CWDSP1650 DSP Core

Technical Manual

January 1998

LSI LOGIC®

Order Number C14049

Document DB14-000071-00, First Edition (January 1998)
This document describes revision A of LSI Logic Corporation's CWDSP1650 DSP Core Technical Manual and will remain the official reference source for all revisions/releases of this product until rescinded by an update.

To receive product literature, call us at 1.800.574.4286 (U.S. and Canada); +32.11.300.531 (Europe); 408.433.7700 (outside U.S., Canada, and Europe) and ask for Department JDS; or visit us at http://www.lsilogic.com.

LSI Logic Corporation reserves the right to make changes to any products herein at any time without notice. LSI Logic does not assume any responsibility or liability arising out of the application or use of any product described herein, except as expressly agreed to in writing by LSI Logic; nor does the purchase or use of a product from LSI Logic convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual property rights of LSI Logic or third parties.

Contents

Preface

Chapter 1 **Introduction**

Chapter 2 **Functional Description**

Chapter 3 **Data Formats, Memory and Addressing**

Chapter 4 **Registers**

Chapter 5 **Signals**

Figures

Tables

Preface

This book is the primary reference and technical manual for LSI Logic's CWDSP1650 DSP Core. It contains a complete functional description for the core and includes both physical and electrical specifications.

Audience

This document assumes that the reader has some familiarity with digital signal processors. The people who benefit the most from this book are:

- Engineers and managers who are evaluating the CWDSP1650 for possible use in a design
- Engineers who are designing the CWDSP1650 into a chip

Engineers who are familiar with the CWDSP1640 can refer to the *Comparison of the CWDSP1640 and CWDSP1650 Cores Technical Note* for a full review of any design changes.

Organization

This document has the following chapters and appendixes:

- Chapter 1, **Introduction**, discusses the CWDSP1650 in general, provides the core's key features, and discusses the CoreWare® program.
- Chapter 2, **Functional Description**, describes the purpose of each component block in the CWDSP1650.
- Chapter 3, **Data Formats, Memory and Addressing**, describes the six supported data formats, the program and data memory spaces, and the addressing modes.

- Chapter 4, **Registers**, defines the CWDSP1650 registers and describes the register bit fields.
- Chapter 5, **Signals**, describes the input, output, and bidirectional signals of the CWDSP1650 core.
- Chapter 6, **Operation**, describes the operation of the CWDSP1650 input/output interface.
- Chapter 7, **Instruction Set**, provides a detailed description of the CWDSP1650 instruction set.
- Chapter 8, **On-Chip Emulation Module (OCEM)**, describes the optional module that provides on-chip emulation for a CWDSP1650-based chip.
- Chapter 9, **ScanICE**, describes the CWDSP1650 ScanICE components and describes an example ScanICE system design.
- Chapter 10, **Specifications**, contains the physical characteristics and AC timing for the CWDSP1650 core.
- Appendix A, **CWDSP1650 Register Summary**, lists all of the CWDSP1650 registers and where a detailed description for each can be found.
- **Customer Feedback**, includes a form that you may use to fax us your comments about this document.

Related Publications

CWDSP1640 OakDSPCore® Assembler and Linker User's Guide,
Order No. C14029

CWDSP1640 OakDSPCore® C Cross Compiler User's Guide,
Order No. C14026.A

CWDSP1640 OakDSPCore® Debugger User's Guide,
Order No. C14027.A

CWDSP1650 Reference Device User's Guide, available from LSI Logic.

CWDSP1650 Evaluation Kit User's Guide, Order No. C14046

Comparison of the CWDSP1640 and CWDSP1650 Cores Technical Note, Order No. C15030

Conventions Used in This Manual

The first time a word or phrase is defined in this manual, it is *italicized.*

The word *assert* means to drive a signal true or active. The word *deassert* means to drive a signal false or inactive.

Hexadecimal numbers are indicated by the prefix “0x”, for example, 0x32CF. Binary numbers are indicated by the prefix “0b”, for example, 0b0011 0010 1100 1111.

Chapter 1
Introduction

This chapter overviews LSI Logic's CWDSP1650 DSP core and contains the following sections:

- Section 1.1, "Core Overview"
- Section 1.2, "Features Summary"
- Section 1.3, "CoreWare Program"

1.1 Core Overview

The CWDSP1650 is a 16-bit, fixed-point digital signal processor (DSP) core designed for middle-end to high-end telecommunications and consumer applications. This core provides a low-cost, high-performance solution for applications where low-power and portability are a necessity. This core is a component of the LSI Logic CoreWare Library, which contains cores for control, high-speed communication, and mixed-signal functions to complement quick time-to-market, customizable solutions. The CWDSP1650 is designed by LSI Logic to be fully compatible with the DSP Group OakDSPCore® Instruction Set architecture. The Oak family of cores are modified Harvard architectures, based on DSP Group's PineDSPCore® architecture.

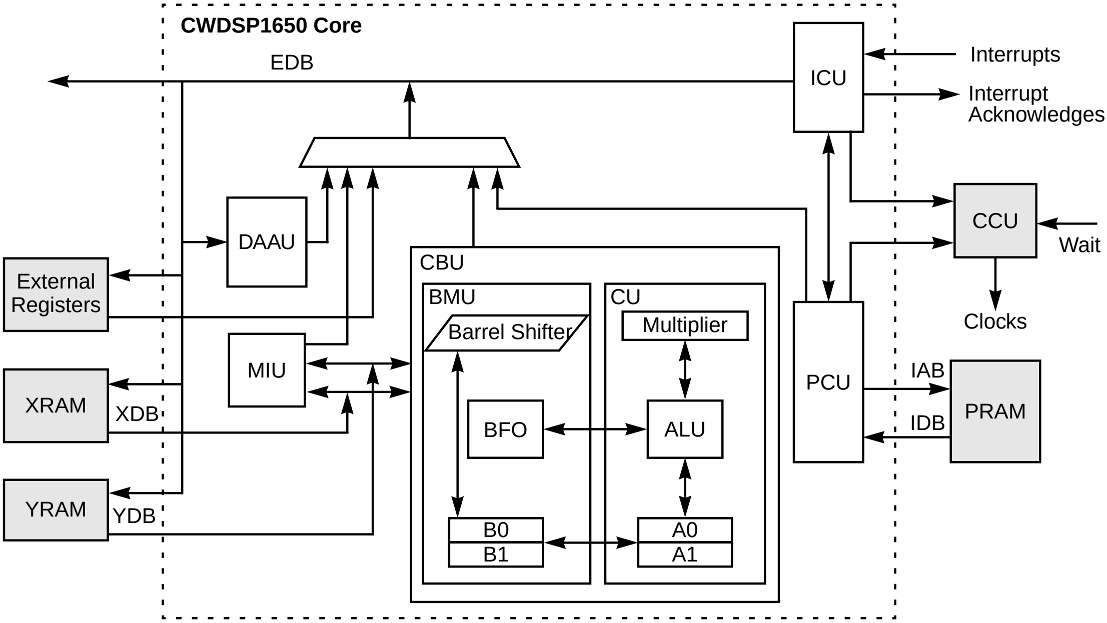
The CWDSP1650 architecture contains dedicated buses, program memories, and data memories. The core is composed of three major components:

- Data Address Arithmetic Unit (DAAU)
- Program Control Unit (PCU)
- Computation and Bit Manipulation Unit (CBU)

The CBU includes a Multiplier, an Arithmetic Logical Unit (ALU), and the Bit Manipulation Unit (BMU). A multiply-accumulate unit performs single-cycle operations.

Figure 1.1 shows a block diagram of the CWDSP1650 core, with modules external to the core shown as shaded regions. Any of the external modules are available as hardmacros from LSI Logic and do not need to be generated by the designer.

Figure 1.1 CWDSP1650 Block Diagram



The CWDSP1650 instruction set allows for straight-forward generation of efficient and compact code for implementation of the DSP functions within a system. Designers may also integrate multiple OakDSPCores onto a single piece of silicon along with other ASIC functions, processors such as the ARM7TDMI core, and application specific logic to build a system-on-a-chip. With many instructions available, the CWDSP1650 can also act as a system controller in its own right.

The CWDSP1650 is supported by a full suite of development tools. Support tools for source code and system-level development include:

- Assembler/Linker
- Optimizing C compiler
- Simulator
- On-Core Emulation Module (OCEM)
- Evaluation board

VHDL or Verilog models implement the top-down design methodology.

1.2 Features Summary

This section summarizes the features of the CWDSP1650.

1.2.1 General Architecture

These features describe general items regarding the hardware architecture of the CWDSP1650:

- Modified Harvard Architecture
- Single-cycle multiply/accumulate instructions
- 36-bit Arithmetic Logical Unit (ALU) and barrel shifter
- Four 36-bit accumulators
- Saturation mode on overflow
- Single-cycle exponent evaluation
- Double-precision multiplication support
- Bit field operations
- Four-level zero-overhead nested looping (interruptible)
- Repeat instruction (interruptible)
- Four user interrupts available (three maskable, one nonmaskable)
- In-circuit debugging support, with on-chip emulator (OCEM)
- Automatic context switching with shadow registers
- Software stack support

- Six 16-bit data pointers for X-data memory and Y-data memory with three additional alternative registers
- Support for direct, indirect, index, and modulo addressing modes
- Off-core X-memory and Y-memory facilitating modular design
- Automatic boot procedure support (self and host booting)

1.2.2 Memory Organization

These features describe the memory support of the CWDSP1650.

- 64 Kword addressable data space
- 64 Kword addressable program memory space
- Off-core data memory: up to 64 Kwords for X-memory, and up to 32 Kwords for Y-memory

1.2.3 Physical Technology

These features summarize the physical attributes and technology of the CWDSP1650 DSP core:

- 0.35-micron G10™ technology
- Fully static design
- 3.3 V ±10% operation
- Core Power - 2.0 mW/MHz
- Core Size - 5.0 mm^2

1.2.4 Instruction Set Summary

Table 1.1 summarizes the instruction set for the core. All instructions are 16 bits long, and most execute in a single cycle. Several instructions including ADDV, SUBV, CMPV, and PUSH can optionally have a long immediate operand (due to the long immediate operand, these instructions are executed in two cycles.)

Table 1.1 CWDSP1650 Instruction Set Summary

Op	Description	Op	Description
Arithmetic and Logical Instructions		**BMU Instructions**	
ADD	Add	SET	Set Bit-field
SUB	Subtract	RST	Reset Bit-field
OR	Logical OR	CHNG	Change Bit-field
AND	Logical AND	TST0	Test Bit-Field for Zeros
XOR	Logical Exclusive OR	TST1	Test Bit-Field for Ones
CMP	Compare	TSTB	Test Specific Bit
ADDL	Add to Low Accumulator	SHFC	Shift Accumulators According to Shift Value Register Conditionally
SUBL	Subtract from Low Accumulator		
ADDH	Add to High Accumulator	SHFI	Shift Accumulators by an Immediate Shift Value
SUBH	Subtract from High Accumulator	EXP	Evaluate the Exponent Value
CMPU	Compare Unsigned	MODB	Modify Bx-Accumulator Conditionally
ADDV	Add Long Immediate Value[1]	MODB Modifications:	
SUBV	Subtract Long Immediate Value[1]	SHR	Shift Right
CMPV	Compare Long Immediate Value[1]	SHR4	Shift Right Four
NORM	Normalize	SHL	Shift Left
DIVS	Division Step	SHL4	Shift Left Four
MAX	Maximum between Two Ax-Accumulators	ROR	Rotate Right through Carry
MAXD	Maximum between Data Memory Location and Ax-Accumulator	ROL	Rotate Left through Carry
		CLR	Clear
MIN	Minimum between Two Ax-Accumulators	**Move Instructions**	
LIM	Limit Ax-Accumulator	MOV	Move Data
MODA	Modify Ax-Accumulator Conditionally	MOVP	Move from Program Memory into Data Memory
MODA Modifications:		MOVD	Move from Data Memory into Program Memory
SHR	Shift Right	MOVS	Move and Shift according to Shift Value Register
SHR4	Shift Right Four	MOVSI	Move and Shift according to an Immediate Shift Value
SHL	Shift Left	MOVR	Move and Round
SHL4	Shift Left Four	PUSH	Push Register or Long Immediate Value onto Stack
ROR	Rotate Right through Carry	POP	Pop from Software Stack into Register
ROL	Rotate Left through Carry	SWAP	Swap Ax and Bx Accumulator
NOT	Logical Not	BANKE	Bank Exchange
NEG	Two's Complement	**Control and Miscellaneous Instructions**	
CLR	Clear	NOP	No Operation
COPY	Copy other Accumulator	MODR	Modify Register N
RND	Round upper 20 bits	EINT	Enable Interrupt
PACR	Product Move and Round	DINT	Disable Interrupt
CLRR	Clear and Round	TRAP	Software Interrupt
INC	Increment by One	LOAD	Load Specific Field into Registers - page, modX, stepX, ps
DEC	Decrement by One	CNTX	Context Switching Store or Restore

Table 1.1 CWDSP1650 Instruction Set Summary (Cont.)

Op	Description	Op	Description
Multiply Instructions		**Branch/Call Instructions**	
MPY	Multiply	BR	Conditional Branch
MPYSU	Multiply Signed by Unsigned	BRR	Relative Conditional Branch
MAC	Multiply and Accumulate Previous Product	CALL	Conditional Call Subroutine
MACSU	Multiply Signed by Unsigned and Accumulate[2]	CALLR	Relative Conditional Call Subroutine
MACUS	Multiply Unsigned by Signed and Accumulate[2]	CALLA	Call Subroutine at Location Specified by Ax-Accumulator
MACUU	Multiply Unsigned by Unsigned and Accumulate[2]	RET	Return Conditionally
MAA	Multiply and Accumulate Aligned Previous Product	RETD	Delayed Return
		RETI	Return from Interrupt Conditionally
MAASU	Multiply Signed by Unsigned and Accumulate Aligned[2]	RETID	Delayed Return from Interrupt
		RETS	Return with Short Immediate Parameter
MSU	Multiply and Subtract Previous Product	**Loop Instructions**	
MPYI	Multiply Signed Short Immediate	REP	Repeat Next Instruction
SQR	Square	BKREP	Block Repeat
SQRA	Square and Accumulate Previous Product	BREAK	Break from a Block-Repeat

[1] To or from a Register or a Data Memory Location.
[2] Previous Product.

The CWDSP1650 supports DSP-specific instructions such as multiply and accumulate, multiply and subtract, nested block repeat (looping), modulo and index operations on data addresses, bit manipulation operations, and control specific operations.

See Chapter 7, "Instruction Set," for a more detailed description of the core instruction set and an alphabetical listing of all CWDSP1650 instructions.

1.3 CoreWare Program

An LSI Logic core is a fully defined, optimized, and reusable block of logic. It supports industry-standard functions and has predefined timing and layout. The core is also an encrypted RTL simulation model for a wide range of VHDL and Verilog simulators.

The CoreWare library contains an extensive set of complex cores for the communications, consumer, and computer markets. The library consists of high-speed interconnect functions such as the GigaBlaze™ G10 core, MIPS embedded microprocessors, MPEG-2 decoders, a PCI core, and many more.

The library also includes megafunctions or building blocks, which provide useful functions for developing a system on a chip. Through the

CoreWare program, you can create a system on a chip uniquely suited to your applications.

Each core has an associated set of deliverables, including:

- RTL simulation models for both Verilog and VHDL environments
- A System Verification Environment (SVE) for RTL-based simulation
- Netlists for full timing simulation
- Complete documentation
- LSI Logic ToolKit support

LSI Logic's ToolKit provides seamless connectivity between products from leading electronic design automation (EDA) vendors and LSI Logic's manufacturing environment. Standard interfaces for formats and languages such as VHDL, Verilog, Waveform Generation Language (WGL), Physical Design Exchange Format (PDEF), and Standard Delay Format (SDF) allow a wide range of tools to interoperate within the LSI Logic ToolKit environment. In addition to design capabilities, full scan Automatic Test Pattern Generation (ATPG) tools and LSI Logic's specialized test solutions can be combined to provide high-fault coverage test programs that assure a fully functional design.

Because your design requirements are unique, LSI Logic is flexible in working with you to develop your system-on-a-chip CoreWare design. Three different work relationships are available:

- You provide LSI Logic with a detailed specification and LSI Logic performs all design work.
- You design some functions while LSI Logic provides you with the cores and megafunctions, and LSI Logic completes the integration.
- You perform the entire design and integration, and LSI Logic provides the core and associated deliverables.

Whatever the work relationship, LSI Logic's advanced CoreWare methodology and ASIC process technologies consistently produce Right-First-Time™ silicon.

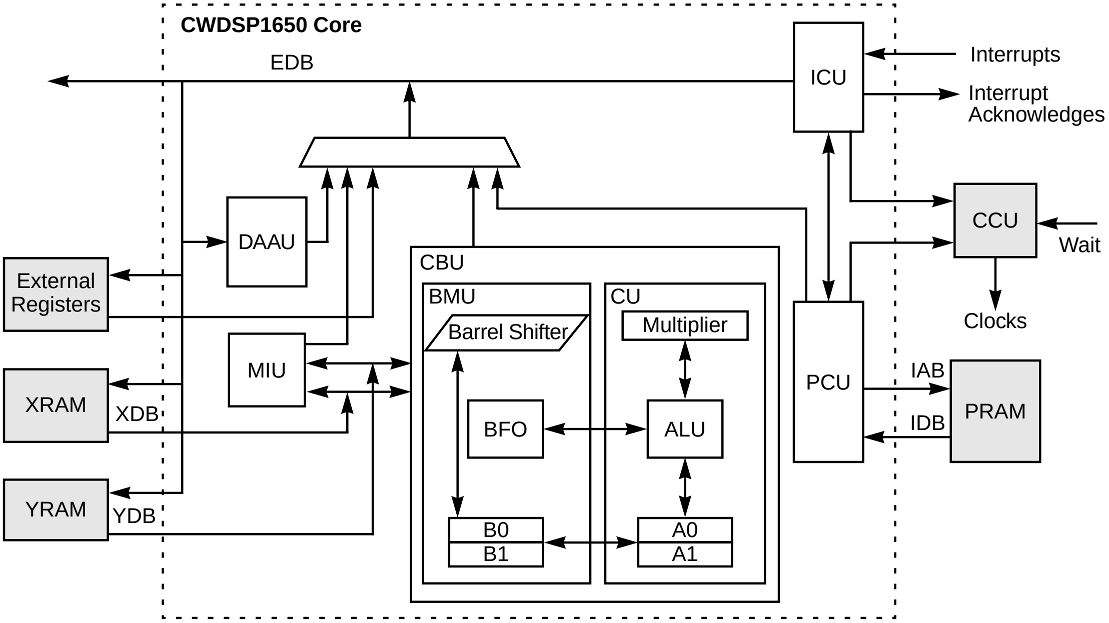
Chapter 2
Functional Description

This chapter describes the components and functional blocks of the CWDSP1650 DSP core. This chapter is divided into the following sections:

2.1 Overview

This section overviews all of the major CWDSP1650 components and the processor pipeline architecture. Figure 2.1 contains a block diagram of the CWDSP1650 architecture. (Please note that the shaded areas are components external to the core.) This section is further divided into the following subsections:

Figure 2.1 CWDSP1650 Block Diagram



2.1.1 CWDSP1650 Core Components

The CWDSP1650 core is composed of the following major elements:

- CWDSP1650 Busses
- Program Control Unit (PCU)
- Computational and Bit Manipulation Unit (CBU)
- Data Addressing Arithmetic Unit (DAAU)
- Memory Interface Unit (MIU)
- Interrupt Control Unit (ICU)

All **CWDSP1650 busses** (data and address) are unidirectional and carry data to and from the core. See Section 2.2, “Buses,” for more detailed information.

The **Program Control Unit (PCU)** controls the sequencing of the core program: it fetches instructions, generates program memory addresses, handles interrupts with the ICU, and sequences branches, calls, and instruction repeats. The PCU generates the control for the rest of the

CWDSP1650 core. See Section 2.3, “Program Control Unit (PCU),” for more detailed information.

The **Computational and Bit Manipulation Unit** (**CBU)** is composed of two units: the Bit Manipulation Unit (BMU) and the Computation Unit (CU). The CU contains the multiplier, Arithmetic Logic Unit (ALU), and Ax Accumulators. The BMU contains a Barrel Shifter, the Bit-Field Operations Unit (BFO), and the Bx Accumulators. The CBU also includes a Saturation Unit and bus alignment/sign-extension logic available for both the BMU and CU. See Section 2.4, “Computation and Bit-Manipulation Unit (CBU),” for more detailed information.

The **Data Addressing Arithmetic Unit (DAAU)** contains the general purpose registers, stack pointer, and index register. It also contains two identical arithmetic units to generate sequences of addresses using the DAAU registers. Two addresses may be generated on each cycle for simultaneous access of both X-memory and Y-memory spaces. See Section 2.5, “Data Address Arithmetic Unit (DAAU),” for more detailed information.

The **Memory Interface Unit (MIU)** routes data memory addresses to the X- and Y-address buses and X-data and Y-data from the data memory. The CWDSP1650 user cannot control the MIU operations, therefore, the MIU is not fully described in this chapter.

The **Interrupt Control Unit (ICU)** handles the interrupt protocols for each of the five interrupts and generates a separate acknowledge signal for each one. The ICU generates the interrupt vector and status signal for the PCU and also prioritizes incoming interrupts. See Section 2.6, “Interrupt Control Unit (ICU),” for more detailed information.

2.1.2 CWDSP1650 External Modules

Besides these internal core components, a CWDSP1650 design may require the following logic blocks external to the core (shaded units in Figure 2.1):

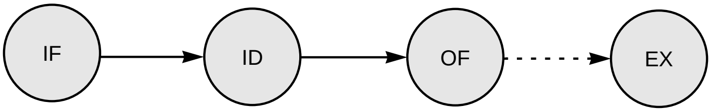
- Data memory (XRAM and YRAM)
- Program memory (PRAM)
- External (User Defined) registers
- Clock Control Unit (CCU)

- On-Chip Emulation Module (OCEM)
- Bus Interface Unit (BIU)

2.1.3 Pipeline Architecture

The CWDSP1650 implements a four-stage pipeline architecture. The Pipeline architecture is designed with a central microcode control that allows easier and better control of the core. Figure 2.2 shows the CWDSP1650 pipeline operation.

Figure 2.2 CWDSP1650 Instruction Pipeline



The execution of a single CWDSP1650 instruction consists of the following stages:

1. IF (Instruction Fetch) - The PCU generates a new program address, fetches the instruction, and stores it in the instruction register.
2. ID (Instruction Decode) - The core decodes the instruction into a wide microinstruction that contains almost all the control signals for the CWDSP1650 processor. During the ID stage, the core decides the program flow and, at the end of the cycle, registers the microinstruction.
3. OF (Operand Fetch and Execute) - During the OF cycle, the core generates the operand addresses and fetches the source operands from either an internal source or from memory. The core also sets up the ALU and other execution units and stores the result in the proper destination. Finally, the condition flags are updated at the end of the cycle.
4. EX (Execute) - This cycle is required only for multiplies and a few other instructions. The multiplier operates on the X and Y registers, storing the result in the Product register. This result is available during the next OF stage.

2.2 Buses

This section covers the different address and data buses that comprise the CWDSP1650. For this section, the CWDSP1650 buses have been divided into two types: data buses and address buses.

2.2.1 Data Buses

The core transfers data on the following unidirectional 16-bit buses:

- External Main Data Bus Out (EDB)
- Instruction Data Bus In (IDB)
- X-Memory Data Bus In (XDB)
- Y-Memory Data Bus In (YDB)
- External User Register Input Bus In (EXT_IN)

The External Data Bus delivers all of the data that passes from the core to the external components (i.e. the XRAM, YRAM, PRAM, and External Registers.) The unidirectional YDB bus carries all data transfers from the Y-Memory to the core. Instruction word fetches take place in parallel over the IDB. The unidirectional XDB bus carries all data transfers from the X-Memory to the core.

Two 16-bit data words and one instruction word can be moved within one instruction cycle.

2.2.2 Address Buses

The core drives addresses on the following unidirectional 16-bit buses:

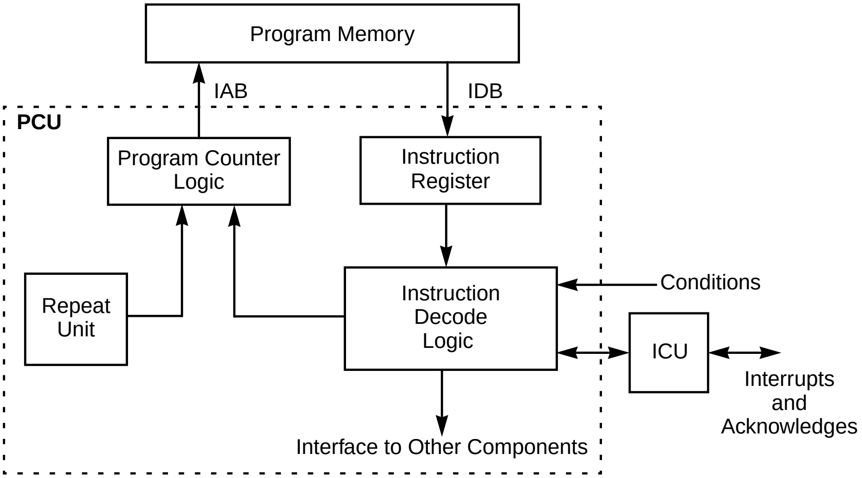
- Instruction Address Bus (IAB)
- X-Memory Address Bus (XAB)
- Y-Memory Address Bus (YAB)

The unidirectional 16-bit X-Memory Address Bus (XAB) provides addresses for the X-Memory. Similarly, the unidirectional 16-bit Y-Memory Address Bus (YAB) provides the addresses for the Y-Memory. The unidirectional 16-bit Instruction Address Bus (IAB) transfers the program memory addresses to the program memory.

2.3 Program Control Unit (PCU)

The Program Control Unit (PCU) performs instruction fetch, instruction decoding, interrupt handling, and hardware loop control. It provides control signals for the rest of the CWDSP1650 core and the on-chip emulation unit (OCEM). The PCU also provides signals to enable program memory protection in the bus interface unit. Figure 2.3 is a block diagram of the PCU and associated logic blocks.

Figure 2.3 Program Control Unit Diagram



The PCU contains:

- Instruction Register
- Instruction Decode Logic
- Program Counter Logic
- Repeat Unit

At the end of the IF cycle, the PCU loads the Instruction register with the instruction to be decoded.

During the ID cycle, instruction decode logic converts the instruction into a number of signals that control the rest of the CWDSP1650 processor. Also during the ID cycle, the PCU initiates branch or vector operations, depending on the conditions and interrupt states during the cycle.

The Program Counter Logic generates the addresses for the program memory and stores the values in the program counter. There are many sources for the program address:

- Current program counter, incremented
- Block repeat start address
- Immediate value (from Instruction Register)
- Current program counter plus short immediate value
- Interrupt vector
- Special Holding Register for data access to program memory

The Repeat Unit controls any zero overhead looping operations, initiated by the REP and BKREP instructions. The number of repetitions for a repeat (REP) or block repeat (BKREP) can be defined as either an 8-bit fixed value embedded in the instruction code, or as a 16-bit value transferred from one of the processor registers. A REP instruction repeats the following instruction and a BKREP repeats a block of program code of at least two instructions in length. Block repeat loops may be nested up to four levels and a block repeat may contain further REP instructions. Both repeat and block repeat loops are interruptible.

During the operation of a repeat loop, the repeat unit register (REPC) stores the current count of repetitions remaining. The REPC value can be read using the instruction `mov repc, ab`. During the operation of a block repeat loop, the last and first addresses of the loop are stored in dedicated registers in the repeat unit.

The loop count is held in the program accessible loop counter (LC). When a block repeat is nested, the start address register, end address register, and LC register for the outer level are stored in the repeat unit, and new values are set up for the new block repeat level. The LP bit in the Internal Configuration Register (ICR) always sets when a block repeat is in progress. Resetting the LP bit (with either the MOV or ICR instruction) stops execution of all current levels of block repeat. The BC[2:0] bits in the ICR indicate the current block repeat nesting level.

A BREAK instruction can stop each of the four nested levels of a block repeat.

2.3.1 Interrupt Handling in the PCU

When the ICU responds to an interrupt, it asserts the ISTAT signal and sends the interrupt vector to the PCU. The asserted ISTAT signal causes the ID cycle to insert an interrupt service pseudo-instruction into the pipeline, so that the next instruction is prevented from being decoded. Instead, the address of the next instruction in the program memory is stored on the stack, and execution continues from the interrupt vector location in the program memory. For more information about the ICU, see Section 2.6, "Interrupt Control Unit (ICU)."

2.4 Computation and Bit-Manipulation Unit (CBU)

The Computation and Bit-Manipulation Unit (CBU) performs all the arithmetic and logical operations within the Core. It contains two main units: the Computation Unit (CU) and the Bit-Manipulation Unit (BMU). The CBU also contains a saturation unit that is shared by both the CU and the BMU units.

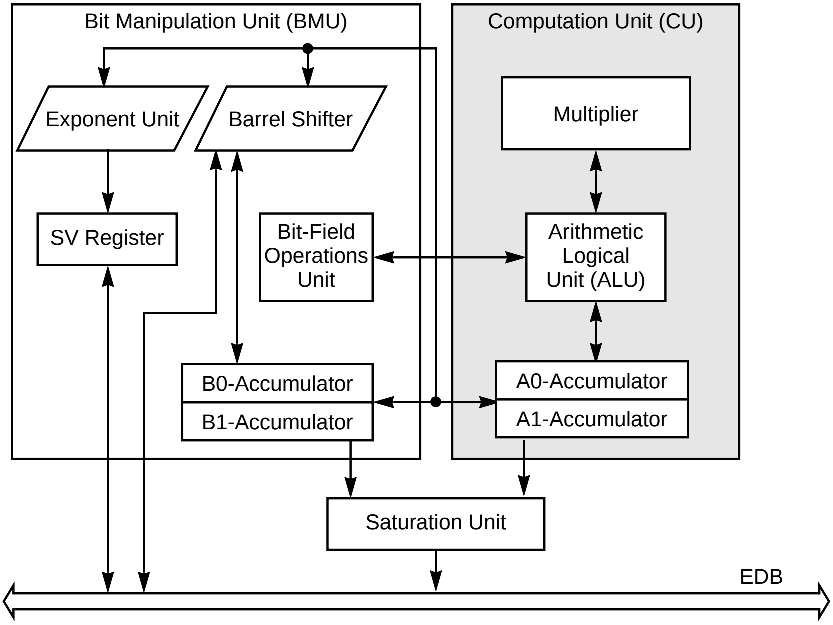
2.4.1 Computation Unit

The Computation Unit (CU) consists of three major parts:

- Two 36-bit accumulators (A0 and A1)
- Arithmetic Logic Unit (ALU)
- Multiplier Unit

The following subsections provide details about each of these three functional blocks. Figure 2.4 shows a block diagram of the CBU with the CU components shaded.

Figure 2.4 CU Block Diagram



2.4.1.1 Ax-Accumulators

Each Ax Accumulator is organized as two 16-bit registers (A0H and A0L, A1H and A1L) with a four-bit extension (A0E and A1E). The Core accesses the two portions of each accumulator as 16-bit data registers, and uses them as either source or destination registers in all relevant instructions. See Section 4.1, “CBU Registers,” for more detailed information on the Ax Accumulators.

2.4.1.2 Arithmetic Logic Unit

The 36-bit Arithmetic Logic Unit (ALU) performs all arithmetic and logical operations on data operands. It can perform positive or negative accumulate, add, subtract, compare and several other operations in a single cycle (operations involving immediate data take two cycles). The ALU is a 36-bit, single cycle, nonpipelined unit that uses two’s complement arithmetic.

The ALU receives one operand from one of the Ax Accumulators and another operand from the output shifter of the multiplier, the XDB (through bus alignment logic), or from the other Ax Accumulator. The source operands are 8, 16, or 36 bits wide and can be addressed using direct or indirect indexed methods, as a register content or as pointed to by the stack pointer. The source and destination Ax Accumulators of an ALU instruction are always the same. For example, in the instruction add `r1, a0` the ALU adds register R1 to the A0 Accumulator, and stores the result in the A0 Accumulator.

The ALU stores results in one of two ways:

1. In one of the Ax Accumulators
2. Transfers results to one of the registers or a data memory location

The ALU uses the second method for addition, subtraction, and compare operations between a 16-bit immediate operand and either a data memory location or one of the registers. This method takes two clock cycles and does not affect the accumulators. The add and subtract operations are read-modify-write instructions. For more information see the ADDV, SUBV, and CMPV instructions in Chapter 7, “Instruction Set.”

Unless otherwise specified, in all operations between an 8-bit or 16-bit operand and a 36-bit Ax Accumulator, the 16-bit operand is regarded as the least-significant word of a 36-bit operand, with sign extension for arithmetic operations and zero extension for logical operations.

The status flags in Status Register 0 and Status Register 1 are affected as a result of the ALU output, the BFO, or the Barrel Shifter operation. A MOV instruction also affects the flags when the entire Ax Accumulator is specified. In most instructions where the ALU result transfers to one of the Ax Accumulators, the flags represent the Ax Accumulator status.

2.4.1.3 Multiplier Unit

The CWDSP1650 uses the two’s complement, single-cycle, nonpipelined multiplier for all core multiplication operations. The CWDSP1650 multiplier supports single-precision and double-precision multiplications, and can perform three types of multiplication:

- Signed-by-signed
- Signed-by-unsigned
- Unsigned-by-unsigned

With the ALU and the multiplier, the CWDSP1650 can perform a single-cycle Multiply-Accumulate (MAC) Instruction.

The Multiplier Unit implements the instructions listed in Table 2.1.

Table 2.1 Multiplier Unit Instructions

Multiply Instructions	
MPY	Multiply
MPYI	Multiply Signed Short Immediate
MPYSU	Multiply Signed by Unsigned
MAA	Multiply and Accumulate Aligned Previous Product
MAASU	Multiply Signed by Unsigned and Accumulate Aligned Previous Product
MAC	Multiply and Accumulate Previous Product
MACSU	Multiply Signed by Unsigned and Accumulate Previous Product
MACUS	Multiply Unsigned by Signed and Accumulate Previous Product
MACUU	Multiply Unsigned by Unsigned and Accumulate Previous Product
MSU	Multiply and Subtract Previous Product
SQR	Square
SQRA	Square and Accumulate Previous Product

The Multiplier Unit consists of the following blocks:

- A 16-bit by 16/32-bit parallel multiplier
- Two 16-bit input registers (X and Y)
- A 32-bit output register (P)
- A product output shifter

Input Registers (X and Y) – The core reads and writes the X and Y Registers as 16-bit operands. The X and Y Registers can also be used as general-purpose temporary data registers. See Section 4.1.2, “X, Y, and P Registers,” for more information.

Output Register (P) – The Multiplier Unit stores results in this 32-bit Register. The core can only move the contents of the P Register to the

A0 and A1 Accumulators. See Section 4.1.2, “X, Y, and P Registers,” for more information.

Product Output Shifter − The P Register is sign-extended to 36 bits and then shifted. The data value can be shifted by one bit to the right, one bit to the left, two bits to the left, or left unshifted. For a right shift, the sign is extended to 36 bits; for a left shift, a zero is appended to the LSBs. The PS bits in Status Register 1 control the shift operations; See Section 4.4.2, “Status Register 1 (ST1),” for more information.

Double-Precision Multiplication – The CWDSP1650 supports double-precision multiplication through several multiplication instructions and an alignment option for the P Register. In multiply-accumulate aligned instructions (MAA and MAASU instructions), the P Register is aligned (shifted 16 bits to the right) before accumulating the partial multiplication result.

Example: Multiplication of 32-bit by 16-bit fractional numbers, where two multiplications are needed.

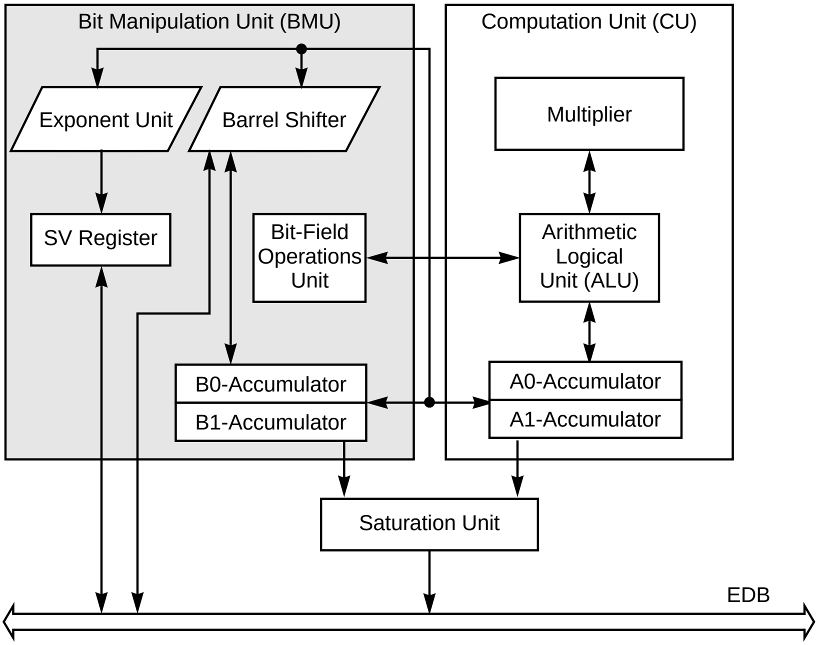
- First, the 16-bit signed multiplier is multiplied with the lower portion of the 32-bit (double-precision) multiplicand using a signed-by-unsigned multiply.
- Then, a signed-by-signed multiplication accumulate operation multiplies the 16-bit signed number with the upper, signed portion of the 32-bit multiplicand and sums this result with the previous result.

For the second operation, it is recommended that the aligned result of the first multiplication is accumulated (using MAA instruction). For the multiplications of two double-precision (32-bit) numbers, the unsigned-by-signed operation can be used. If this operation requires a 64-bit result, the unsigned-by-unsigned operation should be used. For details on the various multiply instructions, see Chapter 7, “Instruction Set.”

2.4.2 Bit-Manipulation Unit (BMU)

The Bit Manipulation Unit (BMU) provides all functionality for shifting, exponent extraction, normalization, saturation, and sign extension. Figure 2.5 shows a block diagram of the CBU with the BMU components shaded.

Figure 2.5 BMU Block Diagram



The Bit-Manipulation Unit (BMU) consists of the following components:

- A full 36-bit Barrel Shifter
- An Exponent unit (EXP)
- A Bit-Field Operation unit (BFO)
- Two 36-bit accumulator registers (B0 and B1)
- A Shift Value (SV) register

2.4.2.1 Barrel Shifter

The 36-bit Barrel Shifter performs arithmetic shift, logical shift, and rotate operations. It is a single-cycle, nonpipelined Barrel Shifter.

The Barrel Shifter receives the source operand from any one of the four accumulators (A0, A1, B0, or B1) or from the EDB (through bus alignment logic). Either the contents of one of the registers or a data memory location may provide the source operands. Source operands can be addressed in either direct memory addressing mode or indirect addressing mode and may be 16 or 36 bits wide. The destination of the shifted value is always one of the four accumulators. The number of bit shifts applied is determined by a constant embedded in the instruction opcode or by a value in the SV register.

When the Barrel Shifter output is put into one of the accumulators, the status of the flag bits represents the accumulator status. See Section 4.4.1, “Status Register 0 (ST0),” for more information on the status flag bits.

2.4.2.2 Exponent Unit

The Exponent Unit (EXP) performs exponent evaluation of an accumulator, a data memory location, or a register. The result of this operation is a signed 6-bit value, sign-extended into 16 bits, which is transferred into the Shift Value register (SV). Optionally, it can also be sign-extended into 36 bits and transferred into one of the Ax Accumulators. The source operand is unaffected by this calculation. The source operand is 36-bits wide when it is an accumulator; 16-bits wide when it is a data memory location or a register.

The Exponent Unit can also be used in floating-point calculations, where it is useful to transfer the exponent result into both the SV register and one of the Ax Accumulators.

2.4.2.3 SV Register

The CWDSP1650 uses the 16-bit Shift Value (SV) Register for shifting operations and exponent calculation. The SV register value determines the number of shifts during shift operations, and enables the core to calculate the number of shifts at run time. The Exponent Unit transfers its output to the SV register for use during floating point calculations.

2.4.2.4 Normalization

The CWDSP1650 performs normalization by one of two methods:

- In the first method, normalization takes two cycles and uses two instructions: EXP and SHFC. EXP evaluates the exponent value of a register, accumulator, or a data memory location. SHFC shifts the evaluated number, according to the exponent result stored in the SV register.

- The second method uses the NORM instruction. This method is slower and has been retained for compatibility with the PineDSPCore instruction set.

2.4.2.5 Bit-Field Operations

The Bit-Field Operation Unit (BFO) is attached to the ALU and sets, resets, changes, or tests up to a 16-bit set within a data memory location or a register. The BFO addresses the data memory location using either a direct or an indirect memory address. The BFO results may affect the flag bits in Status Register 0; see Section 4.4.1, “Status Register 0 (ST0),” for more information on the status flag bits.

The BFO sets, resets, and changes a 16-bit set with the SET, RST, and CHNG read-modify-write instructions. Each instruction requires two cycles and two words, with the 16-bit immediate mask value embedded in the instruction opcode.

Three BFO testing instructions are available: TST0, TST1, and TSTB. TST0 tests up to a 16-bit set for zeroes, TST1 tests for ones, and TSTB tests for a specific bit (1 out of 16) in a data memory location or in a register. BFO testing (TST0 or TST1) requires either one cycle when the mask is in A0L or A1L, or two cycles when the mask value is embedded in the instruction opcode. TSTB always requires a single cycle to execute.

For more details refer to the SET, RST, CHNG, TST0, TST1, and TSTB instructions in Chapter 7, “Instruction Set.”

2.4.2.6 Bx-Accumulators

Each Bx Accumulator is organized as two regular 16-bit registers (B0H and B0L, B1H and B1L) with a four-bit extension nibble. The core accesses the two portions of each accumulator as 16-bit data registers, and uses the Bx Accumulators as 16-bit source or destination data registers in relevant instructions. See Section 4.1, “CBU Registers,” for more details on the Bx Accumulators.

2.4.3 Saturation Unit

The Saturation Unit ranges the outputs of the four accumulators (A0, A1, B0, B1) to fit on the internal bus. Clearing the SAT bit in Status Register 0 to zero enables the saturation unit, and setting SAT to one disables the saturation overflow detection. See Section 4.4.1, “Status Register 0 (ST0),” for more information on the SAT bit.

The enabled saturation overflow process works as follows: with saturation enabled, the Saturation Unit selects one of the four accumulators for transfer and decides whether to transfer the upper or the lower 16 bits of the accumulator. If the accumulator holds an overflowed number (negative or positive number greater than 16 bits), the Saturation Unit transfers the appropriate maximal (0xFFFF) or minimal (0x0000) number that can be represented in 16 bits. Table 2.2 shows the values the Saturation Unit transfers with saturation enabled.

Table 2.2 Saturation Overflow

Value in Accumulator	Transferred Value
Within limits	Value of bits [31:0] unchanged
Above positive limit	0xFFFF
Below negative limit	0x0000

If the accumulator has not overflowed, the saturation unit transfers the real accumulator value, regardless of the value of the SAT flag. If the SAT bit disables saturation, the accumulator value passed is unaffected, regardless of overflow. Saturation arithmetic also selectively limits overflow from the high portion of an accumulator to the sign extension bits.

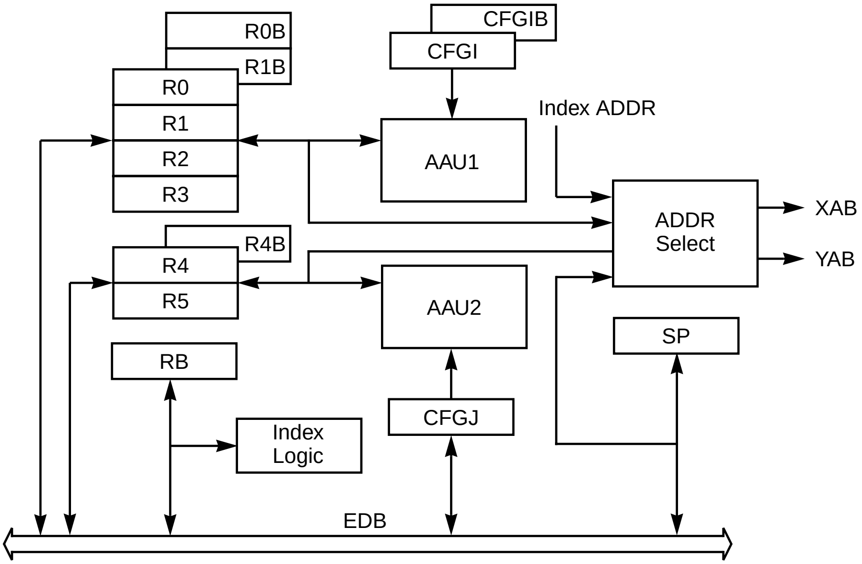
If saturation occurs while performing a move instruction (MOV or PUSH) from one of the accumulators, the core does not change the value of the accumulator. The value transferred is limited to a full-scale 16-bit positive or negative value. Limiting is performed even if the transfer does not immediately follow the accumulator overflow. When an accumulator is swapped using the SWAP instruction, limitation is performed when the value is transferred. The SAT bit enables saturation for move instructions. When limiting occurs, the L flag in ST0 is set to one.

The LIM instruction activates saturation on a 36-bit Ax Accumulator. When there is an overflow from the high portion of an Ax Accumulator to the extension bits and an LIM instruction is executed, the accumulator is limited to a full-scale 32-bit positive (0x7FFF FFFF) or negative (0x8000 0000) value. Limiting is performed even if the LIM instruction does not immediately follow the accumulator overflow. If the core swaps an accumulator, limitation occurs when the value is operated on by the LIM instruction. The LIM instruction can use the same accumulator for both source and destination or it can use one Ax Accumulator, which does not change and then transfer the limited result into the other Ax Accumulator. For more details, refer to the LIM instruction in Chapter 7, "Instruction Set." When limiting occurs, the L flag bit in Status Register 0 is set. Enabling or disabling the SAT bit has no effect on the execution of the LIM instruction.

