{N-1} [(ar(i) + jai(i)) \times b(n-i)]$$

$$cr(n) = \sum_{i=0}^{N-1} ar(i) \times b(n-i) \quad ci(n) = \sum_{i=0}^{N-1} ai(i) \times b(n-i)$$

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r4	b(i)	
r1	cr(n)	ci(n)

				Prog wrds	Clock Cycles
move	#AADDR,r0		;		
move	#BADDR,r4		;		
move	#CADDR,r1		;		
move	#N-1,m4		;		
move	m4,m0		;		
movep	y:input,x:(r4)		;	1	2
clr	a	x:(r0),x0	;	1	1
clr	b	x:(r4)-,x1	y:(r0)+,y0 ;	1	1
do	#N-1,end		;	2	5
mac	x0,x1,a	x:(r0),x0	;	1	1
mac	y0,x1,b	x:(r4)-,x1	y:(r0)+,y0 ;	1	1
end					
macr	x0,x1,a		;	1	1
macr	y0,x1,b	(r4)+	;	1	1
move		a,x:(r1)	;	1	1
move			b,y:(r1) ;	1	1
			Totals	11	2N+11

C-2.7 Complex Multiply

$$cr + jci = (ar + jai) \times (br + jbi)$$

$$cr = ar \times br - ai \times bi \quad ci = ar \times bi + ai \times br$$

Memory map:

pointer	X mem	Y mem
r0	ar	ai
r4	br	bi
r1	cr	ci

					Prog wrds	Clock Cycles
move	#AADDR,r0					
move	#BADDR,r4					
move	#CADDR,r1					
move		x:(r0),x1	y:(r4),y0	;	1	1
mpy	y0,x1,b	x:(r4),x0	y:(r0),y1	;	1	1
macr	x0,y1,b			;	1	1
mpy	x0,x1,a			;	1	1
macr	-y0,y1,a		b,y:(r1)	;	1	1
move		a,x:(r1)		;	1	2 i'lock
Totals					6	7

C-2.8 N Complex Multiplies

$$\begin{aligned}
 cr(i) + jci(i) &= (ar(i) + jai(i)) \times (br(i) + jbi(i)) & i = 1, 2, \dots, N \\
 cr(i) &= ar(i) \times br(i) - ai(i) \times bi(i) \\
 ci(i) &= ar(i) \times bi(i) + ai(i) \times br(i)
 \end{aligned}$$

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r5	cr(i)	ci(i)

				Prog wrds	Clock Cycles
	move	#AADDR,r0	;		
	move	#BADDR,r4	;		
	move	#CADDR-1,r5	;		
	move	x:(r0),x1	y:(r4),y0 ;	1	1
	move	x:(r5),a	;	1	1
	do	#N,end	;	2	5
	mpy	y0,x1,b	x:(r4)+,x0 y:(r0)+,y1 ;	1	1
	macr	x0,y1,b	a,x:(r5)+ ;	1	1
	mpy	-y0,y1,a	y:(r4),y0 ;	1	1
	macr	x0,x1,a	x:(r0),x1 b,y:(r5) ;	1	1
end					
	move	a,x:(r5)	;	1	2 i'lock
		Totals		9	4N+9

C-2.9 Complex Update

$$dr + jdi = (cr + jci) + (ar + jai) \times (br + jbi)$$

$$dr = cr + ar \times br - ai \times bi \quad di = ci + ar \times bi + ai \times br$$

Memory map:

pointer	X mem	Y mem
r0	ar	ai
r4	br	bi
r1	cr	ci
r2	dr	di

				Prog wrds	Clock Cycles
	move	#AADDR,r0			
	move	#BADDR,r4			
	move	#CADDR,r1			
	move	#DADDR,r2			
	move		y:(r1),b ;	1	1
	move	x:(r0),x1	y:(r4),y0 ;	1	1
	mac	y0,x1,b	x:(r4),x0 y:(r0),y1 ;	1	1
	macr	x0,y1,b	x:(r1),a ;	1	1
	mac	x0,x1,a	;	1	1
	macr	-y0,y1,a	b,y:(r2) ;	1	1
	move	a,x:(r2)	;	1	2 i'lock
		Totals		7	8

C-2.10 N Complex Updates

$$dr(i) + jdi(i) = (cr(i) + jci(i)) + (ar(i) + jai(i)) \times (br(i) + jbi(i))$$

$$dr(i) = cr(i) + ar(i) \times br(i) - ai(i) \times bi(i)$$

$$di(i) = ci(i) + ar(i) \times bi(i) + ai(i) \times br(i)$$

$$i = 1, 2, \dots, N$$

Memory map:

pointer	X mem	Y mem
r0	ar(i) ; ai(i)	
r4		br(i) ; bi(i)
r1	cr(i) ; ci(i)	
r5		dr(i) ; di(i)

				Prog wrds	Clock Cycles
	move	#AADDR,r0	;		
	move	#BADDR,r4	;		
	move	#CADDR,r1	;		
	move	#DADDR-1,r5	;		
	move	x:(r0)+,x1	y:(r4)+,y0 ;	1	1
	move	x:(r1)+,b	y:(r5),a ;	1	1
	do	#N,end	;25 ;	2	5
	mac	y0,x1,b	x:(r0)+,x0 y:(r4)+,y1 ;	1	1
	macr	-x0,y1,b	x:(r1)+,a a,y:(r5)+ ;	1	1
	mac	x0,y0,a	x:(r1)+,b b,y:(r5)+ ;	1	2 i'lock
	macr	x1,y1,a	x:(r0)+,x1 y:(r4)+,y0 ;	1	1
end					
	move		a,y:(r5)+ ;	1	2 i'lock
			Totals	9	5N+9

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r1	cr(i)	ci(i)
r5	dr(i)	di(i)

				Prog wrds	Clock Cycles
move	#AADDR,r0		;		
move	#BADDR,r4		;		
move	#CADDR,r1		;		
move	#DADDR-1,r5		;		
move		x:(r5),a	;	1	1
move		x:(r0),x1	y:(r4),y0	1	1
move		x:(r4)+,x0	y:(r1),b	1	1
do	#N,end		;	2	5
mac	y0,x1,b	a,x:(r5)+	y:(r0)+,y1	1	1
macr	x0,y1,b	x:(r1)+,a		1	1
mac	-y0,y1,a	y:(r4),y0		1	1
macr	x0,x1,a	x:(r0),x1	b,y:(r5)	1	1
move		x:(r4)+,x0	y:(r1),b	1	1
end					
move		a,x:(r5)	;	1	1
		Totals		11	5N+9

C-2.11 Complex Correlation Or Convolution (FIR Filter)

$$cr(n) + jci(n) = \sum_{i=0}^{N-1} [(ar(i) + jai(i)) \times (br(n-i) + jbi(n-i))]$$

$$cr(n) = \sum_{i=0}^{N-1} [ar(i) \times br(n-i) - ai(i) \times bi(n-i)]$$

$$ci(n) = \sum_{i=0}^{N-1} [ar(i) \times bi(n-i) + ai(i) \times br(n-i)]$$

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r4	br(i)	bi(i)
r1	cr(i)	ci(i)