2.5 Data Address Arithmetic Unit (DAAU)

The Data Address Arithmetic Unit (DAAU) stores all address and effective address calculations necessary to locate data operands in memory. It contains two data address generators and six different data address pointers. The DAAU provides both linear and modulo arithmetic address generation capabilities. The DAAU operates in parallel with the Computation Unit, thus freeing the ALU for calculation purposes. Figure 2.6 shows a diagram of the DAAU functional blocks.

Figure 2.6 Data Address Arithmetic Unit (DAAU)



2.5.1 DAAU Registers

The DAAU contains the following registers:

R0-R5 – General purpose and address generation registers. These registers are divided into two groups, R0-R3 and R4-R5. Each group is served by its own arithmetic unit. Either group can generate addresses for the X data memory on the XAB address bus, but only R4-R5 can generate addresses for the Y memory. During multiply or multiply-accumulate instructions, a register from each group is used, one from R0-R3 for generating XAB and one from R4-R5 for generating YAB. R4-R5 can also address program memory during MOVD and MOVP instructions.

RB – Index Base Register. This register value is added to the offset in index mode addressing. The RB register is part of the global register set and can be used as a general-purpose register.

SP – Stack Pointer. The SP register controls the pre- and post-modification logic used in DAAU operations.

CFGI, CFGJ – Configuration Registers. These registers specify parameters for modulo addressing and to increment/decrement step control. CFGI and CFGJ affect the Address Arithmetic Units results during modification of R0-R5. The CFGI register is used for registers R0-R3 and CFGJ for registers R4-R5.

R0B, R1B, R4B, CFGIB – Context switch registers for the R0, R1, R4, and CFGI registers.

After a DAAU address pointer (Rn) accesses the memory, it can be postmodified through the STEP field in the Configuration Registers. A length value can be associated with each pointer to implement step modification of the pointer. The Configuration Registers also provide supports for circular buffers through automatic up/down modulo addressing.

The Rn registers may also be used for loop control. The MODR instruction sets the R flag in Status Register 0 if the Rn register being modified is zero following its execution. Conditional branch instructions dependent on the value of the R flag can then implement loops.

2.5.2 Addressing Modes

The DAAU can generate data memory addresses using a number of different methods:

- Long direct addressing
- Short direct (paged) addressing
- Stack Pointer operations
- Indexed addressing
- Indirect addressing

These addressing modes can be used to create data memory structures for circular buffers, delay lines, FIFOs, other pointers to the software stack. The following subsections describe each addressing mode in more detail.

2.5.2.1 Long Direct Addressing Mode

Long Direct Addressing Mode uses 16 bits embedded in the instruction opcode as the 16-bit data memory address. Any location in the 64 Kword memory space can be directly addressed in two cycles.

2.5.2.2 Short Direct Addressing Mode

Short direct addressing uses eight bits embedded in the instruction opcode as the LSB plus eight bits from the PAGE field of Status Register 1 as the MSB to compose the 16-bit data memory address. The PAGE field can then be set to values between 0 and 255, where page 0 corresponds to addresses 0 to 255 in XRAM. This addressing mode allows for single cycle data memory access.

2.5.2.3 Stack Pointer Operations

The core can use RB as either an array pointer or in conjunction with the Stack Pointer (SP). When the stack is used for transferring subroutine parameters, initializing RB with the value of the SP enables quick access to these parameters. This method of operation is particularly useful for high level language compilers, when RB can be used as a frame pointer to hold the value of the SP as subroutines are entered.

The stack pointer is predecremented for a PUSH operation and postincremented for a POP operation. In other words, the Stack grows from a high memory address towards a low memory address.

2.5.2.4 Indexed Addressing Mode

Indexed addresses are derived by adding an offset to the RB register. The offset is either a signed short immediate 7-bit value derived from a field of the instruction opcode or a 16-bit immediate value (a second instruction word). The base register content is unaffected by the instruction. Notice that the index addressing mode, unlike the linear and modulo addressing modes, uses address premodification and not postmodification.

2.5.2.5 Indirect Addressing Modes

The indirect addressing method uses addresses taken from the R0-R5 registers to provide data or program (in MOVP and MOVD instructions) memory addresses. These registers can be postmodified by the AAUs with a linear or modulo operation. Table 2.3 lists the registers and bit fields used during the indirect addressing modes.

Table 2.3 Indirect Addressing Mode Bits

Address Registers	Enable Bits (ST2)	Configuration Registers	Modulo Bit Field	Linear Bit Field
R0-R3	M0-M3	CFGI	MODI	STEPI
R4-R5	M4-M5	CFGJ	MODJ	STEPJ

Each Rn address register has a corresponding modulo enable bit in ST2. Setting one of the Mn bits to one enables modulo addressing for the Rn bit, and clearing the Mn bit to zero disables modulo calculation mode. The MODI and MODJ bit fields of the CFGI and CFGJ registers contain the modulo values to be used in the modulo calculation; the STEPI and STEPJ bit fields contain the linear modifiers. Please remember that CFGI (MODI/STEPI) modifies only the R0-R3 registers, while CFGJ (MODJ/STEPJ) is used exclusively with the R4-R5 registers.

Linear (Step) Modification – In linear modification, the pointer (Rn) is modified by postincrementing by one, postdecrementing by one, or adding the value specified in the STEPI/STEPJ field of the corresponding CFGI/CFGJ register. The STEP bit field value is a two's complement seven-bit number that ranges from -64 to +63.

Modulo Modification – Modulo and linear modification methods operate in a similar fashion, except that the range of address values is limited by the MODI/MODJ bit fields of the CFGI/CFGJ registers. The MODI/MODJ fields are nine bits in length and can specify circular buffers of up to 512 (2^9) words.The six Mn bits in ST2 enable modulo calculation for each register. The MODI and MODJ fields in the CFGI and CFGJ registers determine the modulo settings. One address register from each group may be updated in a single instruction cycle.

Modulo Constraints – For modulo calculation, the following constraints must be satisfied (m = modulo factor; q = stepx, +1 or -1):

1. Only the p least-significant bits (LSBs) of Rn can be modified during modulo operation, where p is the minimal integer that satisfies $2^p \geq m$. Rn should be initiated with a number whose p LSBs are less than m.
2. The constraints when modulo m is a power of 2 (full modulo operation):
   - The lower boundary (base address) must have zeros in at least the k LSBs, where k is the minimal integer that satisfies $2^k > m - 1$.
   - MODn (n denotes either I or J) must be loaded with $m – |q|$, where $|q|$ denotes the absolute value of q.
   - $m \geq q$.
3. The constraints when modulo m is *not* a power of 2:
   - The lower boundary (base address) must have zeros in at least the k least-significant bits, where k is the minimal integer that satisfies $2^k > m – |q|$.
   - MODx (x denotes I or J) must be loaded with $m – |q|$.
   - m must be an integer multiple of q (always true for $q = \pm1$).
   - Rn should be initialized with a value that contains an integer multiple of $|q|$ or zeros in its k least-significant bits.

Modulo Modifier Operation – The modulo modifier operation, which is a postmodification of the Rn register, is defined as follows:

- Rn ← 0 in k LSB; if Rn is equal to MODx in k LSBs and $q \geq 0$,
- Rn ← MODx in k LSB; if Rn is equal to 0 in k LSBs and $q < 0$,
- Rn (k LSBs) ← Rn+q (k LSBs); otherwise
- When $m = |q|$ (for example, MODx = 0), modulo operation is: Rn ← Rn.

Modulo and Step Example 1 – For m = 7 with stepx = 1 (or +1 selected in instruction):

MODx = 7 – 1 = 6,
Rn = 0x0010.

The sequence of Rn values is: 0x0010, 0x0011, 0x0012, 0x0013, 0x0014, 0x0015, 0x0016, 0x0010, 0x0011, and so on.

Modulo and Step Example 2 – For m = 8 with stepx = 2:

MODx = 8 – 2 = 6,
Rn = 0x0010.

The sequence of Rn values is: 0x0010, 0x0012, 0x0014, 0x0016, 0x0010, 0x0012, and so on.

Modulo and Step Example 3 – For m = 9 with stepx = -3:

MODx = 9 – |-3| = 6,
Rn = 0x0016.

The sequence of Rn values is: 0x0016, 0x0013, 0x0010, 0x0016, 0x0013, and so on.

Modulo and Step Example 4 – For m = 8 with stepx = 3, ($2^3 = 8$) - full modulo support:

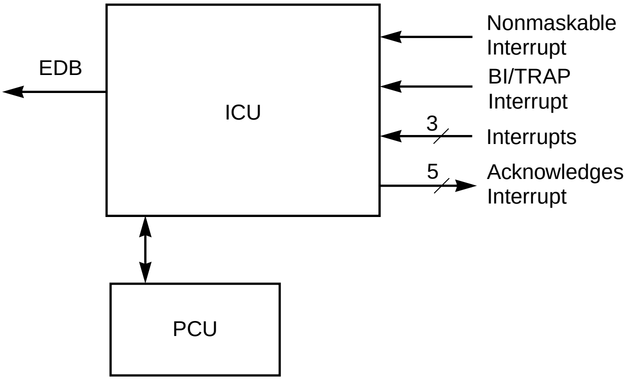
MODx = 8 – 3 = 5,
Rn = 0x0010.

The sequence of Rn values is: 0x0010, 0x0013, 0x0016, 0x0011, 0x0014, 0x0017, 0x0012, 0x0015, 0x0010, 0x0013, and so on.

2.6 Interrupt Control Unit (ICU)

Figure 2.7 shows the Interrupt Control Unit (ICU).

Figure 2.7 Interrupt Control Unit



The ICU implements the protocol for the five external interrupts: BI/TRAP, NMI, I0, I1 and I2. Each interrupt has its own acknowledge, so there are ten signals in all. Interrupts are sampled on the rising edge of ICU_CLK, which is a copy of the core clock but is not affected by wait states. Once sampled as active, an interrupt is held until serviced, no matter what the behavior of the interrupt request signal.

If two interrupts occur simultaneously (or nearly simultaneously) such that both sampled and held in the ICU together, then logic in the ICU prioritizes them, in the order from BI/TRAP (highest), NMI, I0, I1, to I2 (lowest), and services the highest priority interrupt first. The ICU generates an interrupt status signal (ISTAT) to the PCU and an interrupt vector. The interrupt vector is a program location to which program control will jump when the interrupt is serviced. See Section 6.3, “Interrupts,” for more information about interrupts, the interrupt signals, and interrupt priority.

When the PCU services the interrupt request from the ICU, it signals the ICU to acknowledge that interrupt and remove the interrupt status condition. Only one interrupt is acknowledged on its own dedicated acknowledge line to avoid clearing interrupts that have not been serviced. The off-core logic must remove the interrupt request before another interrupt can be serviced for a specific request line.

Chapter 3
Data Formats, Memory and Addressing

The CWDSP1650 allocates two independent memory spaces: the data space and the program space. The data and program memory spaces are both 64 Kwords in size. The data memory space is further divided into the X-Memory and Y-Memory spaces to support parallel data moves. This chapter describes the mapping of the program and data spaces and the different addressing modes to and from these spaces.

This chapter contains the following sections:

- Section 3.1, “Data Formats”
- Section 3.2, “Program Memory”
- Section 3.3, “Data Memory”

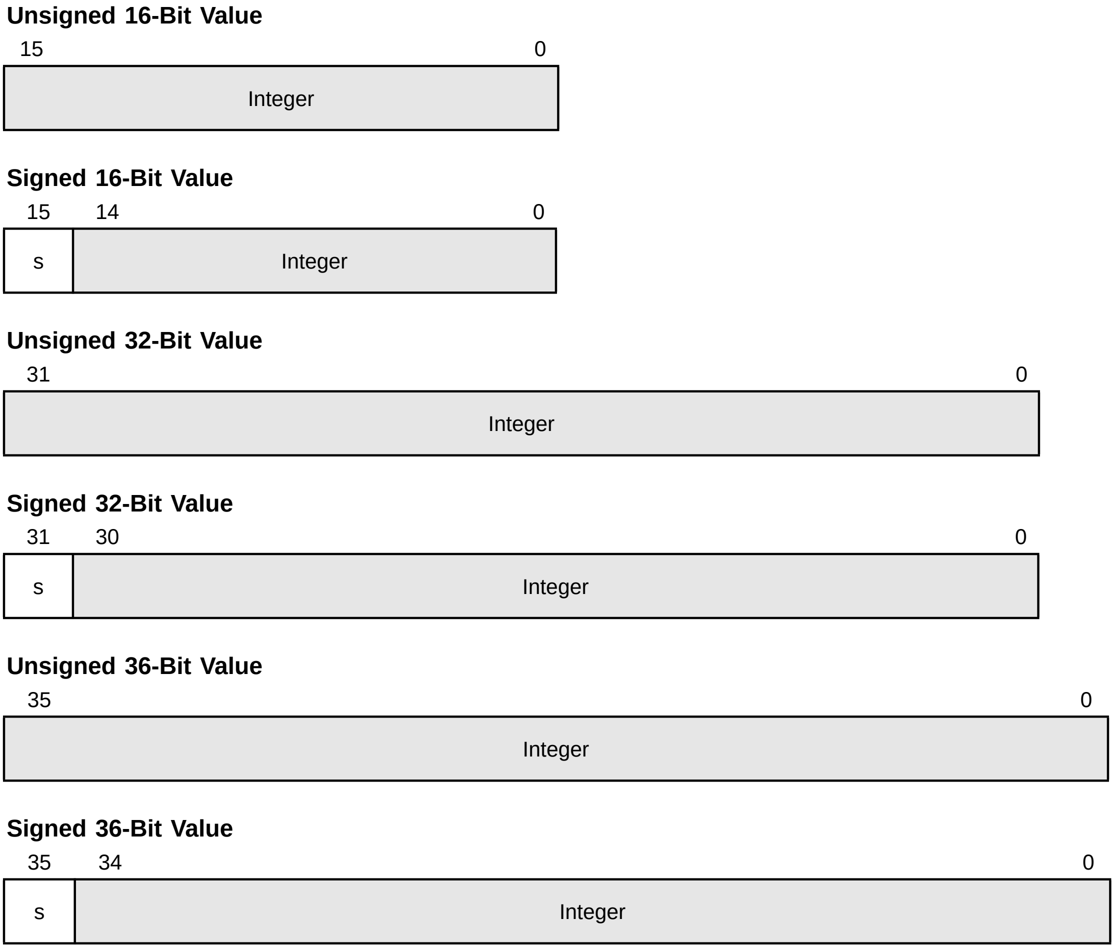
3.1 Data Formats

The CWDSP1650 supports six integer formats:

- 16-bit signed integers
- 16-bit unsigned integers
- 32-bit signed integers
- 32-bit unsigned integers
- 36-bit signed integers
- 36-bit unsigned integers

Figure 3.1 shows the signed and unsigned integer formats. In the figure, an “s” in bit 15 or 31 refers to the sign bit.

Figure 3.1 Signed and Unsigned Integer Formats


Unsigned 16-Bit Value

Signed 16-Bit Value

Unsigned 32-Bit Value

Signed 32-Bit Value

Unsigned 36-Bit Value

Signed 36-Bit Value

Table 3.1 lists the valid ranges for the signed and unsigned integer formats.

Table 3.1 Signed and Unsigned Integer Ranges

Data Type	Width (Bits)	Lower Limit	Upper Limit
Signed	16	-2^{15}	$2^{15} – 1$
Unsigned	16	0	$2^{16} – 1$
Signed	32	-2^{31}	$2^{31} – 1$
Unsigned	32	0	$2^{32} – 1$
Signed	36	-2^{35}	$2^{35} – 1$
Unsigned	36	0	$2^{36} – 1$

In general, the core performs arithmetic operations in 2’s complement and treats the numbers as signed. Some exceptions do occur, however, depending on the specific instruction.

Only the accumulators can support 36-bit integers. Usually, the accumulators are seen as 36-bit signed registers or 32-bit registers if the extension bits are not used (they are the same as bit 31, the sign bit). The add `a0, a0` instruction can involve either two 36-bit signed numbers or two 32-bit signed numbers. See Section 4.1, “CBU Registers,” for more information on the accumulators.

The MACUU instruction multiplies two unsigned numbers and stores the result, which is a 32-bit unsigned number, in the P register. A subsequent instruction that transfers the P register into an accumulator results in a 36-bit unsigned number in the accumulator.

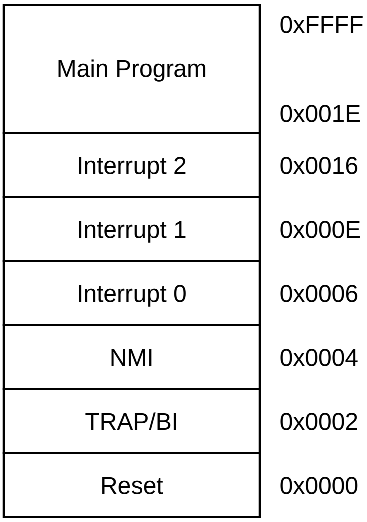
In most cases, a 16-bit register is used as a signed 16-bit number (for example, add `r1, a1`). Whether the register is signed or unsigned depends on the instruction. For example, in the instruction add `r1, a0` r1 is treated as a signed 16-bit number. But for the instruction `addl r1, a0` r1 is treated as an unsigned 16-bit number.

The Loop Counter (LC) Register can be used as a loop counter in the BKREP instruction. In this case, the register value is unsigned. However, the LC register can also be used as a general-purpose register in the add `lc, a0` instruction. In this case, the LC register value is signed. See Chapter 4, “Registers,” for more information on the LC register and the P register.

3.2 Program Memory

The program memory address space is contained in an off-core PRAM module and encompasses the program code, program constants, interrupt routines, and reset routines. Figure 3.2 shows the program memory map.

Figure 3.2 Program Memory Map



Addresses 0x0000 – 0x0016 contain the vector addresses for:

- Reset
- TRAP/BI (software/hardware interrupt)
- NMI (Nonmaskable interrupt)
- Maskable interrupts (Interrupts 0, 1, and 2)

The Reset, TRAP/BI, and NMI vectors are separated by two locations so that branch instructions can be accommodated in those locations, if desired. The maskable interrupt vectors are separated by eight words, so fast interrupt service routines can be accommodated without branch statement delays. Note that the TRAP/BI interrupt vector addresses are reserved for use by the emulator during debug.

The CWDSP1650 supports an off-core wait-state generator to help interface slow program memory devices. To provide this ability, the wait-state generator provides a WAIT input to the core that stops the core clock when asserted. This functionality is usually implemented within the off-core Clock Control Unit.

The CWDSP1650 also supports internal program memory protection. This mechanism allows the implementation of a secured version of an internal program-based chip, which protects the internal program memory from being read without proper authorization. See Section 6.6, "Program Protection Mechanism," for more details about the protection operation.

3.2.1 Program Memory Addressing Modes

Program memory addresses are generated by the Program Control Unit (PCU), which performs all instruction fetching, exception handling and loop control. For the CWDSP1650, the program memory can be addressed either in indirect addressing mode, or in special relative addressing mode.

3.2.1.1 Indirect Addressing Mode

In this mode, the Rn registers of the DAAU and the accumulators are used for addressing the program memory (used in the MOVD and MOVP instructions.)

3.2.1.2 Special Relative Addressing Mode

In this mode, Branch-Relative (BRR) and Call-Relative (CALLR) instructions support jumping relative to the PC (from PC – 63 to PC + 64) that facilitates smaller and more easily relocatable code.

3.3 Data Memory

The CWDSP1650 data memory space is divided into separate X-Memory and Y-Memory spaces to allow single cycle parallel data accesses. The size and configurations of these memories are determined according to the MEM_CFG[2:0] core input. Table 3.2 lists the X-Memory and Y-Memory configurations available and the corresponding MEM_CFG[2:0] value.

Table 3.2 Data Memory Address Mapping

MEM_CFG[2:0]	Y-Memory		X-Memory	
	Size (Words)	Address Range	Size (Words)	Address Range
000	32 K	0x8000 to 0xFFFF	32 K	0 to 0x7FFF
001	16 K	0xC000 to 0xFFFF	48 K	0 to 0xBFFF
010	8 K	0xE000 to 0xFFFF	56 K	0 to 0xDFFF
011	4 K	0xF000 to 0xFFFF	60 K	0 to 0xEFFF
100	2 K	0xF800 to 0xFFFF	62 K	0 to 0xF7FF
101	1 K	0xFC00 to 0xFFFF	63 K	0 to 0xFBFF
110	512	0xFE00 to 0xFFFF	63.5 K	0 to 0xFDFF
111	256	0xFF00 to 0xFFFF	63.75 K	0 to 0xFEFF

The X-Memory and Y-Memory can contain RAM, ROM, and I/O. The core also uses memory-mapped I/O; several addresses of the data space may be reserved for peripherals, depending on the specific chip configuration. The core implements slow peripheral interfacing through an off-core BIU, which controls the number of wait states. To control the wait states, the BIU uses the WAIT input of the core to stop CORE_CLK during wait stated accesses.

3.3.1 Data Memory Addressing Modes

The CWDSP1650 supports five data memory addressing modes:

- Short direct
- Long direct
- Indirect
- Short index
- Long index

The software stack, which is located in the data memory, is addressed using the Stack Pointer (SP) register. See Section 4.2.4, “Stack Pointer Register (SP),” for more information.

3.3.1.1 Short Direct Addressing

In the short direct configuration, the 16-bit address is generated by appending the 8-bit PAGE field from Status Register 1 to the 8 LSBs of the opcode (i.e. the PAGE field forms the MSB of the address). Each page consists of 256 words, as shown in Table 3.3. In short direct addressing, all memory locations in the 64-Kword space can be directly accessed in a single cycle.

Table 3.3 Short Direct Addressing Page Values

Page	X-Memory Address Range
0	0 to 255
1	256 to 511
...	...
255	0xFF00 to 0xFFFF

3.3.1.2 Long Direct Addressing

In the long direct configuration, the 16-bit address is taken directly from the opcode as the second word of the instruction. Using long direct addressing, all memory locations in the 64 Kword space can be directly accessed in two cycles.

3.3.1.3 Indirect Addressing

Indirect addressing has two stages: address generation and pointer postmodification. The Rn registers (pointers) of the DAAU are used as 16-bit addresses for indirect addressing the X-Memory and Y-Memory. In one-operand addressing, one pointer supplies the address for the transaction. In two-operand addressing, two pointers simultaneously select the address to the X-Memory and Y-Memory spaces. See Section 2.5.2.5, “Indirect Addressing Modes,” for details of pointer postmodification.

3.3.1.4 Short Index Addressing

The base register RB plus an index value (offset7) are used for index-based indirect addressing of the X-Memory or the Y-Memory. The index value is a short immediate value embedded in the instruction opcode (offset7), and can range from -64 to +63. The actual address is RB + offset7, the RB contents being unaffected by the operation.

3.3.1.5 Long Index Addressing

The contents of base register RB plus a 16-bit immediate index value are used for index-based indirect addressing of the X-Memory and Y-Memory. The immediate index value is embedded in the instruction opcode, and can range from -32768 to +32767. The contents of RB are unaffected.

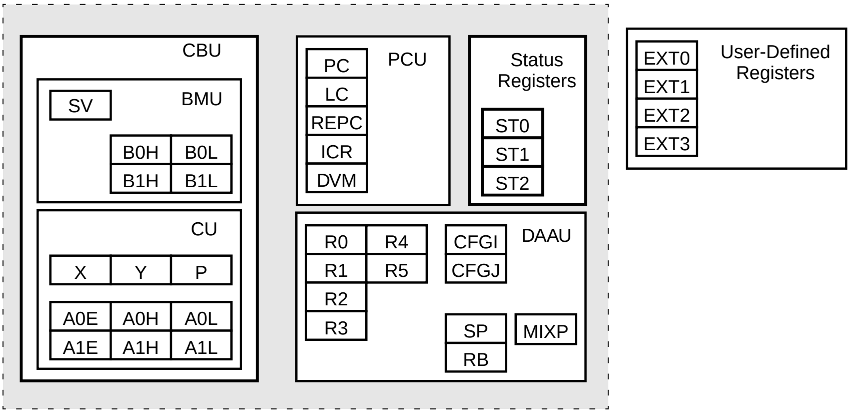
Chapter 4 Registers

This chapter describes the CWDSP1650 registers in detail and defines the bit fields within them. Chapter 4 is further divided into the following sections:

- Section 4.1, “CBU Registers”
- Section 4.2, “DAAU Registers”
- Section 4.3, “PCU Registers”
- Section 4.4, “Status Registers”
- Section 4.5, “User-Defined Registers”

Figure 4.1 shows the accessible registers/counters in their respective functional blocks within the core. Internal CWDSP1650 functional blocks are shown inside the shaded area.

Figure 4.1 CWDSP1650 Registers



The following sections contain detailed descriptions for each of these components and their programmable registers. Unless otherwise stated, all on-core registers, accumulators and counters clear to zero after reset.

4.1 CBU Registers

The Computation and Bit Manipulation Unit contain four accumulators (A0, A1, B0, B1), two multiplier input registers (X, Y), a multiplier product register (P), and a shift value register. Table 4.1 lists the CBU registers; the remainder of this section describes each register set in detail.

Table 4.1 CBU Registers

Register	Abbreviations	Page
Ax Accumulators	(AxE, AxH, or AxL)	4-3
Bx Accumulators	(BxE, BxH, or BxL)	4-5
X Register	–	4-6
Y Register	–	4-6
P Register	–	4-6
Interrupt Context Switching Registers	–	4-7
Shift Value Register	SV	4-9

4.1.1 Ax and Bx Accumulators

There are four accumulators: two Ax Accumulators in the Computation Unit (CU), and two Bx Accumulators in the Bit Manipulation Unit (BMU). Each accumulator is organized as two regular 16-bit registers with 4-bit extensions as shown in Table 4.2.

Table 4.2 Ax and Bx Accumulator Organization

Accumulator	Upper 16-Bit Register	Lower 16-Bit Register	4-Bit Extension
A0	A0H	A0L	A0E
A1	A1H	A1L	A1E
B0	B0H	B0L	B0E
B1	B1H	B1L	B1E

The core accesses the upper and lower portions of each accumulator as separate 16-bit data registers and uses them as 16-bit source or destination registers in all relevant instructions.

Saturation arithmetic selectively limits overflow from the high portion of an accumulator to the extension bits, when performing a move instruction from one of the accumulators through the EDB, or when using the LIM instruction which forces saturation on the 36-bit accumulator. For more information about saturation, see Section 2.4.3, “Saturation Unit.”

All four accumulators clear to zero after a core reset.

4.1.1.1 Ax Accumulators (A0 and A1)

The Ax Accumulators store the source/destination operands of the ALU, Barrel Shifter, or Exponent unit. AxH and AxL may also be used as general purpose 16-bit data registers.

Extension Nibbles – 4-bit extension nibbles A0E and A1E protect against 32-bit overflow. These two nibbles are accessed as bits 15 to 12 of the status registers ST0 (for A0E) and ST1 (for A1E). When the result of an ALU output crosses bit 31, the Extension Flag (E) in ST0 is set, which indicates a crossing of the AxH MSB. An extension nibble allows for up to 15 overflows or underflows. When one of these limits is exceeded, the sign is lost beyond the MSB of the ALU output and/or the extension nibble MSB. When this happens, the Overflow Flag (V) is set, which also causes the Limit Flag (L) to be set. Refer to Section 4.4, “Status Registers,” for more details on the extension, overflow, and limit flags.

Sign Extension – The Core normally sign extends smaller operands written to the 16-bit AxL or AxH registers within the 36-bit Ax Accumulators. This can happen when data from either the EDB, the ALU, or the Exponent Unit (during some CBU operations) is written to one of these accumulators. Specific instructions can also be used to suppress sign extension, for example, `mov 100, a0, eu`. See Chapter 7, "Instruction Set," for details.

Loading of Ax Accumulators – When an instruction loads an Ax Accumulator with a 16-bit data value, the complete 36-bit word of the accumulator is affected as shown in Table 4.3, depending on how the accumulator is specified in the instruction field:

Table 4.3 Ax Accumulator Loading Values

Accumulator	eu[1]	Accumulator Fields after Instruction Execution		
		AXE	**AXH**	**AXL**
AX	–	Sign-extended	Sign-extended	DATA
AXL	–	Cleared	Cleared	DATA
AXH	–	Sign-extended	DATA	Cleared
AXH	eu	Unaffected	DATA	Cleared

[1] Extension unaffected. See Table 7.4 for a more complete description.

For example, the instruction `mov r0, a1l` loads a 16-bit value into A1L, and sign extends A1H; the instruction `mov r1, a1h` loads a 16-bit value into A1H, and clears A1L. The instructions ADDL and ADDH are exceptions. The ADDL instruction treats the 16-bit source operand as an unsigned number, extends this number to 36 bits by adding zeroes, and adds the result to the destination 36-bit Ax Accumulator. The ADDH instruction loads a 16-bit value into AxH, but leaves AxL unaffected. An instruction that loads a 36-bit value into an Ax Accumulator does not follow the above rules, such as a shift instruction or the SWAP instruction.

See Section 4.1.1.3, "Swapping the Accumulators," for a detailed account of accumulator swapping, and Chapter 7, "Instruction Set," for more information on the MOV, ADDL, ADDH, and SWAP instructions.

4.1.1.2 Bx Accumulators (B0 and B1)

Each Bx Accumulator is either a source operand of the BMU Exponent Unit or a source/destination operand of the BMU Barrel Shifter.

Extension Nibbles – Extension nibbles B0E and B1E protect against 32-bit overflows. When the result of the Barrel Shifter crosses bit 31, it sets the Extension Flag (E) in Status Register 0, which indicates a crossing of the BxH MSB. When the sign is lost beyond the MSB of the Barrel Shifter and/or the MSB of the extension nibble, the Overflow (V) and Limit (L) flags in Status Register 0 are set. See Section 4.4, “Status Registers,” for more details about the extension, overflow, and limit flags.

The extension bits B0E and B1E are not accessible directly. They can be accessed with the aid of a single-cycle shift instruction, or by swapping to the Ax Accumulator.

Sign Extension – The core normally sign extends smaller operands written to the 16-bit BxL or BxH registers within the 36-bit Bx Accumulators. This can happen when either data from the EDB or the Barrel Shifter (in shift operations) is written to the Bx Accumulators.

Loading of Bx Accumulators – When an instruction loads a Bx Accumulator with a 16-bit data value, the 36-bit accumulator is affected as shown in Table 4.4, depending on how the accumulator is specified in the instruction field.

Table 4.4 Bx Accumulator Loading Values

	Accumulator Fields after Instruction Execution		
Accumulator	**BXE**	**BXH**	**BXL**
BX	Sign-extended	Sign-extended	DATA
BXL	Cleared	Cleared	DATA
BXH	Sign-extended	DATA	Cleared

For example, the instruction `mov r0, b1l` loads a 16-bit value into B1L, and sign extends B1H; the instruction `mov r1, b1h` loads a 16-bit value into B1H, and clears B1L. However, an instruction that loads a 36-bit value into a Bx Accumulator does not follow the above rules, such as using shift instructions or SWAP instruction.

See Section 4.1.1.3, “Swapping the Accumulators,” for a detailed account of accumulator swapping, and Chapter 7, “Instruction Set,” for more information on the MOV, ADD, and SWAP instructions.

4.1.1.3 Swapping the Accumulators

The SWAP instruction swaps the contents of the Ax Accumulators and the Bx Accumulators in a single cycle. Either two 36-bit registers or all four 36-bit registers can be swapped in one cycle. Swapping can also be enabled between a specific Ax Accumulator and a specific Bx Accumulator such that, in the same cycle, a Bx Accumulator is swapped with an Ax Accumulator. Similarly, swapping is enabled between a specific Bx Accumulator into an Ax Accumulator, and in the same cycle from that Ax Accumulator into another Bx Accumulator.

For a summary of the 14 swap options, refer to the SWAP instruction in Chapter 7, “Instruction Set.”

4.1.2 X, Y, and P Registers

The X, Y, and P registers are part of the Multiplier Unit in the CU and can be used as general-purpose data registers. For more information about these three registers, see Section 2.4.1.3, “Multiplier Unit.”

The 16-bit X and Y registers are input registers, which the core reads or writes as 16-bit operands. The core receives information from the X register through the XDB and from the Y Registers through the YDB; the core writes through the EDB for both registers. Both the X and Y registers clear to zero after reset.

The 32-bit P register is used to store multiplication results, and can be shifted prior to input into the ALU or to support double precision multiplication. The contents of the P register can only be moved to the Ax Accumulators. The most-significant 16 bits of the P register (PH) can be written through the EDB bus, which enables a single-cycle restore of these bits during an interrupt service routine. The core updates the P register only after a multiply instruction and not after a change in the input registers. Figure 4.2 shows the P register bit field.

Figure 4.2 P Register

31 16	15 0
PH	PL

The P register supports double-precision during a multiply instruction. To shift the partial multiplication result, the contents of the P register is shifted 16 bits to the right before the contents are accumulated. For more details, see the multiply instructions in Chapter 7, “Instruction Set.”

4.1.3 Interrupt Context Switching Registers

When an interrupt occurs the contents of the registers used by the interrupt service routine must be saved. This allows the original program to safely resume operation after the interrupt service routine is complete. To reduce the overhead of saving these registers, the core can optionally activate a context switching mechanism for the interrupts NMI, INT0, INT1, and INT2. Setting the corresponding bit in the ICR activates the context switching mechanism for the specified interrupt. See Section 4.3.2, “Internal Configuration Register (ICR),” for more information on the context switching enable bits (IC[2:0], NMIC).

If an interrupt occurs while context switching is enabled, certain registers are saved automatically without an increase in interrupt latency. When returning from the interrupt service routine the original register values are restored automatically (see the RETI and CNTX instructions in Chapter 7, “Instruction Set,” for more information.)

Context switching involves three parallel mechanisms:

1. Push to/pop from dedicated shadow bits
2. Swap of a dedicated page register
3. Swap between the A1 Accumulator and B1 Accumulator

The core saves the ST0[0], ST0[11:2], ST1[11:10], and ST2[7:0] register bits automatically as shadow bits (in a one stack level register.) The data bits can be pushed to or popped from the shadow registers. Figure 4.3 shows the Shadow Register mapping for Status Register 0.

Figure 4.3 ST0 and Shadow Registers

15 12	11	10	9	8	7	6	5	4	3	2	1	0
A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT
Shadow	Z	M	N	V	C	E	L	R	IM1	IM0		SAT

Figure 4.4 shows the Shadow and Alternative Registers available for the shadow mapping of Status Register 1.

Figure 4.4 ST1 and Shadow Registers

15 12	11 10	9 8	7 0
A1E	PS	RES	PAGE
Shadow	PS		
	Alternative		PAGE

The core swaps the page bits in ST1[7:0] to an alternative register instead of the shadow register. When a context switch occurs, the current page is saved into the alternative register, and the previous (stored) value of the page is restored. The stored value should point to the interrupt page to avoid additional initialization. When returning from the interrupt, the interrupt page is saved again into the alternative register for the next interrupt, and the page used before entering the interrupt service routine is swapped with the PAGE bits in ST1. The alternate PAGE bits are not accessible directly. They can be set up through the normal PAGE bits using the CNTX instruction.

Figure 4.5 shows the Shadow Register mapping for Status Register 2.

Figure 4.5 ST2 and Shadow Registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IP1	IP0	IP2	RES	IU1	IU0	OU1	OU0	S	IM2	M5	M4	M3	M2	M1	M0
Shadow								S	IM2	M5	M4	M3	M2	M1	M0

During a context switch the core can automatically swap the A1 Accumulator and B1 Accumulators, therefore, it is normally convenient to store data needed for interrupt routines in the B1 Accumulator.

4.1.4 Shift Value Register

The CWDSP1650 uses the 16-bit Shift Value (SV) Register for shifting operations and exponent calculation. The value in SV determines the number of shifts during shift operations, and enables the Core to calculate the number of shifts at run time. The Exponent Unit transfers its output to the SV Register for use during floating point calculations. This register can also be used as a general-purpose data register. The SV register clears to zero after reset.

4.2 DAAU Registers

This section describes the Data Address Arithmetic Unit (DAAU) registers. These registers are normally used in generation of addresses but some can also be used as general purpose 16-bit registers. Further information about how to use these registers is available in Section 2.5, “Data Address Arithmetic Unit (DAAU).” Table 4.5 lists all DAAU registers, with descriptions of each in the following subsections.

Table 4.5 DAAU Registers

Register	Abbreviations	Page
Address Registers 0-5	R0-R5	4-10
Configuration Registers	CFGI, CFGJ	4-11
Base Register	RB	4-11
Stack Pointer Register	SP	4-12
Alternative Bank Registers	R0B, R1B, R4B, CFGIB	4-12
Min/Max Pointer Latching Register	MIXP	4-13

4.2.1 Address Registers (R0-R5)

The six 16-bit Address Registers (Rn) are divided into two groups: R0 to R3, and R4 to R5. These registers can be used to accesses X and Y memory spaces or program memory space as shown in Table 4.6.

Table 4.6 Rn Register Grouping

Address Bus	Associated Rn Registers
XAB	R0 through R5
YAB	R4 through R5
IAB	R4 through R5

The CWDSP1650 can simultaneously access data in X and Y memory spaces during multiply operations. For example, `mac (r4), (r0), a0` performs a multiply on the Y space data (pointed to by R4) and the data in X space (pointed to by R0) and adds the result to accumulator A0.

The Rn registers can also be used as loop counters with the MODR instruction, which is used to increment or decrement the Rn registers. The R flag in Status Register 0 is set if the MODR (or any Rn modification operation) results in that Rn register being set to zero. Refer to the R flag in Section 4.4.1, “Status Register 0 (ST0),” for more information.

4.2.2 Configuration Registers

The configuration registers, CFGI and CFGJ, define the modulo and/or step values for the addressing modes used in conjunction with the Rn registers. They are split into two different fields: the STEP field and the MODULO field. The STEP field is used in linear and modulo modes, and the MODULO field is used only in modulo mode. Table 4.7 shows when these fields are valid in these modes.

Table 4.7 Validity of STEP and MODULO in Different Addressing Modes

Mode	STEP Field Valid	MODULO Field Valid
Linear	Yes	No
Modulo	Yes	Yes

See Section 2.5.2, “Addressing Modes,” for information on how the two fields affect the Rn register pointers. Figure 4.6 shows the CFGI Register bit fields, and Figure 4.7 shows the CFGJ Register bit fields. The CFGI and CFGJ registers clear to zero after reset.

Figure 4.6 CFGI Configuration Register

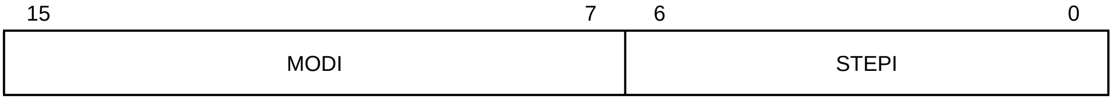
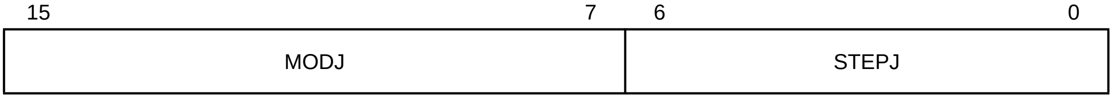


Figure 4.7 CFGJ Configuration Register



CFGI is associated with R0–R3, and CFGJ is associated with R4–R5. This enables simultaneous addressing over both the XAB and either the YAB or the IAB.

4.2.3 Base Register (RB)

The RB register is used in indexed addressing mode to define the base address of the data being addressed. See Section 2.5.2.4, “Indexed Addressing Mode,” for further information. RB can also be used as a 16-bit general-purpose register. The RB register clears to zero after reset.

4.2.4 Stack Pointer Register (SP)

The CWDSP1650 provides a software stack up to 64 Kwords in size. The 16-bit Stack Pointer (SP) is a global register that points to the address of the top value in the stack, which is also the last value pushed onto the stack. The stack fills from high-memory addresses to low-memory addresses. Values can be pushed to and popped from the stack using the PUSH and POP instructions. A POP instruction postincrements the SP; a PUSH instruction predecrements the SP. The Program Counter (PC) is automatically pushed onto the stack whenever a subroutine call or an interrupt occurs and popped off the Stack upon execution of return opcodes. The SP register clears to zero after reset.

To bypass the POP mechanism, execute the instruction `mov (sp), reg`, which causes memory to be read from an address pointed to by the SP. The MOV instruction reads the top of the stack without affecting the SP.

The software stack can reside anywhere in the data space (X-Memory or Y-Memory). The Rn and RB registers can also access the stack.

The stack generally is used for transferring parameters to subroutines and for automatic variables (such as local subroutine variables). Thus, after the initialization of the base register (RB) with the SP value, the MOV, ADD, SUB, CMP, AND, OR, and XOR instructions can typically directly access subroutine parameters using the indexed addressing mode.

4.2.5 Alternative Bank Registers

The DAAU contains an alternative bank of four registers: R0B, R1B, R4B, and CFGIB. These are the alternatives to the normal registers R0, R1, R4, and CFGI. Only one of each pair of normal and alternative registers is accessible at a time, with the bank selection made through the BANKE instruction. The alternative registers can be used as required by the programmer, typically for interrupt service routines. The BANKE instruction exchanges (swaps) the contents of the current register with the alternative register. The instruction includes a list of registers to be exchanged in a single cycle. Up to 4 registers can be included in the list. Table 4.8 shows the Address or Configuration Registers with their corresponding Alternative Bank Registers. The register R0B, R1B, R4B, and CFGIB clear to zero after reset.

Table 4.8 Alternative Bank Registers

Address /Configuration Register	Alternative Bank Register
R0	R0B
R1	R1B
R4	R4B
CFGI	CFGIB

4.2.6 Minimum/Maximum Pointer Latching Register (MIXP)

MIXP is a 16-bit register that is used to latch the value of the R0 register during MIN/MAX operations. Note that the MIXP register cannot be read in the instruction immediately following the MAX/MIN instruction. The MIXP register can also be used as a general-purpose 16-bit register and is cleared to zero after reset.

4.3 PCU Registers

The core uses the Program Control Unit Registers to control program flow, interrupts, loops, and to support on-chip emulation. See Section 2.3, "Program Control Unit (PCU)," for additional information about the PCU architechture. Table 4.9 lists all the PCU registers described in this section.

Table 4.9 PCU Registers

Register	Abbreviations	Page
Data Value Match Register	DVM	4-14
Internal Configuration Register	ICR	4-14
Program Counter	PC	4-15
Loop Counter	LC	4-15

4.3.1 Data Value Match Register

The 16-bit Data Value Match (DVM) Register supports the optional on-chip emulation module (OCEM), which resides off-core. The OCEM uses the DVM to generate a breakpoint on a data value match. A data value match occurs when the DVM register content is the same as the data on the EDB. The DVM register is on-core in order to enable comparisons for any transaction, since data is not always transferred off-core. The DVM register clears to zero after reset.

The DVM register is also used during the service of a software TRAP. The contents of the Program Counter (PC) is transferred to the DVM register and to the software stack. The DVM register contents can only be transferred through the Ax and Bx Accumulators.

4.3.2 Internal Configuration Register (ICR)

Figure 4.8 shows the format of Internal Configuration Register (ICR). The ICR includes the context switching bits and the block-repeat indication. A core reset clears all bits in the ICR to zero.

Figure 4.8 Internal Configuration Register (ICR)

15	8	7	6	5	4	3	2	1	0
RES		BC2	BC1	BC0	LP	IC2	IC1	IC0	NMIC

RES **Reserved** **[15:8]**
These bits are reserved for LSI Logic. These bits always read as zero and are not affected by a write.

BC[2:0] **Block Repeat Nesting Counter** **[7:5], R**
These bits hold the current block-repeat loop nesting level. The BCx bits are read only.

BC2	BC1	BC0	Block-Repeat Counter State Description
0	0	0	Not within a block-repeat loop
0	0	1	First block-repeat level (outer loop)
0	1	0	Second block-repeat level
0	1	1	Third block-repeat level
1	0	0	Fourth block-repeat level (inner loop)

The BCx bits clear when either the processor resets, or the LP bit clears to zero.

LP **INLOOP** **4, R/W**
LP is set when a block loop repeat occurs; otherwise, LP is cleared to zero. Clearing LP also clears the block repeat nesting counter bits (BCx).

Writing a zero to the LP bit has no effect. Writing a one to LP clears the bit to zero and also causes a break from all current levels of block repeat nesting. For more information on block repeats, see Section 2.3, “Program Control Unit (PCU).”

IC2 **INT2 Context Switching Enable** **3, R/W**
IC2 is the context switching enable for INT2. Setting IC2 enables context switching when an INT2 interrupt occurs. Clearing IC2 disables context switching for the INT2 interrupt.

IC1 **INT1 Context Switching Enable** **2, R/W**
IC1 is the context switching enable for INT1. Setting IC1 enables context switching when an INT1 interrupt occurs. Clearing IC1 disables context switching for the INT1 interrupt.

IC0 **INT0 Context Switching Enable** **1, R/W**
IC0 is the context switching enable for INT0. Setting IC0 enables context switching when an INT0 interrupt occurs. Clearing IC0 disables context switching for the INT0 interrupt.

NMIC **NMI Context Switching Enable** **0, R/W**
NMIC is the context switching enable for NMI. Setting NMIC enables context switching when an NMI interrupt occurs. Clearing NMIC disables context switching for the NMI interrupt.

4.3.3 Program and Loop Counters

The PC (Program Counter) and LC (Loop Counter) are directly accessible 16-bit counters. The PC always contains the address of the next instruction to be executed. The LC register can be used as an index inside the block-repeat loop, or for determining the value of the block-repeat counter when a jump out of the block-repeat loop occurs. The block-repeat Loop Counter is a global register and can also serve as a 16-bit general purpose register. PC and LC clear to zero after reset.

4.4 Status Registers

Three status registers hold the flags, status bits, control bits, user I/O bits, and paging bits for direct addressing. The contents of each register and their field definitions are described in the following subsections. Table 4.10 lists the Status Registers for the CWDSP1650.

Table 4.10 Status Registers

Register	Abbreviations	Page
Status Register 0	ST0	4-16
Status Register 1	ST1	4-18
Status Register 2	ST2	4-19

4.4.1 Status Register 0 (ST0)

Figure 4.9 shows the fields within Status Register 0. Each field is described below. The flags (Z, M, N, V, C, E, and L) indicate the result of the last ALU, BFO, or Barrel Shifter output operation. When one of these outputs is latched into a destination accumulator, the flags normally indicate that accumulator status. All bits in this register are read/write and clear during a processor reset.

Figure 4.9 Status Register 0 (ST0)

15	12	11	10	9	8	7	6	5	4	3	2	1	0
A0E		Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

A0E **Accumulator 0 Extension** **[15:12], R/W**
This field contains the contents of the Accumulator 0 Extension after an operation that used Accumulator 0 as a destination.