				Prog wrds	Clock Cycles
	move	#AADDR,r0	;		
	move	#BADDR,r4	;		
	move	#CADDR,r1			
	move	#N-1,m4			
	move	#m4,m0			
	movep	y:input,x:(r4)		1	2
	movep	y:input,y:(r4)		1	2
	clr	a	;	1	1
	clr	b	x:(r0),x1 y:(r4),y0 ;	1	1
	do	#N-1,end	;	2	5
	mac	y0,x1,b	x:(r4)-,x0 y:(r0)+,y1 ;	1	1
	mac	x0,y1,b	;	1	1
	mac	x0,x1,a	;	1	1
	mac	-y0,y1,a	x:(r0),x1 y:(r4),y0 ;	1	1
end					
	mac	y0,x1,b	x:(r4),x0 y:(r0)+,y1 ;	1	1
	macr	x0,y1,b	;	1	1
	mac	x0,x1,a	;	1	1
	macr	-y0,y1,a	;	1	1
	move		b,y:(r1) ;	1	1
	move		a,x:(r1) ;	1	1
			Totals	16	4N+13

C-2.12 Nth Order Power Series (Real)

$$c = \sum_{i=0}^{N-1} [a(i) \times b^i]$$

Memory map:

pointer	X mem	Y mem
r0	a(i)	
r4		b
r1	c	

				Prog wrds	Clock Cycles
	move	#AADDR,r0	;		
	move	#BADDR,r4			
	move	#CADDR,r1			
	move	x:(r0)+,a	;	1	1
	move		y:(r4),x0	1	1
	mpyr	x0,x0,b	x:(r0)+,y0	1	1
	move		b,y1	1	2 i'lock
	do	#N-1,end	;	2	5
	mac	y0,x0,a	x:(r0)+,y0	1	1
	mpyr	x0,y1,b	b,x0	1	1
end					
	macr	y0,x0,a	;	1	1
	move		a,x:(r1)	1	2 i'lock
			Totals	10	2N+11

C-2.13 2nd Order Real Biquad IIR Filter

$$w(n)/2 = x(n)/2 - (a1)/2 \times w(n-1) - (a2)/2 \times w(n-2)$$

$$y(n)/2 = w(n)/2 + (b1)/2 \times w(n-1) + (b2)/2 \times w(n-2)$$

Memory map:

pointer	X mem	Y mem
r0	w(n-2), w(n-1)	
r4		a2/2, a1/2, b2/2, b1/2

					Prog wrds	Clock Cycles
ori	#\$08,mr			;		
move	#AADDR,r0			;		
move	#BADDR,r4			;		
move	#1,m0					
move	#3,m4					
movep	y:input,a			;	1	1
rnd	a	x:(r0)+,x0	y:(r4)+,y0	;	1	1
mac	-y0,x0,a	x:(r0)-,x1	y:(r4)+,y0	;	1	1
mac	-y0,x1,a	x1,x:(r0)+	y:(r4)+,y0	;	1	1
mac	y0,x0,a	a,x:(r0)	y:(r4),y0	;	1	2 i'lock
macr	y0,x1,a			;	1	1
movep	a,y:output			;	1	2 i'lock
				Totals	7	9

C-2.14 N Cascaded Real Biquad IIR Filter

$$w(n)/2 = x(n)/2 - (a1)/2 \times w(n-1) - (a2)/2 \times w(n-2)$$

$$y(n)/2 = w(n)/2 + (b1)/2 \times w(n-1) + (b2)/2 \times w(n-2)$$

Memory map:

pointer	X mem	Y mem
r0	w(n-2)1, w(n-1)1, w(n-2)2, ...	
r4		(a2/2)1, (a1/2)1, (b2/2)1, (b1/2)1, (a2/2)2, ...

				Prog wrds	Clock Cycles
ori	#\$08,mr		;		
move	#AADDR,r0		;		
move	#BADDR,r4		;		
move	\$(2N-1),m0		;		
move	\$(4N-1),m4		;		
move		x:(r0)+,x0	y:(r4)+,y0	;	1
movep	y:input,a			;	1
do	\$(N),end			;	2
mac	-y0,x0,a	x:(r0)-,x1	y:(r4)+,y0	;	1
mac	-y0,x1,a	x1,x:(r0)+	y:(r4)+,y0	;	1
mac	y0,x0,a	a,x:(r0)+	y:(r4)+,y0	;	1
mac	y0,x1,a	x:(r0)+,x0	y:(r4)+,y0	;	1
end					
rnd	a			;	1
movep	a,y:output			;	1
				Totals	10
					5N+10

C-2.15 N Radix-2 FFT Butterflies (DIT, in-place algorithm)

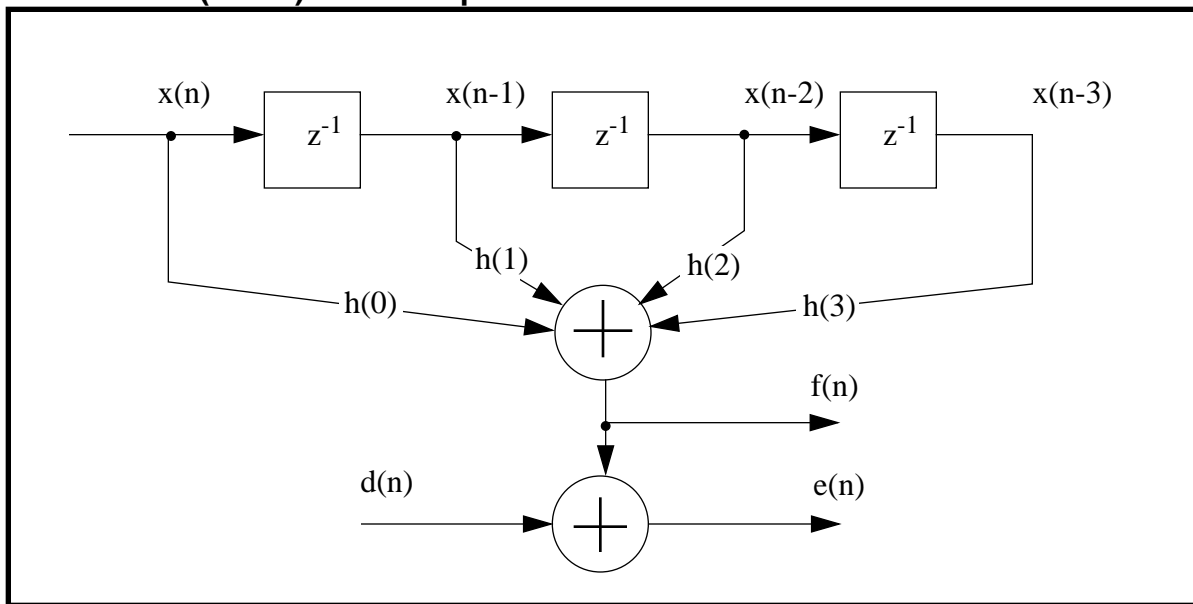
$$\begin{aligned}
 ar' &= ar + cr \times br - ci \times bi & br' &= ar - cr \times br + ci \times bi = 2 \times ar - ar' \\
 ai' &= ai + ci \times br + cr \times bi & bi' &= ai - ci \times br - cr \times bi = 2 \times ai - ai'
 \end{aligned}$$