Z **Zero Flag** **11, R/W**
The Zero Flag is set if the ALU, BFO, or Barrel Shifter output used at the last instruction equals zero. The zero flag also indicates the result of the test bit instructions (TST0, TST1, or TSTB).

M **Minus Flag** **10, R/W**

The Minus Flag is set if the ALU, BFO, or Barrel Shifter output used at the last instruction is a negative number; and cleared otherwise. The minus flag is identical to the most-significant bit of the output (bit 35).

N **Normalized Flag** **9, R/W**

The Normalized Flag is set if the least-significant 32 bits of the ALU or Barrel Shifter output used at the last instruction are normalized; cleared otherwise. In other words, N is set if:

$Z \vee \{(\text{bit } 31 \oplus \text{bit } 30) \wedge \neg E\}$

V **Overflow Flag** **8, R/W**

The Overflow Flag is set if an arithmetic overflow (36-bit overflow) occurs after an arithmetic operation, and is cleared otherwise. The overflow flag indicates that the result of an operation cannot be represented in 36 bits.

C **Carry Flag** **7, R/W**

The Carry Flag is set if the result of an add generates a carry, or if the result of a subtract generates a borrow; it is cleared otherwise. The carry flag also reflects the rotated bit, or the last bit shifted out of the 36-bit result.

E **Extension Flag** **6, R/W**

The Extension Flag is set if bits [35:31] of the ALU or Barrel Shifter output used at the last instruction are not identical; otherwise it is cleared. Clearing the extension flag indicates that the four most-significant bits of the output are sign extensions of bit 31 and can be ignored.

L **Limit Flag** **5, R/W**

The Limit Flag has two functions: to latch the overflow flag, and to indicate limitation during accumulator move or LIM operations. The L bit is set if the overflow flag is set or if a limitation occurs either when an accumulator move instruction (MOV or PUSH) through the data bus is used, or when a limitation occurs when the LIM instruction executes. Otherwise, L is not affected.

R **Rn Register is Zero Flag** **4, R/W**

The R flag is set if the result of an Rn modification operation (Rn; Rn+1; Rn-1; Rn+S) is zero. Only the MODR and NORM instructions affect this flag. The R flag status remains unchanged until one of the above instructions is used.

IM1 **Interrupt 1 Mask** **3, R/W**
IM1 is the interrupt mask for INT1. Clearing IM1 to zero disables the interrupt, setting IM1 to one enables the interrupt.

IM0 **Interrupt 0 Mask** **2, R/W**
IM0 is the interrupt mask for INT0. Clearing IM0 to zero disables the interrupt, setting IM0 to one enables the interrupt.

IE **Interrupt Enable** **1, R/W**
Clearing IE to zero disables all maskable interrupts, setting IE to one enables all maskable interrupts. To modify the interrupt enable bit, use the instructions EINT (enable interrupts) or DINT (disable interrupts).

SAT **Saturation Mode Disable** **0, R/W**
Clearing SAT to zero enables saturation mode when the contents of the accumulator are transferred to the data bus. Setting SAT to one disables the saturation mode.

Note: The SAT bit values have no affect on the LIM instruction operation.

4.4.2 Status Register 1 (ST1)

Figure 4.10 shows the fields within Status Register 1. Each bit field of ST1 is described below. All bits clear to zero after reset.

Figure 4.10 Status Register 1 (ST1)

15 12	11 10	9 8	7 0
A1E	PS	RES	PAGE

A1E **Accumulator 1 Extension** **[15:12], R/W**
This field contains the contents of the Accumulator 1 Extension after an operation that used Accumulator 1 as a destination.

PS **Product Shifter Control** **[11:10], R/W**
The PS bits determine the scaling shift of the P register output. Writing to the ST1 Register modifies the PS bits.

The PS bits control whether the product is shifted right by one, left by one, left by two, or is not shifted.

PS		Number of Shifts
0	0	No shift
0	1	Shift right by one
1	0	Shift left by one
1	1	Shift left by two

RES **Reserved** **[9:8]**
These bits are reserved for LSI Logic. These bits always read as ones and are not affected when written.

PAGE **Data Memory Space Page** **[7:0], R/W**
In direct address mode, these bits address the data memory page. Pages are addressed in 128-byte increments. See Section 3.3.1.1, “Short Direct Addressing,” for more information on direct addressing.

To modify these bits, either write to the ST1 Register, use the LOAD instruction, or use the LPG instruction.

4.4.3 Status Register 2 (ST2)

Figure 4.11 shows the fields within Status Register 2. Each field is described below.

Figure 4.11 Status Register 2 (ST2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IP1	IP0	IP2	RES	IU1	IU0	OU1	OU0	S	IM2	M5	M4	M3	M2	M1	M0

IP1 **Interrupt Pending 1** **15, R**
IP1 is the interrupt pending bit for the INT1 interrupt. IP1 is set when the INT1 interrupt is active. IP1 reflects the interrupt level regardless of the mask bit.

IP0 **Interrupt Pending 0** **14, R**
IP0 is the interrupt pending bit for the INT0 interrupt. IP0 is set when the INT0 interrupt is active. IP0 reflects the interrupt level regardless of the mask bit.

IP2 **Interrupt Pending 2** **13, R**

IP2 is the interrupt pending bit for the INT2 interrupt. IP2 is set when the INT2 interrupt is active. IP2 reflects the interrupt level regardless of the mask bit.

RES **Reserved** **12**

This bit is reserved for LSI Logic. It reads as one and is not affected when written.

IU1 **User-Defined Input 1** **11, R**

IU1 reflects the state of the core input User-Defined Input 1.

IU0 **User-Defined Input 0** **10, R**

IU0 reflects the state of the core input User-Defined Input 0.

OU1 **User-Defined Output 1** **9, R/W**

OU1 determines the state of the core output User-Defined Output 1. Processor reset clears OU1 to zero.

OU0 **User-Defined Output 0** **8, R/W**

OU0 determines the state of the core output User-Defined Output 0. Processor reset clears OU0 to zero.

S **Shift Mode** **7, R/W**

This bit determines the shift method for all of the shift instructions: SHFC, SHFI, MODA, MODB, MOVS, and MOVSI. If S is cleared to zero, the shift instruction is an arithmetic shift. If S is set to one, the shift instruction is a logical shift.

S	Shift Instruction
0	Arithmetic Shift
1	Logical Shift

Processor reset clears the shift mode bit to zero.

IM2 **Interrupt 2 Mask** **6, R/W**

IM2 is the interrupt mask for the INT2 signal. Clearing IM2 to zero disables INT2; setting IM2 to one enables INT2. This bit is cleared to zero during processor reset.

Mn **Modulo Enable** **[5:0], R/W**

Setting an Mn bit to one enables modulo addressing for the corresponding Rn register, as specified by the MOD and STEP values in the relevant CFG register. Processor reset clears the Mn bits.

Modulo Enable Bit	Address Register
M5	R5
M4	R4
M3	R3
M2	R2
M1	R1
M0	R0

Note: The MODR instruction is the only instruction that can use one of the Rn registers without being affected by the corresponding Mn bit. The MODR instruction has an option that disables modulo operation. See Chapter 7, “Instruction Set,” for more information.

4.5 User-Defined Registers

The CWDSP1650 supports four User-Defined Registers that enable expansion of the core in off-core glue logic. The 16-bit User-Defined Registers are part of the core register list, which means they can be accessed by most of the CWDSP1650 instructions. The core does not provide a direct mechanism to clear the user-defined registers after reset.

Chapter 5
Signals

This chapter describes the CWDSP1650 interfaces to logic external to the core. It contains the following subsections:

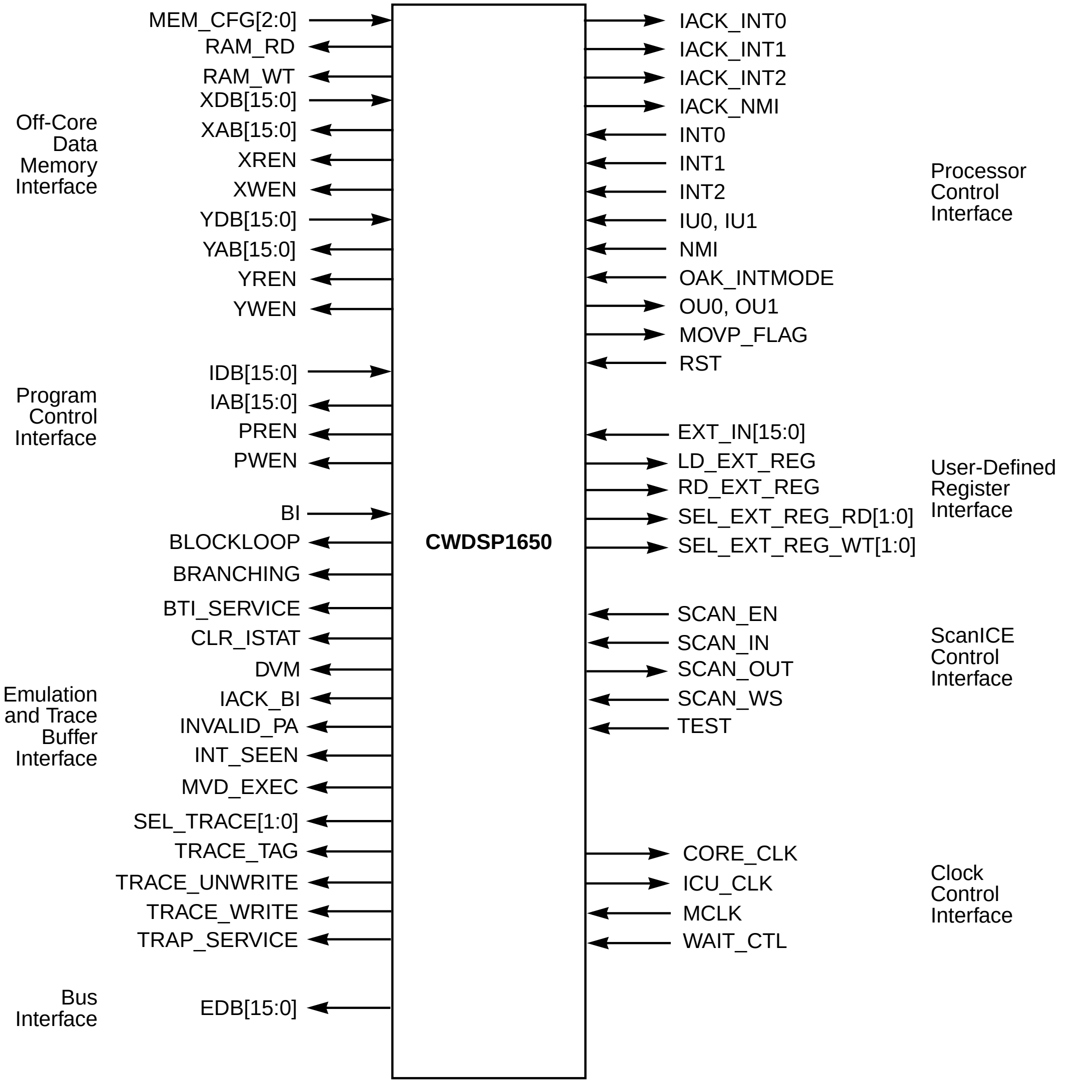
- Section 5.1, “Logic Symbol”
- Section 5.2, “Bus Interface”
- Section 5.3, “Program Control Interface”
- Section 5.4, “Off-Core Data Memory Interface”
- Section 5.5, “User-Defined Register Interface”
- Section 5.6, “Emulation and Trace Buffer Interface”
- Section 5.7, “Processor Control Interface”
- Section 5.8, “ScanICE Control Interface”
- Section 5.9, “Clock Control Interface”

In the descriptions that follow, the verb assert means to drive TRUE or active. The verb deassert means to drive FALSE or inactive.

5.1 Logic Symbol

Figure 5.1 shows the logic symbol for the CWDSP1650 DSP core.

Figure 5.1 CWDSP1650 Logic Symbol



5.2 Bus Interface

The EDB comprises the main bus for most data transactions.

EDB[15:0] **External Data Bus** **Output**
This unidirectional 16-bit data bus transfers data from the internal core components (PCU, DAAU, and so on) to the off-core components (YRAM, XRAM, PRAM, external registers, etc.) EDB[15:0] is the external extension of the Main Data Bus.

5.3 Program Control Interface

The following signals relate to program memory access.

IDB[15:0] **Program Data Bus** **Input**
This 16-bit bus transfers both program instructions and program data to the core from the program memory.

IAB[15:0] **Program Address Bus** **Output**
The core drives this 16-bit bus with the memory address of either the program instruction or the program data.

PREN **Program Read Enable** **Output**
The core drives this signal HIGH to request program data. PREN remains active through the complete program request cycle, including all wait cycles.

PWEN **Program Write Enable** **Output**
The core drives this signal HIGH to indicate either a program word or a data word write to the program memory space through the external data bus. PWEN remains active through the complete program write cycle, including all wait cycles.

5.4 Off-Core Data Memory Interface

The following signals are used in data memory access.

MEM_CFG[2:0]

Memory Configuration **Input**

These signals determine memory mapping for the core system. The memory configuration can range from an equal mix of 32 Kword X-memory / 32 Kword Y-memory, to a 63.75 Kword X-memory / 256 word Y-memory distribution.

MEM_CFG [2:0]	**X-Memory Space**		**Y-Memory Space**	
000	32 K	(0x0000-0x7FFF)	32 K	(0x8000-0xFFFF)
001	48 K	(0x0000-0xBFFF)	16 K	(0xC000-0xFFFF)
010	56 K	(0x0000-0xDFFF)	8 K	(0xE000-0xFFFF)
011	60 K	(0x0000-0xEFFF)	4 K	(0xF000-0xFFFF)
100	62 K	(0x0000-0xF7FF)	2 K	(0xF800-0xFFFF)
101	63 K	(0x0000-0xFBFF)	1 K	(0xFC00-0xFFFF)
110	63.5 K	(0x0000-0xFDFF)	512	(0xFE00-0xFFFF)
111	63.75 K	(0x0000-0xFEFF)	256	(0xFF00-0xFFFF)

RAM_RD **Memory Read Indicator** **Output**

The core drives this signal HIGH to indicate that the instruction about to be decoded will request a read from either the X- or Y-memory data. When a data memory read is in the critical path, RAM_RD can provide a designer with early indication of a memory access.

RAM_WT **Memory Write Indicator** **Output**

The core drives this signal HIGH to indicate that the instruction about to be decoded will request a write to either the X- or Y-memory data. When a data memory write is in the critical path, RAM_RD can provide a designer with early indication of a memory access.

XDB[15:0] **X-Memory Data Bus** **Input**

This unidirectional 16-bit bus transfers data to the core from the X-memory space.

XAB[15:0] **X-Memory Address Bus** **Output**
The core drives this 16-bit bus with the X-memory space address. The available X-memory address range is specified by the MEM_CFG[2:0] signals.

XREN **X-Memory Read Enable** **Output**
The core drives this signal HIGH to request data from the X-memory address specified by the XAB signal. XREN remains active throughout the complete X-memory request cycle, including all wait cycles.

XWEN **X-Memory Write Enable** **Output**
The core drives this signal HIGH to indicate a X-memory data write through the external data bus to the address specified by XAB. XWEN remains active through the complete X-memory write cycle, including all wait cycles.

YDB[15:0] **Y-Memory Data Bus** **Input**
This unidirectional 16-bit bus transfers data to the core from the Y-memory space.

YAB[15:0] **Y-Memory Address Bus** **Output**
The core drives this 16-bit bus with the Y-memory address. The available Y-memory address range is specified by the MEM_CFG[2:0] signals.

YREN **Y-Memory Read Enable** **Output**
The core drives this signal HIGH to request Y-memory data from the address specified by the YAB signal. YREN remains active throughout the complete Y-memory read cycle, including all wait cycles.

YWEN **Y-Memory Write Enable** **Output**
The core drives this signal HIGH to indicate a Y-memory data write through the external data bus to the address specified by YAB. YWEN remains active through the complete Y-memory write cycle, including all wait cycles.

5.5 User-Defined Register Interface

The following signals are used when implementing the optional user-defined registers.

EXT_IN[15:0]

External Registers Input Data Bus **Input**

This 16-bit bus transfers data from the user-defined external registers to the core.

LD_EXT_REG

External Register Write Enable **Output**

The core drives this signal HIGH during an external register write cycle.

RD_EXT_REG

External Register Read Enable **Output**

The core drives this signal HIGH during an external register read cycle.

SEL_EXT_REG_RD[1:0]

Select External Register for Reading **Output**

The core drives these signals with the code of the external register that is the source operand of the external register read instruction.

SEL_EXT_REG_RD[1:0]	External Register
00	0
01	1
10	2
11	3

SEL_EXT_REG_WT[1:0]

Select External Register for Writing **Output**

The core drives these signals with the code of the external register that is the destination operand of the external register write instruction.

SEL_EXT_REG_WT[1:0]	External Register
00	0
01	1
10	2
11	3

5.6 Emulation and Trace Buffer Interface

The following signals are used in conjunction with the on-chip emulation module (OCEM) to implement debug functionality. See Chapter 8, "On-Chip Emulation Module (OCEM)," for further information on the OCEM.

BI **Breakpoint Interrupt** **Input**
Asserting this signal HIGH causes the core to service the TRAP/BI interrupt service routine (vector address 0x0002.)

BLOCKLOOP **Block-Repeat Detected** **Output**
The core drives this signal HIGH whenever it detects a jump back to the first instruction in a block-repeat loop.

BRANCHING **Branch Detected** **Output**
The core drives this signal HIGH whenever a branch-type instruction occurs during program execution (see Chapter 8, "On-Chip Emulation Module (OCEM)," for further information).

BTI_SERVICE

BI/TRAP Service Active **Output**
The core asserts this signal HIGH to indicate the execution of a TRAP/BI service routine. BTI_SERVICE remains active from the decode of address 0x0002 until the third cycle decode of either the RETID or RETI instruction at the end of the service routine.

CLR_ISTAT **Clear Interrupt Status** **Output**
The core asserts this signal HIGH when it serves an interrupt.

DVM **Data Value Match** **Output**
The core asserts this signal HIGH to indicate a match between the data of the DVM register and the value of EDB[15:0].

IACK_BI **Breakpoint Interrupt Acknowledge** **Output**
The core asserts this signal HIGH to acknowledge a breakpoint interrupt.

INVALID_PA **Invalid Program Address** **Output**
The core asserts this signal HIGH whenever either an invalid data read or data write occurs with the program memory space.

INT_SEEN **Interrupt Indication** **Output**
The core asserts this signal HIGH when an interrupt is pending, or if the core is in a noninterruptible state.

MVD_EXEC **Move Data-to-Program Detected** **Output**
The core asserts this signal HIGH after it executes a MOVD instruction. This operation could corrupt the program memory, so the OCEM monitors this condition.

SEL_TRACE[1:0]
Select Addresses for Trace **Output**
The core drives these signals with the program address values selected to be written to the OCEM trace buffer.

TRACE_TAG **Trace Address Tag** **Output**
Asserting this signal HIGH indicates that the current program address is one of two addresses stored for certain nonsequential program flow operations. The core stores the value of TRACE_TAG with the trace address values in the trace buffer.

TRACE_UNWRITE
Unwrite Last Trace Address **Output**
The core drives this signal HIGH to unwrite the last trace address from the trace buffer. An unwrite capability is needed to erase a conditional branch trace buffer entry when the branch is not taken.

TRACE_WRITE
Trace Write **Output**
The core drives this signal HIGH to indicate a data write to the trace buffer.

TRAP_SERVICE
TRAP Service Indicator **Output**
The core drives this signal HIGH when a software trap occurs.

5.7 Processor Control Interface

The following signals relate to program flow control, memory access cycle definition and interrupts:

IACK_INT0 **Maskable Interrupt 0 Acknowledge** **Output**
The core drives this signal HIGH to acknowledge service to maskable interrupt 0.

IACK_INT1 **Maskable Interrupt 1 Acknowledge** **Output**
The core drives this signal HIGH to acknowledge service to maskable interrupt 1.

IACK_INT2 **Maskable Interrupt 2 Acknowledge** **Output**
The core drives this signal HIGH to acknowledge service to maskable interrupt 2.

IACK_NMI **Nonmaskable Interrupt Acknowledge** **Output**
The core drives this signal HIGH to acknowledge service to the nonmaskable interrupt.

INT0 **Maskable Interrupt 0 Request** **Input**
Asserting this signal HIGH through external logic requests a core interrupt. This interrupt causes the core to serve the INT0 interrupt service routine (vector address 0x0006), and can be internally masked by software. INT0 must be synchronized with the rising edge of the main input clock.

INT1 **Maskable Interrupt 1 Request** **Input**
Asserting this signal HIGH through external logic requests a core interrupt. This interrupt causes the core to serve the INT1 interrupt service (vector address 0x000E), and can be internally masked by software. INT1 must be synchronized with the rising edge of the main input clock.

INT2 **Maskable Interrupt 2 Request** **Input**
Asserting this signal HIGH through external logic requests a core interrupt. This interrupt causes the core to serve the INT2 interrupt service (vector address 0x0016), and can be internally masked by software. INT2 must be synchronized with the rising edge of the main input clock.

IU1, IU0 **User Input Pins** **Input**
These signals are discrete inputs to the core and can be read by software as bits of Status Register 2.

MOVP_FLAG **Privacy for the Program Code** **Output**
The core asserts this signal HIGH whenever it executes a MOVP instruction. An external memory controller can protect the on-chip program from unauthorized external reads by monitoring MOVP_FLAG. When MOVP_FLAG is asserted, the external memory controller checks if the access is authorized. If not, the controller disables that transaction enable, thereby protecting the program memory information.

NMI **Nonmaskable Interrupt Request** **Input**
Asserting this signal HIGH through external logic requests a core interrupt. This interrupt causes the core to serve the NMI service routine (vector address 0x0004), and cannot be internally masked by software. NMI must be synchronized with the rising edge of the main input clock.

OAK_INTMODE
Interrupt Mode **Input**
This signal has been reserved for future enhancement. During CWDSP1650 operation, hold the OAK_INTMODE signal LOW at all times.

OU1, OU0 **User Output Pins** **Output**
These discrete control signals can be controlled by software.

RST **Core Reset** **Input**
Asserting this signal HIGH resets the core.

5.8 ScanICE Control Interface

The following signals are used in CWDSP1650 systems which implement the ScanICE debug system.

SCAN_EN **Scan Enable** **Input**
Asserting this signal HIGH configures internal registers for a serial scan.

SCAN_IN **Scan Chain Input** **Input**
This signal connects directly to the test input of the first flip-flop in the kernel scan chain.

SCAN_OUT **Scan Chain Output** **Output**
This signal connects directly to the output of the last flip-flop in the kernel scan-chain.

SCAN_WS **Scan Write Strobe** **Input**
Asserting this signal HIGH disables memory write enables from the core. Disabling memory writes protects the contents of the off-core RAMs during the scan mode.

TEST **Test Mode** **Input**
When SCAN_EN is LOW, asserting TEST forces all kernel registers to be load enabled, implementing a capture function when in scan mode. Asserting TEST also forces the multiplier input latches transparent so they can be controlled when in scan mode. TEST should be asserted whenever SCAN_EN is asserted.

5.9 Clock Control Interface

These signals regulate the CWDSP1650 core clocking system.

CORE_CLK **Main Clock** **Output**
Main clock that drives all CWDSP1650 modules, apart from the ICU.

ICU_CLK **Interrupt Control Unit Clock** **Output**
This signal is the controlling clock for the ICU.

MCLK **Main Input Clock** **Input**
This clock is the main input clock.

WAIT_CTL **Wait Request** **Input**
Asserting this signal HIGH forces the core in a wait state mode. While WAIT_CTL is asserted, the core clock (CORE_CLK) is held LOW. Deasserting WAIT_CTL LOW for one full clock cycle ends this wait state mode. The core clock then begins generating clock pulses again at the rising edge trigger and the core can resume the current transaction.

Chapter 6
Operation

This chapter contains waveforms that depict core signals during various operations. These waveforms are intended to assist hardware designers in understanding the inter-relations of the core signals and how operations initiated on one interface propagate to others.

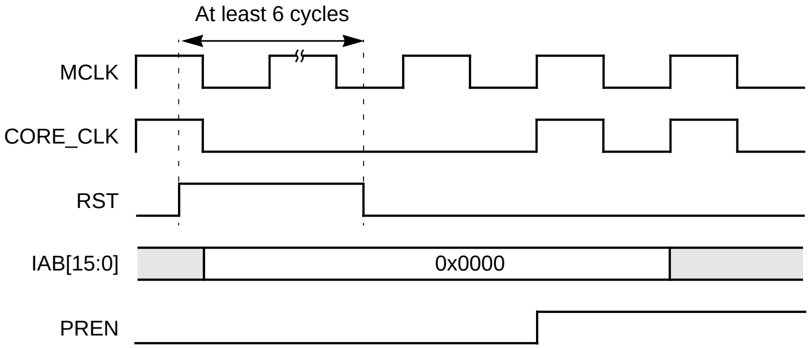
This chapter is further divided into the following sections:

- Section 6.1, “Reset”
- Section 6.2, “Boot Procedure”
- Section 6.3, “Interrupts”
- Section 6.4, “Memory Interface”
- Section 6.5, “User-Defined Register Interface”
- Section 6.6, “Program Protection Mechanism”
- Section 6.7, “Clock Control Unit (CCU)”

6.1 Reset

Reset is an external event used at any time to put the CWDSP1650 core into a known state. Asserting the reset input (RST) HIGH places the core into the reset state. The reset state causes the core to terminate all code execution and initialize (clear) all registers and status bits. Deasserting RST causes program execution to start from location 0x0000. Figure 6.1 shows a waveform for a reset operation.

Figure 6.1 Reset Operation



RST must be held asserted for at least six MCLK cycles. During reset, the core drives the program address bus (IAB) with all zeros and deasserts the program read enable (PREN). After the core deasserts RST, it starts fetching instructions from program address 0x0000 on the next rising edge of CORE_CLK. Since the CWDSP1650 CCU synchronizes RST internally with MCLK, MCLK must not stop while RST is asserted. For details on the effect of RST on specific registers, see the descriptions in Chapter 4, "Registers."

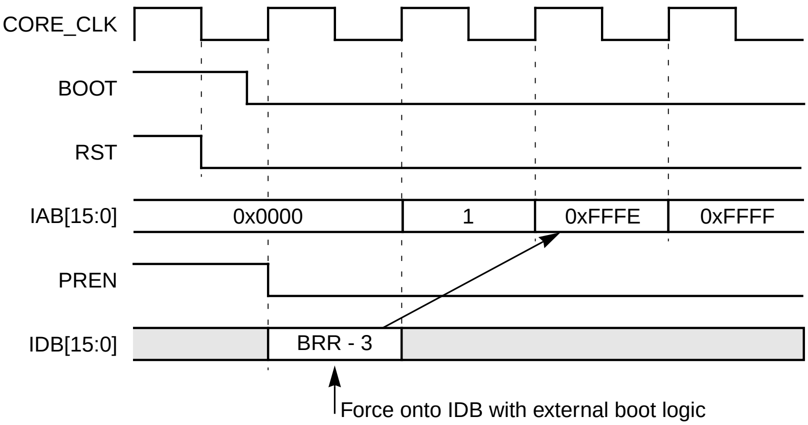
6.2 Boot Procedure

The boot procedure allows automatic code downloading from an external device (for example, EPROM or RAM) to a RAM device located in the program space. The boot procedure is not included in the CWDSP1650 core, but is implemented in the off-core Bus Interface Unit (BIU).

Boot support is transparent to the CWDSP1650 core. An off-core device may force a branch instruction onto the program data bus (IDB) after reset to direct the core to a boot routine. The core fetches this branch instruction and executes it as if it was fetched from memory. Subsequently, the core executes the boot routine to load the program code to the program memory. A boot routine should end by jumping to the program it downloaded.

As an example, an off-core Bus Interface Unit (BIU) on the CWDSP1650 Reference Device samples a BOOT pin during deassertion of RST. If this off-core BOOT pin is valid, the BIU forces the instruction `brr #-3` onto the IDB through a multiplexor. This causes the next instruction to be fetched from address 0xFFFE, the location where the boot routine is held in memory. Inside the boot routine, the instruction movd can be used to download program code from a data memory (slow EPROM or ROM) to a RAM device within the program space. Figure 6.2 illustrates how to initiate the CWDSP1650 boot procedure.

Figure 6.2 Entering Boot Mode



6.3 Interrupts

The CWDSP1650 core supports six different interrupts to handle exceptions:

- Three hardware maskable interrupts (INT0, INT1, INT2)
- One nonmaskable interrupt (NMI)
- Two breakpoint interrupts (software/hardware) in one signal (BTI_SERVICE) for use with the OCEM

The core can either program the registers ST0 and ST2 or use the EINT and DINT instructions to enable or disable maskable interrupts. An off-core device can use any of the hardware interrupts to request services from the core. A program can request system service using the TRAP instruction.

After accepting an interrupt, the core suspends normal program execution and calls a service routine to handle the exception. Each service routine starts from an address, known as the interrupt vector, dedicated to that specific interrupt. The TRAP/BI interrupts share the same interrupt vector. The core acknowledges each hardware interrupt using a dedicated acknowledgment signal while it services that interrupt. After the exception has been handled, an interrupt service routine returns control to the interrupted program, which resumes execution from the suspended point of the program. Note that the BI/TRAP vector is used by the on-chip emulation module (OCEM) when using either the CDI or ScanICE debug.

The remainder of this section is further divided into the following subsections:

- Section 6.3.1, “Maskable Interrupts”
- Section 6.3.2, “Nonmaskable Interrupt (NMI)”
- Section 6.3.3, “TRAP/BI Interrupts”
- Section 6.3.4, “Interrupt Protocol”
- Section 6.3.5, “Interrupt Priority”
- Section 6.3.6, “Context Switching”
- Section 6.3.7, “Interrupt Nesting”
- Section 6.3.8, “Interruptible State”

6.3.1 Maskable Interrupts

INT0, INT1, and INT2 are active HIGH maskable interrupts. The core processes a maskable interrupt provided the following conditions are true:

- CWDSP1650 is in an interruptible state
- NMI or BTI_SERVICE interrupts are not pending or currently being serviced
- Interrupt enable bit (IE) in the ST0 register is set to one
- Corresponding interrupt mask bit (IMn) in the ST0 and ST2 registers is set to one

Table 6.1 lists the bits and signals associated with the maskable interrupts.

Table 6.1 Maskable Interrupt Bits and Signals

Interrupt Signal	Interrupt Acknowledge Signal	Interrupt Enable Bit	Interrupt Mask Bit	Interrupt Pending Bit
INT0	IACK_INT0	IE (ST0)	IM0 (ST0)	IP0 (ST2)
INT1	IACK_INT1	IE (ST0)	IM1 (ST0)	IP1 (ST2)
INT2	IACK_INT2	IE (ST0)	IM2 (ST2)	IP2 (ST2)

Each maskable interrupt is masked independently by clearing to zero one of the mask bits IM0, IM1, or IM2. Clearing the IE bit in the ST0 register disables all maskable interrupts. Always use the instruction EINT and DINT to manipulate the IE bit when enabling and disabling maskable interrupts explicitly. When the core services a maskable interrupt, it clears the IE bit to disable all other maskable interrupts. The interrupt mask bits are unaffected. Pending maskable interrupts are serviced only after the IE bit is set to one. The core acknowledges the pending maskable interrupt through one of the IACK_INTn signals.

To return from a maskable interrupt service routine, one of the four instructions RETI, RETID, RET, or RETD can be used. The instructions RETI and RETID set the IE bit, allowing pending maskable interrupts to be serviced. The RET and RETD instructions do not affect the IE bit.

Interrupt priority arbitrates between simultaneous maskable interrupts. Among the maskable interrupts, INT0 has the highest priority and INT2 has the lowest. The priority between INT0, INT1, and INT2 is significant only if more than one interrupt is received at the same time and the IE bit is set to one. In these cases, the core services the interrupts according to their priorities.

Nesting of maskable interrupts is supported only if enabled by the service routine. When enabled, any maskable interrupt can interrupt the service routine. Interrupt priority does not affect nesting of maskable interrupts.

An NMI or TRAP/BI interrupt can always interrupt a maskable interrupt service routine, regardless of the IE bit. The core sets the IP0, IP1, or

IP2 bit in ST2 register once the corresponding interrupt INT0, INT1, or INT2 has been asserted, even when they are masked and/or disabled. These bits can be used to poll the interrupt status.

6.3.2 Nonmaskable Interrupt (NMI)

NMI is a nonmaskable interrupt that cannot be masked or disabled under software control. The core accepts NMI when it is in an interruptible state and is not already servicing either an NMI or a TRAP/BI request. When servicing an NMI, the core acknowledges the interrupt with the IACK_NMI signal.

While the core is servicing an NMI, all incoming maskable interrupts are held pending. Only a TRAP/BI request can interrupt an NMI service routine. An NMI service routine must return with a RETI or RETID instruction to clear the interrupt logic internal to the core. The IE bit in status register ST0 is not affected by servicing an NMI interrupt.

6.3.3 TRAP/BI Interrupts

TRAP is a software interrupt and is the only interrupt that can be activated directly by a software instruction. BI is a hardware breakpoint interrupt that activates when the BI core input is asserted. The OCEM uses the BI interrupt to provide emulation capability within the core. The core handles the BI and TRAP interrupts almost identically and both share the same interrupt vector. The core takes the following actions in addition to the normal interrupt protocol when servicing a BI/TRAP interrupt:

- The program counter is stored into the Data Value Match Register (PC → DVM.)
- The core asserts the Trap/BI Active Indicator (BTI_SERVICE) HIGH throughout the service routine.
- For TRAP interrupt, the core asserts the Software Trap Indicator (TRAP_SERVICE) HIGH throughout the service routine to indicate that a software trap occurred.

The core cannot disable or mask a TRAP/BI interrupt under software control. It always services a requested TRAP/BI interrupt when it is in an interruptible state and is not already servicing this interrupt. While the core is executing the TRAP/BI service routine, it disables servicing of all other hardware interrupts (NMI, INT0, INT1, and INT2).

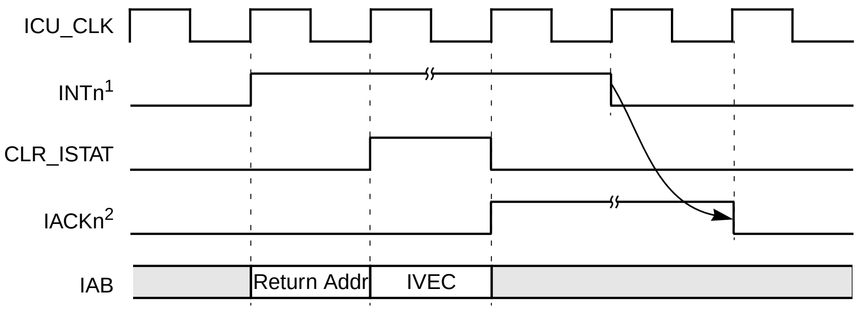
Nesting of BI/TRAP interrupts is not allowed; it is illegal to use a TRAP instruction in a TRAP/BI service routine. A TRAP/BI service routine must end with an RETI or RETID instruction to clear the interrupt logic internal to the core. The IE bit in status register ST0 is not affected by servicing a TRAP/BI interrupt.

The TRAP_SERVICE output signal differentiates between a TRAP (asserted HIGH) or a BI (deasserted LOW) interrupt. If required, a service routine can differentiate between a TRAP and a BI interrupt only with the assistance of external glue logic that latches the signal. For more information on the TRAP or BI service see the Section 8.3, “OCEM Signals.”

6.3.4 Interrupt Protocol

The CWDSP1650 defines an interrupt protocol to avoid race hazards due to the asynchronous nature of external interrupts. Figure 6.3 illustrates this interrupt protocol.

Figure 6.3 Interrupt Protocol



1. INTn can be INT2, INT1, INT0, NMI, or BI.
2. IACKn can be IACK, INT2, IACK_INT1, IACK_INT0, IACK_NMI or IACK_BI.

The core registers interrupts internally on the rising edge of ICU_CLK. The maskable interrupts (INT0, INT1, and INT2) are registered even when disabled, but are not registered when masked. See Section 6.3.1, “Maskable Interrupts,” for further details of masking and disabling the maskable interrupts. A registered interrupt is cleared only when the core services the interrupt or during a core reset.

An on-core priority encoder selects the highest priority interrupt to be serviced when multiple interrupts simultaneously arrive. A lower priority interrupt is kept pending and is serviced only when all higher priority interrupts have been serviced. See Section 6.3.5, "Interrupt Priority," for more information about interrupt priority servicing.

When the core is in an interruptible state, it starts servicing the highest priority registered interrupt at the start of an instruction decode cycle (see Section 6.3.8, "Interruptible State," for further details). Instead of decoding the prefetched instruction, the core passes a pseudo-instruction to the instruction decoder to direct the core to the appropriate service routine. As shown in Figure 6.3, the core performs the following operations on the decode cycle of the pseudo-instruction:

- The vector address of the interrupt being serviced is forced onto IAB to initiate prefetching of the first instruction in the corresponding service routine (IVEC → IAB.)
- Assert CLR_ISTAT for one cycle to clear the internal finite state machine inside the core (CLR_ISTAT HIGH.)
- Complete execution of the instruction already decoded and in the operand fetch stage of the processor pipeline.
- Save the return address by pushing the address of the last prefetched instruction onto the system software stack (PC → (SP), then SP - 1 → SP.)

During the execution cycle of the interrupt pseudo-instruction, the core asserts the corresponding acknowledgment signal (IACKn). IACKn clears only after the corresponding interrupt is deasserted. Hence, a hardware interrupt signal should be held until the core acknowledges it (IACKn HIGH.)

> Note: When servicing a hardware interrupt, the core clears its internal interrupt logic only when the interrupt is deasserted. To comply with the interrupt protocol, a hardware interrupt request should be held asserted until the core acknowledges it. After that, the request must be deasserted before another request is raised on the same interrupt input. Otherwise, the core ignores this constantly asserted interrupt after servicing it once.

6.3.5 Interrupt Priority

The CWDSP1650 core prioritizes interrupts when more than one interrupt request is simultaneously raised. Table 6.2 lists all the interrupts supported by the core in the descending order of their priorities. Their vector addresses and acknowledgment signals are also listed in the table.

Table 6.2 Interrupts and Priorities

Interrupt	Acknowledge	Vector	Priority
TRAP/BI	IACK_BI	0x0002	1
NMI	IACK_NMI	0x0004	2
INT0	IACK_INT0	0x0006	3
INT1	IACK_INT1	0x000E	4
INT2	IACK_INT2	0x0016	5

The core services the highest priority interrupt it registered while the others are held pending. The core services an interrupt only when no higher priority interrupt needs attention. A higher priority interrupt that arrives late is serviced first even if a lower priority interrupt has already been held pending but not yet serviced.

6.3.6 Context Switching

When servicing an interrupt, the service routine has to save the contents of any registers it uses before writing to them. Before returning, the service routine should restore these registers with the saved values so that the interrupted program can resume in the correct context. To reduce the overhead involved, the CWDSP1650 supports automatic context switching for the NMI and INTn interrupts. Automatic context switching on an interrupt is enabled by setting the NMIC and ICn bits in the ICR register. See Section 4.3.2, "Internal Configuration Register (ICR)," for more information. When enabled, the core performs the following operations automatically with zero overhead before executing the interrupt service routine:

- Push ST0[0], ST0[11:2], ST1[11:10], and ST2[7:0] to their shadow registers. The saved status register bits preserve the interrupted program status.
- Swap the PAGE bits of the ST1 register with its shadow register. The swapped PAGE bits allow a service routine to have instant access to a data memory page reserved for interrupt servicing.
- Swap the A1 and B1 accumulators. This makes it convenient to reserve B1 for interrupt servicing.

Automatic restoration of the switched context can be performed when returning from the interrupt service routine using the RETI instruction. An interrupt service routine has to explicitly save and restore any register it uses not preserved by the above operations. The context switching mechanism can also be activated by executing the CNTX instruction. The shadow PAGE bits are not directly accessible by software, but have to be explicitly selected with the CNTX instruction. See Section 4.1.3, "Interrupt Context Switching Registers," for more information about context switching.

BANKE is another useful instruction for context saving and restoration. It can swap up to four of the following pairs of registers in one cycle:

- R0↔R0B
- R1↔R1B
- R4↔R4B
- CFGI↔CFGIB

For more detailed information about the RETI, CNTX, or BANKE instructions, see Chapter 7, "Instruction Set."

6.3.7 Interrupt Nesting

The CWDSP1650 also supports interrupt nesting. In general, an interrupt service routine can be interrupted only by a higher priority interrupt. The lower priority interrupt service routine resumes execution after the higher priority interrupt has been serviced. When servicing a maskable interrupt (INTn), the core clears the IE bit in status register ST0 to disable servicing any further maskable interrupts. A RETI or RETID instruction at the end of a maskable interrupt service routine sets the IE bit to re-enable the maskable interrupts. Nesting maskable interrupts can be enabled inside a maskable interrupt service using the EINT instruction to explicitly set the IE bit to one. When the IE bit is set, a maskable interrupt can interrupt any maskable interrupt service routine. The relative priorities of the maskable interrupts do not affect the nesting of any maskable interrupts.

6.3.8 Interruptible State

While the core is in a noninterruptible state, it will not service any of the interrupts, regardless of their priority. All interrupts received while the core is in a noninterruptible state are held pending. The core is in a noninterruptible state during:

- Multicycle instruction executions
- Wait states
- A nested repeat loop execution

During reset, the core ignores any interrupt. If an interrupt is asserted when RST deasserts, the core services the interrupt after prefetching from location 0x0000.

The core also becomes noninterruptible temporarily after executing some instructions. These cause a delay in servicing an interrupt. Table 6.3 describes all cases in which interrupts are delayed due to a specific instruction execution. The second column lists the interrupt delays in machine cycles (CORE_CLK cycles). CORE_CLK can stretch to multiple-cycles in case of wait states (it stretches until the end of the wait interval.)

Table 6.3 Interrupt Latency after Specific Instructions

Current Instruction	Interrupt Delay after the Instruction	Interrupt[1]
First repetition of instruction during a repeat loop after returning from an interrupt	One cycle	INTn, NMI

Enabling or unmasking interrupts with the following instructions:
`mov soperand, st0`
`movp (aXl), st0`
`set/rst/chng/addv/subv/mov ##long immediate, st0`
`pop st0`
	One cycle	INT0, INT1

Unmasking interrupt with the following instructions: `mov soperand, st2`
`movp (aXl), st2`
`set/rst/chng/addv/subv/mov ##long immediate, st2`
`pop st2`
	One cycle	INT2

`retd, retid, mov soperand, pc`
`movp (aXl), pc`
	Two cycles	INTn, NMI, BI

`mov ##long immediate, pc`	One cycle	INTn, NMI, BI

1. INTn = INT0, INT1, INT2.
2. soperand represents every source operand except for a ##long immediate

6.4 Memory Interface

The CWDSP1650 Harvard architecture separates program memory and data memory spaces. The core further divides the data memory into two separate spaces, X and Y, to increase memory bandwidth. Table 6.4 shows how the core input MEM_CFG[2:0] signals define partitioning of the total 64 Kwords of data memory space into the X- and Y-data spaces.

Table 6.4 Data Memory Space Partitioning

MEM_CFG[2:0]	X-Memory Size (16-Bit Words)	Y-Memory Size (16-Bit Words)
000	32 K	32 K
001	48 K	16 K
010	56 K	8 K
011	60 K	4 K
100	62 K	2 K
101	63 K	1 K
110	63.5 K	512
111	63.75 K	256

The memory interface of the CWDSP1650 allows a flexible selection of memory devices. Only single-port memory is required, although dual or multiport memory can also be supported. There is no restriction on any sequence of read and write transactions on any of the memory spaces.

The core has a single output bus but separate input buses and address buses for each memory space. All memory buses are 16-bits wide.

The external data bus (EDB) is the single output bus to all off-core devices including data memory, program memory and user-defined registers (see Section 6.5, “User-Defined Register Interface,” for more information.) The core indicates which device it is writing to by using dedicated write-enable signals.

The core has three separate memory data input buses. The X-data bus (XDB) receives input from the X-data space, the Y-data bus (YDB) receives input from the Y-data space and the instruction data bus (IDB) receives data and instructions from the program memory. The core uses dedicated read-enable signals to control which memory space or external register is read from.

The core also has a separate address bus for each memory space. The X-address bus (XAB) and Y-address bus (YAB) output addresses for the X- and Y-data spaces respectively. The program address bus (IAB) outputs program memory addresses.

Table 6.5 summaries the memory interface signals of the CWDSP1650.

Table 6.5 Memory Signals Interface

Memory Space	Output Bus	Input Bus	Address Bus	Write Control	Read Control
X data	EDB	XDB	XAB	XWEN	XREN
Y data	EDB	YDB	YAB	YWEN	YREN
Program	EDB	IDB	IAB	PWEN	PREN

For more information about program and data memory, see Chapter 3, “Data Formats, Memory and Addressing.” For program and data memory timing diagrams, see Chapter 10, “Specifications.”

6.4.1 Memory Interface with Slow Memory Devices

The core supports transactions with slow memory devices (in either program or data memory space) through a built-in wait state mechanism. The Clock Control Unit (CCU) suspends CORE_CLK to put the CWDSP1650 into a wait state when the core input WAIT_CTL is asserted HIGH on the rising edge of MCLK. The core remains in the wait state until the rising edge of MCLK after WAIT_CTL is deasserted. This extends the machine cycle time to match the memory access time. During wait stated cycles most parts of the core are effectively put on hold. The Interrupt Control Unit (ICU) and the CCU are not clocked by CORE_CLK and so are not affected. All output signals on the memory interfaces are kept stable throughout the wait states.

Note that for every machine cycle, the core can fetch from several off-core devices concurrently. These include program memory, X-data memory, Y-data memory and the user-defined registers. A wait state request must insert enough wait states to allow enough time for the slowest accessed device.

For example, suppose that the program space is composed of a fast memory device (zero wait states) and a slow memory device (three wait states). The data space is composed of a fast memory device (zero wait states) and a slow device (four wait states). For this example, Table 6.6 summarizes the wait interval for four cases in which the program and data memories are fetched simultaneously.

Table 6.6 Example for Defining the Number of Wait Cycles

Transaction Type	Wait Interval (Cycles)
Program Fast (0 cycles) with Data Fast (0 cycles)	0
Program Slow (3 cycles) with Data Fast (0 cycles)	3
Program Fast (0 cycles) with Data Slow (4 cycles)	4
Program Slow (3 cycles) with Data Slow (4 cycles)	4

For more information about wait state timing and diagrams, see Chapter 10, "Specifications."

6.5 User-Defined Register Interface

The CWDSP1650 core architecture supports up to four user-defined registers. These user-defined registers enable expansion of the core in off-core logic. These 16-bit user-defined registers are considered part of the core register list and can be accessed directly by most CWDSP1650 instructions.

The core reads and writes the user-defined registers using a dedicated interface. When writing to user-defined registers, the core asserts the output signal LD_EXT_REG and specifies the destination with the 2-bit output SEL_EXT_REG_WT. The core outputs the data to the defined registers through the EDB bus. When reading from the user-defined registers, the core asserts the output signal RD_EXT_REG and specifies the source with the 2-bit output SEL_EXT_REG_RD. The core reads the data through the EXT_IN[15:0] bus.

Table 6.7 shows the code (either on SEL_EXT_REG_WT or SEL_EXT_REG_RD) associated with each user-defined register.

Table 6.7 User-Defined Register Coding

User-Defined Register	Code
ext0	00
ext1	01
ext2	10
ext3	11

The user-defined register transactions can have wait states as in the memory transactions. Once a wait state occurs (WAIT_CTL is asserted), the user-defined register interface is held valid throughout the extended access cycle. If a design has a destructive read or a destructive write, where the read or the write operation causes an internal mechanism to be initiated, the read or write must be blocked appropriately by using WAIT_CTL. Otherwise, multiple read/write triggers during the wait interval may cause register data to be lost or corrupted.

For more information about user-defined register timing and diagrams, see Chapter 10, “Specifications.”

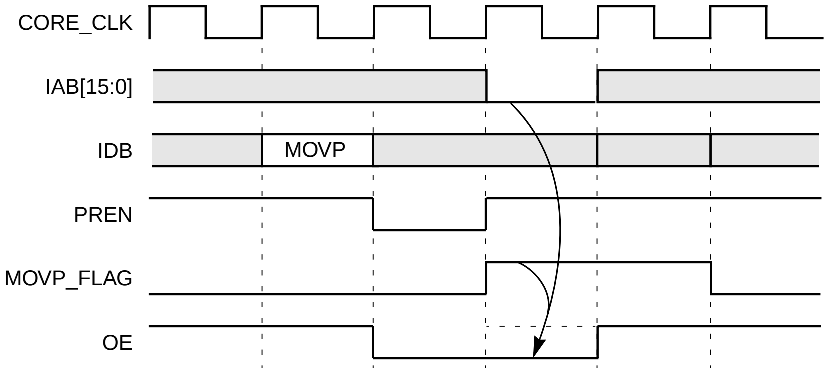
6.6 Program Protection Mechanism

The program protection mechanism protects the user program from unauthorized reading. It prevents infringement on intellectual property rights by reverse engineering of the program code held in on-chip memory.

The CWDSP1650 core supports program protection by asserting the MOVP_FLAG signal when executing a movp instruction. The movp instruction is the only instruction that can read from the program memory space and write the data obtained to the data memory space. An off-core device, typically a BIU, can use MOVP_FLAG to gate the program memory read-enable signal. The protection logic detects any unauthorized reads and prevents the program memory from being read.

Figure 6.4 shows how to protect the lower 32 Kwords of the program space in an application which maps this space to on-chip ROM.

Figure 6.4 Memory Protection Mechanism



Off-chip memory mapped to the upper 32 Kwords of program space can be modified to insert a routine that copies on-chip program code to another off-chip data memory. This can help where effective access control is difficult, if not impossible. To prevent such an intrusion, a BIU can detect the execution of a `movp` instruction fetched from off-chip memory using the MOVP_FLAG and IAB[15]. When detected, the BIU stops the core PREN signal from reaching the CWDSP1650 reference device output-enable (OE) pin of the on-chip program memory.

6.7 Clock Control Unit (CCU)

The CWDSP1650 Clock Control Unit (CCU) is a module external to the core, available as an optimized hard-macro. The CCU provides flexible control of the clock speed and sources for the core and surrounding peripherals. Table 6.8 lists the CCU output clocks.

Table 6.8 CCU Output Clocks

Clock	Description	Aligned Clocks	Wait States	Stop Mode
CORE_CLK	In normal mode, this clock is a division of MASTER clock. In scan mode, it switches to SCAN_CLK.	–	Yes	Yes
ICU_CLK	The ICU registers interrupts using this clock. ICU_CLK is a skewed version of CORE_CLK.	–	No	No
OCEM_CLK	A copy of CORE_CLK. This is the reference clock for the OCEM.	CORE_CLK	Yes	Yes
PMEM_CLK	A copy of CORE_CLK that is skewed optimally for clocking synchronous program memory.	–	Yes	Yes
DMEM_CLK	A copy of CORE_CLK skewed optimally for clocking synchronous data memory. DMEM_CLK is only active during a data memory access. DMEM_CLK skew is programmed through the SKEW bits in the CCU Register.	–	Yes	Yes

6.7.1 CCU Operation

The CCU derives its output clocks from either the MASTER clock or SCAN_CLK (the ScanICE interface generates START_SCAN after a breakpoint in ScanICE mode.) A clock control register provides software or system control over the clocks generated by the CCU. By writing appropriate values to the clock control register, the frequency of the clocks can be varied, the clocks can be stopped pending a restart by an interrupt, or SCAN_CLK can be selected. See Section 6.7.2, “CCU Register,” for more information.

Wait states are controlled using the WAIT_CTL signal. When WAIT_CTL is asserted HIGH, it suppresses the rising edge of CORE_CLK. CORE_CLK stays LOW until the WAIT_CTL signal is deasserted, after which CORE_CLK follows the rising edge of ICU_CLK. CORE_CLK also restarts when the core receives any unmasked interrupt. The restarted

CORE_CLK frequency depends upon the restore value programmed into the CCU register.

Setting the CCU register STOP bit to one initiates the clock stop mode. In stop mode, the CORE_CLK signal is stopped, but ICU_CLK continues to run. When programming the CCU to enter stop mode, the interrupts must be disabled and re-enabled around the instruction that sets the STOP bit in the CCU register using DINT and EINT instructions. Three NOP instructions should also be placed following the EINT instruction, for example:

```
mov ##0xf7e5, r0        ; Set up CCU reg addr in r0
dint                    ; Disable interrupts
chng ##0x0010, (r0)     ; Set STOP bit to one
eint                    ; Enable interrupts
nop
nop
nop
```

6.7.2 CCU Register

Figure 6.5 shows the CCU register and bit fields.

Figure 6.5 CCU Register

15 9	8 5	4	3 0
SKEW	RSV	STOP	SELECT

SKEW **Data Memory Clock Skew Bits** **[15:9], R/W**

The SKEW bits set the skew for DMEM_CLK. These bits should not normally be adjusted during operation since they directly affect memory access interface timings. The following table lists the skew values available through the SKEW bits.

SKEW bits	Read Access (ns)	Write Access (ns)
1000	2.00	6.50
1001	2.25	6.25
1010	2.50	6.50
1011	2.65	6.65
1100	2.80	6.80
1110	3.10	7.10
1111	3.25	7.25
0000[1]	3.40	7.40

SKEW bits	Read Access (ns)	Write Access (ns)
0001	3.55	7.55
0010	3.70	7.70
0011	3.85	7.85
0100	4.00	8.00
0101	4.25	8.25
0110	4.50	8.50
0111	5.00	9.00

[1] Value after reset.

RSV **Restore Value** **[8:5], R/W**

When the STOP bit clears to zero, the CCU transfers the RSV value into the SELECT bits of the clock control register. This allows the core to be put into a low-power idle state with the ICU sampling interrupts at the selected slow clock rate. When the core receives an interrupt, it restarts at the clock rate determined by RSV and then responds to the interrupt.

STOP **Stop Scan Bit** **4, R/W**

The CCU enters stop mode when this bit is set to one. In the stop mode CORE_CLK is stopped, but ICU_CLK clocks are still running. This allows the ICU to remain functional while the core is effectively stopped. Any interrupt allowed through the ICU clears the STOP bit and restarts the CORE_CLK.

SELECT **Select Clock Division** **[3:0], R/W**

This four-bit value selects which division of the MASTER clock generates the output clocks. The MASTER clock is divided by $2^{(\mathrm{SELECT})}$, so the output clock can be equal to MASTER when select is 0x0, or divided by 16384 when select is 0xE. When select is 0xF, SCAN_CLOCK is selected (this selection is reserved for use by the debug system and should not be directly used in application software).

SELECT Value	Output Clock
0x0	MASTER
0x1	MASTER / 2
0x2	MASTER / 4
.	.
.	.
.	.
0xE	MASTER / 16384
0xF	SCAN_CLK

Chapter 7
Instruction Set

This chapter describes the core instruction set, contains a complete alphabetical listing of all instructions, and is composed of the following sections:

- ♦ Section 7.1, “Notations”
- ♦ Section 7.2, “Conventions and General Information”
- ♦ Section 7.3, “Instruction Functional Groups”
- ♦ Section 7.4, “Instruction Set List”
- ♦ Section 7.5, “Instruction Opcode Bit Coding”

7.1 Notations

This section defines the special symbols and notations used throughout this chapter. The areas with special notations include registers, numbers, data and program operands, and flags.

7.1.1 Register Notations

Table 7.1 defines the notations used to describe the core registers.

Table 7.1 Register Notations

Notations	Register
rN	Address registers: r0, r1, r2, r3, r4, r5
rI	Address registers: r0, r1, r2, r3
rJ	Address registers: r4, r5
(Sheet 1 of 2)	

Table 7.1 Register Notations (Cont.)

Notations	Register
aX	a0 or a1
aXl	a-accumulator-low (LSP), x = 0, 1
aXh	a-accumulator-high (MSP), x = 0, 1
aXe	a-accumulator extension, x = 0, 1
bX	b0 or b1
bXl	b-accumulator-low (LSP), x = 0, 1
bXh	b-accumulator-high (MSP), x = 0, 1
ac	a0, a1, a0h, a1h, a0l, a1l
bc	b0, b1, b0h, b1h, b0l, b1l
ab	a0, a1, b0, b1
cfgX	Configuration registers of DAAU (MODI or MODJ, STEPI or STEPJ), x = i, j
sv	Shift Value register
sp	Stack Pointer
pc	Program counter
lc	Loop counter (of block repeat)
extX	External registers, x = 0, 1, 2, 3 (user definable registers)
REG	a0, a1, a0h, a1h, a0l, a1l, b0, b1, b0h, b1h, b0l, b1l, r0, r1, r2, r3, r4, r5, rb, y, p, ph, sv, sp, pc, lc, st0, st1, st2, cfgi, cfgj, ext0, ext1, ext2, ext3
x	x (multiplier input) register
mixp	Minimum/maximum pointer
icr	Internal Configuration Register
repc	Repeat Counter
dvm	Data Value Match register
(Sheet 2 of 2)	

7.1.2 Number Representation

Binary numbers are initiated with either “0b” or “0B.” Hexadecimal numbers are initiated with either “0x” or “0X.”

7.1.3 Data and Program Operands

Table 7.2 and Table 7.3 list the following information for the program and the data operands:

- Number of bits
- Operand range including the assembler mnemonics
- An operand example

Note that negative numbers can be written as four hexadecimal digits. For example, -0x80 can be written as 0xFF80, and -0x20 can be written as 0xFFE0.

Table 7.2 Program Operand Notation

		Assembler Syntax			
Operand	**Number of bits**	**Decimal**	**Hexadecimal**	**Binary**	**Example**
direct address	unsigned 8 bits (offset in page)	0–255	0x0–0xFF	0b0000000–0b11111111	add 120, a1
[##direct address]	unsigned 16 bits	0–65535	0x0–0xFFFF	0b0–0b1111111111	sub [##var1], a0
address	unsigned 16 bits	0–65535	0x0–0xFFFF	0b0–0b1111111111	call 0x5000

Table 7.3 Data Operand Notation

		Assembler Syntax			
Operand	**Number of bits**	**Decimal**	**Hexadecimal**	**Binary**	**Example**
# signed short immediate	8 (2’s complement)	# -128–127	# -0x80–0x7F	# -0b1000.0000 –0b0111.1111	mov #-12, r0
# signed 6 bit immediate	16 (2’s complement)	# -32–31	# -0x20–0x1F	# -0b100000–0b011111	shfi b0, a0, #-4
# signed 5 bit immediate	5 (2’s complement)	# -16–15	# -0x10–0xF	# -0b10000–0b01111	movsi r1, a0, #3
# unsigned 9 bit immediate	9 (unsigned)	# 0–511	# 0x0–0x1FF	# 0b0000000–0b1111.11111	load #270, modi
# unsigned short immediate	8 (unsigned)	# 0–255	# 0x0–0xFF	# 0b0000000–0b11111111	add #0b10, a0
# unsigned 7 bit immediate	7 (unsigned)	# 0–127	# 0x0–0x7F	# 0b0000000–0b01111111	load #3, stepj
# unsigned 5 bit immediate	5 (unsigned)	# 0–31	# 0x0–0x1F	# 0b00000–0b011111	mov #0x5, icr
# unsigned 2 bit immediate	2 (unsigned)	# 0–3	# 0x0–0x3	# 0b00–0b11	load #0b11, ps
# bit number	4 (unsigned)	# 0–15	# 0x0–0xF	# 0b0000–0b01111	tstb r0, #12
## long immediate	16 (2’s complement)	## -32768–32767	## -0x8000–0x7FFF	## -0b100000–0b01111111	mov ## -0x9000, a0
## offset	16 (unsigned)	## 0–65535	## 0x0–0xFFFF	## 0b0–0b1111111111	mov ## 0xF000, r0
# offset7	7 (2’s complement)	-64–63	-0x40–0x3F	-0b1000000–0b0111111	add (rb-5), a1

7.1.4 Option Fields

The option fields contain the notations listed in Table 7.4.

Table 7.4 Option Field Notations

Notations	Description
eu	Extension unaffected. This field is optional in the `mov direct address, axh, [eu]` instruction. When eu is mentioned, the data is transferred into aXh without affecting aXe. When eu is not mentioned, the data is transferred into aXh with sign-extension into aXe.
context	Context switching. This field is optional in the `reti` instruction. When context is used, automatic context switching occurs. When context is not used, there is no context switching.
dmod	Disable modulo. This field is optional in the `modr` instruction. When dmod is mentioned, the rN is postmodified with modulo modifier disabled. When not mentioned, the Mn bit influences the postmodification of rN.

7.1.5 Condition Field (cond) Notations

Table 7.5 shows the notations for the condition field.

Table 7.5 Condition Field Notations

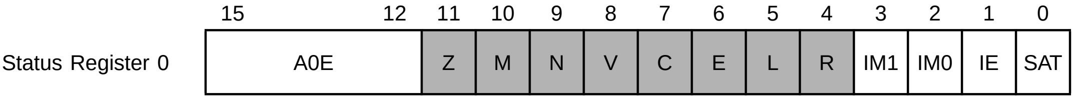
Mnemonics	Description	Condition
true	Always	–
eq	Equal to zero	Z = 1
neq	Not equal to zero	Z = 0
gt	Greater than zero	$M = 0 \cap Z = 0$
ge	Greater or equal to zero	M = 0
lt	Less than zero	M = 1
le	Less or equal to zero	$M = 1 \cup Z = 1$
nn	Normalize flag is cleared	N = 0
v	Overflow flag is set	V = 1
c	Carry flag is set	C = 1
(Sheet 1 of 2)		

Table 7.5 Condition Field Notations (Cont.)

Mnemonics	Description	Condition
e	Extension flag is set	E = 1
l	Limit flag is set	L = 1
nr	R flag is cleared	R = 0
niu0	Input user pin 0, IUSER0, is cleared	–
iu0	Input user pin 0, IUSER0, is set	–
iu1	Input user pin 1, IUSER1, is set	–
(Sheet 2 of 2)		

7.1.6 Flag Notations

The eight flags are found in Status Register 0. There are eight flags: Z, M, N, V, C, E, L, and R. The flags are depicted as shown below.



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Status Register 0	A0E				Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Flags that are affected by the specific instruction are shaded. Flags that are not affected are not shaded. For full definitions and descriptions of the flag bits, see Section 4.4.1, “Status Register 0 (ST0).”

7.1.7 Miscellaneous Notations

Table 7.6 lists all other notations.

Table 7.6 Miscellaneous Notations

Notation	Description
(x)	The contents of x
|	One of the options should be included
[ ]	Optional field at the instruction
<x>	Specific notes
(Sheet 1 of 2)	

Table 7.6 Miscellaneous Notations (Cont.)

Notation	Description
->	Is assigned to
>>	Shift right
<<	Shift left
exp(x)	Exponent of x
_	Not
»	Or
«	And
(Sheet 2 of 2)	

7.2 Conventions and General Information

The conventions used through the instruction set are:

1. The arithmetic operations are performed in two’s complement.
2. The following instructions allow postmodification of the rN registers:
   - instructions that use indirect addressing mode
   - MODR
   - NORM
   - MAX, MAXD, MIN (use r0 only)

   In these instructions, the contents of the rN register are postmodified through the following:

   - Instruction-Controlled Options
   - rN, rN + 1, rN – 1, rN + step
   - Configuration Register (CFGI or CFGJ) Controlled Options

     Step size: STEPI, STEPJ - 2’s complement 7 bits (-64 to 63)
     Modulo size: MODI, MODJ - unsigned 9 bits (1 to 512)

- Status Register ST2 Controlled Options

  For each rN register, it should be defined if MODULO is enabled or disabled.

  When using MODI or MODJ, the relative Mn bit must be set. The only exception is with the MODR instruction, which has an optional field for disabling the modulo.

  For more details on the modulo arithmetic unit, see the subsection entitled “Modulo Modification” on page 2-21.

Whenever the operand field in the instruction is (rN), the rN can be postmodified in one of the following four options.

Assembler syntax: `(rN), (rN)+, (rN)-, (rN) + s`

Four examples of instructions with postmodified rNs are:

```
mov (r0)-, r1
mac (r4)+, (r0)+s, a0
add (r2), a1
modr (r5)-
```

3. The direct addressing mode assembler syntax is as follows:

   - The syntax when a one-word instruction is used is either direct address or [direct address]. (This instruction uses the contents of the PAGE bits.)

   - The syntax when a two-word instruction is used—even if the address is an eight-bit value—is [## direct address].

4. The 16 most-significant bits of the P register (denoted as PH) are write only. The 32-bit P register is updated after a multiply operation. The only way to read the P register is to transfer its contents into the Ax accumulators or set the P register as an operand for an arithmetic and logic operation. The contents of the P register are sign-extended to 36 bits when transferred into the accumulator. This method allows you to store and restore the P register.

5. The P register is used as a source operand for different instructions as follows:

   - as one of the REG registers,

   - in a moda instruction—*pacr* function,

   - in multiply instructions where the P register is added or subtracted from one of the accumulators.

When the P register is used as a source operand, it is referred to as the “shifted P register.” Shifted P register means that the P register is sign-extended into 36 bits and then shifted as defined by the PS field in Status register ST1. In a right shift, the sign is extended; whereas in a left shift, a zero is appended to the LSB. The contents of the P register remain unchanged. In two multiply instructions, `maa` and `maasu`, the P register is also aligned. In other words, after the P register is sign-extended and shifted according to the PS field, it is also shifted 16 bits to the right.

6. Move instructions that use the accumulator (aX or bX) as a destination are sign extended. Instructions that specify the accumulator-low (aXl or bXl) as a destination clear the accumulator-high and the accumulator-extension. Therefore, these instructions are sign-extension suppressed.

   All instructions using the accumulator-high (aXh or bXh) as a destination clear the accumulator-low. These instructions are sign extended. An exception is:

   `mov direct address, axh, [eu]`.

   When the eu option is used, data is moved into accumulator-high without sign extension (the accumulator-extension aXe is unaffected).

7. In all arithmetic operations between 16-bit registers and aX (36 bits), the 16-bit register is regarded as the 16 low-order bits of a 36-bit operand with a sign extension in the most-significant bits.

8. The condition field is almost always an optional field, except when it is followed by another optional field, as in the `reti` instruction. When the condition field is the last field of the instruction and the condition is omitted, the condition defaults to true. For example, `shr4 true` is the same as `shr4`. In the instruction `reti true, context`, the ‘`true`’ cannot be omitted.

9. Arithmetic and logical operations (not bit manipulation operations) using the same accumulator as the source (soperand) and the destination (doperand) are not permitted. For example, add `a0, a0` is not invalid but `shfc a0, a0` is.

10. An instruction that immediately follows another which has modified the rb register may not use the index addressing mode. The only exception is when rb is modified using a long immediate operand (`mov ##long immediate, rb`).

11. All commands that use pc as a destination register must be followed with two nop instructions. The only exception to this statement is the `mov ##long immediate,` pc instruction, where only one nop is required.

7.3 Instruction Functional Groups

This section contains an overview of all core instructions, and provides additional information on selected groups of instructions. Table 7.7 contains a comprehensive list of all the CWDSP1650 instructions, grouped by function, and the page number where information about each instruction can be found.

Table 7.7 CWDSP1650 Instruction Set Listing

Function	Operation	Description	Page
Arithmetic and Logical Instructions	ADD	Add	7-18
	SUB	Subtract	7-180
	OR	Logical OR	7-137
	AND	Logical AND	7-28
	XOR	Logical Exclusive OR	7-200
	CMP	Compare	7-48
	ADDL	Add to Low Accumulator	7-23
	SUBL	Subtract from Low Accumulator	7-185
	ADDH	Add to High Accumulator	7-21
	SUBH	Subtract from High Accumulator	7-183
	CMPU	Compare Unsigned	7-51
	ADDV	Add Long Immediate Value[1]	7-25
	SUBV	Subtract Long Immediate Value[1]	7-187
	CMPV	Compare Long Immediate Value[1]	7-53
	NORM	Normalize	7-134
(Sheet 1 of 5)			

Table 7.7 CWDSP1650 Instruction Set Listing (Cont.)

Function	Operation	Description	Page
Arithmetic and Logical Instructions (Cont.)	DIVS	Division Step	7-61
	MAX	Maximum between two Ax-Accumulators	7-86
	MAXD	Maximum between Data Memory Location and Ax-Accumulator	7-88
	MIN	Minimum between two Ax-Accumulators	7-90
	LIM	Limit Ax-Accumulator	7-67
	MODA	Modify Ax-Accumulator Conditionally	7-92
	MODA Modifications:[2]		
	SHR	Shift Right	7-172
	SHR4	Shift Right Four	7-174
	SHL	Shift Left	7-168
	SHL4	Shift Left Four	7-170
	ROR	Rotate Right Through Carry	7-159
	ROL	Rotate Left Through Carry	7-158
	NOT	Logical NOT	7-136
	NEG	Two’s Compliment	7-132
	CLR	Clear	7-46
	COPY	Copy other Accumulator	7-58
	RND	Round upper 20 bits	7-157
	PACR	Product Move and Round	7-140
	CLRR	Clear and Round	7-47
	INC	Increment by One	7-66
	DEC	Decrement by One	7-59
Multiply Instructions	MPY	Multiply	7-121
	MPYSU	Multiply Signed by Unsigned	7-127
	MAC	Multiply and Accumulate Previous Product	7-76
(Sheet 2 of 5)			

Table 7.7 CWDSP1650 Instruction Set Listing (Cont.)

Function	Operation	Description	Page
Multiply Instructions (Cont.)	MACSU	Multiply Signed by Unsigned and Accumulate Previous Product	7-78
	MACUS	Multiply Unsigned by Signed and Accumulate Previous Product	7-81
	MACUU	Multiply Unsigned by Unsigned and Accumulate Previous Product	7-83
	MAA	Multiply and Accumulate Aligned Previous Product	7-72
	MAASU	Multiply Signed by Unsigned and Accumulate Aligned Previous Product	7-74
	MSU	Multiply and Subtract Previous Product	7-129
	MPYI	Multiply Signed Short Immediate	7-123
	SQR	Square	7-176
	SQRA	Square and Accumulate Previous Product	7-178
BMU Instructions	SET	Set Bit Field	7-162
	RST	Reset Bit Field	7-160
	CHNG	Change Bit Field	7-44
	TST0	Test Bit Field for Zeroes	7-193
	TST1	Test Bit Field for Ones	7-195
	TSTB	Test Specific Bit	7-198
	SHFC	Shift Accumulators According to Shift Value Register Conditionally	7-164
	SHFI	Shift Accumulators by an Immediate Shift Value	7-166
	EXP	Evaluate the Exponent Value	7-64
	MODB	Modify Bx-Accumulator Conditionally	7-96
	MODB Modifications:[2]		
	SHR	Shift Right	7-172
	SHR4	Shift Right Four	7-174
	SHL	Shift Left	7-168
(Sheet 3 of 5)			

Table 7.7 CWDSP1650 Instruction Set Listing (Cont.)

Function	Operation	Description	Page
BMU Instructions (Cont.)	MODB Modifications: (Cont.)		
	SHL4	Shift Left Four	7-170
	ROR	Rotate Right through Carry	7-159
	ROL	Rotate Left through Carry	7-158
	CLR	Clear	7-46
Move Instructions	MOV	Move Data	7-101
	MOVP	Move from Program Memory into Data Memory	7-113
	MOVD	Move from Data Memory into Program Memory	7-112
	MOVS	Move and Shift according to Shift Value Register	7-117
	MOVSI	Move and Shift according to an Immediate Shift Value	7-119
	MOVR	Move and Round	7-115
	PUSH	Push Register or Long Immediate Value onto Stack	7-143
	POP	Pop from Software Stack into Register	7-141
	SWAP	Swap Ax and Bx Accumulators	7-190
	BANKE	Bank Exchange	7-32
Branch/Call Instructions	BR	Conditional Branch	7-37
	BRR	Relative Conditional Branch	7-39
	CALL	Conditional Call Subroutine	7-40
	CALLR	Relative Condition Call Subroutine	7-42
	CALLA	Call Subroutine at Location Specified by Ax Accumulator	7-41
	RET	Return Conditionally	7-147
	RETD	Delayed Return	7-149
	RETI	Return from Interrupt Conditionally	7-151
	RETID	Delayed Return from Interrupt	7-153
	RETS	Return with Short Immediate Parameter	7-155
(Sheet 4 of 5)			

Table 7.7 CWDSP1650 Instruction Set Listing (Cont.)

Function	Operation	Description	Page
Control and Miscellaneous Instructions	NOP	No Operation	7-133
	MODR	Modify Register N	7-99
	EINT	Enable Interrupt	7-63
	DINT	Disable Interrupt	7-60
	TRAP	Software Interrupt	7-192
	LOAD	Load Specific Field into Registers - page, modx, stepx, ps	7-69
	CNTX	Context Switching Store or Restore	7-55
Loop Instructions	REP	Repeat Next Instruction	7-145
	BKREP	Block Repeat	7-34
	BREAK	Break from a Block-Repeat	7-38
(Sheet 5 of 5)			

1. To or from a register or a data memory location.
2. Shaded cells are modifications of either MODA or MODB.

7.3.1 Shift Operations

All shift operations are performed in a single cycle. Each of the four accumulators can be shifted according to a six-bit signed number, representing -32 to +31 shifts, embedded in the instruction opcode. A left shift is indicated by a positive number; a right shift by a negative number.

The SV register content can be used to cause a conditional shift of between -36 and +36 bits. Conditional shifting supports calculating the amount of shifts at run time, as in normalization operations (refer to Section 2.4.2.4, “Normalization.”) The source and the destination accumulators do not have to be the same. If the accumulators are different, the source accumulator is unaffected. For more information, refer to the SHFC and SHFI instructions.

The shift and rotate operations included in the MODA and MODB instructions are also performed conditionally. MODA and MODB include one-bit right and left arithmetic-shift and rotate, and four-bit right and left arithmetic-shift. Refer to the MODA and MODB instructions for more information.

The Status Mode bit (S) in the ST2 register determines whether the shift is arithmetic or logical. Figure 7.1 shows an arithmetic right shift of an accumulator. During an arithmetic right shift, the most-significant bits are sign extended and the least significant bit is loaded into the C flag.

Figure 7.1 Arithmetic Shift Right

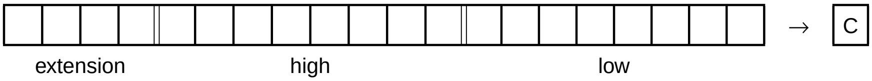
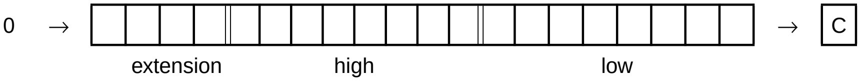


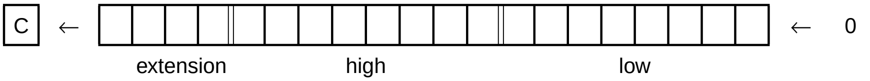
Figure 7.2 shows a logical right shift of an accumulator. During a logical right shift, the most-significant bits are filled with zeros.

Figure 7.2 Logical Shift Right



Arithmetic left shifts and logical left shifts are performed identically (see Figure 7.3). The arithmetic left shifts and logical left shifts both fill the least-significant bits with zeros.

Figure 7.3 Arithmetic and Logical Shift Left



7.3.2 Move and Shift Operations

The move and shift operations (MOVS and MOVSI) perform in a single cycle. The accumulators are loaded from one of the registers or from a data memory location, addressed in direct addressing mode, or in indirect addressing mode, according to the SV shift value. The shifting range is +36 to -36, (shift left is indicated by a positive number; shift right by a negative number). The accumulators can also be loaded by one of the DAAU registers (Rn registers) and shifted according to a constant embedded in the instruction. The shifting range in this case is +15 to -16. The Status Mode bit (S) in the ST2 register determines whether the shift is arithmetic or logical, as described in Section 7.3.1, “Shift Operations.” For more information, see the MOVS and MOVSI instructions in Section 7.4, “Instruction Set List.”

7.3.3 Rounding Operations

The MOVR and MODA instructions can round the contents of the accumulator either in a single cycle or in parallel with other operations. For more information, see the MOVR and MODA instructions.

7.3.4 Division Step Operations

The ALU supports a single-cycle division step. For more details, see the DIVS instruction.

7.3.5 Logical Operations

The AND, OR, and XOR logical operations operate on 36 bit wide data. 16-bit operands are zero extended when used with these logical operations. The source and destination Ax Accumulators of these instructions are always the same. For example, the instruction `and (r1), a0` ANDs the contents of register R1 with the contents of the A0 accumulator and stores the result in the A0 Accumulator.

Logical operations between the two Ax Accumulators are also possible, for example, the instruction `and a0, a1` ANDs the contents of the A0 and A1 Accumulators together. The results are stored in Accumulator A1. For details, refer to the detailed descriptions of the AND, OR, and XOR instructions in this chapter.

Other logical operations are SET, RST, CHNG, TST0, TST1, and TSTB. These operations are executed on one of the registers or on data memory contents. Refer to Section 2.4.2.5, “Bit-Field Operations.”

7.3.6 MAX and MIN Instructions

The MAX and MIN instructions are single-cycle operations used to find maximum and minimum values. The two Ax accumulators are compared and if the non-specified accumulator is found to be the larger (or smaller for the MIN instruction) then this value overwrites the value in the specified accumulator. Hence, the specified accumulator holds the maximum/minimum value of the two accumulators after the operation.

The R0 pointer can be used in the same instruction as a buffer pointer, for example. R0 can be postmodified according to the specified mode in the instruction. If a new maximal or minimal number is found, the R0 pointer is also latched into the minimum/maximum pointer latching (MIXP) register.

The maximum operation can also be performed directly on a data memory location pointed to by the R0 register (MAXD instruction). With this single-cycle instruction, the maximal value is saved in the defined Ax Accumulator and the R0 value is written into the MIXP register.

In finding the maximum value, the pointer of the first element or the last element is latched, using greater than (>), or equal or greater than (≥) conditions, respectively. In finding the minimum value, the pointer of the first element or the last element is latched, using less than (<), or equal or less than (≤) conditions, respectively.

Refer to the MAX, MAXD, and MIN instructions in this chapter. For more details on the R0 and MIXP registers, refer to Section 4.2, “DAAU Registers.”

7.3.7 Multiplication Instructions

The core supports single- and double-precision multiplications. The signed-by-signed operation is used to multiply or multiply-accumulate the signed portions of the numbers. The signed-by-unsigned operation is used to multiply or multiply-accumulate the signed portion of the number with the lower, unsigned portion of the double-precision number. Additional support for double-precision operations is given by shifting the P register 16 bits to the right before accumulating it, during a multiply-accumulate instruction, or to shift the partial multiplication result.

7.4 Instruction Set List

The instruction set descriptions are listed alphabetically. Each instruction definition contains the instruction format, syntax, instruction description, operation, flags, cycles, and an example. See Section 7.3, “Instruction Functional Groups,” for a listing of all instructions, grouped by function, and specific instruction page numbers.

Most core instructions are executed in a single cycle. The instruction execution time is denoted by Xc, where X is the number of cycles. Instructions that have a long immediate, long direct, or long index operand as one of the operand options take an additional cycle to execute. Instructions that break the pipeline may have a different execution time depending on whether their condition is met or not met (for example, the CALL instruction). The execution time is indicated for both conditions in the Cycles section.

ADD Add

Opcode

	15 13	12 9	8	7 0
Short Direct	101	0011	i	direct

	15 13	12 9	8	7 3	2 0
Long Direct (MSW)	110	1010	i	11111	011

	15 0
Long Direct (LSW)	long direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	0011	i	100	mod	rn

	15 13	12 9	8	7 5	4 0
Register	100	0011	i	101	REG

	15 12	11 9	8	7 0
Short Immediate	1100	011	i	short immediate

	15 12	11 9	8	7 5	4 0
Long Immediate (MSW)	1000	011	i	110	xxxxx

	15 0
Long Immediate (LSW)	long immediate

Syntax

`add operand, ai`

Description

The contents of ai and operand are added to form either a 16-bit or a 32-bit result. The result is stored in ai[35:0]. If an operand other than p or aj register is selected, the contents of operand are added to ai[15:0] to form a 16-bit addition. If p or aj register is selected for the operand, p[31:0] is added to ai[31:0]. In both cases, the sign is extended through ai[35:32].

ADD Add

Operation

```
ai ← ai + operand

operand = REG[1]
          (rN)
          short direct address
          [##direct address]
          #unsigned short immediate
          ##long immediate
          (rb + offset7)
          (rb+##offset)
```

1. The REG cannot be ai or bi.

Flags affected

	15	12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E		Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Long Direct	2	2
Indirect	1	1
Register	1	1
Short Immediate	1	1
Long Immediate	2	2
Short Index	1	1
Long Index	2	2

ADD Add

Examples

```
add (r0), a0
```

Before Execution:

a0:	0	8000	0000	(r0):	001E

After Execution:

a0:	0	8000	001E	(r0):	001E

```
add 0x100, a0
```

Before Execution:

a0:	0	0000	0000	0x100:	00FF

After Execution:

a0:	0	0000	00FF	0x100:	00FF

```
add ##0x1001, a0
```

Before Execution:

a0:	0	1234	FFE0

After Execution:

a0:	0	1235	0FE1

ADDH Add to High Accumulator

Opcode

Short Direct

15–13	12–9	8	7–0
101	1001	i	direct

Indirect

15–13	12–9	8	7–5	4–3	2–0
100	1001	i	100	mod	rN

Register

15–13	12–9	8	7–5	4–0
100	1001	i	101	REG

Syntax

`addh operand, ai`

Description

The contents of operand are added to ai[31:16] to form a 16-bit addition. ai[15:0] is unaffected after the operation.

Operation

$\texttt{ai + operand * } 2^{16} \rightarrow \texttt{ai}$

The ail is unaffected.

operand = REG[1]
(rN)
short direct address

1. The REG cannot be bi, ai, or p.

Flags affected

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1

ADDH Add to High Accumulator

Examples `addh b1h, a0`

Before Execution:

a0:	0	10EE	8000	b1h:	E320

After Execution:

a0:	F	F40E	8000	b1h:	E320

`addh (r0)+, a0`

Before Execution:

a0:	0	10EE	8000	r0:	0006
				0x0006:	0200
				0x0007:	9476

After Execution:

a0:	0	10EE	8200	r0:	0007
				0x0006:	0200
				0x0007:	9476

ADDL Add to Low Accumulator

Opcode

	15 13	12 9	8	7 0
Short Direct	101	1010	i	direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	1010	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	1010	i	101	REG

Syntax

```
addl operand, ai
```

Description

The contents of operand are added to ai[15:0] to form a 16-bit addition. Sign-extension of operand is suppressed for this operation.

Operation

```
ai + operand → ai

operand = REG[1]
          (rN)
          short direct address
```

1. The REG cannot be bi, ai, or p.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1

ADDL Add to Low Accumulator

Example `addl lc, a0`

Before Execution:

a0:	0	4320	8000	lc:	FFFF

After Execution:

a0:	0	4321	7FFF	lc:	FFFF

ADDV Add Long Immediate Value to a Register or a Data Memory Location

Opcode

Short Direct

15 12	11 9	8	7 0
1110	011	1	direct

Indirect

15 12	11 9	8 5	4 3	2 0
1000	011	0111	mod	rN

Register

15 12	11 9	8 5	4 0
1000	011	1111	REG

Syntax

`addv ##long immediate, doperand`

Description

The contents of long immediate value are added to doperand to form a 16-bit result. The result of the operation is stored in the doperand. The operand and long immediate values are sign-extended before the addition. If the doperand is not part of an accumulator (`ail`, `aih`, `aie`, `bil`, or `bih`) then the accumulators are unaffected. If the operand is part of an accumulator, only the addressed part is affected.

Operation

```
doperand ← ##long immediate + doperand

doperand = REG[1]
           (rN)
           short direct address
```

1. The REG cannot be bi, ai, p, or pc. Note that ai can be used in add `##long immediate, ai` instruction.

ADDV Add Long Immediate Value to a Register or a Data Memory Location

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Z, M, and C are a result of the 16-bit operation. Bit 15 affects the M flag.

When the operand is not st0:

When adding a long immediate value to ST0, the result is stored in ST0. When adding a long immediate value to ST1, the flags are affected by the ALU output, as usual.

Note that when the operand is part of an accumulator (`ail`, `aih`, `aie`, `bil`, or `bih`), only the addressed part is affected. For example, if the instruction addv `##long immediate`, `a0l` generates a carry, the carry flag is set, however, a0h is unchanged. On the other hand, the instruction `addl ##long immediate, a0l` (with same a0 and immediate values) changes the a0h and affects the carry flag according to bit 36 of the ALU result.

When the operand is st0:

ST0 (including the flags) accepts the addition result, regardless of the a0e bits.

Cycles

	Cycles	Words
Short Direct	2	2
Indirect	2	2
Register	2	2

ADDV — Add Long Immediate Value to a Register or a Data Memory Location

Examples

```
addv ##0xFE00, a0l
```

Before Execution:

a0:	0	3000	3F01

After Execution:

a0:	0	3000	3D01

```
addv ##0x0002, 0
```

Before Execution:

0x100:	0120	page:	01

After Execution:

0x100:	0122	page:	01

AND And

Opcode

	15 13	12 9	8	7 0
Short Direct	101	0001	i	direct

	15 13	12 9	8	7 3	2 0
Long Direct (MSW)	110	1010	i	11111	001

	15 0
Long Direct (LSW)	long direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	0001	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	0001	i	101	REG

	15 12	11 9	8	7 0
Short Immediate	1100	001	i	short immediate

	15 12	11 9	8	7 5	4 0
Long Immediate (MSW)	1000	001	i	110	xxxxx

	15 0
Long Immediate (LSW)	long immediate

	15 12	11 9	8	7	6 0
Short Index	0100	001	i	0	Offset

	15 12	11 9	8	7	6 3	2 0
Long Index (MSW)	1101	010	i	0	11011	001

	15 0
Long Index (LSW)	long index

AND And

Syntax `and operand, ai`

Description The contents of operand are combined with the contents of ai in a bitwise logical-AND operation. If p or aj is selected for operand, ai[35:0] is logically ANDed with operand and the result is stored in ai[35:0]. If the operand is short immediate, the operand is zero-extended to form a 36-bit operand, then ANDed with the destination accumulator. Bits [15:8] are unaffected; other bits of the accumulator are cleared.

The instruction and `#unsigned short immediate, ai` can be used for masking out (clearing) some of the low-order bits at a 16-bit destination. The example below shows masking out of the top 8 bits of a memory location:

```
mov (r0), a0
and #0xf, a0
mov a0, (r0)
```

Using the and instruction, bits [15:8] are unaffected, therefore the high-order bits at the destination do not change. See also the `rst` instruction.

If the operand is a 16-bit register or a long immediate value, the operand is zero-extended to form a 36-bit operand, then ANDed with the accumulator. Therefore, this instruction clears the upper bits of the accumulator.

Operation

```
If operand is ai or p
   ai[35:0] AND operand → ai[35:0]

If operand is unsigned short immediate
   ai[7:0] AND operand → ai[7:0]
   ai[15:8] → ai[15:8]¹
   0 → ai[35:16]

If operand is REG, (rN), long immediate
   ai[15:0] AND operand → ai[15:0]
   0 → ai[35:16]
```

AND And

```
operand = REG1
          (rN)
          short direct address
          [##direct address]
          #unsigned short immediate
          ##long immediate
          (rb+offset7)
          (rb+##offset)
```

[1] The REG cannot be bi.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Note: The Z flag is set if all ALU output bits after the operation are zero; otherwise, it is cleared. When the operand is unsigned short immediate, ALU output bit [35:8] = 0 bits [7:0] = ai[7:0] AND operand.

Cycles

	Cycles	Words
Short Direct	1	1
Long Direct	2	2
Indirect	1	1
Register	1	1
Short Immediate	1	1
Long Immediate	2	2
Short Index	1	1
Long Index	2	2

AND And

Example `and #30, a0`

Before Execution:

a0:	0	0820	FFFF

After Execution:

a0:	0	0000	FF1E

BANKE Bank Exchange

Opcode

	15 7	6 4	3 0
banke	010010111	xxx	bank

Syntax

`banke [r0] [,r1] [,r4] [,cfgi]`

Description

Exchange the list of operands with their respective shadow registers.

The bank bits select the registers to be exchanged, as shown below:

Bit Number	Register
0	r0
1	r1
2	r4
3	cfgi

Assembler syntax examples:

```
banke r0
banke r1, cfgi
banke r1, r0
banke cfgi, r1, r4
```

For more details refer to Section 6.3.6, “Context Switching.”

Operation

operand ↔ operandb

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

BANKE **Bank Exchange**

Flags affected This instruction does not affect the flags.

Cycles

	Cycles	Words
banke	1	1

Example banke `r0`

Before Execution:

r0:	2F01	r0b:	0100

After Execution:

r0:	0100	r0b:	2F01

BKREP Block Repeat

Opcode

Register

15 … 8	7 … 5	4 … 0
01011101	xxx	REG

# unsigned short immediate

15 … 8	7 … 0
01011100	short immediate

Syntax

`bkrep operand, addr`

Description

This instruction begins a block repeat that is to be repeated operand + 1 times. The value of operand can range from 0 to 65535.

The start address of the code block to be repeated is the address after the `bkrep` instruction, and the last address is the one specified by the `addr` field. If the last instruction at the block repeat is a one-word instruction, `addr` is the address of this instruction. If the last instruction at the block repeat is a two-word instruction, add is the address of the second word of the instruction.

The operand is inserted into the loop counter (LC) register. The inloop status bit LP is set to one indicating a block repeat loop. The block repeat nesting level counter is incremented by one. The repeated block is interruptible.

This instruction can be nested. Four levels of block repeat can be used. The block-repeat minimum length is two words.

Restrictions when using `bkrep`:

1. The `break` instruction cannot start at the last address of the block repeat loop.
2. The following instructions cannot start at addr – 1 of the block repeat loop: `mov soperand, icr; mov icr,` ab.
3. The following instructions cannot start at the last two addresses of the block repeat loop: instructions with `lc` or `icr` as a source, `br`, `brr, call, callr, calla, ret, reti, rets, retd, retid, bkrep`, and `rep` instructions with pc or `lc` as a destination.

BKREP **Block Repeat**

Description (Cont.)

4. The `mov soperand, icr` instruction and instructions with lc as a destination cannot start at addr – 2 of the block repeat loop.
5. The following instructions cannot start at the addr – 3 of the block repeat loop: `set/rst/chng/addv/subv` with lc as a destination.
6. Notice that illegal instruction sequences are also restricted as the last and first instructions of a block-repeat loop.
7. Two block-repeats cannot have the same last address.

Operation

```
operand → lc
1 → LP status bit
BCx + 1 → BCx

   operand:     #unsigned short immediate¹
                REG²,³
```

1. When using an unsigned short immediate operand, the number of repetitions is between 1 to 256. When transferring the `#unsigned short immediate` number into the lc register, it is copied to the low-order eight bits of the lc. The high-order eight bits are cleared.
2. In the block-repeat outer level, the REG cannot be ai, bi, or p. In other block-repeat levels, the REG cannot be ai, bi, p, or lc. Note: The assembler cannot check the restriction on the lc register in a nested block-repeat.
3. The data read while reading lc during a block repeat loop execution is of the loop counter. If the outer block repeat loop has finished normally the contents of lc is 0; if it was finished using break, the contents of lc will be the value of the loop counter at the breakpoint.

Flags affected This instruction does not affect the flags.

BKREP Block Repeat

Cycles

	Cycles	Words
Register	2	2
short immediate	2	2

Example

```
bkrep #num1-1, >%main
.
.
   bkrep #num2-1, >%inner
         moda clr a0
         .
         .
   %inner: mov a0h, (r0)+
   .
   .
%main: modr (r0)+
```

The outer loop that ends with label %main is executed num1 times while the inner loop that ends with label %inner is executed num2 times. These nested loops are executed without any overhead in wrapping around from the last statement of the loops to the first statement of the loops.

BR Conditional Branch

Opcode

	15 6	5 4	3 0
br	0100000110	xx	cond

Syntax

```
br address [, cond]
```

Description

Branch to the specified address, if cond is met. If condition is met, `address` is the address/label of the new program memory location. The address is the second word of the instruction.

Operation

```
If condition then
   address → pc
```

If condition is met, branch to the program memory location specified by `address`.

Flags affected

This instruction does not affect the flags.

Cycles

	Cycles	**Words**
br	2 (if no branch) 3 (if branch occurs)	2

Example

```
mov @var, a0
br Process, eq
```

The flags are set according to the contents of variable located at `var` by the `mov @var, a0` instruction. If `Z = 1` in the ST0 Status Register, the branch is executed. Otherwise, the instruction following the branch instruction is executed.