Memory map:

pointer	X mem	Y mem
r0	ar(i)	ai(i)
r1	br(i)	bi(i)
r6	cr(i)	ci(i)
r4	ar'(i)	ai'(i)
r5	br'(i)	bi'(i)

				Prog wrds	Clock Cycles
move	#AADDR,r0		;		
move	#BADDR,r1		;		
move	#CADDR,r6		;		
move	#ATADDR,r4		;		
move	#BTADDR-1,r5		;		
move		x:(r1),x1	y:(r6),y0	1	1
move		x:(r5),a	y:(r0),b	1	1
do	#N,end		;	2	5
mac	y0,x1,b	x:(r6)+n,x0	y:(r1)+,y1	1	1
macr	x0,y1,b	a,x:(r5)+	y:(r0),a	1	1
subl	b,a		;	1	1
move		x:(r0),b	b,y:(r4)	1	1
mac	x0,x1,b	x:(r0)+,a	a,y:(r5)	1	1
macr	-y0,y1,b	x:(r1),x1	y:(r6),y0	1	1
subl	b,a	b,x:(r4)+	y:(r0),b	1	2 i'lock
end					
move		a,x:(r5)+	;	1	2 i'lock
			Totals	12	8N+9

C-2.16 True (Exact) LMS Adaptive Filter



Notation and symbols:

$x(n)$	-	Input sample at time n .
$d(n)$	-	Desired signal at time n .
$f(n)$	-	FIR filter output at time n .
$H(n)$	-	Filter coefficient vector at time n . $H=\{h_0, h_1, h_2, h_3\}$
$X(n)$	-	Filter state variable vector at time N , $X=\{x(n), x(n-1), x(n-2), x(n-3)\}$.
u	-	Adaptation gain.
NTAPS	-	Number of coefficient taps in the filter. For this example, $ntaps=4$.

System equations:

True LMS Algorithm	Delayed LMS Algorithm
$e(n)=d(n)-H(n)X(n)$	$e(n)=d(n)-H(n)X(n)$
$H(n+1)=H(n)+uX(n)e(n)$	$H(n+1)=H(n)+uX(n-1)e(n-1)$

LMS Algorithm:

True LMS Algorithm	Delayed LMS Algorithm
Get input sample	Get input sample
Save input sample	Save input sample
Do FIR	Do FIR
Get $d(n)$, find $e(n)$	Update coefficients
Update coefficients	Get $d(n)$, find $e(n)$
Output $f(n)$	Output $f(n)$
Shift vector X	Shift vector X

Memory map:

pointer	X mem	Y mem
r0	$x(n), x(n-1), x(n-2), x(n-3)$	
r4, r5		$h(0), h(1), h(2), h(3)$

					Prog wrds	Clock Cycles
	move	#-2,n0		;		
	move	n0,n4				
	move	#NTAPS-1,m0		;		
	move	m0,m4		;		
	move	m0,m5		;		
	move	#AADDR+NTAPS-1,r0		;		
	move	#BADDR,r4		;		
	move	r4,r5		;		
_getsmp						
	movep	y:input,x0		;get input sample	1	1
	clr	a	x0,x:(r0)+	y:(r4)+,y0 ;save	1	1
				;X(n), get h0		
	rep	#NTAPS-1		;do fir	1	5
				;do taps		
	mac	x0,y0,b	x:(r0)+,x0	y:(r4)+,y0 ;	1	1
				;last tap		
	macr	x0,y0,b		;	1	1
;Get d(n), subtract fir output, multiply by "u",						
;put the result in y1.						
;This section is application dependent.						
	move	x:(r0)+,x0	y:(r4)+,a		1	1
	movep	b,y:output		;output fir if desired	1	1
	move		y:(r4)+,b		1	1
	do	#NTAPS/2,cup		;	2	5
	macr	x0,x1,a	x:(r0)+,x0	y:(r4)+,y0 ;	1	1
	macr	x0,x1,b	x:(r0)+,x0	y:(r4)+,y1 ;	1	1
	tfr	y0,a		a,y:(r5)+	1	1
	tfr	y0,b		b,y:(r5)+	1	1
cup						
	move		x:(r0)+n0,x0	y:(r4)+n4,y0 ;	1	1
;continue looping (jmp _getsmp)						
Total					15	3N+16

C-2.17 Delayed LMS Adaptive Filter

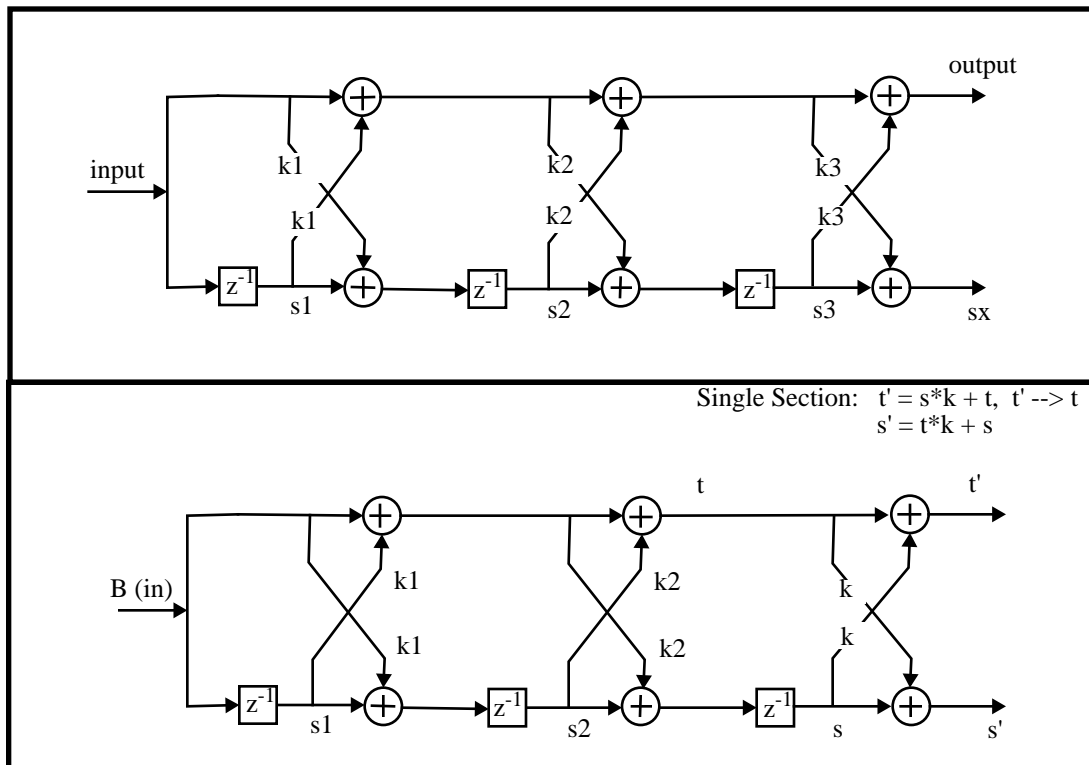
- error signal is in y1
- FIR sum in a = a + h(k)old*x(n-k)
- h(k)new in b = h(k)old + error*x(n-k-1)