BREAK — Break from Block Repeat

Opcode

	15 … 6	5 … 0
break	1101001111	xxxxxx

Syntax `break`

Description This instruction is used for breaking out of the current block-repeat loop. The internal registers which contain the first address, last address and loop-counter are popped.

The `break` instruction cannot be the last instruction of a block-repeat loop.

A break at the outer level does not change `lc` and resets LP bit.

Operation None.

Flags affected This instruction does not affect the flags.

Cycles

	Cycles	Words
Break	1	1

Example

```
bkrep #num, <%main
    .
    tsb @var, #15
    brr <%continue, eq
    break
    brr %leaveloop
    .
%continue:
    .
    .
%main:
    .
%leaveloop:
```

Bit 15 of the variable located at label var is tested. If the bit is not clear, the loop is terminated. If it is cleared, the loop continues.

BRR Relative Conditional Branch

Opcode

	15 11	10 4	3 0
brr	01010	offset	cond

Syntax `brr offset [, cond]`

Description If condition is met, a branch is executed to the program memory location: address of the `brr` instruction + offset + 1. The offset range is -63 to +64. (Offset range is offset +1.)

Operation

```
If condition then
   'address of brr inst.' + offset + 1 → pc

Assembler syntax:
   brr offset [,cond]
or
   brr $-offset [,cond]
or
   brr label [,cond]
```

When `label` is the new program memory location, the instruction word includes the “relative address” calculated by the assembler as follows: (label address) – (`brr` address) – 1.

Flags affected This instruction does not affect the flags.

Cycles

	Cycles	Words
brr	2	1

Example This example tests bit 2 of the variable var. The branch instruction will cause a jump back to this test until the bit is cleared.

```
%init: tstb @var, #2
    brr <%init, eq
```

CALL Conditional Call Subroutine

Opcode

	15 6	5 4	3 0
call	0100000111	xx	cond

Syntax `call address [,cond]`

Description If condition is met, address is the address/label of the new program memory location. The address is the second word of the instruction.

Operation

```
If condition then
   sp – 1 → sp
   pc → (sp)
   address → pc
```

If the condition is met, the stack pointer (sp) is predecremented, the program counter (pc) is pushed into the software stack, and a branch is performed to the program memory location specified by address.

Flags affected This instruction does not affect the flags.

Cycles

	Cycles	Words
call	2 (if condition not met) 3 (if condition met)	2

Example The code segment `ini` of the routine `subroutine` is called within the `main` loop.

```
bkrep #sample, >%main
   .
   call subroutine.ini
   .
%main:
```

CALLA Call Subroutine at Location Specified by the ai Accumulator

Opcode

	15 9	8	7	6 5	4	3 0
calla	1101010	i	1	xx	0	xxxx

Syntax `calla ail`

Description Call subroutine indirect (address from ai1). The stack pointer (sp) is predecremented. The program counter (pc) is pushed onto the software stack, and a branch is executed to the address pointed to by accumulator-low.

This instruction can be used to perform computed subroutine calls.

Operation

```
sp – 1 → sp
pc → sp
ail → pc
```

Flags affected This instruction does not affect the flags.

Cycles

	Cycles	**Words**
calla	3	1

Example In this example, the program address contained in a0l is called within the `main` loop.

```
bkrep #sample, >%main
   .
   calla a0l
.
.
%main:
```

CALLR Relative Conditional Call Subroutine

Opcode

	15 11	10 4	3 0
callr	00010	offset	cond

Syntax `callr offset [,cond]`

Description If the condition is met, the stack pointer (sp) is predecremented, the program counter (pc) is pushed on to the software stack and a branch is executed to the program memory location (the `callr` instruction + `offset` + 1). The offset range is between -63 to +64, defined as `offset` + 1.

Operation

```
If condition then
   sp – 1 → sp
   pc → (sp
   address → pc

Assembler syntax:
   callr $+offset [,cond]
or
   callr $-offset [,cond]
or
   callr label [,cond]
```

New execution address is label. The instruction word includes the “relative address” calculated by the assembler as follows: (label address) – (`callr` address) – 1.

Flags This instruction does not affect the flags.

CALLR Relative Conditional Call Subroutine

Cycles

	Cycles	Words
callr	2	1

Example

In this example, code segment ini is called within the main loop if register r0 has not been decremented to zero (R flag clear).

```
bkrep #sample, >%main
   .
   modr (r0)-
   callr >%ini, nr
   .
%main:
.
.
.
%ini:
.
.
ret
```

CHNG Change Bit Field

Opcode

	15 12	11 9	8	7 0
Short Direct	1110	010	1	direct

	15 12	11 9	8 5	4 3	2 0
Indirect	1000	010	0111	mod	rN

	15 12	11 9	8 5	4 0
Register	1000	010	1111	REG

Syntax

`chng ##long immediate, operand`

Description

Change specific bit-field in a 16-bit operand according to a long immediate value. The long immediate value contains ones in the bit-field location.

If the operand is not part of an accumulator (`ail`, `aih`, `aie`, `bil`, or `bih`) then the accumulators are unaffected. If the operand is part of an accumulator, only the addressed part is affected.

The operand and the long immediate values are sign-extension suppressed.

Operation

```
operand XOR ##long immediate → operand

operand = REG[1]
          (rN)
          short direct address
```

1. The REG cannot be ai, bi, or p.

CHNG **Change Bit Field**

Flags affected **When the operand is not st0**:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

When the operand is st0:

The specified bits are changed according to the bit-field in the long immediate value, regardless whether or not the A0E bits have changed.

When changing the A0E bits (chng `##long immediate, st0`) the flags are affected according to the long immediate value. When changing the A1E bits (chng `##long immediate, st1`), the flags are updated according to the ALU output.

Cycles

	Cycles	**Words**
Short Direct	2	2
Indirect	2	2
Register	2	2

Example chng `##0x7FFF, a0l`

Before Execution:

a0:	0	FFFF	1011

After Execution:

a0:	0	FFFF	6FEE

CLR Clear Accumulator

Opcode

	15 13	12	11 8	7 4	3 0
clr ai	011	i	0111	0110	cond

	15 13	12	11 8	7	6 4	3 0
clr bi	011	i	0111	x	110	cond

Syntax

```
clr ai [, cond] or
clr bi [, cond]
```

Description

Clear specified accumulator to zero. Also refer to the moda and modb instructions on pages 7-92 and 7-96.

Operation

```
clr ai:   ai = 0
clr bi:   bi = 0
```

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
clr	1	1

Example

```
clr a0, true
```

Before Execution:

a1:	0	0013	3A05

After Execution:

a1:	0	0000	0000

CLRR — Clear and Round A-Accumulator

Opcode

	15 13	12	11 8	7 4	3 0
clrr	011	i	0111	1100	cond

Syntax

`clrr ai [, cond]`

Description

Refer to the moda instruction on page 7-92 for complete descriptions of this instruction.

Operation

`ai = 0x8000`

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
clrr	1	1

Example

In this example, assume the M flag in ST0 is set.

`clrr a1, lt`

Before Execution:

a1:	0	0013	3A05

After Execution:

a1:	0	0000	8000

CMP Compare

Opcode

	15 13	12 9	8	7 0
Short Direct	101	0110	i	direct

	15 13	12 9	8	7 3	2 0
Long Direct (MSW)	110	1010	i	11111	110

	15 0
Long Direct (LSW)	long direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	0110	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	0110	i	101	REG

	15 12	11 9	8	7 0
Short Immediate	1100	110	i	short immediate

	15 12	11 9	8	7 5	4 0
Long Immediate (MSW)	1000	110	i	110	xxxxx

	15 0
Long Immediate (LSW)	long immediate

	15 12	11 9	8	7	6 0
Short Index	0100	110	i	0	Offset

	15 12	11 9	8	7 3	2 0
Long Index (MSW)	1101	010	i	11011	110

	15 0
Long Index (LSW)	long index

CMP Compare

Syntax `cmp operand, ai`

Description The contents of ai are compared with the contents of operand and the appropriate flags are set accordingly. Both ai and operand are treated as signed numbers. The operand is LSB adjusted and the comparison is made against ai.

Operation

```
ai - operand

operand = REG[1]
          (rN)
          short direct address
          [##direct address]
          #unsigned short immediate
          ##long immediate
          (rb + offset 7)
          (rb+##offset)
```

[1] The REG cannot be bi.

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	**Words**
Short Direct	1	1
Long Direct	2	2
Indirect	1	1
Register	1	1
Short Immediate	1	1
Long Immediate	2	2
Short Index	1	1
Long Index	2	2

CMP Compare

Example `cmp #8, a0`

Before Execution:

a0:	0	0820	FFFF

After Execution:

a0:	0	0820	FFFF

Z, M, N, V, C bits cleared, E bit set.

CMPU Compare Unsigned

Opcode

	15 13	12 9	8	7 0
Short Direct	101	1111	i	direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	1111	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	1111	i	101	REG

Syntax

`cmpu operand, ai`

Description

The contents of operand are compared with ai[15:0] with sign bits being ignored, and the flags are set accordingly. The sign extension of the operand is suppressed for this operation.

Operation

```
ai - operand

operand = REG¹
          (rN)
          short direct address
```

1. The REG cannot be bi, ai, or p.

In order to compare ai with an unsigned 16-bit operand, ai[35:16] should be cleared using either the `mov` → `ail` instruction or other instructions).

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

CMPU Compare Unsigned

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1

Example

```
cmpu r4, a0
```

Before Execution:

a0:	0	0000	3A05	r4:	4000

After Execution:

a0:	0	0000	3A05	r4:	4000

The Z, N, V, E bits are cleared, M and C bits are set.

CMPV — Compare Long Immediate Value to a Register or a Data Memory Location

Opcode

	15 12	11 9	8	7 0
Short Direct	1110	110	1	direct

	15 12	11 9	8 5	4 3	2 0
Indirect	1000	110	0111	mod	rN

	15 12	11 9	8 5	4 0
Register	1000	110	1111	REG

Syntax

`cmpv ##long immediate, operand`

Description

The contents of long immediate value are compared with operand and the flags updated accordingly. The contents of the operand are unaffected after the operation. The operand and long immediate values are sign-extended prior to the comparison.

Operation

```
operand - ##long immediate value

operand = REG[1]
          (rN)
          short direct address
```

1. The REG cannot be bi, ai, p, or pc.

Note that ai can be used in the `cmp ##long immediate, ai` instruction.

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Z, M, and C are the results of the 16-bit operation. M is affected by bit 15.

Note that the flags are set differently for the `subv ##long immediate, st0` and `cmpv ##long immediate, st0` instructions.

CMPV Compare Long Immediate Value to a Register or a Data Memory Location

Cycles

	Cycles	Words
Short Direct	2	2
Indirect	2	2
Register	2	2

Example

```
cmpv ##0x004F, b0h
```

Before Execution:

b0:	0	0013	3A05

After Execution:

b0:	0	0013	3A05

The M and C bits are set, and the Z bit cleared.

CNTX Context Switching Store or Restore

Opcode

	15 6	5	4	3 0
cntx (store)	1101001110	x	0	xxxx

	15 6	5	4	3 0
cntx (restore)	1101001110	x	1	xxxx

Syntax `cntx s | r`

Description This instruction activates the context switching mechanism.

Operation s - store the shadow/swap bits and swap a1 and b1 accumulators contents:

- The st0[0], st0[11:2], st1[11:10], and st2[7:0] bits are pushed to their shadow bits.
- The page bits st1[7:0] are swapped with their alternative register.
- a1 is transferred into b1, b1 is transferred into a1.

r - restore the shadow/swap bits and swap a1 and b1 accumulators contents:

- The st0[0], st9[11:2], st1[11:10], and st2[7:0] bits are popped from their shadow bits.
- The page bits st1[7:0] are swapped with their alternative register.
- a1 is transferred into b1, b1 is transferred into a1.

CNTX — Context Switching Store or Restore

Flags affected In a store, flags represent the data transferred into a1:

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

In a restore, flags are restored from their shadow bits:

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
cntx	1	1

Example `cntx s`

Before Execution:

a1:	0	0000	0000
b1:	0	0007	00E0
st0:	0E20		
st1:	0B00		
st2:	1000		

CNTX Context Switching Store or Restore

Example **After Execution:**

a1:	0	0007	00E0	b1:	0	0000	0000
st0:	0A20			st0s:	0E20		
st1:	0B00			st1s:	0002		
st2:	1000			st2s:	0000		

COPY Copy Other A Accumulator

Opcode

	15 13	12	11 8	7 4	3 0
copy	011	i	0111	1111	cond

Syntax

`copy ai [, cond]`

Description

Copy from contents of other A accumulator. Also see the moda instruction on page 7-92.

Operation

$ai = a\bar{i}$

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
copy	1	1

Example

`copy a0, eq`

(assume Z bit is set before execution of the instruction)

Before Execution:

a0:	0	0780	0400	a1:	0	0007	00E0

After Execution:

a0:	0	0007	00E0	a1:	0	0007	00E0

Z, M, N, E are cleared.

DEC Decrement A Accumulator by One

Opcode

	15 13	12	11 8	7 4	3 0
dec	011	i	0111	1110	cond

Syntax `dec ai [, cond]`

Description Decrement `ai` accumulator by one if cond is true. Also see the `moda` instruction on page 7-92.

Operation `ai = ai - 1 [if cond=true]`

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
copy	1	1

Example `dec a, true`

Before Execution:

a0:	0	0000	0001

After Execution:

a0:	0	0000	0000

Z, N are set, M, V, C, E are cleared, L unaffected.

DINT

DINT **Disable Interrupt**

Opcode

	15 6	5 0
dint	0100001111	xxxxxx

Syntax `dint`

Description The IE bit is cleared, disabling interrupts.

Operation `0 → IE`

Flags affected This instruction has no affect on flags.

Cycles

	Cycles	**Words**
dint	1	1

DIVS Division Step

Opcode

	15 9	8	7 0
divs	0000111	i	direct

Syntax

`divs direct address, ai`

Description

This instruction performs a division step for division of ai by the contents of the direct address. The 16-bit dividend is stored in ail and the divisor is stored in the direct address. The aie and aih registers are overwritten during operation of the instruction. Each DIVS operation calculates one quotient bit using a nonrestoring fractional division algorithm. To produce an N-bit quotient, the division instruction is executed N times, where N is the number of bits of precision desired in the quotient ($0 \le N \le 16$). After the execution of the operation, the quotient is stored in ail and the remainder is stored in aih. Both the dividend and the divisor must be positive.

Operation

```
ai - (direct address * $2^{15}$) $\rightarrow$ALU output
   If ALU output < 0
         then   ai = ai * 2
         else   ai = ALU output * 2 + 1
```

The 16-bit dividend is placed at accumulator low; the accumulator high and the accumulator extension are cleared. The divisor is placed at the direct address.

For a 16-bit division, DIVS should be executed 16 times. After 16 times, the quotient is in the accumulator low and the remainder is in the accumulator high.

The dividend and the divisor must both be positive.

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

DIVS Division Step

Cycles

	Cycles	Words
divs	1	1

Example

```
rep #15
divs 0x0050, a0
```

Before Execution:

a0:	0	0000	3C04	0050:	3420

After Execution:

a0:	0	07E4	0001	0050:	3420

EINT Enable Interrupt

Opcode

	15 6	5 0
eint	0100001110	xxxxxx

Syntax `eint`

Description The IE bit is set, enabling interrupts.

Operation 1 → IE

Flags affected This instruction has no affect on flags.

Cycles

	Cycles	Words
eint	1	1

Example After the parameters are initialized, an interrupt is enabled by setting IE.

```
init:
    .
    mov ##Ptr, r4
    .
eint
```

EXP Evaluate the Exponent Value

Opcode

	15 9	8	7 6	5	4 3	2 0
Indirect ai	1001100	i	01	x	mod	rN

	15 9	8	7 5	4 0
Register ai	1001000	i	010	REG

	15 9	8	7 6	5	4 3	2 0
Indirect sv	1001110	x	01	x	mod	rN

	15 9	8	7 5	4 0
Register sv	1001010	x	010	REG

	15 9	8	7 5	4 1	0
bj, ai	1001000	i	011	xxxx	j

	15 9	8	7 5	4 1	0
bi, sv	1001010	x	011	xxxx	i

Syntax

`exp soperand [, ai]`

Description

Evaluate the exponent value of the operand. The operand can be either REG (one of the registers) or a data memory location. The exponent result, a signed six-bit value, is sign extended and written into the Shift Value register (sv) and optionally into one of the A-accumulators. The source operand (soperand) is unchanged.

The instruction following an exp instruction cannot move from/to the sv register. The sv register is be used in `shfc` and `movs` instructions or as a temporary general purpose register.

EXP Evaluate the Exponent Value

Operation

```
When using exp soperand:
   Exponent (soperand) → sv
   The soperand is unaffected.

When using exp soperand, ai:
   Exponent (soperand) → sv and ai
   The soperand is unaffected.

soperand: REG[1]
          (rN)
```

1. The REG cannot be p.

Flags

	15	12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E		Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

This instruction does not affect the flags.

Cycles

	Cycles	Words
Indirect, ai	1	1
Register, ai	1	1
Indirect, sv	1	1
Register, sv	1	1

Example

```
exp(r0)+, a0
```

Before Execution:

a0:	0	8000	0000	(r0):	0020

After Execution:

a0:	0	0000	0009	(r0):	0020

INC

Increment A Accumulator by One

Opcode

	15 13	12	8	7 4	3 0
inc	011	i	0111	1101	cond

Syntax

`inc ai [, cond]`

Description

Increments specified A accumulator. Also see the moda instruction on page 7-92.

Operation

`ai = ai + 1 [if cond=true]`

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
copy	1	1

Example

`inc a, true`

Before Execution::

a0:	0	0000	0040

After Execution::

a0:	0	0000	0041

Z, N, M, V, C, E are cleared, L unaffected.

LIM Limit ai Accumulator

Opcode

	15 – 6	5 – 4	3 – 0
lim a0	0100100111	00	xxxx
lim a0, a1	0100100111	01	xxxx
lim a1, a0	0100100111	10	xxxx
lim a1	0100100111	11	xxxx

Syntax

`lim ai[,aī]`

Description

This instruction saturates `ai` accumulator with maximum positive or minimum negative value when the value in the accumulator exceeds maximum or minimum representable value.

Operation

When using `lim ai`:

```
   if ai > 0x07FFFFFF then
         ai = 0x07FFFFFF
   else
         if ai < 0xF80000000 then
         ai = 0xF80000000
   else
         ai is unaffected

When using lim ai, aī:
         ai is unaffected
         if aī ≥      0x07FFFFFF then
         ai = 0x07FFFFFF
   else
   if ai ≤ 0xF80000000 then
         aī = 0xF80000000
   else
         aī = ai
```

LIM Limit ai Accumulator

Flags affected

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT
							0						

Note: L flag is set when limitation occurs.

Cycles

	Cycles	Words
lim	1	1

Example

```
lim a0
```

Before Execution:

a0:	0	8490	0000

After Execution:

a0:	0	7FFF	FFFF

LOAD Load Specific Fields into Registers

Opcode

	15 8	7 0
load page	00000100	short immediate

	15 12	11	10 9	8 0
load modi	0000	0	01	short immediate

	15 12	11	10 9	8 0
load modj	0000	1	01	short immediate

	15 11	10	9 7	6 0
load stepi	11011	0	111	short immediate

	15 11	10	9 7	6 0
load stepj	11011	1	111	short immediate

	15 7	6 2	1 0
load ps	010011011	xxxxx	immed

Syntax

`load #immediate, operand`

Description

Load designated portion with immediate value.

Operation

```
operand:  #unsigned immediate 8 bit, page
                The page bits, the low-order 8 bits of
                st1, are loaded with an 8-bit constant
                (0 to 255).

          #unsigned immediate 9 bit, modi
          #unsigned immediate 9 bit, modj
                The modX bits, the high-order 9 bits of
                cfgi are loaded with a 9-bit constant
                (0 to 511 which means 1 to 512 modulo
                options).
```

LOAD **Load Specific Fields into Registers**

Operation (Cont.)

```
#immediate 7 bit, stepi
#immediate 7 bit, stepj
      The stepX bits, the low-order 7 bits of
      cfgi, are loaded with a 7-bit constant.
#unsigned immediate 2-bit, ps
      The ps status bits, bits 10 and 11 of ST1
      are loaded with a two-bit constant. Refer
      to Figure 4.10 Status Register 1 (ST1)
      for the encoding of the ps bits.
```

The assembler syntax permits the use of `lpg #unsigned short immediate`, which is equivalent to `load #unsigned short immediate, page`. The page bits (the low-order eight bits of ST1) are loaded with an eight-bit constant (0 to 255).

Flags affected This instruction does not affect the flags.

Cycles

	Cycles	Words
Load	1	1

Example Load integer value 4 into stepi in cfgi register.

`load #4, stepi`

Before Execution:

cfgi: 0020

After Execution:

cfgi: 0004

LPG Load the Page Bits

Opcode

	15 … 8	7 … 0
lpg	00000100	short immediate

Syntax `lpg #unsigned short immediate`

Description Refer to the `load` instruction for a complete description of this instruction.

MAA Multiply and Accumulate Aligned Previous Product

Opcode

	15 12	11	10 9	8	7 0
Y Direct	1110	i	10	0	direct

	15 12	11	10 8	7 5	4 3	2 0
Y Indirect	1000	i	100	001	mod	rN

	15 12	11	10 8	7 5	4 0
Y Register	1000	i	100	010	REG

	15 12	11	10 8	7 5	4 3	2 0
Indirect Long Immediate (MSW)	1000	i	100	000	mod	rN

	15 0
Indirect Long Immediate (LSW)	long immediate

	15 12	11	10 8	7	6 5	4 3	2	1 0
(rJ), (rI)	1101	i	100	0	jj	ii	w	qq

Syntax

```
maa operand1, operand2, ai
```

Description

The previous product (p) is sign-extended to 36 bits and shifted as defined in the PS field in ST1. The shifted value is aligned with sign-extension 16 bits to the right and is added to ai. The result is stored in ai. The signed operands are multiplied together and stored in p.

Operation

```
ai + aligned & shifted p → ai
operand1 → y[1]
operand2 → x
signed y * signed x → p

operand1, operand2:     y, short direct address
                        y, (rN)
                        y, REG[2]
                        (rJ), (rI)[3]
                        (rN), ##long immediate
```

1. y → y means that y retains its value.
2. The REG cannot be ai, bi, or p.
3. The multiplication in maa (rJ), (rI), ai is between Y-Memory and X-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

MAA Multiply and Accumulate Aligned Previous Product

Flags

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

Example

```
maa (r4)+, (r0)-, a1
```

Before Execution:

(r4):	FF20	(r0):	0200

a1:	0	4350	FFE3
p:	2345	6789	

After Execution:

a1:	0	4351	2328
p:	FFFE	4000	

MAASU — Multiply Signed by Unsigned and Accumulate Aligned Previous Product

Opcode

	15 12	11	10 8	7 5	4 3	2 0
Y Indirect	1000	i	111	001	mod	rN

	15 12	11	10 8	7 5	4 0
Y Register	1000	i	111	010	REG

	15 12	11	10 8	7 5	4 3	2 0
Indirect Long Immediate (MSW)	1000	i	111	000	mod	rN

	15 0
Indirect Long Immediate (LSW)	long immediate

	15 12	11	10 8	7	6 5	4 3	2	1 0
(rJ), (rI)	1101	i	111	0	jj	ii	w	qq

Syntax

```
maasu operand1, operand2, ai
```

Description

The previous product (p) is sign-extended to 36 bits and shifted as defined in the PS field in ST1. The shifted value is aligned with sign-extension 16 bits to the right and is added to ai. The result is stored in ai. The signed operand1 is multiplied with the unsigned operand2, and the result is stored in p.

Operation

```
ai + aligned & shifted p → ai
operand1 → y[1]
operand2 → x
signed y * unsigned x → p

operand1, operand2:     y, (rN)
                        y, REG[2]
                        (rJ), (rI)[3]
                        (rN), ##long immediate
```

1. y → y means that y retains its value.
2. The REG cannot be ai, bi, or p.
3. The multiplication in maasu (rJ), (rI), ai is between X-Memory and Y-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

MAASU Multiply Signed by Unsigned and Accumulate Aligned Previous Product

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Indirect	1	1
Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

Example

```
maasu (r4)+, (r0)-, a1
```

Before Execution:

(r4):	FF20		
(r0):	0200		
a1:	0	4350	FFE3
p:	2345	6789	

After Execution:

a1:	0	4351	466D
p:	FFFE	4000	

MAC Multiply and Accumulate Previous Product

Opcode

Y Direct

15 12	11	10 9	8	7 0
1110	i	01	0	direct

Y Indirect

15 12	11	10 8	7 5	4 3	2 0
1000	i	010	001	mod	rN

Y Register

15 12	11	10 8	7 5	4 0
1000	i	010	010	REG

Long Immediate (MSW)

15 12	11	10 8	7 5	4 3	2 0
1000	i	010	000	mod	rN

Long Immediate (LSW)

15 0
long immediate

(rJ), (rI)

15 12	11	10 8	7	6 5	4 3	2	1 0
1101	i	010	0	jj	ii	w	qq

Syntax

`mac operand1, operand2, ai`

Description

The previous product (p) is sign-extended to 36 bits and shifted as defined in the PS field in ST1. The shifted value is added to ai. The result is stored in ai. The signed operands are multiplied together and stored in p.

Operation

```
ai + shifted p → ai
operand1 → y[1]
operand2 → x
signed y * signed x → p
```

```
operand1, operand2:    y, short direct address
                       y, (rN)
                       y, REG[2]
                       (rJ), (rI)[3]
                       (rN), ##long immediate
```

1. y → y means that y retains its value
2. The REG cannot be ai, bi, or p.
3. The multiplication in mac (rJ), (rI), ai is between Y-Memory and X-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

MAC Multiply and Accumulate Previous Product

Flags

	15	12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E		Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

Example

```
mac (r4)+, (r0)-, a1
```

Before Execution:

(r4):	01AD			(r0):	1980	
a1:	0	4304	1768	p:	0000	8000

After Execution:

p:	002A	BB80	
a1:	0	42C8	0CB4

MACSU — Multiply Signed by Unsigned and Accumulate Previous Product

Opcode

	15 12	11	10 9	8	7 0
Y Direct	1110	i	11	0	direct

	15 12	11	10 8	7 5	4 3	2 0
Y Indirect	1000	i	110	001	mod	rN

	15 12	11	10 8	7 5	4 0
Y Register	1000	i	110	010	REG

	15 12	11	10 8	7 5	4 3	2 0
Long Immediate (MSW)	1000	i	110	000	mod	rN

	15 0
Long Immediate (LSW)	long immediate

	15 12	11	10 8	7	6 5	4 3	2	1 0
(rJ), (rI)	1101	i	110	0	jj	ii	w	qq

Syntax

```
macsu operand1, operand2, ai
```

Description

The previous product (p) is sign-extended to 36 bits and shifted as defined in the PS field in ST1. The shifted value is added to ai. The result is stored in ai. The signed operand1 is multiplied with the unsigned operand2, and the result is stored in p.

MACSU Multiply Signed by Unsigned and Accumulate Previous Product

Operation

ai + shifted p → ai
operand1 → y[1]
operand2 → x
signed y * unsigned x → p

operand1, operand2: y, short direct address
y, (rN)
y, REG[2]
(rJ), (rI)[3]
(rN), ##long immediate

1. y → y means that y retains its value.
2. The REG cannot be ai, bi, or p.
3. The multiplication in macsu (rJ), (rI), ai is between Y-Memory and X-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

MACSU Multiply Signed by Unsigned and Accumulate Previous Product

Example `macsu (r4)+, (r0)-, a1`

Before Execution:

(r4):	FF26			(r0):	2341	
a1:	0	4303	9768	p:	0000	4000

After Execution:

p:	FFE1	FAA6	
a1:	0	4304	1768

MACUS — Multiply Unsigned by Signed and Accumulate Previous Product

Opcode

Y Indirect

15–12	11	10–8	7–5	4–3	2–0
1000	i	011	001	mod	rN

Y Register

15–12	11	10–8	7–5	4–0
1000	i	011	010	REG

Long Immediate (MSW)

15–12	11	10–8	7–5	4–3	2–0
1000	i	011	000	mod	rN

Long Immediate (LSW)

15–0
long immediate

(rJ),(rI)

15–12	11	10–8	7	6–5	4–3	2	1–0
1101	i	011	0	jj	ii	w	qq

Syntax

`macus operand1, operand2, ai`

Description

The previous product (p) is sign-extended to 36 bits and shifted as defined in the PS field in ST1. The shifted value is added to ai. The result is stored in ai. The unsigned operand1 is multiplied with the signed operand2, and the result is stored in p.

Operation

```
ai + shifted p → ai
operand1 → y[1]
operand2 → x
unsigned y * signed x → p

operand1, operand2:     y, (rN)
                        y, REG[2]
                        (rJ), (rI)[3]
                        (rN), ##long immediate
```

1. y → y means that y retains its value.
2. The REG cannot be ai, bi, or p.
3. The multiplication in macus (rJ), (rI), ai is between Y-Memory and X-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

MACUS — Multiply Unsigned by Signed and Accumulate Previous Product

Flags

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Indirect	1	1
Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

Example

```
macus (r4)+, (r1)-, a1
```

Before Execution:

(r4):	FF26		
(r0):	2341		
a1:	0	42C8	0CB4
p:	002A	BB80	

After Execution:

a1:	0	431D	83B4
p:	2322	FAA6	

MACUU Multiply Unsigned by Unsigned and Accumulate Previous Product

Opcode

	15 12	11	10 8	7 5	4 3	2 0
Y Indirect	1000	i	101	001	mod	rN

	15 12	11	10 8	7 5	4 0
Y Register	1000	i	101	010	REG

	15 12	11	10 8	7 5	4 3	2 0
Long Immediate (MSW)	1000	i	101	000	mod	rN

	15 0
Long Immediate (LSW)	long immediate

	15 12	11	10 8	7	6 5	4 3	2	1 0
(rJ), (rI)	1101	i	101	0	jj	ii	w	qq

Syntax

`macuu operand1, operand2, ai`

Description

The previous product (p) is sign-extended to 36 bits and shifted as defined in the PS field in ST1. The shifted value is added to ai. The result is stored in ai. The unsigned operands are multiplied together, and the result is stored in p.

After using this instruction, the p register cannot be reconstructed. During an interrupt service routine that uses the p register, the p register should be saved before it is used and restored before returning from the interrupt. However, the p register cannot be reconstructed after a macuu instruction; it is, therefore, recommended that a `dint` instruction be put before the macuu instruction and an `eint` instruction be placed after the instruction which uses the result in the p register (the unsigned product).

The instruction that uses the p register or the 'shifted p register' as a source operand after a macuu instruction uses the unsigned result in p zero extended into 36 bits, and then shifted as defined in the PS field. This behavior is in effect until a new signed product is generated or a new value is written to ph.

MACUU Multiply Unsigned by Unsigned and Accumulate Previous Product

Operation

```
ai + shifted p → ai
operand1 → y[1]
operand2 → x
unsigned y * unsigned x → p

operand1, operand2:     y, (rN)
                        y, REG[2]
                        (rJ), (rI)[3]
                        (rN), ##long immediate
```

1. y → y means that y retains its value.
2. The REG cannot be ai, bi, or p.
3. The multiplication in macuu (rJ), (rI), ai is between X-Memory and Y-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

Flags

	15	12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E		Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Indirect	1	1
Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

MACUU **Multiply Unsigned by Unsigned and Accumulate Previous Product**

Example `macuu (r4)+, (r0)-, a1`

Before Execution:

(r4):	FF20			(r0):	0200	
a1:	0	431D	83B4	p:	2322	FAA6

After Execution:

a1:	0	8963	7900	p:	01FE	4000

MAX — Maximum of Two ai Accumulators

Opcode

	15 – 10	9	8	7 – 5	4 – 3	2 – 0
max (ge)	100001	0	i	011	mod	xxx

	15 – 10	9	8	7 – 5	4 – 3	2 – 0
max (gt)	100001	1	i	011	mod	xxx

Syntax

`max ai, (r0), ge | gt`

Description

This instruction is used to find the maximal value of the two A accumulators. If a new maximal value is found, this value is saved in the defined accumulator (ai) and the r0 pointer value is saved in the MIXP register. (Note that the MIXP cannot be read by the instruction following the `max` instruction). The r0 register is postmodified as specified in the instruction, regardless of the result of the comparison.

Operation

When using ge:

```
If aī ≥ ai then
      ai = aī
      mixp = r0
r0 is postmodified
```

When using gt:

```
If aī > ai then
      ai = aī
      mixp = r0
r0 is postmodified
```

Flags affected

	15 – 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

MAX — Maximum of Two ai Accumulators

Cycles

	Cycles	Words
Max	1	1

Example

```
max a0, (r0), ge
```

Before Execution:

a0:	0	0000	001B	r0	FF00
a1:	0	040B	2B52	(r0):	0100

After Execution: (assuming M=0)

a0:	0	040B	2B52	mixp:	FF00

MAXD — Maximum of Data Memory Location and ai Accumulator

Opcode

	15 10	9	8	7 5	4 3	2 0
maxd (ge)	100000	0	i	011	mod	xxx

	15 10	9	8	7 5	4 3	2 0
maxd (gt)	100000	1	i	011	mod	xxx

Syntax

`maxd ai, (r0), ge | gt`

Description

This instruction is used for finding the maximal value of a data memory location pointed to by r0 and one of the A accumulators. In case r0 points to a larger or equal value than the accumulator, the new maximal value is transferred to the defined accumulator (ai) and the r0 pointer value is transferred into the mixp register. (Note that the mixp cannot be read in the instruction following the `maxd` instruction). The r0 register is postmodified as specified in the instruction, regardless of whether the new maximal value is updated.

Operation

When using ge:

```
if (r0) ≥ ai then
      ai = (r0)
      mixp = r0
r0 is postmodified
```

When using gt:

```
if (r0) > ai then
      ai = (r0)
      mixp = r0
r0 is postmodified
```

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Note: M is set when the max value was found and the accumulator and mixp register were updated; cleared otherwise.

MAXD Maximum of Data Memory Location and ai Accumulator

Cycles

	Cycles	Words
maxd	1	1

Example

This example searches for the maximum value in a data array pointed to by r0, stores the maximum value in a0, and stores its pointer in mixp.

```
moda clr, a0
rep #7
   maxd a0, (r0)+, ge
nop
mov mixp, r1
```

The array is as follows:

Array Address	Contents
FF00	0x0240
FF01	0x0053
FF02	0xAACC
FF03	0x08C7
FF04	0x1CCC
FF05	0x0020
FF06	0x0100
FF07	0x2540

Before Execution:

a0:	0	0000	0000	r0:	FF00

After Execution:

a0:	0	0000	2540	mixp:	FF07

MIN — Minimum of Two ai Accumulators

Opcode

	15 11	10	9	8	7 5	4 3	2 0
min (le)	10001	x	0	i	011	mod	xxx

	15 11	10	9	8	7 5	4 3	2 0
min (lt)	10001	x	1	i	011	mod	xxx

Syntax

```
min ai, (r0), le | lt
```

Description

This instruction is used to find the minimal value of the two A accumulators. If a new minimal value is found the new value is saved in the defined accumulator ai and the r0 pointer value saved in the mixp register. The r0 register is postmodified as specified at the instruction, regardless of the result of the comparison of the accumulators.

Operation

When using le:

```
If aī ≤ ai then
      ai = aī
      mixp = r0
r0 is postmodified
```

When using lt:

```
If aī < ai then
      ai = aī
      mixp = r0
r0 is postmodified
```

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Note: M is set when the min value was found and the accumulator and mixp register were updated; cleared otherwise.

MIN Minimum of Two ai Accumulators

Cycles

	Cycles	Words
min	1	1

Example

```
min a0, (r0), lt
```

Before Execution:

a0:	0	0000	3907	a1:	0	5128	0338

After Execution: (assuming M=1)

a0:	0	0000	3907	a1:	0	5128	0338

In this example, a0 was the minimum value, so no changes occurred.

MODA — Modify ai Accumulator Conditionally

Opcode

	15 13	12	8	7 4	3 0
moda	011	i	0111	func	cond

func:	0000	shr
	0001	shr4
	0010	shl
	0011	shl4
	0100	ror
	0101	rol
	0110	clr
	0111	reserved
	1000	not
	1001	neg
	1010	rnd
	1011	pacr
	1100	clrr
	1101	inc
	1110	dec
	1111	copy

Syntax

`moda func, ai [,cond]`

The assembler syntax permits omitting the moda; for example, `shr a0` is equivalent to `moda shr, a0`.

Description

The contents of ai are modified according to `func`, and the flags are set accordingly. If cond is specified, `ai` is modified only when the condition is true and is unaffected when false.

MODA Modify ai Accumulator Conditionally

Operation

```
func:    shr    ai = ai >> 1
         shl    ai = ai << 1
         shr4   ai = ai >> 4
         shl4   ai = ai << 4
         ror    rotate ai right through carry
         rol    rotate ai left through carry
         clr    ai = 0
         copy   ai = aī
         neg    ai = -ai
         not    ai = not (ai)
         rnd    Round upper 20 bits of the ai
                ai = ai+0x8000
         pacr   ai = shifted p[1] + 0x8000
         clrr   ai = 0x8000
         inc    ai = ai + 1
         dec    ai = ai - 1
```

[1] Shifted p register means that the p register is sign-extended to 36 bits and then shifted as defined in the PS field, status register ST1.

Flags affected

Arithmetic Shift: `shr`, `shl`, `shr4`, and `shl4`

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

An arithmetic shift is performed when the S status bit in status register ST2 is cleared.

The C flag is set according to the last bit shifted out of the operand (shr = bit 0, shr4 = bit 3, shl = bit 35, shl4 = bit 32).

For shl/shl4, the V flag is cleared if the operand being shifted could be represented in 35/31 bits for shl/shl4, respectively; set otherwise. For shr/shr4, the V flag is always cleared.

MODA Modify ai Accumulator Conditionally

Flags affected (cont.)

Logical Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	**Z**	**M**	**N**	V	**C**	**E**	L	R	IM1	IM0	IE	SAT

A logical shift is performed when the S status bit in status register ST2 is set.

The C flag is set according to the last bit shifted out of the operand (shr = bit 0, shr4 = bit 3, shl = bit 35, shl4 = bit 32).

`ror`

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	**Z**	**M**	**N**	V	**C**	**E**	L	R	IM1	IM0	IE	SAT

C- Set according to the LSB (bit 0) shifted out of the operand.

`rol`

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	**Z**	**M**	**N**	V	**C**	**E**	L	R	IM1	IM0	IE	SAT

C- Set according to the MSB (bit 35) shifted out of the operand.

`not, copy, clr, clrr`

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	**Z**	**M**	**N**	V	C	**E**	L	R	IM1	IM0	IE	SAT

`neg, rnd, pacr`

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	**Z**	**M**	**N**	**V**	**C**	**E**	**L**	R	IM1	IM0	IE	SAT

`inc, dec`

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	**Z**	**M**	**N**	**V**	**C**	**E**	**L**	R	IM1	IM0	IE	SAT

MODA Modify ai Accumulator Conditionally

Cycles

	Cycles	Words
moda	1	1

Example

In this example, assume the M flag in ST0 is set.

```
moda neg, a1, lt
```

Before Execution:

a1:	0	0013	3A05

After Execution:

a1:	0	FFEC	C5FB

MODB — Modify bi Accumulator Conditionally

Opcode

	15 13	12	11 8	7	6 4	3 0
modb	011	i	1111	x	func	cond

func:		
	000	shr
	001	shr4
	010	shl
	011	shl4
	100	ror
	101	rol
	110	clr
	111	reserved

Syntax

```
modb func, bi [,cond] or func bX [, cond]
```

The assembler syntax permits omitting the `modb`; e.g. `shr b0` is equivalent to `modb shr, b0`.

Description

The contents of bi are modified according to `func`, and the flags are set accordingly. If cond is specified, `bi` is modified only when the condition is true and is unaffected when false. Refer to Section 7.1.2, “Number Representation,” and Section 4.4, “Status Registers,” for more information.

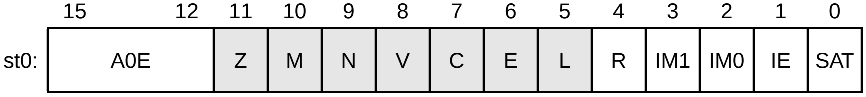
Operation

```
func:    shr        bi = bi >> 1
         shl        bi = bi << 1
         shr4       bi = bi >> 4
         shl4       bi = bi << 4
         ror        rotate bi right through carry
         rol        rotate bi left through carry
         clr        bi = 0
```

MODB Modify bi Accumulator Conditionally

Flags

Arithmetic Shift: `shr, shl, shr4, and shl4`



	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

An arithmetic shift is performed when the S status bit in status register ST2 is cleared.

The C flag is set according to the last bit shifted out of the operand (shr = bit 0, shr4 = bit 3, shl = bit 35, shl4 = bit 32).

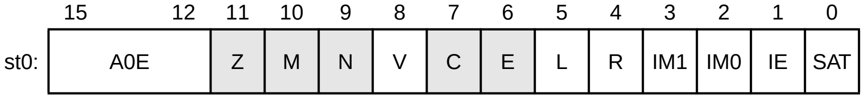
For shl and shl4, the V flag is cleared if the operand being shifted could be represented in 35/31 bits for shl/shl4, respectively; V flag is set otherwise.

The flags are always cleared with the `shr` and `shrf` instructions.

Logical Shift:

Logical shift is performed when the S status bit in status register ST2 is set.

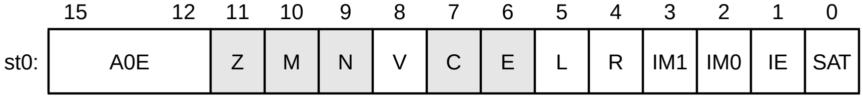
`shr/shr4/shl/shl4`



	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

The C flag is set according to the last bit shifted out of the operand (shr = bit 0, shr4 = bit 3, shl = bit 35, shl4 = bit 32).

Rotate right: `ror`



	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

The C flag is set according to the LSB (bit 0) shifted out of the operand.

MODB Modify bi Accumulator Conditionally

Flags (Cont.)

Rotate left: `rol`

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

The C flag is set according to the MSB (bit 35) shifted out of the operand

Clear: `clr`

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
modb	1	1

Example

In this example, assume the M flag in ST0 and the S flag in ST2 are set (logical shift mode selected).

```
modb neg, b1, lt
```

Before Execution:

b1:	0	4860	FE30

After Execution:

b1:	4	860F	E300

MODR Modify rN

Opcode

	15 7	6	5	4 3	2 0
modr (modulo enabled)	000000001	x	0	mod	rN

	15 7	6	5	4 3	2 0
modr (modulo disabled)	000000001	x	1	mod	rN

Syntax

```
modr (rN)[+|-|+s] [,dmod]
```

Description

The contents of rN are modified, and the flags are set accordingly. If dmod is specified, rN is modified based on the corresponding Mn bit.

Operation

```
When using modr (rN):
   rN is modified, influenced by the corresponding Mn bit.

When using modr (rN), dmod:
   rN is modified with modulo disabled.
```

This instruction can also be used for loop control, the R bit providing an indication that the end of the loop has been reached:

```
add:
   .
   modr (r0) -
   brr add, nr
```

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

R flag is set if the 16-bit rN register is zero; otherwise it is cleared.

Cycles

	Cycles	Words
Modr	1	1

MODR **Modify rN**

Example In this example, r3 is postincremented by one.

```
modr (r3)+
```

Before Execution:

r3:	FF00

After Execution:

r3:	FF01

MOV Move Data

Opcode

	15 10	9 5	4 0
register, register	010110	REG	reg

	15 12	11 10	9 6	5	4 3	2 0
ABL, dvm	1101	ABL	1010	x	11	xxx

	15 12	11 10	9 6	5	4 3	2 0
ABL, x	1101	ABL	1011	x	11	xxx

	15 12	11 10	9 7	6 5	4 3	2 0
ab, AB	1101	ab	101	AB	10	xxx

	15 6	5	4 0
register, mixp	0101111010	x	REG

	15 10	9 5	4 3	2 0
register, indirect	000110	REG	mod	rN

	15 5	4 0
mixp, register	01000111110	REG

	15 9	8	7	6 5	4 3	2	1 0
repc, ab	1101010	x	1	AB	10	x	00

	15 6	5	4 0
register, icr	0100111111	x	REG

	15 9	8	7	6 5	4 2	1 0
dvm, ab	1101010	x	1	AB	101	xx

MOV Move Data

Opcode (Cont.)

	15 9	8	7	6 5	4 3	2	1 0
icr, ab	1101010	x	1	AB	10	x	10

	15 9	8	7	6 5	4 3	2	1 0
x, ab	1101010	x	1	AB	10	x	11

	15 10	9 5	4 3	2 0
indirect, register	000111	REG	mod	rN

	15 5	4 0
(sp), register	01000111111	REG

	15 12	11 9	8	7 0
indirect, direct	0010	rN*	0	direct

	15 12	11	10	9	8	7 0
ail, direct	0011	1	i	0	0	direct

	15 12	11	10	9	8	7 0
aih, direct	0011	1	i	1	0	direct

	15 12	11	10	9	8	7 0
bil, direct	0011	0	i	0	0	direct

	15 12	11	10	9	8	7 0
bih, direct	0011	0	i	1	0	direct

	15 6	5	4 0
register, bi	010111011	i	REG

MOV Move Data

Opcode (Cont.)

indirect, bi

15 9	8	7 6	5	4 3	2 0
1001100	i	11	x	mod	rN

sv, direct

15 8	7 0
01111101	direct

direct, indirect

15 13	12 10	9 8	7 0
011	rN*	00	direct

direct, ai

15 13	12	11	10 8	7 0
011	1	i	001	direct

direct, bi

15 13	12	11	10 8	7 0
011	0	i	001	direct

direct, bil

15 13	12	11	10	9 8	7 0
011	0	i	0	10	direct

direct, ail

15 13	12	11	10	9 8	7 0
011	1	i	0	10	direct

direct, bih

15 13	12	11	10	9 8	7 0
011	0	i	1	10	direct

direct, aih

15 13	12	11	10	9 8	7 0
011	1	i	1	10	direct

direct, aih [eu]

15 13	12	11 8	7 0
011	i	0101	direct

direct, sv

15 8	7 0
01101101	direct

MOV Move Data

Opcode (Cont.)