Memory map:

pointer	X mem	Y mem
r0	x(n), x(n-1), x(n-2), x(n-3), x(n-4)	
r5, r4		dummy, h(0), h(1), h(2), h(3)

			Prog wrds	Clock Cycles
move	#STATE,r0	;start of X		
move	#2,n0	;used for pointer update		
move	#NTAPS,m0	;number of filter taps		
move	#COEF+1,r4	;start of H		
move	m0,m4	;number of filter taps		
move	#COEF,r5	;start of H-1		
move	m4,m5	;number of filter taps		
movep	y:input,a	;get input sample	1	1
move	a,x:(r0)	;save input sample	1	1
clr	a	x:(r0)+,x0 ;x0<-x(n)	1	1
move		x:(r0)+,x1 y:(r4)+,y0 ;x1<-x(n-1); y0<-h(0)	1	1
do	#TAPS/2,lms	;	2	5
	;a<-h(0)*x(n) b<-h(0) Y<-dummy			
mac	x0,y0,a	y0,b b,y:(r5)+	1	2 i'lock
	;b<-H(0)=h(0)+e*x(n-1), x0<-x(n-2), y0<-h(1)			
macr	x1,y1,b	x:(r0)+,x0 y:(r4)+,y0 ;	1	1
	;a<-a+h(1)*x(n-1); b<-h(1); Y(0)<-H(0)			
mac	x1,y0,a	y0,b b,y:(r5)+ ;	1	2 i'lock
	;b<-H(1)=h(1)+e*x(n-2); x1<-x(n-3); y0<-h(2)			
macr	x0,y1,b	x:(r0)+,x1 y:(r4)+,y0 ;	1	1
lms				
movep	a,y:output		1	1
move	b,y:(r5)+	;Y<-last coef	1	1
move	(r0)-n0	;update pointer	1	1
Totals			13	3N+12

C-2.18 FIR Lattice Filter

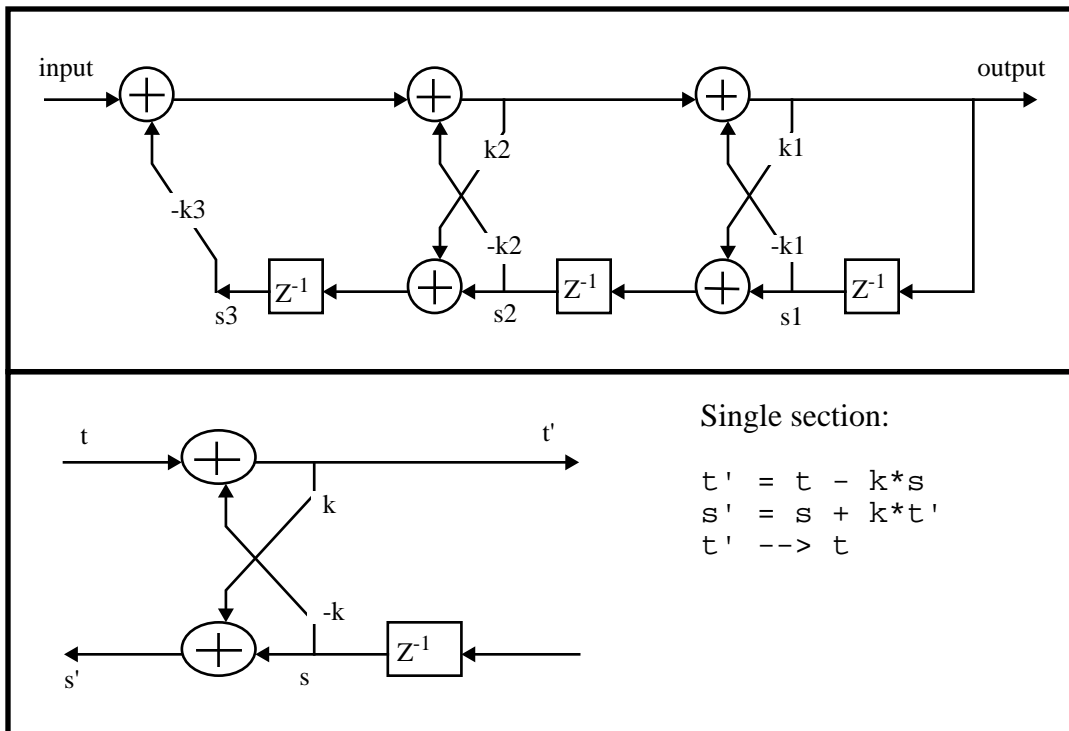


Memory map:

pointer	X mem	Y mem
r0	s1, s2, s3, sx	
r4		k1, k2, k3

				Prog wrds	Clock cycles
	move	#S,r0	;point to s		
	move	#N,m0	;N=number of k coefficients		
	move	#K,r4	;point to k coefficients		
	move	#N-1,m4	;mod for k's		
	movep	y:datin,b	;get input	1	1
	move	b,a	;save first state	1	1
	move	x:(r0),x0	y:(r4)+,y0 ;get s, get k	1	1
	do	#N,_elat	;	2	5
	macr	x0,y0,b	b,y1 ;s*k+t,copy t for mul	1	1
	tfr	x0,a	a,x:(r0)+ ;save s', copy next s	1	1
	macr	y1,y0,a	x:(r0),x0 y:(r4)+,y0 ;t*k+s, get s, get k	1	1
_elat					
	move	a,x:(r0)+	y:(r4)-,y0 ;adj r4,dummy load	1	1
	movep	b,y:datout	;output sample	1	1
			Totals	10	3N+10