	15 9	8	7 5	4 0
long immediate, bi (MSW)	0101111	i	001	xxxxx

	15 0
long immediate, bi (LSW)	long immediate

	15 9	8	7 5	4 0
long immediate, register (MSW)	0101111	x	000	REG

	15 0
long immediate, register (LSW)	long immediate

	15 9	8	7 2	1 0
long direct, ai (MSW)	1101010	i	101110	xx

	15 0
long direct, ai (LSW)	long direct

	15 13	12	11 8	7 0
short immediate, ail	001	i	0001	short immediate

	15 13	12	11 8	7 0
short immediate, aih	001	i	0101	short immediate

	15 13	12	11	10	9 8	7 0
direct, bih	011	0	i	1	10	direct

	15 13	12	11	10	9 8	7 0
direct, aih	011	1	i	1	10	direct

	15 9	8	7 2	1 0
ail, long direct (MSW)	1101010	i	101111	xx

MOV Move Data

Opcode (Cont.)

ail, long direct (LSW)

15 0
long direct

short index, ai

15 9	8	7	6 0
1101100	i	1	offset

long index, ai (MSW)

15 9	8	7 2	1 0
1101010	i	100110	xx

long index, ai (LSW)

15 0
long index

ail, short index

15 9	8	7	6 0
1101110	i	1	offset

ail, long index (MSW)

15 9	8	7 2	1 0
1101010	i	100111	xx

ail, long index (LSW)

15 0
long index

short immediate, indirect

15 13	12 10	9 8	7 0
001	rN*	11	short immediate

short immediate, extx

15 13	12 10	9 8	7 0
001	ext	01	short immediate

short immediate, sv

15 8	7 0
00000101	short immediate

short immediate, icr

15 5	4 0
0100111110	short immediate

MOV Move Data

Syntax MOV soperand, doperand

Description Move the contents of the specified source operand, soperand, to the specified destination operand, doperand.

Operation soperand → doperand

soperand, doperand: REG, REG[1,2,3,4]

REG, (rN)[1,2,5]
(rN), REG[4,5]
mixp, REG[4,6]
REG, mixp[1,2,6]
icr, ab
x, ab
dvm, ab
repc, ab

soperand, doperand: ail, x
bil, x
ail, dvm
bil, dvm

rN, direct address
ail, direct address
aih, direct address
bil, direct address
bih, direct address
y, direct address
rb, direct address
sv, direct address

direct address, rN
direct address, ai
direct address, ail
direct address, aih [,eu][10]
direct address, bi
direct address, bil
direct address, bih
direct address, y
direct address, rb
direct address, sv
[##direct address], ai
ail, [##direct address]
(sp), REG[4,6]
(rb+#offset7), ai

MOV **Move Data**

Operation (Cont.)

```
(rb+##offset), ai
ail, (rb+#offset7)
ail, (rb+##offset)

##long immediate, REG[4]
#unsigned short immediate, ail
#signed short immediate, aih
#signed short immediate, rN[9]
#signed short immediate, y[9]
#signed short immediate, rb[9]
#signed short immediate, extX[9]
#signed short immediate, sv[9]
#signed immediate (5 bits), icr[7,8]
```

1. The 32-bit P register can be transferred only to ai *(*`mov p, ai`*).* ph is a write-only register, therefore soperand cannot be ph.
2. A 36-bit accumulator can be a soperand only with a `mov ab, ab` instruction.
3. With `mov reg, reg`, the soperand cannot be the same as the doperand.
4. When the doperand REG is the pc register, two nop instructions must be placed after the `mov soperand, pc` instruction, except for the instruction `mov ##long immediate, pc` where only one nop should be inserted.
5. It is not permitted to move data from data address pointed by one of the indirect registers to the same indirect register (and vice versa) with post-modification.
6. The REG cannot be bi.
7. Enable or disable of context switching (by a write to `icr`) takes effect after the next sequential instruction (e.g. when the user enables context switching for a specific interrupt, if the same interrupt is accepted immediately after the write to `icr`, it will not activate the context switching mechanism).
8. A `mov soperand, icr` cannot be followed by a `bkrep` instruction.
9. Loading the doperand with a short immediate number causes sign-extension.
10. The eu field is an optional field. eu = accumulator extension is unaffected. Refer to the following table.

Instruction Fields		**Accumulator Fields Contents after the Instruction**		
Accumulator	**eu**	**Extension Bits**	**16 MSB aih/bih**	**16 LSB ail/bil**
ai/bi	-	sign-extended	sign-extended	DATA
ail/bil	-	clear	clear	DATA
aih/bih	-	sign-extended	DATA	clear
aih	eu	unaffected	DATA	clear

MOV — Move Data

Operation (Cont.)

Conventions:

- The instruction at program memory address 0x0100: `mov pc, ram`

  After execution: (ram) = 0x0101.
- `mov (r0), r0`:

Before Execution:

r0:	0x20	RAM Address:	0x1000

After Execution:

r0:	0x1000	RAM Address:	0x1000

Flags

When soperand is ail, aih, bil or bih:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

When doperand is ai or bi:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

If doperand is st0, st0 (including the flags) accepts the transferred data. There is no effect when doperand is not ac, bc, or st0, or when soperand is not `ail`, `aih`, `bil`, or `bih`.

MOV Move Data

Cycles

	Cycles	Words
register, register	1	1
register, (rN)	1	1
(rN), register	1	1
mixp, register	1	1
register, mixp	1	1
icr, ab	1	1
x, ab	1	1
ab, AB	1	1
dvm, ab	1	1
repc, ab	1	1
ail, x	1	1
bil, x	1	1
ail, dvm	1	1
bil, dvm	1	1
register, icr	1	1
rN*, direct	1	1
ail, direct	1	1
aih, direct	1	1
bil, direct	1	1
bih, direct	1	1
sv, direct	1	1
direct, rN*	1	1
direct, ai	1	1
direct, ail	1	1

MOV Move Data

Cycles (Cont.)

	Cycles	Words
direct, aih	1	1
direct, bi	1	1
direct, bil	1	1
direct, bih	1	1
direct, aih, eu	1	1
direct, sv	1	1
long direct, ai	2	2
ail, long direct	2	2
(sp), register	1	1
short index, ai	1	1
long index, ai	2	2
ail, short index	1	1
ail, long index	2	2
long immediate, register	2	2
short immediate, ail	1	1
short immediate, aih	1	1
short immediate, rN*	1	1
short immediate, extX	1	1
short, sv	1	1
short immediate (5 bits), icr	1	1
register, bi	1	1
(rN), bi	1	1
long immediate, bi	2	2

MOV Move Data

Example

```
mov ##0x4560, a0
```

Before Execution:

a0:	0	4500	6780

After Execution:

a0:	0	0000	4560

This operation affects the ALU flags in the ST0 Status Register.

MOVD — Move from Data Memory into Program Memory

Opcode

	15 7	6 5	4 3	2	1 0
(rI), (rJ)	010111111	jj	ii	w	qq

Syntax

```
movd (rI), (rJ)
```

Description

Move a word from data memory location pointed to by rI into program memory location pointed to by rJ.

Operation

```
rI - points to data memory location
rJ[1] - points to program memory location
   (rI) → (rJ)

(rI) is postmodified
(rJ) is postmodified
```

[1] The rJ register cannot point to the movd instruction address or to (movd address) + 1.

Flags affected

This instruction does not affect the flags.

Cycles

	Cycles	Words
(rI), (rJ)	4	1

Example

```
movd (r0)+, (r4)+
```

In this example, r0 points to a location in data memory, and r4 points to a location in program memory. Both r0 and r4 are postincremented by one.

Before Execution:

(r4):	0000	(r0):	1234

After Execution:

(r4):	1234	(r0):	1234

MOVP — Move from Program Memory into Data Memory

Opcode

	15 6	5	4 0
(ai), register	0000000001	i	REG

	15 9	8 7	6 5	4 3	2 0
(rI)	0000011	ii	qq	mod	rN

Syntax `movp soperand, doperand`

Description This instruction is used to move a word from program memory location pointed to by soperand into a data memory location pointed to by doperand or into REG. When using ai as soperand, the address is defined by ai-accumulator-low.

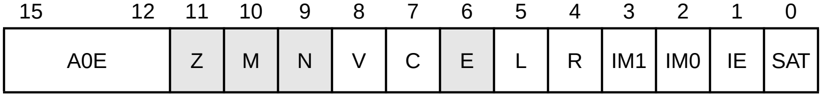
Operation

```
soperand points to program memory location
soperand → doperand
soperand, doperand:    (ail), REG[1,2]
                       (rN), (rI)
```

1. When the operand REG is the pc register, two nop instructions must be placed after the movp (ai), pc instruction.
2. The REG cannot be bi.

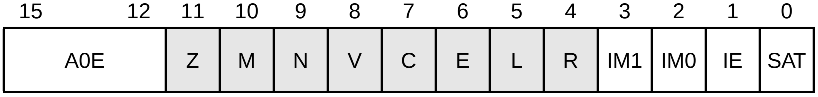
Flags affected **When doperand is not an A accumulator or st0:** this instruction does not affect the flags.

When doperand is an A accumulator:



	15 12	11	10	9	8	7	6	5	4	3	2	1	0
ac:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

When doperand is st0:



	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

MOVP Move from Program Memory into Data Memory

Cycles

	Cycles	Words
(ai), register	3	1
(rN), (rI)	3	1

Example

```
movp (r0)+, (r3)+
```

In this example, r0 points to a location in program memory, and r3 points to a location in data memory. Both r0 and r3 are postincremented by one.

Before Execution:

(r3):	0000	(r0):	1234

After Execution:

(r3):	1234	(r0):	1234

MOVR Move and Round

Opcode

	15 9	8	7 5	4 3	2 0
Indirect	1001110	i	111	mod	rN

	15 9	8	7 5	4 0
Register	1001110	i	110	REG

Syntax

`movr operand, ai`

Description

The value stored in the `operand` is rounded and loaded into the `ai`.

Operation

`operand + 0x8000` → `ai`

`operand:` `REG`[1]
`(rN)`

1. The REG cannot be bi.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	**Words**
indirect	1	1
register	1	1

MOVR — Move and Round

Example `movr (r0)+, a0`

Before Execution:

a0:	0	0000	3D03	(r0):	5678

After Execution:

a0:	0	0000	D678	(r0):	5678

MOVS Move and Shift according to Shift Value Register

Opcode

	15 13	12 11	10 8	7 0
Short Direct	011	AB	011	direct

	15 7	6 5	4 3	2 0
Indirect	000000011	AB	mod	rN

	15 7	6 5	4 0
Register	000000010	AB	REG

Syntax

`movs operand, ab`

Description

The value stored in the operand is shifted by the amount stored in sv and loaded into ab. If the value in the sv is positive, the left shift is executed. If the value in the sv is negative, right shift is executed. If S flag in the ST2 Status Register is set, the operation performs a logical shift. If cleared, it performs an arithmetic shift.

Operation

```
If 0 < sv ≤ 36 then
          operand << sv → ab

If -36 ≤ sv < 0 then
          operand >> |sv| → ab

If sv = 0 then
          operand → ab

operand[2]: REG[1]
          (rN)
          direct address
```

1. The REG cannot be p.
2. When operand is ab, the assembler translates the instruction into a `shfc`: `shfc a0, b0, true`.

MOVS

Move and Shift according to Shift Value Register

Flags

Arithmetic Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

If -36 ≤ sv ≤ 0 (shift right), then the V flag is cleared. If 0 < sv < 36 (shift left) and the operand before being shifted can be represented in (36 –sv) bits then V is cleared. If sv = 36 (shift left) and the operand ≠ 0 then V is set.

Logical Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Note: If sv = 0, the C flag is cleared.

Cycles

	Cycles	**Words**
Short Direct	1	1
Indirect	1	1
Register	1	1

Example

This example assumes sv = 8.

```
movs (r4), a0
```

Before Execution:

a0:	0	0000	3F01	(r4):	001E

After Execution:

a0:	0	0000	1E00	(r0):	001E

MOVSI Move and Shift according to Immediate Shift Value

Opcode

	15 12	11 9	8 7	6 5	4 0
Indirect	0100	rN*	01	AB	immediate

	15 12	11 9	8 7	6 5	4 0
y					

	15 12	11 9	8 7	6 5	4 0
rb					

Syntax `movsi operand, ab, #signed 5-bit immediate`

Description The value stored in the operand is shifted by the immediate value and stored into the ab accumulator. If the immediate value is positive, a left shift is executed. If the immediate value is negative, a right shift is executed. If the S flag in the ST2 Status Register is set, the operation is a logical shift. If S is cleared, an arithmetic shift is performed.

Operation

```
The operand is sign-extended to 36 bits
if 0 < #immediate ≤ 15 then
          operand << #immediate → ab

if -16 ≤ #immediate < 0 then
          operand >> #|immediate| → ab

iF #immediate = 0 then
          operand → ab

          operand:     (rN)
                       y
                       rb
```

Flags

Arithmetic Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

MOVSI Move and Shift according to Immediate Shift Value

Flags (Cont.) **Logical Shift:**

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Note: If `#immediate = 0`, the C flag is cleared.

Cycles

	Cycles	Words
Indirect, ab	1	1

Example This example assumes that the S flag in ST2 is cleared.

```
movsi r4, a0, #8
```

Before Execution:

a0:	0	0000	3E02	r4:	FF00

After Execution:

a0:	F	FFFF	0000	r4:	FF00

MPY Multiply

Opcode

	15 12	11	10 9	8	7 0
Y Direct	1110	i	00	0	direct

	15 12	11	10 8	7 5	4 3	2 0
Y Indirect	1000	i	000	001	mod	rN

	15 12	11	10 8	7 5	4 0
Y Register	1000	i	000	010	REG

	15 12	11	10 8	7 5	4 3	2 0
Indirect Long Immediate (MSW)	1000	i	000	000	mod	rN

	15 0
Indirect Long Immediate (LSW)	long immediate

	15 12	11	10 8	7	6 5	4 3	2	1 0
(rJ), (rI)	1101	i	000	0	jj	ii	w	qq

Syntax

`mpy operand1, operand2`

Description

This instruction multiplies signed `operand1` by signed `operand2` and stores the result in p register. The `operand1` is loaded into y input register with sign extension, and the `operand2` is loaded into x multiplier input register with sign extension.

MPY Multiply

Operation

```
operand1 → y[1]
operand2 → x
signed y * signed x → p

operand1, operand2:     y, direct address
                        y, (rN)
                        y, REG[2]
                        (rJ), (rI)[3]
                        (rN), ##long immediate
```

1. y → y means that y retains its value.
2. The REG cannot be ai, bi, or p.
3. The multiplication in mpy (rJ), (rI) is between X-Memory and Y-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

This instruction does not affect the flags.

Cycles

	Cycles	**Words**
Short Direct	1	1
Indirect	1	1
Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

Example

```
mpy y, (r4)+
```

Before Execution:

y:	3709	(r4):	3240

After Execution:

p:	0ACD	8440

MPYI — Multiply Signed Short Immediate

Opcode

	15 8	7 0
Short Immediate	00001000	immediate

Syntax

`mpyi y, #signed short immediate`

Description

The immediate value is loaded into the x multiplier input register with sign extension. This instruction then multiplies the y input register with sign extension by the x multiplier input register. The result is stored in the product register.

For PineDSPCore compatibility the assembler syntax permits the use of mnemonic `mpys y, #signed short immediate` which is equivalent to `mpyi y, #signed short immediate`.

Operation

`#signed short immediate` → `x`

`signed y * signed x` → `p`

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

This instruction does not affect the flags.

Cycles

	Cycles	Words
Short Immediate	1	1

MPYI Multiply Signed Short Immediate

Example `mpyi y, #0x20`

Before Execution:

y:	01AD

After Execution:

p:	0000	35A0

MPYS Multiply Signed Short Immediate

Opcode

	15 … 8	7 … 0
mpys	00001000	immediate

Syntax

`mpys y, #signed short immediate`

Description

The immediate value is loaded into the x multiplier input register with sign extension. This instruction then multiplies the y input register with sign extension by the x multiplier input register. The result is stored in the product register.

For PineDSPCore compatibility, the assembler syntax permits the use of mnemonic `mpys y, #signed short immediate`, which is equivalent to `mpyi y, #signed short immediate`.

Operation

`#signed short immediate` → `x`

`signed y * signed x` → `p`

Flags

	15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

This instruction does not affect the flags.

Cycles

	Cycles	Words
Short Immediate	1	1

MPYS Multiply Signed Short Immediate

Example

```
mpys y, #0x20
```

Before Execution:

y:	01AD

After Execution:

p:	0000	35A0

MPYSU Multiply Signed by Unsigned

Opcode

	15 12	11	10 8	7 5	4 3	2 0
Y Indirect	1000	i	001	001	mod	rN

	15 12	11	10 8	7 5	4 0
Y Register	1000	i	001	010	REG

	15 12	11	10 8	7 5	4 3	2 0
Indirect Long Immediate (MSW)	1000	i	001	000	mod	rN

	15 0
Indirect Long Immediate (LSW)	long immediate

	15 12	11	10 8	7	6 5	4 3	2	1 0
(rJ), (rI)	1101	i	001	0	jj	ii	w	qq

Syntax

`mpysu operand1, operand2`

Description

This instruction multiplies signed operand1 by unsigned operand2 and stores the result in product register. The operand1 is loaded into y input register with sign extension, and the operand2 is loaded into x multiplier input register with sign extension.

Operation

operand1 → y[1]
operand2 → x
signed y * unsigned x → p

operand1, operand2: y, (rN)
y, REG[2]
(rJ), (rI)[3]
(rN), ##long immediate

1. y → y means that y retains its value.
2. The REG cannot be ai, bi, p.
3. The multiplication in mpysu (rJ), (rI) is between X-Memory and Y-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

MPYSU Multiply Signed by Unsigned

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

This instruction does not affect the flags.

Cycles

	Cycles	Words
Indirect	1	1
Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

Example

```
mpysu y, (r4)+
```

Before Execution:

y:	3F01	(r4):	3240

After Execution:

p:	0C5D	F240

MSU Multiply and Subtract Previous Product

Opcode

	15 13	12 9	8	7 0
Direct	101	1000	i	direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	1000	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	1000	i	101	REG

	15 9	8	7	6 5	4 3	2	1 0
(rJ), (rI)	1101000	i	1	jj	ii	w	qq

	15 9	8	7 6	5	4 3	2 0
Indirect Long Immediate (LSW)	1001000	i	11	x	mod	rN

	15 0
Indirect Long Immediate (MSW)	long immediate

Syntax

`msu operand1, operand2, ai`

Description

The previous product (p) is sign-extended to 36 bits and shifted as defined in the PS field in ST1. The shifted value is subtracted from ai. The result is stored in ai. The signed operands are multiplied together and stored in p.

MSU Multiply and Subtract Previous Product

Operation

```
ai - shifted p → ai
operand1 → y[1]
operand2 → x
signed y * signed x → p

operand1, operand2:     y, direct address
                        y, (rN)
                        y, REG[2]
                        (rJ), (rI)[3]
                        (rN), ##long immediate
```

1. y → y means that y retains its value.
2. The REG cannot be ai, bi, or p.
3. The multiplication in msu (rJ), (rI), ai is between X-Memory and Y-Memory only, where rJ points to Y-Memory, rI points to X-Memory.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
y, Direct	1	1
y, Indirect	1	1
y, Register	1	1
(rJ), (rI)	1	1
Long Immediate	2	2

MSU Multiply and Subtract Previous Product

Example `msu (r4)+, (r0)-, a1`

Before Execution:

(r0):	01AD			(r4):	1980	
a1:	0	0001	8000	p:	01FE	4000

After Execution:

a1:	F	FE03	4000		
x:	01AD			y:	1980
p:	002A	BB80			

NEG Two’s Complement of A Accumulator

Opcode

	15–13	12	11–8	7–4	3–0
neg	011	i	0111	1001	cond

Syntax `neg ai [, cond]`

Description Negate A accumulator.

Operation `ai = -ai`

Flags affected

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Example `neg a1`

Before Execution:

a1:	0	0013	3A05

After Execution:

a1:	F	FFEC	C5FB

NOP No Operation

Opcode

	15 5	4 0
nop	00000000000	xxxxx

Syntax nop

Description No operation occurs.

Operation nop `No operation`

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

This instruction does not affect the flags.

Cycles

	Cycles	**Words**
nop	1	1

NORM Normalize

Opcode

	15 9	8	7 6	5	4 3	2 0
norm	1001010	i	11	x	mod	rN

Syntax `norm ai, (rN)`

Description

This instruction normalizes a signed number in ai by evaluating the N flag in ST1 each time `norm` is executed. If N = 0, rN is modified as specified to generate the magnitude of the exponent. It is assumed that rN and N are initialized before normalization begins.

Multiple executions of `norm` in conjunction with `rep` and `brr` may be required to completely normalize a value in ai.

Normalization can be also performed with exp and shift operations.

The following code examples may be used to normalize a signed value in ai:

```
Option 1:
   rep #n
    norm a0, (r0)+
```

In this example the N flag is set when normalization is complete and no operation is performed for the remainder of the repeat loop.

```
Option 2:
   nrm:   norm a0, (r0)+
          brr nrm, nn
```

In this example the norm instruction is only repeated while N is not set.

Normalization can also be performed (more efficiently) using the exp and shift instructions. For more details refer to Section 2.4.2.4, “Normalization.”

NORM Normalize

Operation

```
If N = 0 (ai is not normalized)
   ai = ai * 2
   rN is modified as specified
else
   nop
   nop
```

This instruction is used to normalize the signed number at the accumulator. It affects the rN register.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

The R flag is updated in `norm` instruction only when rN pointer is modified.

The C flag is set according to the last bit shifted out of the operand (bit 35).

Cycles

	Cycles	Words
norm	2	1

Example

```
rep #10
   norm a0, (r0)+
```

Before Execution:

a0:	0	0134	AEC3	r0:	0000

After Execution:

a0:	0	4D2B	B0C0	r0:	0006

NOT

NOT Logical Not

Opcode

	15 13	12	8	7 4	3 0
not	011	i	0111	1000	cond

Syntax `not ai [, cond]`

Description Performs logical NOT on specified A accumulator. Also see the `moda` instruction on page 7-92.

Operation `ai = not(ai)`

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Example `not a1`

Before Execution:

a1:	0	0013	3A05

After Execution:

a1:	F	FFEC	C5FA

OR Logical Or

Opcode

	15 13	12 9	8	7 0
Short Direct	101	0000	i	direct

	15 12	11 9	8	7 3	2 0
Long Direct (MSW)	1101	010	i	11111	000

	15 0
Long Direct (LSW)	long direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	0000	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	0000	i	101	REG

	15 12	11 9	8	7 0
Short Immediate	1100	000	i	short immediate

	15 12	11 9	8	7 5	4 0
Long Immediate (MSW)	1000	000	i	110	xxxxx

	15 0
Long Immediate (LSW)	long immediate

	15 12	11 9	8	7	6 0
Short Index	0100	000	i	0	Offset

	15 12	11 9	8	7 3	2 0
Long Index (MSW)	1101	010	i	11011	000

	15 0
Long Index (LSW)	long index

OR Logical Or

Syntax `or operand, ai`

Description

The contents of operand are combined with the contents of operand in a bitwise logical-OR operation. If ai is selected as the operand, ai[35:0] is logically ORed with operand and the result is stored in ai[35:0]. If p is selected as the operand, ai[31:0] is logically ORed with operand and the result is stored in ai[31:0]. If operand is Register or (rN), unsigned short immediate, long immediate then ai[15:0] ORed with operand is stored in ai[15:0].

If the operand is one of the A-accumulators or the P register, it is ORed with the destination accumulator.

If the operand is a 16-bit register or an immediate value, the operand is zero-extended to form a 36-bit operand, then ORed with the accumulator. Therefore, this instruction does not affect the upper bits of the accumulator.

Operation

```
or operand, ai

If operand is ai
          ai[35:0] OR ai[35:0] → ai[35:0]

If operand is p
          ai[31:0] OR p[31:0] → ai[31:0]

If operand is REG, (rN), unsigned short immediate, long
immediate
          ai[15:0] OR operand → ai[15:0]
          ai[35:16] → ai[35:16]

operand = REG[1]
          (rN)
          direct address
          [##direct address]
          #unsigned short
          ##long immediate
          (rb+offset7)
          (rb+##offset)
```

[1] The REG cannot be bi.

OR Logical Or

Flags

	15	12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E		Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Long Direct	2	2
Indirect	1	1
Register	1	1
Long Immediate	2	2
Short Immediate	1	1
Index	1	1
Long index	2	2

Example

```
or (r2)+, a0
```

Before Execution:

a0:	0	7520	0000	(r2):	0020

After Execution:

a0:	0	7520	0020	(r2):	0020

PACR Product Move and Round to A Accumulator

Opcode

	15 13	12	11 8	7 4	3 0
pacr	011	i	0111	1011	cond

Syntax

`pacr ai [, cond]`

Description

Accumulator `ai` is loaded with the shifted value of the p register and rounded. If cond is specified, ai is modified only when the condition is true and is unaffected when it is false.

Operation

`pacr ai = shifted p`^{1}` + 0x8000`

1. Shifted p register means that the p register is sign-extended to 36 bits and then shifted as defined in the PS field, status register ST1.

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

POP Pop from Stack into Register

Opcode

	15 … 5	4 … 0
Register	01011110011	REG

Syntax pop REG

Description The top of stack is popped into one of the registers (REG) and the stack pointer, sp, is post-incremented.

Operation

```
(sp) → REG[1,2]
sp + 1 → sp
```

1. REG cannot be sp or bi.
2. A write into p is transferred into p-high (ph).

Flags

If REG is not an A accumulator:

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

If REG is an A accumulator:

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
ac:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Register	1	1

POP — Pop from Stack into Register

Example `pop r3`

Before Execution:

sp:	0384	r3:	00FA
0x0384:	001B		

After Execution:

sp:	0385	r3:	001B

PUSH Push Register or Long Immediate Value onto Stack

Opcode

	15 ... 5	4 ... 0
Register	01011110010	REG

	15 ... 6	5 ... 0
Long immediate (MSW)	0101111101	xxxxxx

	15 ... 0
Long immediate (LSW)	long immediate

Syntax

push operand

Description

The stack pointer, sp, is predecremented, and the operand is pushed onto the software stack.

The push instruction cannot follow mov soperand, sp (except for `mov ##long immediate, sp`); movp `(ail), sp;` `addv/subv/set/rst/chng ##long immediate, sp`.

Operation

```
sp - 1 → sp
operand → (sp)

operand:  REG1
          ##long immediate
```

1. The REG cannot be ai, bi, p, or sp.

Flags affected

15 ... 12	11	10	9	8	7	6	5	4	3	2	1	0
A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

If operand is `ail`, aih, `bil`, or bih, the `L` flag is affected; otherwise the flags are not affected.

PUSH — Push Register or Long Immediate Value onto Stack

Cycles

	Cycles	**Words**
Register	1	1
Long immediate	2	2

Example

```
push r3
```

Before Execution:

sp:	0384	r3:	00FF

After Execution:

sp:	0383	0x383:	00FF

REP Repeat Next Instruction

Opcode

	15 8	7 5	4 0
Register	00001101	xxx	REG

	15 8	7 0
Short immediate	00001100	short immediate

Syntax

`rep operand`

Description

Rep begins a single word instruction loop that is to be repeated operand + 1 times. Repetition times are between 1–65536. The repeat mechanism is interruptible and the interrupt service routine can use another repeat (nested repeats). The nested repeat is uninterruptible.

The following single word instructions cannot be repeated (most of these instructions break the pipeline): `brr`; `callr`; `trap`; `ret`; `reti`; `retd`; `retid`; `rets`; `rep`; `calla`; mov `operand, pc`; pop `pc`; movp `(ai), pc`; `mov repc, ab`.

Rep can be performed inside block-repeat (bkrep).

Operation

operand: #unsigned short immediate[1]
REG[2]

1. When using an unsigned short immediate operand the number of repetitions is between 1 and 256. When transferring the #unsigned `short immediate` number into the repc register, it is copied to the low-order 8-bits of the repc. The high-order 8-bits are zero-extension of the low-order bits.
2. The REG cannot be ai, bi, or p.

REP **Repeat Next Instruction**

Flags affected This instruction does not affect the flags.

Cycles

	Cycles	Words
Register	1	1
Short immediate	1	1

Example

```
rep #tap-1
mac (r4)+, (r1)-, a0

mac is executed #tap times.
```

RET Return Conditionally

Opcode

	15 6	5 4	3 0
ret	0100010110	xx	cond

Syntax

`ret [cond]`

Description

This instruction is used to return from subroutines or interrupts. If the condition is met, the program counter (pc) is pulled from the software stack, while the previous program counter is lost, and the stack pointer (sp) is postincremented.

This instruction can also be used as return from the maskable interrupt service routines (INT0, INT1, or INT2). The IE bit in ST0 must be set to one in order to re-enable interrupts if this is done. The TRAP/BI and NMI interrupt service routines must end with either a `reti` or a `retid` instruction.

Operation

```
If condition then
   (sp) → pc
   sp + 1 → sp
```

Flags affected

This instruction does not affect the flags (or IE bit).

Cycles

	Cycles	Words
ret	2 (return not performed) 3 (return performed)	1

RET Return Conditionally

Example The main program calls the `init` section of the subroutine sub. After execution of `ret`, the program returns to the main routine and executes the next instruction (`mov a0h, @var`).

```
Main Program:                  Subroutine “sub”:

Main:                          .CODE sub
mov @var, a0                   init:
.                              .
.                              .
call sub.init                  ret
mov a0h, @var.
.
```

RETD Delayed Return

Opcode

	15 6	5 0
retd	1101011110	xxxxxx

Syntax `retd`

Description This instruction is used for a delayed return from subroutines or interrupts. Two one-cycle instructions or one two-cycle instruction are/is fetched and executed, before executing the return. (The one-cycle instruction and the two-cycle instruction cannot be instructions that break the pipeline: `brr; callr`; `rep; trap`; `retd; retid; mov` operand`, pc;` `movp (ai), pc; pop pc`.) When the return occurs, the program counter (pc) is popped from the software stack (the previous program counter is lost) and the stack pointer (sp) is post-incremented.

The `retd` instruction and the instruction(s) following the `retd` (two one-cycle instructions or one two-cycle instruction) are uninterruptible.

This instruction can also be used as a return from the maskable interrupt service routines (INT0, INT1, or INT2). To re-enable interrupts, the IE bit in ST0 must be set. The TRAP/BI and NMI interrupt service routines must end with either a `reti` or a `retid` instruction.

Operation

```
(sp) → temporary storage
sp + 1 → sp
Two one-cycle instructions or one two-cycle instruction
are/is executed
temporary storage → pc
```

Flags affected This instruction does not affect the flags.

RETD Delayed Return

Cycles

	Cycles	Words
retd	1	1

Example

The main program calls the `init` section of the subroutine sub. After executing add `p, a1`, the program returns to the main routine and executes the next instruction (mov `a1h, @var`) after add `p, a1` is executed.

```
Main Program:                 Subroutine “sub”:

Main:                         .CODE sub
mov @var, a0                  init:
.                             .
.                             retd
call sub.init                 mac (r4)+, (r0)-, a1
mov a1h, @var                 add p, a1
```

RETI Return from Interrupt Conditionally

Opcode

	15 6	5	4	3 0
reti (context switching disabled)	0100010111	x	0	cond

	15 6	5	4	3 0
reti (context switching enabled)	0100010111	x	1	cond

Syntax

`reti [cond [,context]]`

Description

This instruction is used to return from interrupt with or without interrupt context switching. If the condition is met, the program counter (pc) is pulled from the software stack, while the previous program counter is lost. The stack pointer (sp) is postincremented, and the IE bit in the ST0 register is set to enable interrupts. IE is set only when returning from the INT0, INT1, or INT2 service routine.

The TRAP/BI and NMI interrupt service routines must end with either a `reti` or a `retid` instruction.

Operation

```
If condition then
   (sp) → pc
   sp + 1 → sp
   1 → IE      ;IE is set only when returning from INT0,
               ;INT1, or INT2 service routine.
```

```
If context selected then same context restore operation is
performed as for a cntx r instruction (see page 7-55.)
```

Some assembler syntax examples are:

```
reti
reti ge
reti true, context
reti ge, context
```

Flags affected

This instruction does not affect the flags.

RETI Return from Interrupt Conditionally

Cycles

	Cycles	Words
reti	2 (return not performed) 3 (return performed)	1

Example

After the execution of `reti`, `context` is restored from the shadow registers, and the IE bit in the ST0 Status Register is set. The context switching assumes that the IC0 bit in ICR is set.

```
Int0:
   .
   .
   reti true, context
```

RETID Delayed Return from Interrupt

Opcode

	15 6	5	4	3 0
retid	1101011111	x	0	xxxx

Syntax

`retid`

Description

Delayed return from interrupt. The IE bit is set to enable interrupts, and the stack pointer (sp) is postincremented. IE is set only when returning from INT0, INT1, or INT2 service routine. The two one-cycle instructions or one two-cycle instruction are/is fetched and executed before executing the return. The one-cycle instruction and the two-cycle instruction cannot be instructions that break the pipeline: `brr`; `callr`; `rep`; `trap`; `retd`; `retid`; move operand, `pc`; pop pc*.* When the return occurs, the program counter (pc) is popped from the software stack, and the previous program counter is lost.

The `retid` instruction and the instruction(s) following the `retid` (two one-cycle instructions or one two-cycle instruction) are uninterruptible.

Operation

```
(sp) → temporary storage
sp + 1 → sp
1 → IE    ;IE is set only when returning from INTO,
          ;INT1, or INT2 service routine.
          Two one-cycle instructions or one two-cycle
          instruction following retid instruction are/is
          executed
temporary storage → pc
```

RETID Delayed Return from Interrupt

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

This instruction does not affect the flags.

Cycles

	Cycles	**Words**
retid	1	1

Example

Interrupted instruction is resumed after the execution of pop r2.

```
Int0:
   .
   .
   retid
   pop r3
   pop r2
```

RETS — Return and Adjust Stack Pointer with a Short Immediate Offset

Opcode

	15 8	7 0
rets	00001001	short immediate

Syntax

```
rets #unsigned short immediate
```

Description

This instruction is used to return from subroutines or interrupt service routines (for INT0, INT1, or INT2) and to delete unnecessary parameters from the stack. The program counter (pc) is pulled from the software stack, while the previous program counter is lost. The stack pointer (sp) is post-incremented by one plus an eight-bit unsigned short immediate value.

To enable more interrupts, you must set to one the IE bit in ST0.

Operation

```
(sp) → pc
sp + 1 + #immediate → sp
```

Flags affected

This instruction does not affect the flags.

Cycles

	Cycles	Words
rets	3	1

RETS Return and Adjust Stack Pointer with a Short Immediate Offset

Example

```
rets ##0x004F
```

Before Execution:

sp:	0x0020	0x0020:	0x0001

After Execution:

sp:	0x0070	pc:	0x0001

RND Round Upper 20 bits of A Accumulator

Opcode

	15 13	12	11 8	7 4	3 0
rnd	011	i	0111	1010	cond

Syntax

`rnd ai [, cond]`

Description

Round upper 20 bits of A accumulator if condition is met. Also see `moda` instruction on page 7-92.

Operation

```
Round upper 20 bits of ai
ai = ai+0x8000
```

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Example

`rnd a1`

Before Execution:

a1:	0	0013	3A05

After Execution:

a1:	0	0013	BA05

ROL Rotate Accumulator Left through Carry

Opcode

	15 13	12	11 8	7 4	3 0
rol ai	011	i	0111	0101	cond

	15 13	12	11 8	7	6 4	3 0
rol bi	011	i	0111	x	101	cond

Syntax

```
rol ai [, cond] or
rol bi [, cond]
```

Description

Rotate specified accumulator left if condition is met. The C flag bit is shifted into the LSB of the accumulator and then overwritten with bit 35 of that accumulator. Also see moda instruction on page 7-92 and modb instruction on page 7-96.

Operation

```
if condition is true then
    ai/bi[35] = temporary storage
    ai/bi = (ai/bi << 1) + C
    C = temporary storage
```

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Example

```
rol a1
```

Before Execution: (assuming C flag cleared)

a1:	0	1234	5678

After Execution:

a1:	0	2468	ACF0

C flag = 0

ROR Rotate Accumulator Right through Carry

Opcode

ror ai

15 13	12	11 8	7 4	3 0
011	i	0111	0100	cond

ror bi

15 13	12	11 8	7	6 4	3 0
011	i	0111	x	100	cond

Syntax

```
ror ai [, cond] or
ror bi [, cond]
```

Description

Rotate specified accumulator right if condition is met. The C flag bit is shifted into the MSB of the accumulator and then overwritten with bit 0 of that accumulator. Also see moda instruction on page 7-92 and modb instruction on page 7-96.

Operation

```
if condition is true then
    ai/bi[0] = temporary storage
    ai/bi = (ai/bi >> 1) + C * 2^35
    C = temporary storage
```

Flags affected

st0:

15 12	11	10	9	8	7	6	5	4	3	2	1	0
A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Example

```
ror a1
```

Before Execution: (assuming C flag cleared)

a1:

0	1234	5678

After Execution:

a1:

0	091a	2b3c

C flag = 0

RST Reset Bit Field

Opcode

	15 12	11 9	8	7 0
Short Direct	1110	001	1	direct

	15 12	11 9	8 5	4 3	2 0
Indirect	1000	001	0111	mod	rN

	15 12	11 9	8 5	4 0
Register	1000	001	1111	REG

Syntax `rst ##long immediate, operand`

Description Reset bit-fields specified in a 16-bit operand according to a long immediate value. The long immediate value contains ones in the bit-field locations to be reset.

If the operand is not part of an accumulator (`ail`, `aih`, `aie`, `bil`, or `bih`) then the accumulators are unaffected. If the operand is part of an accumulator, only the addressed part is affected.

The operand and the long immediate values are sign-extension suppressed.

Operation

```
operand AND ##long immediate → operand

operand = REG[1]
          (rN)
          direct address
```

[1] The REG cannot be ai, bi, p, or pc.

RST Reset Bit Field

Flags

When the operand is not st0:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

When the operand is st0:

The specified bits are reset according to the bit-field in the long immediate value, whether or not the A0E bits have changed.

When resetting the A0E bits (`rst ##long immediate, st0`) the flags are reset according to the long immediate value. When setting the A1E bits (`rst ##long immediate, st1`), the flags are reset according to the ALU output.

Cycles

	Cycles	Words
Short Direct	2	2
Indirect	2	2
Register	2	2

Example

```
rst ##0x004F, b0l
```

Before Execution:

b0:	0	0013	3A05

After Execution:

b0:	0	0013	3A00

SET Set Bit Field

Opcode

	15–12	11–9	8	7–0
Short Direct	1110	000	1	direct

	15–12	11–9	8–5	4–3	2–0
Indirect	1000	000	0111	mod	rN

	15–12	11–9	8–5	4–0
Register	1000	000	1111	REG

Syntax

`set ##long immediate, operand`

Description

Set specific bit-fields in a 16-bit operand according to a long immediate value, the long immediate value containing ones in the bit-field locations to be set.

If the operand is not part of an accumulator (`ail`, `aih`, `bil`, or `bih`) then the accumulators are unaffected. If the operand is part of an accumulator, only the addressed part is affected.

The operand and the long immediate values are sign-extension suppressed.

Operation

```
##long immediate OR operand → operand

operand = REG[1]
          (rN)
          direct address
```

[1] The REG cannot be ai, bi, p, or pc.

SET Set Bit Field

Flags

When the operand is not st0:

15 12	11	10	9	8	7	6	5	4	3	2	1	0
A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

When the operand is st0:

The specified bits are set according to the bit-field in the long immediate value, regardless of whether or not the A0E bits have changed.

When setting the A0E bits (`set ##long immediate, st0`) flags are set according to the long immediate value. When setting the A1E bits (`set ##long immediate`, `st1`), the flags are set according to the ALU output.

Cycles

	Cycles	Words
Short Direct	2	2
Indirect	2	2
Register	2	2

Example

```
set ##7FFF, a0l
```

Before Execution:

a0:	0	0013	3A05

After Execution:

a0:	0	0013	7FFF

SHFC — Shift Accumulators According to Shift Value Register Conditionally

Opcode

	15 12	11 10	9 7	6 5	4	3 0
shfc	1101	ab	101	AB	0	cond

Syntax

```
shfc soperand, doperand [, cond]
```

Description

The contents of soperand are shifted according to the value in the sv register, and the result is stored into doperand if the condition is met. If a condition is not specified, the operation is always executed.

If the value in the sv register is positive, a left shift is executed. If the value in the sv register is negative, a right shift is executed.

If the S flag in the ST2 Status register is set, a logical shift is executed. If S is zero, an arithmetic shift is executed.

Operation

```
If condition is true then

    If 0 < sv ≤ 36 then
            soperand << sv → doperand

    If -36 ≤ sv < 0 then
            soperand >> |sv| → doperand

    If sv = 0 then
            soperand → doperand[1]

    If soperand ≠ doperand then
            soperand is unaffected

soperand, doperand: ab, ab
```

[1] In case the sv content is zero, this instruction is a conditional move between accumulators.

SHFC — Shift Accumulators According to Shift Value Register Conditionally

Flags

Arithmetic Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

If -36 ≤ sv ≤ 0 (shift right), then V is cleared. If 0 < sv < 36 (shift left) and the operand before being shifted can be represented in (36 = sv) bits, then V is cleared; V is set otherwise. If sv = 36 (shift left) and the operand ≠ 0, then V is set; V is cleared otherwise.

Logical Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Note: If sv = 0, the C flag is cleared.

Cycles

	Cycles	Words
Shfc	1	1

Example

```
shfc a0, a0
(sv = 8)
```

Before Execution:

a0:	0	0001	0000

After Execution:

a0:	0	0100	0000

SHFI Shift Accumulators by an Immediate Shift Value

Opcode

	15–12	11–10	9	8–7	6	5–0
shfi	1001	ab	1	AB	1	immediate

Syntax

`shfi soperand, doperand, #signed 6-bit immediate`

Description

The contents of soperand are shifted according to the immediate value, and the result is stored in doperand.

If the immediate value is positive, a left shift is executed; if negative, a right shift is executed.

If the S flag in the ST2 status register is set, a logical shift is executed. If zero, arithmetic shift is executed.

Operation

```
If 0 < #immediate ≤ 31 then
    soperand << #immediate → doperand

If -32 ≤ #immediate < 0 then
    soperand >> #|immediate| → doperand

If #immediate = 0 then
    soperand → doperand[1]

If soperand ≠ doperand then
    soperand is unaffected

soperand, doperand: ab, ab
```

[1] In case the immediate shift value is zero, this instruction can be used as a move instruction between the 36-bit accumulators.

SHFI Shift Accumulators by an Immediate Shift Value

Flags

Arithmetic Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

If -32 ≤ #immediate ≤ 0 (shift right), then V is cleared. If 0 < #immediate ≤ 31 (shift left) and the operand before being shifted can be represented in 36 - immediate bits, then V is cleared.

Logical Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Note: If #immediate = 0, the C flag is cleared.

Cycles

	Cycles	Words
shfi	1	1

Example

```
shfi a0, a0, #-16
```

Before Execution:

a0:	0	8000	0000

After Execution:

a0:	0	0000	8000

SHL Shift Left

Opcode

	15 13	12	11 8	7 4	3 0
shl ai	011	i	0111	0010	cond

	15 13	12	11 8	7	6 4	3 0
shl bi	011	i	0111	x	010	cond

Syntax

```
shl ai [, cond] or
shl bi [, cond]
```

Description

Shift specified accumulator left if condition is met. Also see moda instruction on page 7-92 and modb instruction on page 7-96.

If the S flag in the ST2 status register is set, a logical shift is executed. If zero, arithmetic shift is executed.

Operation

```
if condition is true then
    c = ai/bi[35]
    ai/bi = ai/bi <<1
```

Flags affected

Arithmetic Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	**Z**	**M**	**N**	**V**	**C**	**E**	**L**	R	IM1	IM0	IE	SAT

The C flag is set according to bit 35.

The V flag is cleared if the operand being shifted could be represented in 35 bits.

Logical Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	**Z**	**M**	**N**	V	**C**	**E**	L	R	IM1	IM0	IE	SAT

A logical shift is performed when the S status bit in status register ST2 is set.

The C flag is set according to bit 35 shifted out of the operand.

SHL Shift Left

Example `shl a1`

Before Execution:

a1:	0	1987	6543

After Execution:

a1:	0	330e	CA86

SHL4 Shift Left by Four

Opcode

	15–13	12	11–8	7–4	3–0
shl4 ai	011	i	0111	0011	cond

	15–13	12	11–8	7	6–4	3–0
shl4 bi	011	i	0111	x	011	cond

Syntax

```
shl4 ai [, cond] or
shl4 bi [, cond]
```

Description

Shift specified accumulator left by 4 if condition is met. Also see moda instruction on page 7-92 and modb instruction on page 7-96.

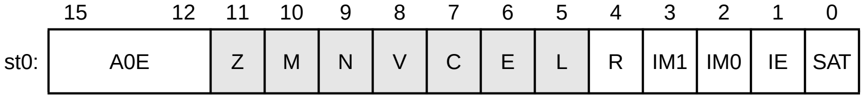
If the S flag in the ST2 status register is set, a logical shift is executed. If zero, arithmetic shift is executed.

Operation

```
if condition is true then
    c = ai/bi[32]
    ai/bi = ai/bi << 4
```

Flags affected

Arithmetic Shift:

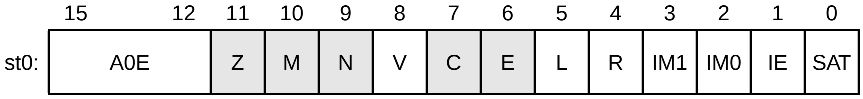


	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

The C flag is set according to bit 35.

The V flag is cleared if the operand being shifted could be represented in 31 bits.

Logical Shift:



	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

A logical shift is performed when the S status bit in status register ST2 is set.

The C flag is set according to bit 32 shifted out of the operand.

SHL4 Shift Left by Four

Example `shl4 a1`

Before Execution:

a1:	0	1987	6543

After Execution:

a1:	1	9876	5430

SHR Shift Right

Opcode

	15 13	12	11 8	7 4	3 0
shr ai	011	i	0111	0000	cond

	15 13	12	11 8	7	6 4	3 0
shr bi	011	i	0111	x	000	cond

Syntax

```
shr ai [, cond] or
shr bi [, cond]
```

Description

Shift specified accumulator right if condition is met. Also see moda instruction on page 7-92 and modb instruction on page 7-96.

If the S flag in the ST2 status register is set, a logical shift is executed. If zero, arithmetic shift is executed.