C-2.19 All Pole IIR Lattice Filter

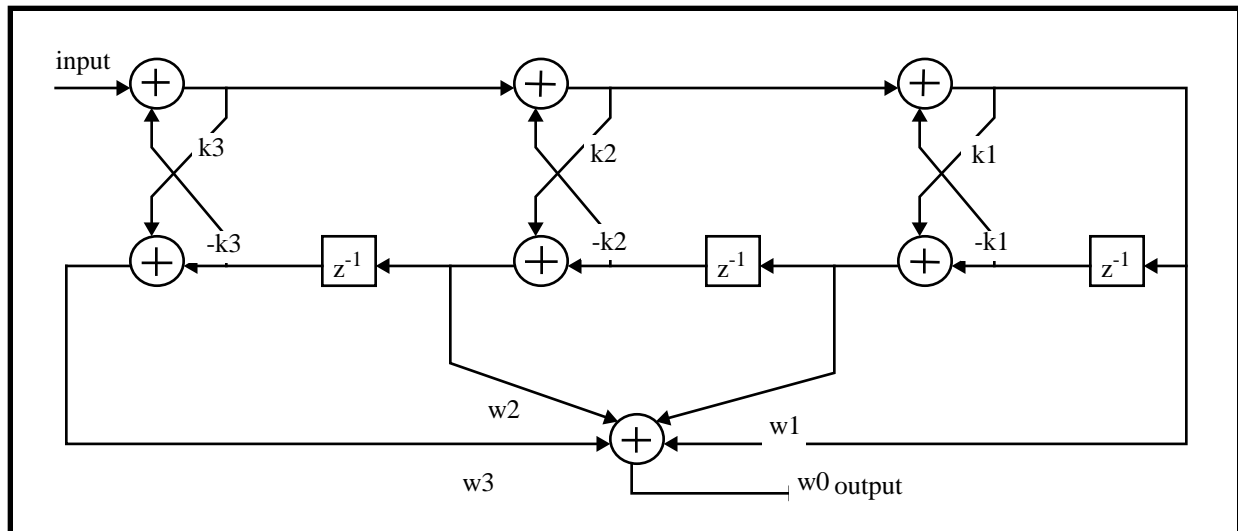


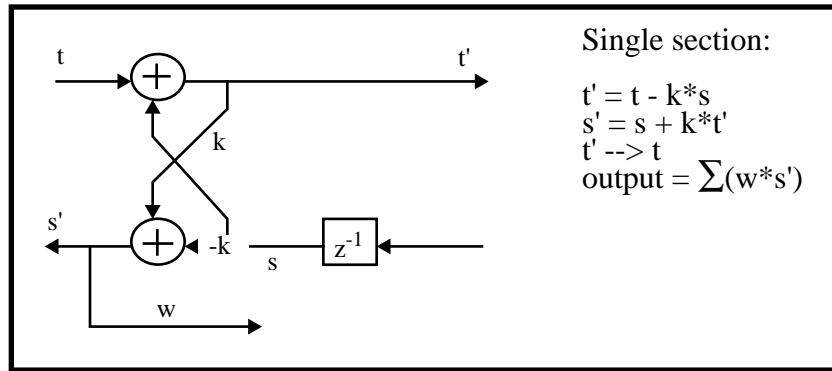
Memory map:

pointer	X mem	Y mem
r0	k3, k2, k1	
r4		s3, s2, s1

					Prog wrds	Clock Cycles
move	#k+N-1,r0			;point to k		
move	#N-1,m0			;number of k's-1		
move	#STATE,r4			;point to filter states		
move	m0,m4			;mod for states		
move	#1,n4			;		
movep	y:datin,a		y:(r4)+,b	;get input	1	1
move		x:(r0)-,x0	y:(r4)+,y0	;get s, get k	1	1
macr	-x0,y0,a	x:(r0)-,x0	y:(r4),y0	;s*k+t	1	1
do	#N-1,_endlat			;do sections	2	5
macr	-x0,y0,a		y:(r4)+,y1	;	1	1
tfr	y1,b	a,x1	b,y:(r4)	;	1	2 i'lock
macr	x1,x0,b	x:(r0)-,x0	y:(r4),y0		1	1
_endlat						
movep	a,y:datout				1	1
move		x:(r0)+,x0	y:(r4)+,r0	;output sample	1	1
move	b,y:(r4)+			;save s'	1	1
;save last s', update r4						
move		a,y:(r4)			1	1
				Totals	12	4N+8

C-2.20 General Lattice Filter



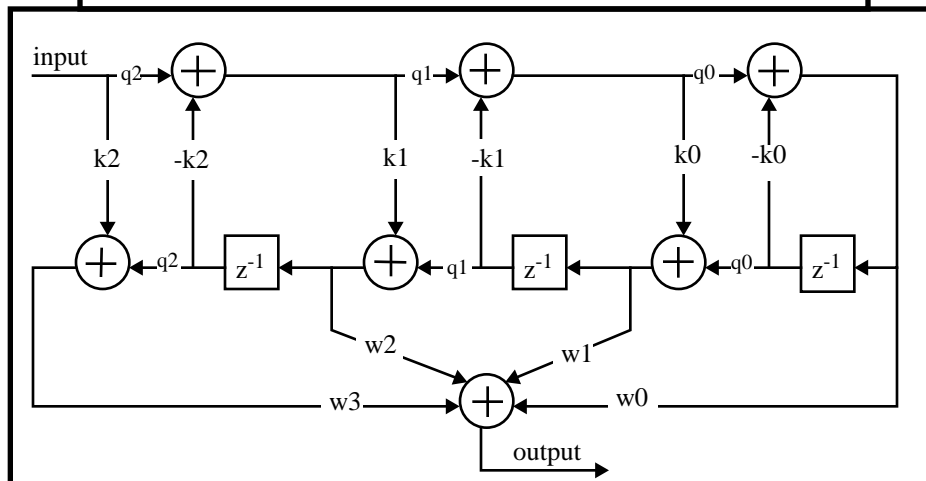
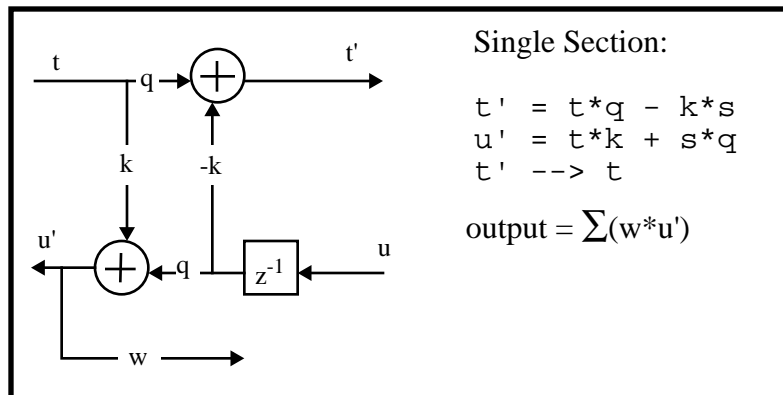


Memory map:

pointer	X mem	Y mem
r0	k3, k2, k1, w3, w2, w1, w0	
r4		s4, s3, s2, s1

				Prog wrds	Clock Cycles
	move	#K,r0	;point to coefficients		
	move	#2*N,m0	;mod 2*(# of k's)+1		
	move	#STATE,r4	;point to filter states		
	move	#-2,n4			
	move	#N,m4	;mod on filter states		
	movep	y:datin,a	;get input	1	1
	move	x:(r0)+,x0	y:(r4)-,y0	1	1
	do	#N,_endlat		2	5
	macr	-x0,y0,a		1	1
	tfr	y0,b	a,x1 b,y:(r4)+n4	1	2 i'lock
	macr	x1,x0,b	x:(r0)+,x0 y:(r4)-,y0	1	1
_endlat					
	move		b,y:(r4)+ ;save s'	1	2 i'lock
	clr	a	a,y:(r4)+ ;save last s', update r4	1	1
	move		y:(r4)+,y0	1	1
	rep	#N		1	5
	mac	x0,y0,a	x:(r0)+,x0 y:(r4)+,y0 ;s*w+out, get s, get w	1	1
	macr	x0,y0,a	;last mac	1	1
	movep	a,y:datout	;output sample	1	2 i'lock
			Totals	14	5N+19

C-2.21 Normalized Lattice Filter



Memory map:

pointer	X mem	Y mem
r0	q2, k2, q1, k1, q0, k0, w3, w2, w1, w0	
r4		sx, s2, s1, s0

					Prog wrds	Clock Cycles
	move	#COEF,r0		;point to coefficients		
	move	#3*N,m0		;mod on coefficients		
	move	#STATE+1,r4		;point to state variables		
	move	#N,m4		;mod on filter states		
	movep	y:datin,y0		;get input sample	1	1
	move		x:(r0)+,x1	;get q in the table	1	1
	do	#N,_elat			2	5
	mpy	x1,y0,a	x:(r0)+,x0	y:(r4),y1 ;q*t,get k,get s	1	1
	macr	-x0,y1,a		b,y:(r4)+ ;q*t-k*s,save new s	1	1
	mpy	x0,y0,b		;k*t	1	1
	macr	x1,y1,b	x:(r0)+,x1	a,y0 ;k*t+q*s,get next q,set t'	1	1
_elat						
	move	b,y:(r4)+		;save second last state	1	2 i'lock
	move	a,y:(r4)+		;save last state	1	1
	clr	a	y:(r4)+,y0	;clear a, get first state	1	1
	rep	#N			1	5
	mac	x1,y0,a	x:(r0)+,x1	y:(r4)+,y0 ;fir taps	1	1
	macr	x1,y0,a	(r4)+	; round, adj pointer	1	1
	movep	a,y:datout		;output sample	1	2 i'lock
				Total	15	5N+19

C-2.22 [1x3][3x3] Matrix Multiplication

					Prog wrds	Clock Cycles
_init						
	move	#MAT_A,r0		;point to A matrix		
	move	#MAT_B,r4		;point to B matrix		
	move	#MAT_X,r1		;output X matrix		
	move	#2,m0		;mod 3		
	move	#8,m4		;mod 9		
	move	m0,m1		;mod 3		
_start						
	move	x:(r0)+,x0	y:(r4)+,y0		1	1
	mpy	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	1	1
	mac	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	1	1
	macr	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	1	1
	mpy	x0,y0,b	x:(r0)+,x0	y:(r4)+,y0	1	1
	move		a,y:(r1)+		1	1

mac	x0,y0,b	x:(r0)+,x0	y:(r4)+,y0	1	1
macr	x0,y0,b	x:(r0)+,x0	y:(r4)+,y0	1	1
mpy	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	1	1
move			b,y:(r1)+	1	1
mac	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	1	1
macr	x0,y0,a			1	1
move			a,y:(r1)+	1	2 i'lock
end					
Totals				13	14

C-2.23 N Point 3x3 2-D FIR Convolution

The two dimensional FIR uses a [3x3] coefficient mask:

c(1,1) c(1,2) c(1,3)

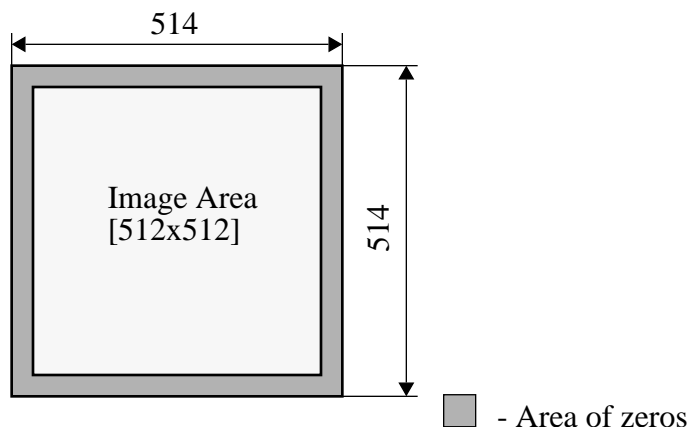
c(2,1) c(2,2) c(2,3)

c(3,1) c(3,2) c(3,3)

stored in Y memory in the order:

c(1,1), c(1,2), c(1,3), c(2,1), c(2,2), c(2,3), c(3,1), c(3,2), c(3,3).

The image is an array of 512x512 pixels. To provide boundary conditions for the FIR filtering, the image is surrounded by a set of zeros such that the image is actually stored as a 514x514 array. i.e.



The image (with boundary) is stored in row major storage. The first element of the array image(.) is image(1,1) followed by image(1,2). The last element of the first row is image(1,514) followed by the beginning of the next column image(2,1). These are stored sequentially in the array "im" in X memory:

Image(1,1) maps to index 0, image(1,514) maps to index 513;

Image(2,1) maps to index 514 (row major storage).

Although many other implementations are possible, this is a realistic type of image environment where the actual size of the image may not be an exact power of 2. Other possibilities include storing a 512x512 image but computing only a 511x511 result, computing a 512x512 result without boundary conditions but throwing away the pixels on the border, etc.

Memory map:

r0	-->	image(n,m) image(n,m+1) image(n,m+2)
r1	-->	image(n+514,m) image(n+514,m+1) image(n+514,m+2)
r2	-->	image(n+2*514,m) image(n+2*514,m+2) image(n+2*514,m+3)
r4	-->	FIR coefficients
r5	-->	output image

				Prog wrds	Clock Cycles
move	#MASK,r4		;point to coeffi- cients		
move	#8,m4		;mod 9		
move	#IMAGE,r0		;top boundary		
move	#IMAGE+514,r1		;left of first pixel		
;left of first pixel 2nd row					
move	#IMAGE+2*514,r2		;		
;adjust. for end of row					
move	#2,n1		;		
move	n1,n2		;		
move	#IMAGEOUT,r5		;output image		
;first element, c(1,1)					
move		x:(r0)+,x0 y:(r4)+,y0	;	1	1
do	#512,row		;	2	5
do	#512,col		;	2	5
mpy	x0,y0,a	x:(r0)+,x0 y:(r4)+,y0	;c(1,2)	1	1

mac	x0,y0,a	x:(r0)-,x0	y:(r4)+,y0	;c(1,3)	1	1
mac	x0,y0,a	x:(r1)+,x0	y:(r4)+,y0	;c(2,1)	1	1
mac	x0,y0,a	x:(r1)+,x0	y:(r4)+,y0	;c(2,2)	1	1
mac	x0,y0,a	x:(r1)-,x0	y:(r4)+,y0	;c(2,3)	1	1
mac	x0,y0,a	x:(r2)+,x0	y:(r4)+,y0	;c(3,1)	1	1
mac	x0,y0,a	x:(r2)+,x0	y:(r4)+,y0	;c(3,2)	1	1
mac	x0,y0,a	x:(r2)-,x0	y:(r4)+,y0	;c(3,3)	1	1
;preload, get c(1,1)						
macr	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	;	1	1
;output image sample						
move			a,y:(r5)+	;	1	2 i'lock
col						
; adjust pointers for frame boundary						
;adj r0,r5 w/dummy loads						
move		x:(r0)+,x0	y:(r5)+,y1	;	1	1
;adj r1,r5 w/dummy loads						
move		x:(r1)+n1,x0	y:(r5)+,y1	;	1	1
;adj r2 (dummy load y1), preload x0 for next pass						
move		x:(r0)+,x0		;	1	1
move			y:(r2)+n2,y1	;	1	1
row						
Total				19 (prog. words)	11N ² +8N+7 (clock cycles)	

C-2.24 Parsing data stream

This routine implements parsing of data stream for MPEG audio.