Operation

```
if condition is true then
    c = ai/bi[0]
    ai/bi = ai/bi >> 1
```

Flags affected

Arithmetic Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

The C flag is set according to bit 0.

Logical Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

A logical shift is performed when the S status bit in status register ST2 is set.

The C flag is set according to bit 0 shifted out of the operand.

SHR **Shift Right**

Example

```
shr a1
```

Before Execution:

a1:	0	1987	6543

After Execution:

a1:	0	0CC3	B2A1

SHR4 Shift Right by Four

Opcode

	15 13	12	11 8	7 4	3 0
shr4 ai	011	i	0111	0001	cond

	15 13	12	11 8	7	6 4	3 0
shr4 bi	011	i	0111	x	001	cond

Syntax

```
shr4 ai [, cond] or
shr4 bi [, cond]
```

Description

Shift specified accumulator right by 4 if condition is met. Also see `moda` instruction on page 7-92 and `modb` instruction on page 7-96.

If the S flag in the ST2 status register is set, a logical shift is executed. If zero, arithmetic shift is executed.

Operation

```
if condition is true then
   c = ai/bi[3]
   ai/bi = ai/bi >> 4
```

Flags affected

Arithmetic Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

The C flag is set according to bit 3.

Logical Shift:

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

A logical shift is performed when the S status bit in status register ST2 is set.

The C flag is set according to bit 3 shifted out of the operand.

SHR4 Shift Right by Four

Example `shr4 a1`

Before Execution:

a1:	0	1987	6543

After Execution:

a1:	0	0198	7654

SQR Square

Opcode

	15 13	12 9	8	7 0
Short Direct	101	1101	i	direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	1101	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	1101	i	101	REG

Syntax

`sqr operand`

Description

The operand is loaded into both the y input and x registers. Both values are sign extended. The contents of the registers are multiplied together and stored in p.

Operation

```
operand → y
operand → x
signed y * signed x → p

operand:  (rN)
          REG[1]
          direct address
```

[1] The REG cannot be ai, bi, or p.

Flags affected

This instruction does not affect the flags.

SQR Square

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1

Example

```
sqr r1
```

Before Execution:

r1:	1001

After Execution:

p:	0100	2001

SQRA Square and Accumulate Previous Product

Opcode

	15 13	12 9	8	7 0
Short Direct	101	1110	i	direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	1110	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	1110	i	101	REG

Syntax `sqra operand, ai`

Description The previous product (p) is sign-extended to 36 bits and shifted as defined in the PS field in ST1. The shifted value is added to ai and the result is stored in ai. The operand is loaded into both the y and x inputs. Both x and y are sign extended and are then multiplied together, the product is stored in the p register.

Operation

```
ai + shifted p → ai
operand → y
operand → x
signed y * signed x → p

operand:  (rN)
          REG[1]
          direct address
```

[1] The REG cannot be ai, bi, or p.

Flags affected

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

SQRA Square and Accumulate Previous Product

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1

Example

```
sqra (r1)+, a0
```

Before Execution:

a0:	0	0000	007F	(r1):	4000

p:	0000	1000

After Execution: (assume PS bits in ST1 are cleared)

a0:	0	0000	107F

p:	1000	0000

SUB Subtract

Opcode

	15 13	12 9	8	7 0
Short Direct	101	0111	i	direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	0111	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	0111	i	101	REG

	15 12	11 9	8	7 0
Short Immediate	1100	111	i	short immediate

	15 12	11 9	8	7 5	4 0
Long Immediate (MSW)	1000	111	i	110	xxxxx

	15 0
Long Immediate (LSW)	long immediate

	15 12	11 9	8	7	6 0
Short Index	0100	111	i	0	Offset

	15 12	11 9	8	7 3	2 0
Long Index (MSW)	1101	010	i	11011	111

	15 0
Long Index (LSW)	long index

	15 12	11 9	8	7 3	2 0
Long Direct (MSW)	1101	010	i	11111	111

	15 0
Long Direct (LSW)	long direct

SUB Subtract

Syntax `sub operand, ai`

Description The contents of operand are subtracted from that of ai. The result is stored in ai[35:0]. If an operand other than p or aj register is selected, the contents of operand are subtracted from ai[15:0] to form a 16-bit subtraction. If the p or aj register is selected for the operand, aj/p[31:0] is subtracted from ai[31:0]. In both cases, the sign is extended through ai[35:32].

Operation ai - operand → ai

```
operand = REG[1]
          (rN)
          direct address
          [##direct address]
          #unsigned short immediate
          ##long immediate
          (rb+offset7)
          (rb+##offset)
```

[1] The REG cannot be bi.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

SUB Subtract

Cycles

	Cycles	Words
Short Direct	1	1
Long Direct	2	2
Indirect	1	1
Register	1	1
Short Immediate	1	1
Long Immediate	2	2
Index	1	1
Long Index	2	2

Example

```
sub (r4), a0
```

Before Execution:

a0:	0	0820	1000	(r4):	0020

After Execution:

a0:	0	0820	0FE0

SUBH Subtract from High Accumulator

Opcode

	15–13	12–9	8	7–0
Short Direct	101	1011	i	direct

	15–13	12–9	8	7–5	4–3	2–0
Indirect	100	1011	i	100	mod	rN

	15–13	12–9	8	7–5	4–0
Register	100	1011	i	101	REG

Syntax

`subh operand, ai`

Description

The contents of operand are subtracted from ai[31:16] to form 16-bit result. ai[15:0] is unaffected after the operation.

Operation

$ai - operand * 2^{16} \rightarrow ai$

```
operand = REG¹
          (rN)
          direct address
```

1. The REG cannot be bi, ai, or p.

Flags affected

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1

SUBH Subtract from High Accumulator

Example `subh rb, a0`

Before Execution:

a0:	0	2045	4000	rb:	3450

After Execution:

a0:	F	EBF5	4000	rb:	3450

SUBL Subtract from Low Accumulator

Opcode

	15 13	12 9	8	7 0
Short Direct	101	1100	i	direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	1100	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	1100	i	101	REG

Syntax

```
subl operand, ai
```

Description

The contents of operand are subtracted from ai[15:0] to form 16-bit result. Sign-extension of operand is suppressed for this operation.

Operation

```
ai - operand → ai
The operand is sign-extension suppressed.

operand = REG¹
          (rN)
          direct address
```

1. The REG cannot be bi, ai, or p.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Direct	1	1
Indirect	1	1
Register	1	1

SUBL Subtract from Low Accumulator

Example

```
subl r1, a0
```

Before Execution:

a0:	0	2020	0000	r1:	FF00

After Execution:

a0:	0	201F	0100	r1:	FF00

SUBV — Subtract Long Immediate Value from a Register or a Data Memory Location

Opcode

Short Direct

15 12	11 9	8	7 0
1110	111	1	direct

Indirect

15 12	11 9	8 5	4 3	2 0
1000	111	0111	mod	rN

Register

15 12	11 9	8 5	4 0
1000	111	1111	REG

Long Immediate (LSW)

15 0
long immediate

Syntax

`subv ##long immediate, operand`

Description

The contents of long immediate value are subtracted from operand to form a 16-bit result. The result of the operation is stored in operand. The operand and long immediate values are sign-extended. If the operand is not part of an accumulator (ail, aih, aie, bil, and bih) then the accumulators are unaffected. If the operand is part of an accumulator, only the addressed part is affected.

Operation

```
operand - ##long immediate → operand

operand = REG¹
          (rN)
          direct address
```

1. The REG cannot be bi, ai, or p.

Note that ai can be used in sub ##long immediate, ai instructions.

SUBV Subtract Long Immediate Value from a Register or a Data Memory Location

Flags

15 12	11	10	9	8	7	6	5	4	3	2	1	0
A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Z, M, and C are the results of the 16-bit operation. M is affected by bit 15.

When the operand is st0:

ST0 (including the flags) accepts the subtraction result, regardless of a0e bits.

When the operand is not st0:

When subtracting a long immediate value from ST1, the ALU output affects the flags.

When the operand is part of an accumulator, only the addressed part is affected. For example, if the instruction subv `##long immediate, a0l` generates a borrow and the carry flag is set, but a0h is unchanged. However, the instruction `subl ##long immediate, a0l` (with same a0 and immediate values) changes the a0h and affects the carry flag according to the 36-bit ALU result.

Note: When using subv `##long immediate, st0` and `cmpv ##long immediate, st0`, the flags are set differently.

Cycles

	Cycles	Words
Short Direct	2	2
Indirect	2	2
Register	2	2

SUBV Subtract Long Immediate Value from a Register or a Data Memory Location

Example `subv ##0x004F, b0h`

Before Execution:

b0:	0	0013	3A05

After Execution:

b0:	0	FFC4	3A05

SWAP Swap Accumulators

Opcode

	15 6	5	4 0
swap	0100100110	xx	swap

swap =	0000	a0 ↔ b0
	0001	a0 ↔ b1
	0010	a1 ↔ b0
	0011	a1 ↔ b1
	0100	a0 ↔ b0 and a1 ↔ b1
	0101	a0 ↔ b1 and a1 ↔ b0
	0110	a0 → b0 → a1
	0111	a0 → b1 → a1
	1000	a1 → b0 → a0
	1001	a1 → b1 → a0
	1010	b0 → a0 → b1
	1011	b0 → a1 → b1
	1100	b1 → a0 → b0
	1101	b1 → a1 → b0

Syntax

Refer to the Operation section for the different swap mnemonics.

Description

Contents of selected ai and bi accumulators are exchanged. When the operation is x → y → z, then y → z and x → y.

Operation

Assembler mnemonics		Operation
swap	(a0, b0), (a1, b1)	a0↔b0 and a1 ↔ b1
swap	(a0, b1), (a1, b0)	a0 ↔ b1 and a1 ↔ b0
swap	(a0, b0)	a0 ↔ b0
swap	(a0, b1)	a0 ↔b1
swap	(a1, b0)	a1 ↔ b0
swap	(a1, b1)	a1 ↔ b1
swap	(a0, b0, a1)	a0 → b0 → a1
swap	(a0, b1, a1)	a0 → b1 → a1
swap	(a1, b0, a0)	a1 → b0 → a0
swap	(a1, b1, a0)	a1 → b1 → a0
swap	(b0, a0, b1)	b0 → a0 → b1
swap	(b0, a1, b1)	b0 → a1 → b1
swap	(b1, a0, b0	b1 → a0 → b0
swap	(b1, a0, b0)	b1 → a0 → b0
swap	(b1, a1, b0)	b1 → a1 → b0

SWAP Swap Accumulators

Flags affected

	15–12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

In the case of: `swap (a0, b0), (a1, b1)`
`swap (a0, b1), (a1, b0)`
The flags represent the data transferred into `a0`.

In all other cases: the flags represent the data transferred into ai.

Cycles

	Cycles	Words
swap	1	1

Example

```
swap (a0, b0), (a1, b1)
```

Before Execution:

a0:	0	0000	1091	b0:	0	0346	5490
a1:	0	0000	8000	b1:	0	0FFF	2300

After Execution:

a0:	0	0346	5490	b0	0	0000	1091
a1:	0	0FFF	2300	b1:	0	0000	8000

TRAP

TRAP **Software Interrupt**

Opcode

	15	5	4	0
trap	00000000001		xxxxx	

Syntax `trap`

Description

The stack pointer (sp) is predecremented. The program counter (pc), which points to the next instruction, is pushed onto the stack and into the DVM register. A branch to address location 0x0002 is executed. The interrupts (INT0, INT1, INT2, NMI, or BI) are disabled regardless of the interrupt mask bits: IE, IM0, IM1, and IM2 in ST0 and IM2 in ST2.

The `trap` instruction cannot be used in the TRAP/BI service routine. To return from the TRAP/BI service routine, use either the `reti` or `retid` instruction.

The software interrupt (TRAP) and the breakpoint interrupt (BI) shares the same interrupt vector address. For more details on TRAP/BI, refer to Section 6.3.3, “TRAP/BI Interrupts.”

Note that both CDI and ScanICE based debug methods use the TRAP/BI interrupt. This precludes the use of the TRAP instruction when using the CWDSP1650 debugger.

Operation

```
sp – 1 → sp
pc → (sp)
pc → dvm
0x0002 → pc
Disable interrupts (INT0, INT1, INT2, NMI, BI)
```

Flags affected This instruction does not affect the flags.

Cycles

	Cycles	Words
Trap	2	1

TST0 Test Bit Field for Zeros

Opcode

Direct ai

15 13	12 9	8	7 0
101	0100	i	direct

Indirect ai

15 13	12 9	8	7 5	4 3	2 0
100	0100	i	100	mod	rN

Register ai

15 13	12 9	8	7 5	4 0
100	0100	i	101	REG

Direct mask

15 12	11 9	8	7 0
1110	100	1	direct

Syntax

`tst0 mask, operand`

Description

Test whether a specified bit-field of REG (one of the registers) or a data space location (using a direct or indirect addressing mode) is all zeros. The field to be tested is specified by a mask containing ones in the bit field location. The mask is the content of an Ax Accumulator (ail) or a long immediate operand. The test operation affects the zero flag—the flag is set if the specified bit-field is all zeros, cleared otherwise.

Operation

```
If (operand AND mask) = 0x0000 then Z = 1
   else Z = 0

mask =    ail
          ##long immediate

operand = REG[1]
          (rN)
          direct address
```

[1] The REG cannot be ai, bi, or p.

The instructions `tst0 a0l, a0l`; `tst0 a1l, a1l` are illegal. The operand and the mask are sign-extension suppressed.

TST0 Test Bit Field for Zeros

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1
Long Immediate	2	2

Example

```
tst0 ##0x004F, r1
```

Before Execution:

r1:	0000

After Execution:

r1:	0000

Z is set in the ST0 Status Register.

TST1 Test Bit Field for Ones

Opcode

Direct ai

15 13	12 9	8	7 0
101	0101	i	direct

Indirect ai

15 13	12 9	8	7 5	4 3	2 0
100	0101	i	100	mod	rN

Register ai

15 13	12 9	8	7 5	4 0
100	0101	i	101	REG

Direct mask

15 12	11 9	8	7 0
1110	101	i	direct

Long immediate (LSW)

15 0
long immediate

Syntax

```
tst1 mask, operand
```

Description

Test whether all bits in the operand, specified by the bit-field in mask, are set. The mask contains ones in the bit field locations to be tested. The test operation affects the zero flag which is set if the specified bit-field is all ones, cleared otherwise.

TST1 Test Bit Field for Ones

Operation

```
If ($\overline{\text{operand}}$ AND mask) = 0x0000 then Z = 1
   else Z = 0

mask =    ail
          ##long immediate

operand = REG[1]
          (rN)
          direct address
```

[1] The REG cannot be ai, bi, or p.

The instructions `tst1 a0l, a0l`; `tstl a1l, a1l` are illegal. The operand and the mask are sign-extension suppressed.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1
Long Immediate	2	2

TST1 Test Bit Field for Ones

Example

```
tst1 ##0x004F, r1
```

Before Execution:

r1: FFFF

After Execution:

r1: FFFF

Z is set in the ST0 Status Register.

TSTB Test Specific Bit

Opcode

	15 12	11 8	7 0
Direct	1111	bbbb	direct

	15 12	11 8	7 5	4 3	2 0
Indirect	1001	bbbb	001	mod	rN

	15 12	11 8	7 5	4 0
Register	1001	bbbb	000	REG

Syntax

```
tstb operand, #bit number
```

Description

Test whether a specified bit from operand is a one or a zero. The bit to be tested is specified by the bit number (0-15). The test operation affects the zero flag—the flag is set if the specified bit is a one, cleared otherwise.

Operation

```
If operand [bit number] = 1 then Z = 1

If operand [bit number] = 0 then Z = 0

operand = REG[1]
          (rN)
          direct address

0 ≤ bit number ≤ 15
```

[1] The REG cannot be ai, bi, or p.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

The Z flag reflects the status of the tested bit.

TSTB Test Specific Bit

Cycles

	Cycles	Words
Short Direct	1	1
Indirect	1	1
Register	1	1

Example

```
tstb a0l, #7
```

Before Execution:

a0:	0	1010	008F

After Execution:

a0:	0	1010	008F

Z is set in the ST0 Status Register.

XOR Exclusive Or

Opcode

	15 13	12 9	8	7 0
Short Direct	101	0010	i	direct

	15 13	12 9	8	7 5	4 3	2 0
Indirect	100	0010	i	100	mod	rN

	15 13	12 9	8	7 5	4 0
Register	100	0010	i	101	REG

	15 12	11 9	8	7 0
Short Immediate	1100	010	i	short immediate

	15 12	11 9	8	7 5	4 0
Long Immediate (MSW)	1000	010	i	110	xxxxx

	15 0
Long Immediate (LSW)	long immediate

	15 12	11 9	8	7 3	2 0
Long Direct (MSW)	1101	010	i	11111	010

	15 0
Long Direct (LSW)	long direct

	15 12	11 9	8	7	6 0
Index	0100	010	i	0	Offset

	15 12	11 9	8	7 3	2 0
Long Index (MSW)	1101	010	i	11011	010

	15 0
Long Index (LSW)	long index

XOR Exclusive Or

Syntax `xor operand, ai`

Description The operand and ai are logically-XORed and the result stored in ai. If p or ai is selected for operand, ai[35:0] is logically XORed with operand and the result is stored in ai[35:0]. If the operand is a 16-bit register or an immediate value, the operand is zero-extended to form a 36-bit operand, then XORed with the accumulator. Therefore, this instruction does not affect the upper bits of the accumulator.

Operation

```
ai ← ai XOR operand

operand = REG[1]
          (rN)
          direct address
          long direct address
          #unsigned short
          ##long immediate
          (rb+offset7)
          (rb+##offset)
```

[1] REG cannot be bi.

Flags

	15 12	11	10	9	8	7	6	5	4	3	2	1	0
st0:	A0E	Z	M	N	V	C	E	L	R	IM1	IM0	IE	SAT

XOR Exclusive Or

Cycles

	Cycles	Words
Short Direct	1	1
Long Direct	2	2
Indirect	1	1
Register	1	1
Short Immediate	1	1
Long Immediate	2	2
Index	1	1
Long Index	2	2

Example

```
xor ##0xFFFF, a0
```

Before Execution:

a0:	0	4320	E3DD

After Execution:

a0:	0	4320	1C22

7.5 Instruction Opcode Bit Coding

This section lists the opcodes with their respective bit encodings in tabular form. Table 7.8 through Table 7.24 list opcodes for the instructions in Section 7.4, “Instruction Set List.”

Table 7.8 Opcode i

i (ai)	Description
0	Accumulator a0
1	Accumulator a1

Table 7.9 Opcode i or j

i/j (bi)	Description
0	Accumulator b0
1	Accumulator b1

Table 7.10 Opcode AB

AB/ab	Description
00	b0
01	b1
10	a0
11	a1

Table 7.11 Opcode ABL

ABL	Description
00	b0l
01	b1l
10	a0l
11	a1l

Table 7.12 Opcode rn

rn (rN)	Description
000	r0
001	r1
010	r2
011	r3
100	r4
101	r5

Table 7.13 Opcode rn*

rn* (rN*)	Description
000	r0
001	r1
010	r2
011	r3
100	r4
101	r5
110	rb
111	y

Table 7.14 Opcode mod

mod	Description
00	No modification
01	+1
10	-1
11	+step

Table 7.15 Opcode w

w (rJ)	Description
0	r4
1	r5

Table 7.16 Opcode REG/reg

REG	Description	REG	Description
00000	r0	10000	b0h
00001	r1	10001	b1h
00010	r2	10010	b01
00011	r3	10011	b11
00100	r4	10100	ext0
00101	r5	10101	ext1
00110	rb	10110	ext2
00111	y	10111	ext3
01000	st0	11000	a0
01001	st1	11001	a1
(Sheet 1 of 2)			

Table 7.16 Opcode REG/reg (Cont.)

REG	Description	REG	Description
01010	st2	11010	a01
01011	p / ph	11011	a11
01100	pc	11100	a0h
01101	sp	11101	a1h
01110	cfgi	11110	lc
01111	cfgj	11111	sv
(Sheet 2 of 2)			

Table 7.17 Opcode ii

ii	Description
00	No modification
01	+1
10	-1
11	+step

Table 7.18 Opcode jj

jj	Description
00	No modification
01	+1
10	-1
11	+step

Table 7.19 Opcode qq

qq (rl)	Description
00	r0
01	r1
10	r2
11	r3

Table 7.20 Opcode cond

cond	Description	cond	Description
0000	true	1000	c
0001	eq	1001	v
0010	neq	1010	e
0011	gt	1011	l
0100	ge	1100	nr
0101	lt	1101	niu0
0110	le	1110	iu0
0111	nn	1111	iu1
		x	Don’t Care

Table 7.21 Opcode x

x	Description
x	don’t care

Table 7.22 Opcode bank in BANKE Instructions

bank	Description
0001	cfgi
0010	r4
0100	r1
1000	r0

Table 7.23 Opcode ext

ext	Description
010	ext0
011	ext1
110	ext2
111	ext3

Table 7.24 Opcode bbbb

bbbb	Bit Number	bbbb	Bit Number
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Chapter 8
On-Chip Emulation Module (OCEM)

This chapter describes the on-chip emulation module (OCEM), which is an optional module for debug support with the CWDSP1650. This chapter covers all aspects of the operation and interface of the OCEM.

This chapter contains the following sections:

8.1 OCEM Overview

The OCEM is an optional module that can provide on-chip emulation for a CWDSP1650-based chip. The OCEM uses the BI/TRAP breakpoint of the CWDSP1650 core to implement all emulation functions. The OCEM includes the following features:

- Breakpoint generation
- Program flow tracing
- ScanICE debug support
- Suspended mode operation

8.1.1 Breakpoint Generation

Predefined conditions programmed into the OCEM registers determine the breakpoint generation conditions. Once a condition is met, the OCEM activates the breakpoint mechanism (BI/TRAP), causing the core to suspend any current action and jump to the BI/TRAP interrupt vector. The OCEM provides multiple breakpoints:

- Program address breakpoints with separate counters
- Data address breakpoint
- Data value breakpoint
- Combined data address and data value breakpoints
- External registers breakpoint
- Abort breakpoint
- Branch instruction breakpoint
- Block repeat loop breakpoint
- Interrupt breakpoint
- Illegal access breakpoint
- Program flow trace buffer full breakpoint
- Single step breakpoint

See Section 8.4, “OCEM Breakpoints,” for more information about using these OCEM breakpoints.

8.1.2 Program Flow Tracing

Program flow tracing produces a dynamic record of program addresses that may be used for debugging a program. These addresses provide a full program trace of instructions executed by the core. The OCEM contains a 16-bit program flow trace register and a 17-bit FIFO program flow trace buffer, both are described in more detail in Section 8.2.8, “Program Flow Trace Register and Program Flow Trace Buffer.”

8.1.3 ScanICE Debug Support

To minimize chip pin count of a design, the OCEM is designed to support the ScanICE serial testing and debugging interface. It also allows an external scan controller to take control of a debugging session without the need for the core to run a monitor program. See Chapter 9, “ScanICE,” for more information.

8.1.4 Suspended Mode Operation

The OCEM supports a low power suspended mode. When in this mode, the OCEM consumes negligible power and is effectively disabled. In the suspended mode, nothing inside the OCEM is clocked. Therefore, the OCEM can be incorporated into final designs with little penalty in terms of silicon area and power consumption. This enables the designer to debug a design in its final form without the need for a special debugging prototype.

8.2 OCEM Programming Model

This section describes the OCEM programming model. All registers and counters within the model are memory-mapped into the data memory space of the CWDSP1650 core. Table 8.1 lists these components with their address locations and page numbers.

Table 8.1 OCEM Programming Model

Register or Counter	Physical Address	Page
Status 0 Register	0xF7FF	8-4
Status 1 Register	0xF7FE	8-6
Mode Register	0xF7FD	8-7
Data Address Breakpoint	0xF7FB	8-9
Data Address Mask	0xF7FA	8-9
Program Address Breakpoint Counter 3	0xF7F9	8-10
Program Address Breakpoint Counter 2	0xF7F8	8-10
Program Address Breakpoint Counter 1	0xF7F7	8-10
Program Address Breakpoint 3	0xF7F3	8-10
Program Address Breakpoint 2	0xF7F2	8-10
Program Address Breakpoint 1	0xF7F1	8-10
Program Flow Trace Register	0xF7F0	8-10

The OCEM registers do not have to be memory mapped in systems using the ScanICE debug as the mapping is implemented in off-core logic. Apart from the two status registers, the content of all the OCEM registers and counters after a reset depends on the DBG_PIN input. If DBG_PIN is asserted HIGH on the falling edge of RST, their contents before reset are preserved. Otherwise, they are all cleared to zero. The same rule also applies to the program flow trace buffer. The two status registers are always cleared to zero after reset. The rest of this section describes the registers and counters from Table 8.1 in more detail.

8.2.1 Status 0 Register

The Status 0 Register contains status information for different breakpoints. The OCEM updates this register automatically to reflect the current breakpoint status when the core is not servicing a TRAP/BI interrupt. While servicing a TRAP/BI interrupt, this register is modified only when written to explicitly. The Status 0 Register is located at address 0xF7FF. The value of this register after reset is zero. The core can read and write all bits in the register unless otherwise stated. Figure 8.1 shows the bit fields for the OCEM Status 0 Register.

Figure 8.1 Status 0 Register

15	14	13	12	11	10	8	7	6	5	4	3	2	1	0
SFT	ILL	TBF	INT	BR	RES		PA3	PA2	PA1	ABORT	EREG	CDVA	DA	DV

SFT **Software Trap** **15, R**
The OCEM sets this bit to one when it detects a software trap. A software trap occurs when the core executes the TRAP instruction.

ILL **Illegal Breakpoint** **14, R/W**
The OCEM sets this bit to one when it detects an illegal breakpoint. An illegal breakpoint occurs when the program attempts an access to the mailbox space or OCEM registers outside a breakpoint handler.

TBF **Trace Buffer Full** **13, R/W**
The OCEM sets this bit to one when it detects a program flow trace buffer full breakpoint.

INT **Interrupt Breakpoint** **12, R/W**
The OCEM sets this bit to one when it detects a breakpoint due to the core servicing an interrupt.

BR **Branch Breakpoint** **11, R/W**
The OCEM sets this bit to one when it detects a branch or block repeat breakpoint.

RES **Reserved** **[10:8]**
These bits are reserved for LSI Logic. These bits always read as zeroes. Writing to these bits has no effect.

PA3 **Program Address 3** **7, R/W**
When the OCEM detects a program address breakpoint initiated by Program Address Breakpoint Register 3, PA3 is set to one.

PA2 **Program Address 2** **6, R/W**
When the OCEM detects a program address breakpoint initiated by Program Address Breakpoint Register 2, PA2 is set to one.

PA1 **Program Address 1** **5, R/W**
When the OCEM detects a program address breakpoint initiated by Program Address Breakpoint Register 1, PA1 is set to one.

ABORT **Abort** **4, R/W**
When the OCEM detects a breakpoint due to its EVENT input signal, ABORT is set to one.

EREG **External Register** **3, R/W**
The OCEM sets this bit to one when it detects a breakpoint due to an user-defined register transaction supported by the CWDSP1650 core.

CDVA **Combined Data Value and Address Breakpoint** **2,R/W**
When the OCEM detects a breakpoint due to both a data value match and a data address match, CDVA is set to one.

DA **Data Address Breakpoint** **1, R/W**
The OCEM sets this bit to one when it detects a breakpoint due to a data address match.

DV **Data Value Breakpoint** **0, R/W**

When the OCEM detects a breakpoint due to a data value match, it sets DV to one.

8.2.2 Status 1 Register

The Status 1 Register contains miscellaneous status information. The OCEM updates this register automatically to reflect the current status. Unless otherwise stated, the core can read and write all bits in the register. The Status 1 Register is located at address 0xF7FE. Value held at each bit after reset is described below. Figure 8.2 shows the bit fields for OCEM Status 1 Register.

Figure 8.2 Status 1 Register

15	14	13	12	11	1	0
DBG	BOOT	ERR	MVD	RES		TREI

DBG **Debug** **15, R/W**

A one on this bit indicates debug mode. The OCEM sets DBG to one when the DBG_PIN input is asserted on the falling edge of RST.

BOOT **Boot** **14, R/W**

A one on this bit indicates boot mode. The OCEM sets BOOT to one when the BOOT_PIN input is asserted on the falling edge of RST.

ERR **Error** **13, R/W**

The OCEM sets this bit to one when a reset occurs during a breakpoint service routine. In this case, the emulation session might continue improperly after the reset.

For example, if reset occurs through a hardware signal and not through the CWDSP1650 debugger, the core to debugger communication might be suspended indefinitely, so a new emulation session must be initiated.

MVD **MOVD** **12, R/W**

When the OCEM detects the core executing a MOVD instruction, it sets this bit to one.

RES **Reserved** **[11:1]**
These bits are reserved for LSI Logic. These bits always read as zeroes. Writing to these bits has no effect.

TREI **Trace Entry Indication** **0, R**
TREI acts as a tag bit for the current program flow trace buffer entry. Refer to Section 8.2.8, "Program Flow Trace Register and Program Flow Trace Buffer," for details of the program flow trace buffer.

8.2.3 Mode Register

The Mode Register contains bits that enable various OCEM functions, such as breakpoints and single stepping. The core can read and write all Mode Register bits. The Mode Register is located at address 0xF7FD. The content of this register after reset depends on the input signal DBG_PIN. If DBG_PIN is asserted high on the falling edge of RST, the register content before reset is preserved. Otherwise, all bits are cleared to zero. Note that the OCEM is prevented from raising any breakpoint while the core is servicing a TRAP/BI interrupt, regardless of the content of the Mode Register. Figure 8.3 shows the bit-fields for the OCEM Mode Register.

Figure 8.3 Mode Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SSE	ILLE	BKRE	TBFE	INTE	BRE	P3E	P2E	P1E	EXTRE	EXTWE	CDVAE	DARE	DAWE	DVRE	DVWE

SSE **Single Step Enable** **15, R/W**
Setting this bit to one enables single-step operation.

ILLE **Illegal Enable** **14, R/W**
Setting this bit to one enables a breakpoint on an illegal condition (for example, trying to access an OCEM register not through the Trap Handler).

BKRE **Block Repeat Enable** **13, R/W**
Setting this bit to one enables a breakpoint when returning to the beginning of a block repeat loop. Executing a REP loop will not raise this breakpoint.

TBFE **Trace Buffer Full Enable** **12, R/W**

Setting this bit to one enables a breakpoint when the program flow trace buffer is full.

INTE **Interrupt Enable** **11, R/W**

Setting this bit to one enables a breakpoint upon detection of the servicing of an interrupt.

BRE **BR Enable** **10, R/W**

Setting this bit to one enables a breakpoint when the core executes a branch-type instruction. A branch-type instruction could be one of the following: BR, BRR, CALL CALLR, CALLA, RET, RETD, RETS, RETI, RETID, or any instruction that has the program counter register (PC) in the core as a destination.

P3E **P3 Enable** **9, R/W**

Setting this bit to one enables Program Address Breakpoint 3. A breakpoint occurs when the core fetches an instruction from a program address specified at Program Address Breakpoint Register 3 (address 0xF7F3).

P2E **P2 Enable** **8, R/W**

Setting this bit to one enables Program Address Breakpoint 2. A breakpoint occurs when the core fetches an instruction from a program address specified at Program Address Breakpoint Register 2 (address 0xF7F2).

P1E **P1 Enable** **7, R/W**

Setting this bit to one enables Program Address Breakpoint 1. A breakpoint occurs when the core fetches an instruction from a program address specified at Program Address Breakpoint Register 1 (address 0xF7F1).

EXTRE **External Register Read Enable** **6, R/W**

Setting this bit to one enables a breakpoint as a result of a user-defined register read transaction.

EXTWE **External Register Write Enable** **5, R/W**

Setting this bit to one enables a breakpoint as a result of a user-defined register write transaction.

CDVAE **Combined Data Value and Address Enable** **4, R/W**
Setting this bit to one enables a breakpoint as a result of a simultaneous data address and data value match.

DARE **Data Address Read Enable** **3, R/W**
Setting this bit to one enables data address breakpoints for data read transactions. A break occurs when the core reads from a data address location that matches the address in the Data Address Breakpoint Register.

DAWE **Data Address Write Enable** **2, R/W**
Setting this bit to one enables data address breakpoints for data write transactions. A break occurs when the core writes to a data address location that matches the address in the Data Address Breakpoint Register.

DVRE **Data Value Read Enable** **1, R/W**
Setting this bit to one enables a data value breakpoint on a data read transaction when the value read matches the Data Value Breakpoint Register (DVM) within the core.

DVWE **Data Value Write Enable** **0, R/W**
Setting this bit to one enables a data value breakpoint on a data write transaction when the value written matches the Data Value Breakpoint Register (DVM) within the core.

8.2.4 Data Address Breakpoint Register

The Data Address Breakpoint register is a 16-bit register containing the data address location that triggers a data address breakpoint. This register is located at address 0xF7FB. The core can read and write this register. The content of this register after reset depends on the input signal DBG_PIN. If DBG_PIN is asserted HIGH on the falling edge of RST, the register content before reset is preserved. Otherwise, all bits are cleared to zero.

8.2.5 Data Address Mask Register

The Data Address Mask register is a 16-bit register containing the data address mask. This register is located at address 0xF7FA. The core can read and write this register. A one on any bit in this register masks out the corresponding bit when matching the Data Address Breakpoint Register with the current data address on the XAB and the YAB. The content of this register after reset depends on the input signal DBG_PIN. If DBG_PIN is asserted HIGH on the falling edge of RST, the register content before reset is preserved. Otherwise, all bits are cleared to zero.

8.2.6 Program Address Breakpoint Counters

The OCEM has three 16-bit read/write program address breakpoint counters. Each counter has a corresponding Program Address Breakpoint Register (see Section 8.2.7, “Program Address Breakpoint Registers.”) When a program address breakpoint is enabled, the OCEM decrements a counter by one every time there is a valid program fetch address on the instruction address bus (IAB) that matches the address in the corresponding Program Address Breakpoint Register. The counter stops decrementing when it reaches zero. The contents of these counters after reset depends upon the input signal DBG_PIN. If DBG_PIN is asserted HIGH on the falling edge of RST, the content of the counters before reset are preserved. Otherwise, the counters are cleared to zero.

8.2.7 Program Address Breakpoint Registers

The OCEM has three 16-bit read/write program address breakpoint registers containing the address locations that triggers program address breakpoints. When enabled, the OCEM raises a program address breakpoint if a valid instruction fetch address on the IAB matches the content of any one of these registers and the current value held in the corresponding Program Address Breakpoint Counter is either zero or one. The contents of these registers after reset depend on the input signal DBG_PIN. If DBG_PIN is asserted HIGH on the falling edge of RST, the contents of the registers before reset are preserved. Otherwise, the registers are cleared to zero.

8.2.8 Program Flow Trace Register and Program Flow Trace Buffer

The OCEM has a 16-bit Program Flow Trace Register and a 16-stage, 17-bit wide FIFO Program Flow Trace Buffer. The buffer dynamically holds the most recent nonlinear program addresses, which are locations of instructions that cause discontinuity in the sequential program flow. Examples of these instructions include branches and interrupts. The nonlinear addressees are stored in the program flow trace buffer in the order of their occurrence. Each 17-bit entry in the trace buffer consists of a 16-bit nonsequential address and a tag bit. When set to zero, this tag bit identifies entries that need to be decoded with adjacent entries. The entire graph of the program flow can be reconstructed by applying the recorded nonlinear addresses to the program source file containing instruction address information. The trace buffer reduces the volume of addresses that need to be kept to accurately reconstruct the program flow.

The core cannot write to the Program Flow Trace Buffer, but it can read the buffer indirectly by reading the Program Flow Trace Register. This register reflects the address field of the current output of the FIFO trace buffer. Reading this register causes the next entry in the buffer to become the current output. The contents of the register and the buffer after reset depend on the input signal DBG_PIN. If DBG_PIN is asserted HIGH on the falling edge of RST, the contents of the register and the buffer before reset are preserved. Otherwise, they are cleared to zero.

The trace buffer methodology uses the following principles:

- Only taken branches, or *nonsequential* program fetches, are recorded.
- If the nonsequential instruction has a distinct destination (for example: BR, BRR, CALL, or CALLR), only the instruction address is recorded.
- If the nonsequential instruction does not have a distinct destination, both the instruction address and the target address are recorded. These include the instructions CALLA, RET, RETI, RETD, RETID, RESTS, and any instruction with the PC register as a destination. Note that MOV ##long_immediate, pc is also in this group although it has a distinct target.
- Interrupts are treated as nonsequential instructions. However, they can occur anywhere. Although the target (destination) is known, the source, the last instruction being executed prior to the service routine, cannot be easily derived. Therefore, the source address (the last instruction being executed before servicing the interrupt) and the destination address (the vector address) are both kept within the trace buffer.
- Nonsequential addresses within the breakpoint handler are not recorded (including the BR instruction that is at the TRAP vector address).

- Nonsequential fetches that are due to the TRAP instruction are not recorded.
- The read address of a MOVP instruction and the write address of a MOVD instruction are not recorded although they appear on the IAB. They are not used for fetching program instructions and do not affect the normal sequential program flow.

8.2.8.1 Program Flow Trace Buffer Reading and Decoding

When reconstructing the program flow recorded in the Program Flow Trace Buffer, alternate reading of the TREI bit in the Status 1 register and the Program Flow Trace Register is required. The TREI bit is always read first as reading the Program Flow Trace Register causes the next entry in the buffer to become its current output. Both registers should be read 16 times to flush the entire buffer. Each pair of values read from the two registers represent one entry in the trace buffer. Program flow reconstruction should start from the last entry read and proceed backwards. An entry with the tag bit set to one is a single entry record that contains the source address of an instruction causing a nonsequential program flow to a distinct target location. An entry with the tag bit set to zero is part of a double entry record recording a nonsequential program flow without a distinct target location. In a double word record, the first entry contains the source address which is the location of the instruction that caused the branch. The second entry in the record contains the target address of the branch. Figure 8.4 shows a program fragment with corresponding trace buffer entries.

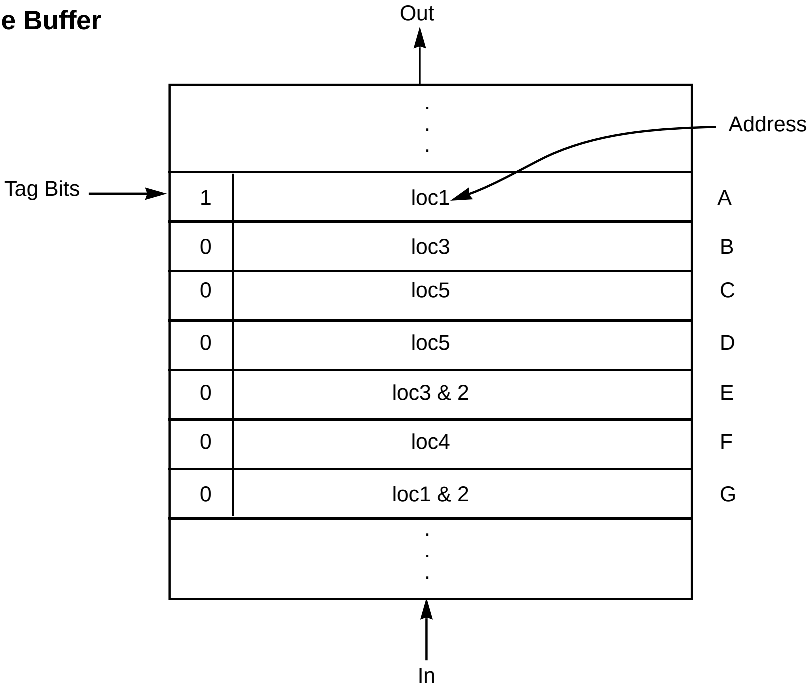
The first few entries read from the trace buffer might have all bits set to one. These entries are not used since the buffer was last flushed. They contain no useful information and can be safely ignored. When the MVD bit in the OCEM Status 1 register is set to one, the integrity of the program flow trace cannot be guaranteed. This is because the core has executed a MOVD instruction, which might have modified the program code.

Figure 8.4   Program Flow Trace with Corresponding Trace Buffer Entries

(a) Program Fragment

```
    .
    .
    .
loc1:call ##loc2 ;distinct target location, single-entry record: A
    .
    .
    .
loc2:mov  ##loc5, a1l
loc3:callaa1l    ;nondistinct target location, two-entry record: B,C
    .
    .
    .
loc 4:ret        ;nondistinct target location, two-entry record: F,G
    .
    .
    .
loc 5:ret        ;nondistinct target location, two-entry record: D,E
    .
    .
    .
```

(b) Flow Trace Buffer



1	loc1	A
0	loc3	B
0	loc5	C
0	loc5	D
0	loc3 & 2	E
0	loc4	F
0	loc1 & 2	G

8.3 OCEM Signals

This section defines the OCEM signals, which are organized according to function in the following subsections:

- Section 8.3.1, “Service Interface”
- Section 8.3.2, “Boot Logic Interface”
- Section 8.3.3, “Core Memory Bus Interface”
- Section 8.3.4, “User-Defined Register Interface”
- Section 8.3.5, “Illegal Access Interface”
- Section 8.3.6, “Core Control Interface”
- Section 8.3.7, “Breakpoint Interface”
- Section 8.3.8, “ScanICE Interface”
- Section 8.3.9, “Clocking and Miscellaneous OCEM Signals”

Table 8.2 lists all OCEM and monitored CWDSP1650 core signals.

Table 8.2 OCEM Signal List

Interface	Signal Mnemonic	Signal Name	Input/ Output
Service Interface	SV_A[3:0]	Service Address Bus	Input
	SV_DI[15:0]	Service Data Bus In	Input
	SV_DO[15:0]	Service Data Bus Out	Output
	SV_G	Service Chip Select	Input
	SV_R	Service Read Enable	Input
	SV_W	Service Write Enable	Input
Boot Logic Interface	BOOT_EN	Boot Mode Enable	Output
	BOOT_PIN	Boot Pin	Input
	DBG_EN	Debug Mode Enable	Output
	DBG_PIN	Debug Pin	Input
	URST_PIN	User Reset Pin	Input
(Sheet 1 of 3)			

Table 8.2 OCEM Signal List (Cont.)

Interface	Signal Mnemonic	Signal Name	Input/ Output
Core Memory Bus Interface	IAB[15:0]	Program Address Bus	Input
	PREN	Program Read Enable	Input
	PWEN	Program Write Enable	Input
	XAB[15:0]	X-Memory Address Bus	Input
	XREN	X-Memory Read Enable	Input
	XWEN	X-Memory Write Enable	Input
	YAB[15:0]	Y-Memory Address Bus	Input
	YREN	Y-Memory Read Enable	Input
	YWEN	Y-Memory Write Enable	Input
User-Defined Register Interface	LD_EXT_REG	External Register Write Enable	Input
	RD_EXT_REG	External Register Read Enable	Input
Illegal Access Interface	ILLE	Illegal Breakpoint Enabled	Output
	ILLEGAL_ACCESS	Illegal Access	Input
Core Control Interface	BLOCKLOOP	Block-Repeat Detected	Input
	BRANCHING	Branch Detected	Input
	BTI_SERVICE	BI/TRAP Service Active	Input
	CLR_ISTAT	Clear Interrupt Status	Input
	DVM	Data Value Match	Input
	INT_SEEN	Interrupt Indication	Input
	INVALID_PA	Invalid Program Address	Input
	MVD_EXEC	Move Data-to-Program Detected	Input
	SEL_TRACE[1:0]	Select Address for Trace	Input
	TRACE_TAG	Trace Tag	Input
	TRACE_UNWRITE	Trace Unwrite	Input
	TRACE_WRITE	Trace Write	Input
	TRAP_SERVICE	Trap Service Indicator	Input
(Sheet 2 of 3)			

Table 8.2 OCEM Signal List (Cont.)

Interface	Signal Mnemonic	Signal Name	Input/ Output
Breakpoint Interface	BI	Breakpoint Interrupt	Output
	IACK_BI	Breakpoint Interrupt Acknowledge	Input
ScanICE Interface	SCAN_ALERT	Scan Breakpoint Alert	Output
	EXT_START_SCAN	External Start Scan	Output
	OCEM_SCAN_ALERT	OCEM Scan Alert	Output
	SCAN_EN	Scan Enable	Input
	SCANICE_EN	ScanICE Enable	Input
	SCAN_IN	Scan Chain Input	Input
	SCAN_OUT	Scan Chain Output	Output
	SCAN_WS	Scan Write Strobe	Input
	TEST	Test Mode	Input
Clocking and Miscellaneous OCEM Signals	LD_CC	Load Clock Control Register	Input
	EVENT	External Event	Input
	RST	Reset	Input
	SECOND_CYCLE	Second Cycle Indicator	Input
	SUSPEND	OCEM Suspended Mode Enable	Input
(Sheet 3 of 3)			

8.3.1 Service Interface

This section describes the signals that support the OCEM service interface. The service interface exists to imitate a static RAM style interface, which eases OCEM interfacing with other CWDSP1650 modules. The service interface occupies 16 memory address locations which can be mapped to anywhere in the data memory space. (Note that the CWDSP1650 debugger assumes the OCEM is mapped to locations 0xF7F0 – 0xF7FF when using CDI debug.)

SV_A[3:0] **Service Address Bus** **Input**
SV_A is a four-bit address bus selecting an OCEM register or counter for access.

SV_DI[15:0] **Service Data Bus** **Input**
SV_DI is a 16-bit data input bus through which the OCEM registers and counters are written.

SV_DO[15:0] **Service Data Bus** **Output**
SV_DO is a 16-bit data output bus through which the OCEM registers and counters are read.

SV_G **Service Chip Select** **Input**
Assert SV_G HIGH when accessing the OCEM.

SV_R **Service Read Enable** **Input**
Assert SV_R HIGH to read from the OCEM.

SV_W **Service Write Enable** **Input**
Assert SV_W HIGH to write to the OCEM.

8.3.2 Boot Logic Interface

This section describes the OCEM signals that interact with the boot logic in a CWDSP1650 design. The OCEM samples the input signals in this section only on the falling edge of RST when returning from a reset operation. Note that the boot mechanism is always implemented off-core in a CWDSP1650 design. Signals described in this section are most likely to be connected to off-chip strap pins and to the off-core boot logic.

BOOT_EN **Boot Mode Enable** **Output**
The OCEM drives this signal HIGH when the BOOT bit in the Status 1 Register is set to one, which indicates to off-core boot logic that the boot mode is enabled.