The data stream, composed by concatenated words of variable length, is allocated in consecutive memory words. The words lengths reside in another memory buffer.

The routine extracts words from data stream according to their length.

Two consecutive words are read from the stream buffer and are concatenated in the accumulator. Using bit offset and the specified length, a field of variable length can be extracted. The decision whether to load a new memory word into the accumulator from the stream is determined when bit offset overflow to the LSP of the accumulator.

The following describes the pointers and registers used by the routine:

- r0 - pointer to the buffer in X memory containing the variable length stream.
- r5 - pointer to buffer in Y memory where the length of each field is stored.
- r4 - pointer to a location that stores the "bits offset", number of bits left to be consumed. 48 initially.
- r3 - pointer to a location storing the constant 24.
- r1 - used as temporary storage (no need to initialize).

- y1 - stores the length of the field to be extracted.
- x0 - stores 24.

Memory map:

pointer	X mem	Y mem
r0	stream buffer	
r5		length buffer
r4		"bits offset"
r3	'24'	

init_ ;this is the initialization code

```

move #stream_buffer,r0
move #length_buffer,r5
move #bits_offset,r4
move #boundary,r3
move #>48,b
move #>24,x0
move x0,x:(r3)    b,y:(r4)

```

Prog
wrds Clock
 Cycles

Get_bits

```

;bring length of next field and '24'
move x:(r3),x0    y:(r5)+,y1    1    1
;bring word for parsing and "bits offset"
move x:(r0)+,a    y:(r4),b      1    1
;bring next word for parsing, point back to first word
move x:(r0)-,a0    1    1
;calculate new "bits offset", r1 points to current word
sub y1,b          r0,r1          1    1
;save "bits offset" in x1
move b,x1          1    2
;merge width and offset
merge y1,b         1    1
;extract the field according to b, place it in a
extract b1,a,a      1    1
;restore "bits offset", r0 points to next word
tfr x1,b           (r0)+         1    1
;compare "bits offset" to 24, extracted word to a1

```

cmp	x0,b	a0,a	1	1
	;if "bits offset" is less or equal 24 another word is needed - update "bits offset" and point to next word			
add	x0,b	ifle	1	1
tgt		r1,r0	1	1
	;save "bits field" in memory			
move		b1,y:(r4)	1	1
Totals			12	13

C-2.25 Creating data stream

This routine implements creation of data stream for MPEG audio.

Words of variable length are concatenated and stored in consecutive memory words.

The words for generating the stream are allocated in a memory buffer, and are aligned to the right. The words lengths reside in another memory buffer.

The word and its length are loaded for insertion. A word is read from the stream buffer into the accumulator. Using a bit offset and the specified length, a field of variable length is inserted into the accumulator. The accumulator is stored back containing the new concatenated field. The decision whether to read a new word from the stream is determined when bit offset overflow to the LSP of the accumulator.

The following describes the pointers and registers used by the routine:

- r0 - pointer to a buffer in X memory, containing the variable length codes.
The code is right aligned at each location.
- r2 - pointer to a buffer in X memory containing the stream generated.
- r4 - pointer to a buffer in Y memory where the actual length of each field is stored.
- r3 - pointer to a location that stores the "bits offset", number of bits left to be consumed. 48 initially.
- r5 - pointer to a location storing the constant 24.
- r1 - used as temporary storage (no need to initialize).
- x0 - stores the current word to be inserted
- y1 - stores the length of the code brought in x0.
- y0 - stores 24.

Memory map:

pointer	X mem	Y mem
r0	data buffer	
r2	stream buffer	
r4		length buffer
r3		"bits offset"
r5		24

init_ ;this is the initialization code

```

move #data_buffer,r0
move #stream_buffer,r2
move #length_buffer,r4
move #bits_offset,r3
move #boundary,r5
move #>48,b
move #>24,y0
move          b,x:(r3)      y0,y:(r5)

```

Prog
wrds Clock
 Cycles

Put_bits

```

;bring code and its length
move          x:(r0)+,x0      y:(r4)+,y1      1      1
;bring "bits offset" and '24'
move          x:(r3),b        y:(r5),y0      1      1
;calculate new "bits offset", bring current word from
stream buffer
sub   y1,b          x:(r2),a      1      1
;save "bits offset" in x1
move   b,x1          1      2
;merge width and offset
merge  y1,b          1      1
;insert the field according to b, place it in a
insert b1,x0,a        1      1
;restore "bits offset", r1 points to current word
tfr    x1,b          r2,r1        1      1
;compare "bits offset " to 24, send new word to
stream buffer

```

cmp	y0,b	a1,x:(r2)+	1	1
	;send a0 to next location in stream buffer in case of crossing boundary			
move		a0,x:(r2)	1	2
	;if "bits offset" is less or equal 24 then update "bits offset " and point to the next word in stream buffer			
add	y0,b	ifle	1	1
tgt		r1,r2	1	1
	;save "bits offset" in memory			
move		b1,y:(r4)	1	1
Totals			12	14

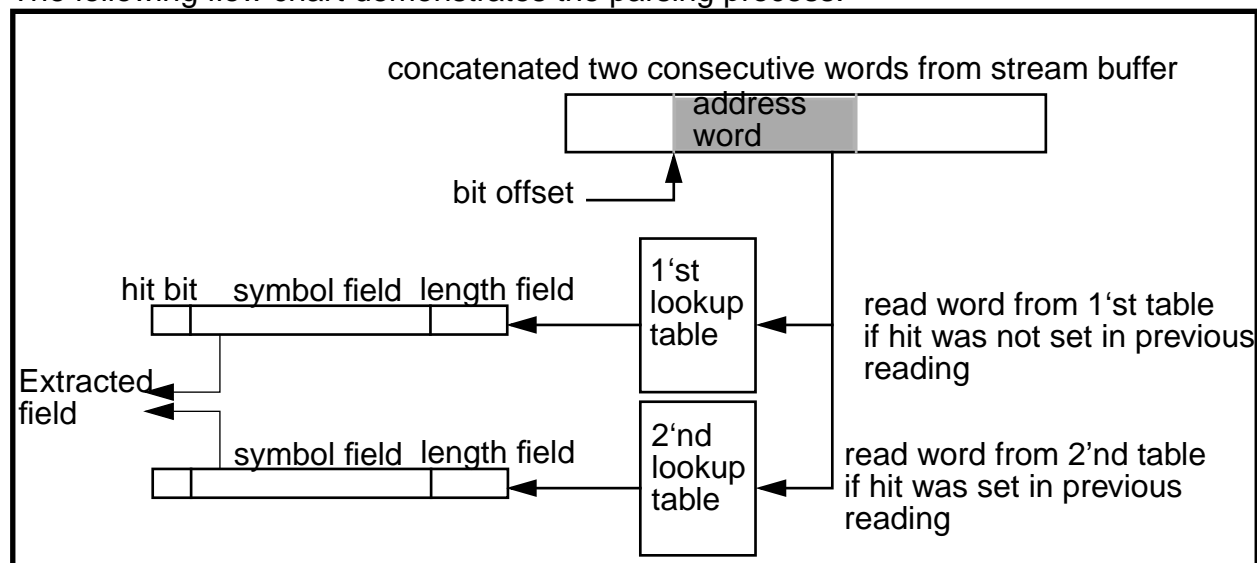
C-2.26 Parsing Hoffman code data stream

This routine implements the parsing of Hoffman code data stream.

The routine extracts a bit field from the stream. Two consecutive words are brought to the accumulator from the stream buffer. An address word is extracted using a bit offset and a field length. The field length is determined by the number of bits needed by the address of the two Hoffman code lookup tables. A word is loaded from the first lookup table. If the hit bit in the word is not set then a field of variable length is extracted. The length of the extracted field is specified in the length field in the word. The bit offset is updated according to the length of the extracted word.

If the hit bit in the word is set then a new address word is read from the stream. A word is brought from the second lookup table. The bit field is extracted according to the same guidelines.

The following flow chart demonstrates the parsing process:



Thek following describes the pointers and registers used by the routine:

- r0 - pointer to the buffer in X memory containing the stream.
- r1 - used as temporary storage (no need to initialize).

- r3 - pointer to buffer in Y memory where the extracted fields are stored.
- r5 - pointer to a location that stores the "bits offset", number of bits left to be consumed. 48 initially.
- r2 - pointer to the right table.
- r6 - pointer to the first lookup table.
- r7 - pointer to the second lookup table.
- r4 - pointer to constants.

Memory map:

pointer	X mem	Y mem
r0	stream buffer	
r3	extracted data buffer	
r5		"bits offset"
r4		#no.1 address bus length
		#no.2 mask word for length field
		#no.3 merged width and offset
		'24'
r6	first lookup table	
r7	second lookup table	

```

init_           ;this is the initialization code
               move    #stream_buffer,r0
               move    #data_buffer,r3
               move    #bits_offset,r5
               move    #constants,r4
               move    #first_table,r2
               move    #first_table,r6
               move    #second_table,r7
               ;move constants to memory
               move    #>48,b
               move    b,y:(r5)
               move    #>3,n4
               move    #n0_1,y1
               move    y1,y:(r4)+
               move    #n0_2,y1
               move    y1,y:(r4)+
               move    #n0_3,y1

```

	move	y1,y:(r4)+		
	move	#>24,y1		
	move	y1,y:(r4)-n4		
			Prog	Clock
			wrds	Cycles
Get_bits				
		;bring word from stream, and "bits-offset"		
	move	x:(r0)+,a y:(r5)+,b	1	1
		;bring next word from stream, and address length		
	move	y:(r4)+,y0	1	1
	move	x:(r0)-,a0	1	1
		;calculate new "bits offset", and save old one in x1		
	sub	y0,b b,x1	1	1
		;merge width and offset		
	merge	y0,b	1	1
		;extract the field according to b, place it in a		
	extract	b1,a,a	1	1
		;move address to n2		
	move	a0,n2	1	1
		;bring mask for length field in lookup table words		
	move	y:(r4)+,y1	1	1
		;bring the merged offset and length for extactionf		
	move	y:(r4)+,x0	1	1
		;r1 points to current address for extracted field		
	move	r3,r1	1	1
		;bring word from lookup table		
	move	x:(r2+n2),a	1	1
		;extract the field according to x0, place it in b		
	extract	x0,a,b	1	1
		;test if hit bit is set, r2 points s first lookup table		
	tst	a r6,r2	1	1
		; if hit bit is set, r2 points second lookup table, a holds address length		
	tmi	y0,a r7,r2	1	1
		;restore "bit offset" , send extracted field to memory		
	tfr	x1,b b0,x:(r3)+	1	1
		; if hit bit is set, restore r3		
	tmi	r1,r3	1	1
		;mask length field , save pointer to current stream word		
	and	y1,a r0,r1	1	1

	;calculate new "bits offset", y1 holds '24'		
sub	a,b	y:(r4)-n4,y1	1 1
	;compare "bits offset " to 24, update steam pointer		
cmp	y1,b	(r0)+	1 1
	;if "bits offset" is less or equal 24 another word is needed - update "bits offset " and point to next word		
add	y1,b	ifle	1 1
tgt		r1,r0	1 1
	;save "bits field" in memory		
move		b1,y:(r5)	1 1
Totals			22 22

C-3 BENCHMARK OVERVIEW

Benchmark	Program Length in Words	Program Length in Clock Cycles	Sample Rate or Execution Time for 66MHz Clock Cycle	Sample Rate or Execution Time for 80MHz Clock Cycle
Real Multiply on page 3	3	4	61 ns	50 ns
N Real Multiplies on page 3	7	2N+8	30N+122 ns	25N+100 ns
Real Update on page 4	4	5	76 ns	62.5 ns
N Real Updates on page 4	9	2N+8	30N+122 ns	25N+100 ns
Real Correlation Or Convolution (FIR Filter) on page 5	6	N+14	66/(N+14) MHz	80/(N+14) MHz
Real * Complex Correlation Or Convolution (FIR Filter) on page 7	9	2N+10	33/(N+5) MHz	40/(N+5) MHz
Complex Multiply on page 7	6	7	106 ns	87.5 ns
N Complex Multiplies on page 8	9	5N+9	76N+137 ns	62.5N+113 ns
Complex Update on page 9	7	8	122 ns	100 ns
N Complex Updates on page 10	9	4N+9	61N+137 ns	50N+113 ns
Complex Correlation Or Convolution (FIR Filter) on page 11	16	4N+13	66/(4N+13) MHz	80/(4N+13) MHz
Nth Order Power Series (Real) on page 12	10	2N+11	30N+167 ns	25N+137ns
2nd Order Real Biquad IIR Filter on page 13	7	9	137 ns	113 ns
N Cascaded Real Biquad IIR Filter on page 14	10	5N+10	66/(5N+10) MHz	16/(N+2) MHz
N Radix-2 FFT Butterflies (DIT, in-place algorithm) on page 15	12	8N+9	122N+137 ns	100N+113 ns

Benchmark	Program Length in Words	Program Length in Clock Cycles	Sample Rate or Execution Time for 66MHz Clock Cycle	Sample Rate or Execution Time for 80MHz Clock Cycle
True (Exact) LMS Adaptive Filter on page 16	15	$3N+16$	$66/(3N+16)$ MHz	$80/(3N+16)$ MHz
Delayed LMS Adaptive Filter on page 19	13	$3N+12$	$66/(3N+12)$ MHz	$80/(3N+12)$ MHz
FIR Lattice Filter on page 20	10	$3N+10$	$66/(3N+10)$ MHz	$80/(3N+10)$ MHz
All Pole IIR Lattice Filter on page 21	12	$4N+8$	$33/(2N+4)$ MHz	$20/(N+2)$ MHz
General Lattice Filter on page 22	14	$5N+19$	$66/(5N+19)$ MHz	$80/(5N+19)$ MHz
Normalized Lattice Filter on page 24	15	$5N+19$	$66/(5N+19)$ MHz	$80/(5N+19)$ MHz
[1x3][3x3] Matrix Multiplication on page 25	13	14	213 ns	175 ns
N Point 3x3 2-D FIR Convolution on page 26	19	$11N^2+8N+7$	$66/(11N^2+8N+7)$ MHz	$80/(11N^2+8N+7)$ MHz