BOOT_PIN **Boot Pin** **Input**
Asserting this signal HIGH on the falling edge of RST sets the BOOT bit to one in the Status 1 register. Connecting this signal to an off-chip strap pin allows off-chip control of program boot loading.

DBG_EN **Debug Mode Enabled** **Output**
The OCEM drives this signal HIGH when the DBG bit in Status 1 Register is one indicating debug mode to the boot logic.

DBG_PIN **Debug Pin** **Input**
Asserting this signal HIGH on the falling edge of RST sets the DBG bit to one in the Status 1 register.

Connecting this signal to an off-chip strap pin allows off-chip control of on-chip debugging logic.

For example, the CWDSP1650 debugger forces a combined boot and debug mode to boot load a monitor program.

URST_PIN **User Reset Pin** **Input**
Asserting this signal HIGH on the falling edge of RST sets the ERR bit to one in Status 1 register. Note that this signal does not trigger a reset. Instead, it is intended to be used by the CWDSP1650 debugger to detect a reset caused by an external event.

For example, a user pushed an on-board reset button while the monitor program is running. In this case, the reset button should assert both RST and URST_PIN.

8.3.3 Core Memory Bus Interface

This section describes the OCEM interface to the memory buses of the core. The OCEM monitors these core signals to trigger the OCEM program and data breakpoints.

IAB[15:0] **Program Address Bus** **Input**
The core drives this 16-bit bus with the memory address of either the program instruction or the program data.

PREN **Program Read Enable** **Input**
The core drives this signal HIGH to request program data.

PWEN **Program Write Enable** **Input**
The core drives this signal HIGH to indicate a write to the program memory space.

XAB[15:0] **X-memory Address Bus** **Input**
The core drives this 16-bit bus with the X-memory space address.

XREN **X-memory Read Enable** **Input**
The core drives this signal HIGH to request data from the X-memory address specified by XAB[15:0].

XWEN **X-memory Write Enable** **Input**
The core drives this signal HIGH to indicate a X-memory data write to the address specified by XAB[15:0].

YAB[15:0] **Y-memory Address Bus** **Input**

The core drives this 16-bit bus with the Y-memory space address.

YREN **Y-memory Read Enable** **Input**

The core drives this signal HIGH to request data from the Y-memory address specified by YAB[15:0].

YWEN **Y-memory Write Enable** **Input**

The core drives this signal HIGH to indicate a Y-memory data write to the address specified by YAB[15:0].

8.3.4 User-Defined Register Interface

This section describes signals driven by the core to access user-defined registers. The OCEM monitors these core signals to trigger the OCEM external register breakpoints.

LD_EXT_REG

External Register Write Enable **Input**

The core drives this signal HIGH during a user-defined register write cycle.

RD_EXT_REG

External Register Read Enable **Input**

The core drives this signal HIGH during a user-defined register read cycle.

8.3.5 Illegal Access Interface

These OCEM signals connect to a Bus Interface Unit (BIU) to provide program protection from illegal memory accesses. The OCEM triggers an illegal breakpoint when it detects an illegal access.

ILLE **Illegal Breakpoint Enabled** **Output**

When the ILLE bit of the Mode register is set to one, the OCEM drives this signal HIGH to indicate that the illegal access breakpoint is enabled.

ILLEGAL_ACCESS

Illegal Access **Input**

The BIU drives this signal HIGH to inform the OCEM that an illegal access has occurred (such as an attempt to access the OCEM registers not through the TRAP interrupt service routine).

8.3.6 Core Control Interface

This section describes the core control signals that the OCEM decodes to monitor various core activities.

BLOCKLOOP

Block-Repeat Detected **Input**
The core drives this signal HIGH whenever it detects a block-repeat loop returning to its start address from its end address (completing one loop.)

BRANCHING **Branch Detected** **Input**
The core drives this signal HIGH whenever a a branch-type instruction causes a nonsequential program flow.

BTI_SERVICE

BI/TRAP Service Active **Input**
The core asserts this signal HIGH upon execution of a TRAP/BI service routine.

CLR_ISTAT **Clear Interrupt Status** **Input**
The core asserts this signal HIGH when it services an interrupt.

DVM **Data Value Match** **Input**
The core asserts this signal HIGH to indicate a match between the content of the DVM register and the current value on the core internal data bus.

INT_SEEN **Interrupt Indication** **Input**
The core asserts this signal HIGH when it is preparing to service an interrupt.

INVALID_PA **Invalid Program Address** **Input**
The core asserts this signal HIGH when it is not using the current address on the IAB to fetch an instruction. This occurs when the core executes the MOVD and MOVP instructions to write or read the program memory.

MVD_EXEC **Move Data-to-Program Detected** **Input**
The core asserts this signal HIGH after it executes a MOVD instruction. The OCEM logs this event to indicate potential corruption of program memory.

SEL_TRACE[1:0]

Select Addresses for Trace **Input**

The OCEM keeps a record of the last four addresses on the IAB, including the current address. The core uses SEL_TRACE[1:0] to specify which of these address values is selected to be written to the OCEM trace buffer.

TRACE_TAG **Trace Tag** **Input**

The core asserts this signal HIGH to indicate that the current program address is one of two addresses stored for certain nonsequential program flow operations. The OCEM stores the value of TRACE_TAG with the trace address values in the trace buffer.

TRACE_UNWRITE

Trace Unwrite **Input**

The core drives this signal HIGH to unwrite the last trace address from the trace buffer. An unwrite capability is needed to erase a conditional branch trace buffer entry when the branch is not taken.

TRACE_WRITE

Trace Write **Input**

The core drives this signal HIGH to write an entry into the trace buffer.

TRAP_SERVICE

Trap Service Indicator **Input**

The core drives this signal HIGH when a software trap occurs.

8.3.7 Breakpoint Interface

This section describes the OCEM signals interfacing with the breakpoint interrupt logic of the core.

BI **Breakpoint Interrupt** **Output**

The OCEM asserts this signal HIGH to send a breakpoint interrupt to the core.

IACK_BI **Breakpoint Interrupt Acknowledge** **Input**

The core asserts this signal HIGH to acknowledge a breakpoint interrupt sent by the OCEM.

8.3.8 ScanICE Interface

This section describes the OCEM signals that support the ScanICE Interface.

EXT_START_SCAN
External Start Scan **Input**
The ScanICE interface asserts this signal HIGH to force a breakpoint interrupt allowing the scan controller to take control of the CWDSP1650 core.

OCEM_SCAN_ALERT
OCEM Scan Alert **Output**
The OCEM asserts this signal when the core jumps to the BI/TRAP vector address while in scanICE mode, which indicates that the scanICE mode has been entered.

SCAN_EN **Scan Enable** **Input**
The ScanICE interface asserts this signal HIGH to configure all registers in the OCEM scan chain in serial scan mode.

SCANICE_EN **ScanICE Enable** **Input**
This signal is asserted HIGH to configure the OCEM for ScanICE debug.

SCAN_IN **Scan Chain Input** **Input**
This input signal connects directly to the test input of the first flip-flop in the OCEM scan chain.

SCAN_OUT **Scan Chain Output** **Output**
This output signal connects directly to the test output of the last flip-flop in the OCEM scan chain.

SCAN_WS **Scan Write Strobe** **Input**
Asserting this signal HIGH disables all write strobe memory signals from the core. Disabling all core write transactions protects the contents of the off-core RAMs during scan mode.

TEST **Test Mode** **Input**
When SCAN_EN is LOW, driving TEST HIGH forces all registers in the OCEM scan chain to be load enabled, implementing a capture function when in scan mode. TEST should also be asserted whenever SCAN_EN is asserted.

8.3.9 Clocking and Miscellaneous OCEM Signals

This section describes the clocking and miscellaneous signals of the OCEM.

LD_CC **Load Clock Control Register** **Input**

This external input signal is the strobe for loading the Clock Control register. This signal has a lower priority than the START_SCAN or STOP_SCAN signals and should not be used as a substitution for either of these signals (by writing 0b1111 or 0b0000 to the CCU register.)

OCEM_CLK **OCEM Clock** **Input**

The CWDSP1650 CCU drives this signal to provide a clock to the OCEM. The OCEM_CLK is extended by wait states, as the CORE_CLK is.

EVENT **External Event** **Input**

Off-core glue logic drives this signal HIGH to initiate an ABORT breakpoint.

RST **Reset** **Input**

Driving this signal HIGH resets the OCEM.

SECOND_CYCLE

Second Cycle Indicator **Input**

Off-core glue logic (normally a BIU) asserts this signal HIGH for two ICU_CLK cycles when the core comes out of a reset.

SUSPEND **OCEM Suspended Mode Enable** **Input**

Asserting this signal HIGH forces the OCEM into suspended mode, which ensures that the OCEM registers are not clocked. In suspended mode the OCEM consumes negligible power and is effectively isolated from the other CWDSP1650 modules.

8.4 OCEM Breakpoints

This section describes the operations of different OCEM breakpoints. The breakpoints available for the OCEM are:

- Program address
- Data address
- Data value
- Combined data address and data value
- External register
- Abort
- Illegal access
- Branch
- Block repeat
- Interrupt
- Single step
- Program flow trace buffer full

8.4.1 Program Address Breakpoint

The OCEM supports three program address breakpoints. Each breakpoint has three controlling components:

- A program address breakpoint register to hold the breakpoint address
- A program address breakpoint counter that holds the number of matches needed until a breakpoint is issued
- An enable bit in the Mode register

Table 8.3 lists these three breakpoints and components. For more information on any of these counters or registers, see Section 8.2, “OCEM Programming Model.”

Table 8.3 Program Address Breakpoint Components

Program Address Breakpoint #	Program Address Breakpoint Register	Program Address Breakpoint Counter	Mode Register Enable Bit
1	1	1	PE1
2	2	2	PE2
3	3	3	PE3

When the core fetches an instruction, the OCEM detects a match between the address appearing on the instruction address bus (IAB) and an address in any of the Program Address Breakpoint Registers. If a match with any of the registers occurs while the corresponding program address breakpoint is enabled, the OCEM decrements the appropriate breakpoint counter by one. The OCEM also triggers a breakpoint if the match occurs when the value held by the counter is one or zero. A counter stops decrementing once it reaches zero. A program address breakpoint stops a program at the specified location. The instruction at that location is not executed until after the breakpoint is serviced. The OCEM ignores addresses on the IAB that are not used for valid program instruction fetches.

To cause a breakpoint on every occurrence of a program address, the address counter should be set to either zero or one. To cause a breakpoint on the n^{th} occurrence only, the counter should be set to *n.* Writing zeros to the P1E, P2E, and P3E bits within the OCEM Mode Register disables the program address breakpoints. The PA1, PA2, and PA3 bits in the Status 0 Register indicate the source of the program address breakpoint.

8.4.2 Data Address Breakpoint

A data address breakpoint occurs on a match between the OCEM Data Address Register and the CWDSP1650 data address buses, XAB and YAB. The OCEM sets the DA bit in the Status 0 Register to one when it detects a data address breakpoint.

The Data Address Mask Register allows the data address breakpoint to expand into an address space rather than a singular address. A one in any bit in this register masks the corresponding bit in the address match. A match occurs when all other bits in the Data Address Register and the data address buses are equal.

The data address breakpoints are separately enabled for read and write transactions. To break on read transactions, set the DARE bit in the Mode Register to one. To break on write transactions, set the DAWE bit in the Mode Register to one.

A data address breakpoint is serviced on the completion of the data transaction causing the breakpoint. Hence, the breakpoint is serviced at least two cycles after the prefetch of the instruction causing the breakpoint. Depending on whether the instruction causing the breakpoint is followed by a multicycle instruction, up to two instructions are executed before the core executes the breakpoint service routine.

8.4.3 Data Value Breakpoint

The data value breakpoint is activated when the OCEM detects a data match between the CWDSP1650 internal data bus and the contents of the DVM Register. Note that a data value breakpoint is due only to a matched data during *memory* transactions. The OCEM sets the DV bit in the Status 0 Register to one when it detects a data value breakpoint.

The data value breakpoint is separately enabled for read and write transactions. To break on read transactions, set the DVRE bit in the Mode Register to one. To break on write transactions, set the DVWE bit in the Mode Register to one.

A data value breakpoint is serviced on the completion of the data transaction causing the breakpoint. Hence, the breakpoint is serviced at least two cycles after the prefetch of the instruction causing the breakpoint. Depending on whether or not the instruction causing the breakpoint is followed by a multicycle instruction, up to two instructions can be executed before the core executes the breakpoint service routine.

8.4.4 Combined Data Address and Data Value Breakpoints

To enable the OCEM to detect combined data address and data value conditions, the CDVAE bit in the Mode Register should be set to one. Note that the DARE and DVRE bits and/or the DAWE and DVWE bits must also be set to one for the OCEM to detect a dual breakpoint condition.

When a combination of data address and data value breakpoints is enabled, a breakpoint is triggered only when both a data value match and a data address match occur. A combined data address and data value breakpoint is serviced on the completion of the data transaction causing the breakpoint.

8.4.5 External Register Breakpoint

An external register breakpoint occurs when the CWDSP1650 core accesses one of its four user-defined registers. The OCEM sets the EREG bit to one in the Status 0 Register when it detects an external register breakpoint.

The external register breakpoint is enabled separately for read and write transactions. To break on read transactions, the EXTRE bit in the Mode Register is set to one. To break on write transactions, set the EXTWE bit in the Mode Register to one. An external register breakpoint is serviced on the completion of an external register transaction.

8.4.6 Abort Breakpoint

An external breakpoint occurs when the EVENT input of the OCEM is asserted. The OCEM registers the EVENT input on the rising edge of OCEM_CLK. EVENT should be synchronized with OCEM_CLK before entering the OCEM. The abort breakpoint is always enabled. The OCEM sets the ABORT bit of Status 0 Register when it detects this breakpoint.

8.4.7 Illegal Access Breakpoint

One feature of the OCEM is to protect the reserved mail box area (addresses 0xF400 – 0xF7DF) and the OCEM registers (addresses 0xF7F0 – 0xF7FF) from illegal accesses. Legal access of the mailbox and register spaces occur only through the breakpoint handler (the routine that is executed as a result of the TRAP/BI interrupt). Attempting to access the mailbox or OCEM space through anything other than the breakpoint routine causes a breakpoint and sets the ILL bit to one in the Status 0 Register. The illegal breakpoint is enabled/disabled through the ILLE bit within the Mode Register. Both the ILLE bit in the Mode Register and the ILL bit in the Status 0 Register must be written to through the breakpoint handler. Any other attempts to write to these bits are disabled.

The illegal breakpoint mechanism also protects the breakpoint handler routine from other illegal accesses, such as when a user program jumps into the breakpoint handler not through a TRAP/BI interrupt. In such a case, if the handler accesses the mailbox area, it will trigger an illegal breakpoint.

8.4.8 Branch and Block Repeat Breakpoints

The OCEM activates a branch breakpoint when it detects a branch-type instruction. A branch-type instruction could be one of the following: BR, BRR, CALL, CALLR, CALLA, RET, RETD, RETS, RETI, RETID, and any instruction that has the program counter register (PC) in the core as destination. The OCEM activates a block repeat breakpoint when it detects the switch from the last to the first address in a block repeat instruction (BKREP). No breakpoint is triggered when the core fetches the first instruction for the first time when entering the block repeat loop. The BR bit in the Mode Register enables/disables the branch breakpoint. The BKRE bit in the Mode Register enables/disables the block repeat breakpoint. The OCEM sets the BR bit in the Status 0 register when it detects either a branch breakpoint or a block repeat breakpoint.

Note that the REP instruction does not cause a branch breakpoint. The breakpoint occurs for taken branches only. The next instruction executed after returning from the breakpoint service routine is at the target address. Two exceptions are the BRR and CALLR instructions, where the next instruction executed after returning from the breakpoint service routine is at the target address plus one.

8.4.9 Interrupt Breakpoint

An interrupt breakpoint occurs when the OCEM detects the core is going to service one of four interrupts: INT0, INT1, INT2, or NMI. The core services this breakpoint and then services the interrupt causing the breakpoint before returning control to the interrupted program. The OCEM sets the INT bit in the Status 0 Register to one when an interrupt breakpoint occurs. Setting the INTE bit in the Mode Register enables the interrupt breakpoint.

8.4.10 Single-Step Operation

Setting the SSE bit to one in the Mode Register enables single-step operation. In the single-step mode, the OCEM automatically breaks after each instruction the core executes. The OCEM does not set a specific status bit for a single-step breakpoint.

8.4.11 Program Flow Trace Buffer Full Breakpoint

Setting the TBE bit in the Mode Register to one enables the trace buffer full breakpoint. The OCEM sets the TBF bit in the Status 0 Register to one when the trace buffer is full. The trace buffer is a 16-stage FIFO buffer logging the most recent *nonsequential* operations. A nonsequential program flow can be recorded with one or two buffer entries. To prevent data loss due to an overflow, the trace buffer full breakpoint is activated when the buffer has either 15 or 16 valid entries. Once the whole trace buffer is read, the OCEM fills it with all ones, thus isolating new incoming addresses from the old ones.

Chapter 9
ScanICE

This chapter describes the CWDSP1650 Scan based In-Circuit Emulation (or ScanICE) implementation, which is an optional module for embedded debugging and emulation. This chapter covers the operation and interface of the CWDSP1650 ScanICE implementation.

Scan based In-Circuit Emulation (ScanICE) enables complete observation and control of circuit states for debug purposes through circuitry included in most devices for testing during manufacture. This technique allows the entire processor state to be clocked out of the design and, if desired, a modified state to be clocked back in over a serial bus (or *scan chain*) using a lower frequency clock. Inserting a scan chain into a core design is normally performed by a scan insertion software tool as part of the Design-For-Test process. A major benefit of the ScanICE approach is that very few package pins and additional logic are required to implement a comprehensive debugging system, thus removing the need for debug prototypes and reducing the failure risk for production silicon.

This chapter contains the following sections:

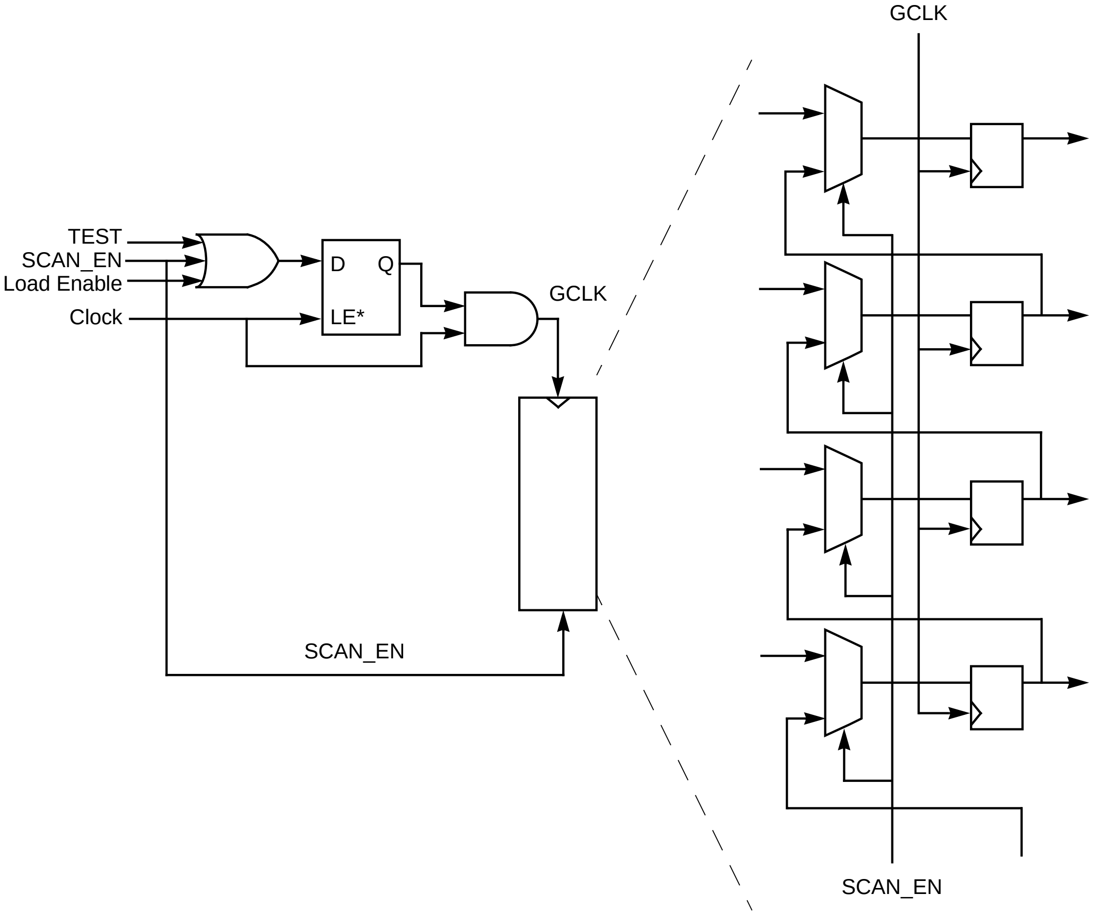
- Section 9.1, “ScanICE Power Saving Registers”
- Section 9.2, “ScanICE Requirements”
- Section 9.3, “ScanICE Interface”
- Section 9.4, “CWDSP1650 ScanICE Support”
- Section 9.5, “Memory Access during ScanICE”
- Section 9.6, “ScanICE Reset”

This chapter describes ScanICE designs in general terms whenever possible, but uses the CWDSP1650 Reference Device design as an example when needed. If you do not recognize some of the ScanICE signal names, they are most likely Reference Device pins described in the *CWDSP1650 Reference Device User's Guide*.

9.1 ScanICE Power Saving Registers

Figure 9.1 illustrates a power-saving, scan-inserted register bank.

Figure 9.1 Power Saving Register with Scan Inserted



There are two input pins that control its mode of operation, SCAN_EN and TEST, that are common for every register on a particular scan chain and are routed to each flip-flop in the bank. Table 9.1 lists the ScanICE operations enabled with the SCAN_EN and TEST pins.

Table 9.1 ScanICE Operational Modes

TEST	SCAN_EN	Operation Mode
0	0	Normal
x	1	Serial Scan
1	0	Capture

During production testing, test data is first shifted into the power saving registers using the Serial Scan mode. Then a capture cycle forces every register to be clocked and load-enabled. The result of this capture cycle is then shifted out of the device and compared with the desired state.

It is common for all registers of a design to be located on a single scan chain. In systems with significant amounts of memory, an additional scan chain may be provided specifically for memory access. Since the core will normally contain many thousands of register bits, the main scan chain could take many thousands of scan clock cycles to completely clock in/out of the chip. This is not significant for core testing since the scan clock will usually be running at speeds in the order of 5 MHz. However, when the memory block fills and block reads or program loading is required, the scan delay time could become unacceptably long. A second and much shorter scan chain, dedicated for memory accesses, can be added to improve overall scan performance. In this scenario the core generates the addresses for memory accesses using its internal address registers while data is loaded to or read from the memory scan chain.

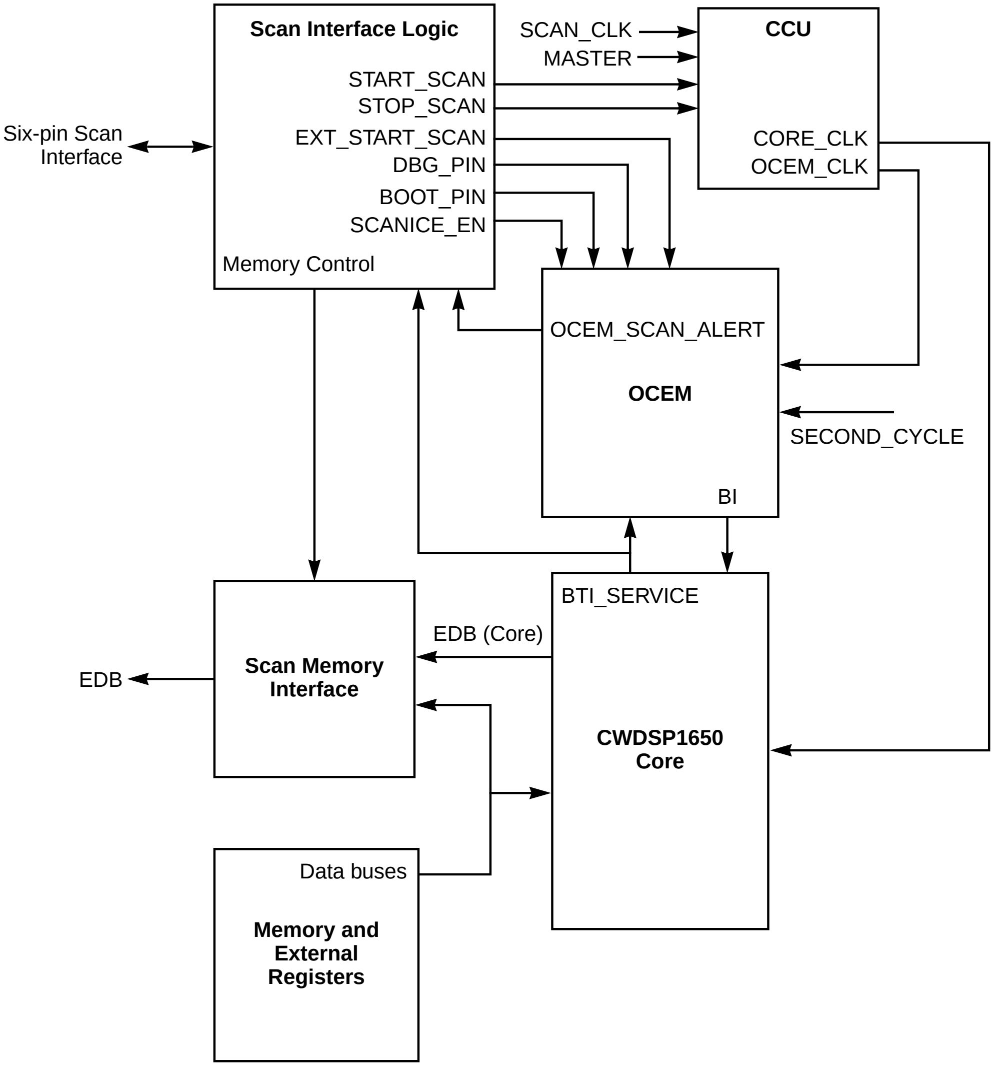
In a typical CWDSP1650 system, ScanICE replaces the functionality provided by a software monitor program, by giving read/write access to core registers, OCEM registers, and memory. The key difference is that a monitor program typically communicates this information through an off-chip bus to a memory mapped mailbox, while ScanICE utilizes a serial link to an off-chip scan controller.

9.2 ScanICE Requirements

During scan, when the contents of every flip-flop in a particular scan chain is serially shifted out of the device, the clock to each flip-flop is sourced by the off-chip scan controller. Since this clock is different from the one used in normal operation, the clock-controller must stop and switch clocks before scanning can begin. It is important that when the main clock is stopped, the core is in a stable state; all data that a programmer expects to be stored in the registers is stored, and the core is not in the middle of a wait-stated access to memory or off-chip peripherals. To achieve this condition, the CWDSP1650 enters scan mode only when it has just executed a breakpoint interrupt or software TRAP. By ensuring that two NOPs are read from the interrupt service vector, the clock can be stopped cleanly while executing these instructions, guaranteeing the state of the pipeline.

Proper operation of the OCEM is integral to the functioning of ScanICE. A block diagram of a typical configuration including the core, OCEM, and ScanICE supporting circuitry is illustrated in Figure 9.2.

Figure 9.2 ScanICE Support in a CWDSP1650 System



The following sections describe the functionality of the main blocks of Figure 9.2 that control the operation of ScanICE. It is noted that although the CWDSP1650, CCU and OCEM are supplied as hard-macros, the Scan Interface Logic and Scan Memory Interface are application dependent and must be designed specifically into each target system.

9.3 ScanICE Interface

Table 9.2 details the six-wire bus between the scan interface and the off-chip scan controller in Figure 9.2 as implemented in the CWDSP1650 Reference Device. For a more detailed description of these signals (pins), please refer to the *CWDSP1650 Reference Device User's Guide*.

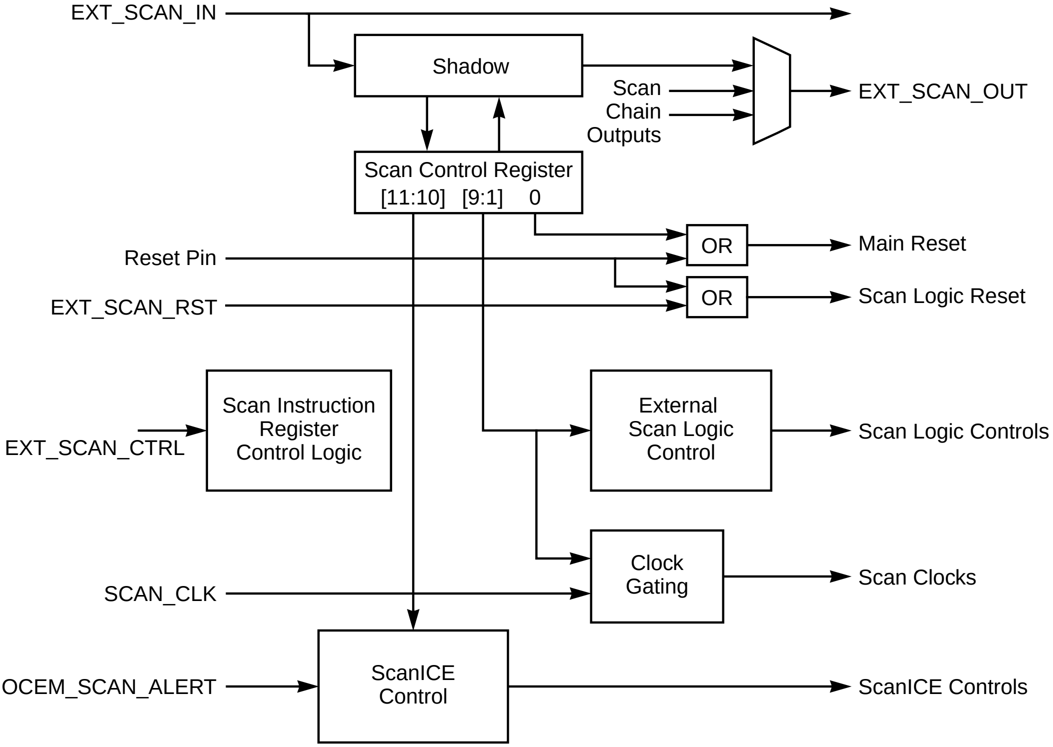
Table 9.2 Six-pin ScanICE Interface

Signal	Input / Output to ASIC	Function
EXT_SCAN_IN	Input	Serial scan data input
EXT_SCAN_OUT	Output	Serial scan data output
EXT_SCAN_CTRL	Input	Start of scan indication to on-chip scan interface
SCAN_CLK	Input	Scan clock input
SCAN_ALERT	Output	Event indication output from scan controllers
EXT_SCAN_RST	Input	Resets on-chip scan control logic

This six-pin interface serially loads the Scan Control Register, which controls all ScanICE operations while the CWDSP1650 is in the Serial Scan Mode. For more information on the Scan Control Register, see Section 9.3.1, "Scan Control Register."

Scan interface logic details are design dependent, but a block diagram of an example circuit is shown in Figure 9.3. The main elements from the figure are further described in the following subsections.

Figure 9.3 Example Scan Interface



9.3.1 Scan Control Register

To program the Scan Control Register, first serially load the shadow register with EXT_SCAN_IN data clocked by SCAN_CLK. Then parallel load this shadow register data to the Scan Control Register at the end of the programming sequence. This technique ensures that the control lines do not change state incorrectly during the serial information transfer.

The shadow register is selected as the destination of the EXT_SCAN_IN signal whenever the EXT_SCAN_CTRL signal is asserted. Once all the required bits have been clocked into the shadow register, the EXT_SCAN_CTRL signal is deasserted and the parallel load to the Scan Control Register occurs on the second next edge of SCAN_CLK.

The length and function of the Scan Control Register will vary with each application depending on the variety of functions ScanICE has to perform. Figure 9.4 shows the bit-fields of an example CWDSP1650 Scan Control Register.

Figure 9.4 Scan Control Register

STSC	ESS	CCMR	SCNE	BTM	DBGM	EDBC	FFSE	FFT	SCS	MWD	SCRST

STSC **Stop Scan**
When asserted HIGH this bit forces exit from scan mode. Note that the CCU waits for this bit to be deasserted before restarting the CWDSP1650 clocks.

ESS **External Start Scan**
When asserted HIGH this bit asserts BI, allowing the off-chip scan controller to take control of the CWDSP1650.

CCMR **Clock Core and Memory Register**
When asserted HIGH this bit routes SCAN_CLK to both the CWDSP1650 and the memory interface logic.

SCNE **Scan Enable**
This bit configures the OCEM for ScanICE debug and controls what happens when a Breakpoint Interrupt or TRAP is detected.

BTM **Boot Mode**
This bit drives the BOOT_PIN signal of the OCEM.

DBGM **Debug Mode**
This bit drives the DEBUG_PIN signal of the OCEM.

EDBC **EDB Control**
In the Scan Memory Interface logic, the source of the EDB can be either the EDB from the CWDSP1650 core or an internal 16-bit register loaded through the Scan Interface Logic. EDBC selects between either the EDB or a 16-bit register for EDB control.

FFSE **Scan Flip-Flop Scan Enable Input**
This bit drives the Scan Enable input of every flip-flop on the scan chain selected by the SCS bit.

FFT **Scan Flip-Flop Test Input**
This bit drives the TEST input of every flip-flop on the scan chain selected by the SCS bit.

SCS **Scan Chain Select**
This bit selects between the available target scan chains. In the CWDSP1650 Reference Device, there are two scan chains in addition to the shadow register in the Scan Interface Logic; the main scan chain and a 16-bit scan chain for the Scan Memory Interface. In an application with more than two scan chains, SCS may be a multi-bit field.

MWD **Memory Write Disable**
When asserted HIGH, this bit disables all memory write enable signals, thus protecting the CWDSP1650 memory during scan operations.

SCRST **Scan Control Reset**
This bit connects to the reset signals of all CWDSP1650 logic blocks except the scan interface logic. This allows resetting of all CWDSP1650 components during a serial scan, without affecting the current scan operation.

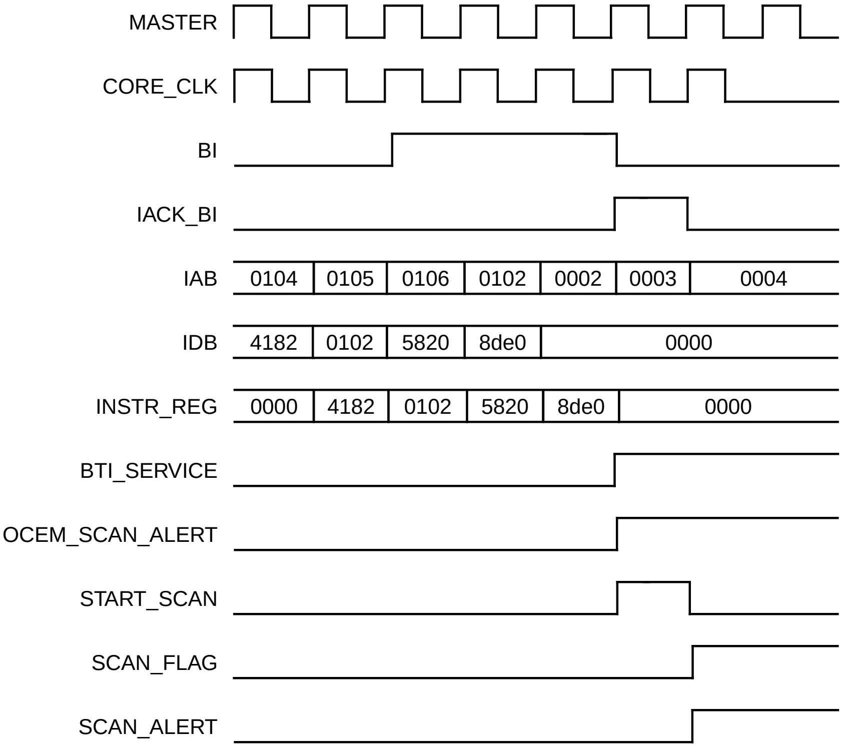
9.3.2 ScanICE Control

The ScanICE control block in Figure 9.3 controls the entry and exit from scan mode. There are several ways in which scan mode can be entered:

- Hardware Breakpoint
- Software TRAP
- Debug Boot
- Debug Reset
- Abort

Figure 9.5 shows an example entry into Scan Mode through an OCEM breakpoint interrupt. This example is equally valid for any of the scan mode entry methods.

Figure 9.5   Entry to Scan Using a Breakpoint Interrupt



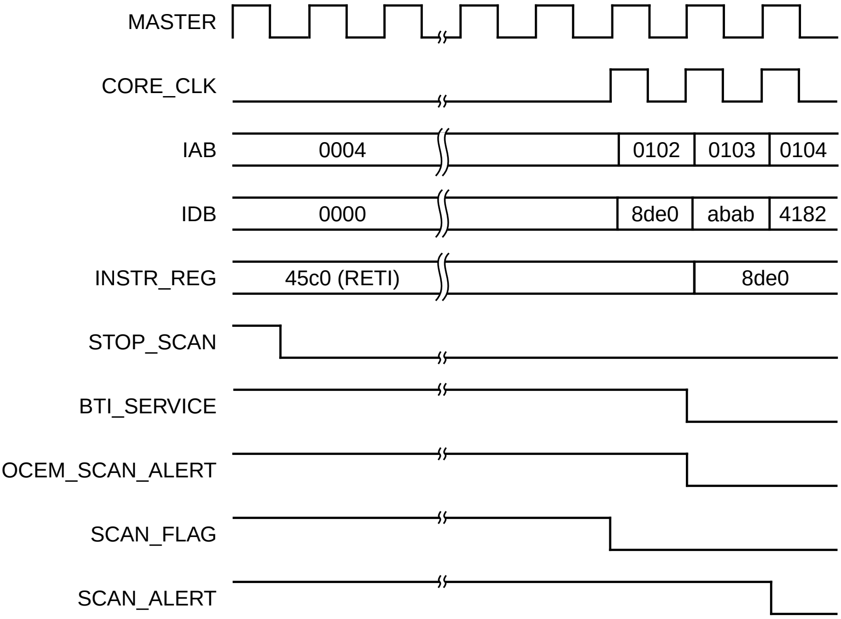
Note: INSTR_REG - Contents of the instruction register inside the CWDSP1650.

The assertion of OCEM_SCAN_ALERT indicates entry into scan mode, but the triggering of the OCEM BI signal differs. The OCEM_SCAN_ALERT assertion is synchronous to BTI_SERVICE when SCANICE_EN is asserted. For hardware breakpoints and the software TRAP, BI is asserted in the usual way by the OCEM and the core. The external scan controller initiates scan by asserting the ESS bit in the Scan Control Register, which drives the EXT_START_SCAN signal from the Scan Interface. Finally a debug boot initiates a breakpoint interrupt immediately following reset through dedicated circuitry in the OCEM.

Following assertion of OCEM_SCAN_ALERT, the ScanICE Interface asserts EXT_START_SCAN to control the CCU, asserts SCAN_FLAG to indicate entry into scan mode for on-chip logic and asserts SCAN_ALERT to inform the off-chip scan controller that it now has control of the CWDSP1650 and support logic.

The only mechanism for leaving the Scan Mode is through the STSC bit in the Scan Control Register. This bit must be set to one and cleared back to zero before the CCU clocks stopped during scan mode are started again. Figure 9.6 shows the CWDSP1650 leaving Scan Mode through the STSC bit.

Figure 9.6 Stop Scan Mode



9.3.3 External Scan Logic Control

Each register on a destination scan chain has two inputs that control its operation; SCAN_EN and TEST. The function of these signals is described in Section 9.1, “ScanICE Power Saving Registers.” The External Scan Logic Control block drives the Scan Enable and Test inputs of each scan chain depending on the state of the SCNE, FFT, CCMR and SCS bits of the Scan Control Register. See Section 9.3.1, “Scan Control Register,” for more information on these bits.

9.3.4 Clock Gating

It is often convenient to adjust which scan chains the scan-clock controls. This is handled by the Clock Gating block and configured with the CCMR and SCS bits of the Scan Control Register. See Section 9.3.1, “Scan Control Register,” for more information on these bits.

9.4 CWDSP1650 ScanICE Support

The OCEM and CCU module both contain special features to assist the ScanICE interface.

9.4.1 OCEM ScanICE Support

To support ScanICE, the OCEM has the following features:

- A Breakpoint Interrupt is triggered when EXT_START_SCAN from the Scan Interface is asserted.
- A Breakpoint Interrupt is triggered when BOOT_PIN, DBG_PIN, SCANICE_EN, and SECOND_CYCLE are asserted. This forces entry to scan mode during a debug-boot before any instructions are decoded.
- OCEM_SCAN_ALERT is asserted whenever BTI_SERVICE and SCANICE_EN are asserted.
- All OCEM registers are scan enabled, which allows hardware breakpoints to be set when in scan mode by the scan controller.

9.4.2 CCU ScanICE Support

The CCU has the following features to support ScanICE:

- SCAN_CLK is multiplexed with MASTER clock to source the main clock and during scan mode, SCAN_CLK is routed to most output clocks.
- The START_SCAN and STOP_SCAN signals control the switching to and from scan mode as indicated in Figure 9.5 and Figure 9.6. The CCU controls the clean stopping and switching of clocks.

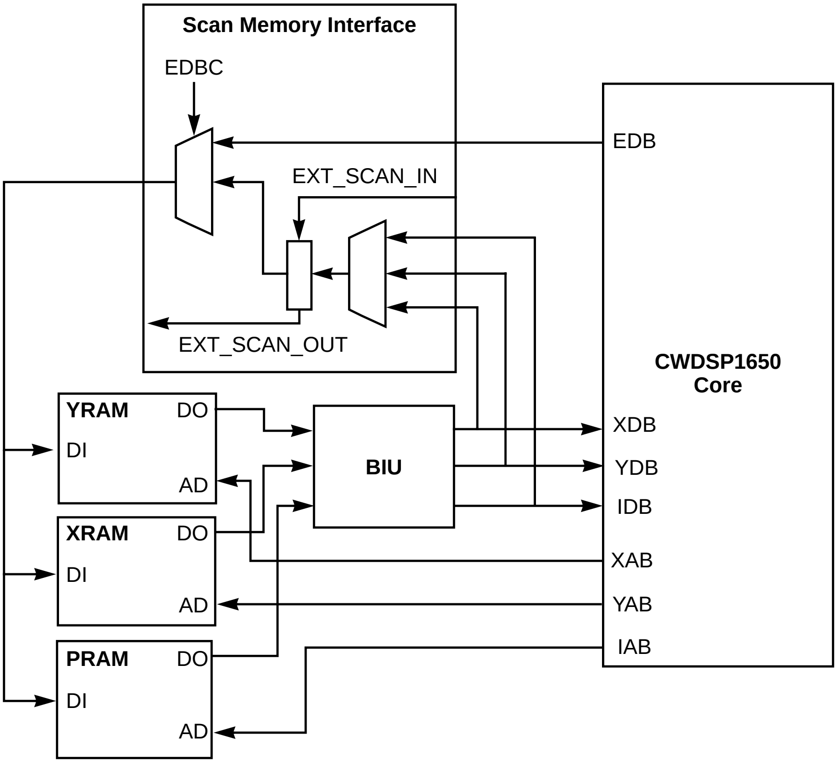
- During scan mode, the state of the Clock Control Register before entering scan mode is stored in a shadow register. This shadow register is scan enabled, allowing the resumption clock speed on leaving scan mode to be changed and the memory clock skews to be programmed.

9.5 Memory Access during ScanICE

During a conventional debug when a monitor program runs on the core, access to memory is carried out in the same way as during normal operation. When in scan mode, this would involve preparing the core through the scan chain for a memory read/write operation and would thus require a complete scan for every data value. This is prohibitively slow when large blocks of data or program memory must be transferred between the debugger and a device. A faster approach is to provide an alternative scan enabled source for data to the EDB and destination for the core input buses. By using a separate and much smaller scan chain, the time for accessing memory can be considerably reduced.

Figure 9.7 shows the support provided on the CWDSP1650 Reference Device for memory access.

Figure 9.7 Example ScanICE Memory Access Scheme



A 16-bit register with its own 16-bit scan chain is located within the Memory Scan Interface block; this register will be referred to as the Memory Register. The Memory Register output is multiplexed to the EDB as determined by the EDBC bit in the Scan Control Register. In this way, the Memory Register can be quickly loaded during ScanICE, and the data presented to all on-chip memory blocks. The input to the Memory Register is derived from the data-buses entering the core; in Figure 9.7, just the memory data buses are shown. The input bus selected is determined by the various read enable control signals driven by the core.

During a read or write memory access when in scan mode, the core is configured to supply the appropriate addresses and control signals for the memory. For contiguous blocks of data, the core internal address modification circuitry can increment the presented address on successive clock cycles, thus avoiding the need to scan the core for every data value. For a memory read the Memory Register must be clocked to load the data being read, and the core must be clocked to update the address it is sourcing; this is the reason for the CCWR bit in the Scan Control Register.

9.6 ScanICE Reset

A complication that arises when developing embedded core debug systems is handling a reset. There are numerous methods for triggering a reset, the most common being board-level push-buttons and debugger commands. On leaving reset it is often useful to be able to configure the device for a particular mode of operation (Boot pin and Debug pin, for example, that are driven to the OCEM). On the CWDSP1650 Reference Device, strap or configuration pins are brought out to the chip periphery, limiting the configurability and flexibility of the pin-count. The Scan Control Register offers a unique opportunity to extend the configurability of a device during debug without impacting the pin-count. This method is illustrated above by including the Boot and Debug Pin bits in the Scan Control Register, and can be extended to include other functions as needed.

One of the problems with embedded debug is how to communicate to the debug software that a hardware reset has occurred. The OCEM sets the ERR bit in Status 1 Register to one when a reset occurs during a breakpoint service routine. This tells the debugger that its view of the core state may be incorrect. This concept could be extended by including a bit in the Scan Interface Control Shadow Register that is set during normal operation. When a reset occurs, this bit would be cleared and the next time the scan controller does anything it will see that a reset was executed since the last scan. This bit would then probably reset the device again, but also ensure that the Scan Control Register was appropriately set up when releasing reset.

For this reset scheme to work, it is required that the SCRST bit in the Scan Control Register not reset the scan control logic. This can be achieved as indicated in Figure 9.3. This ScanICE reset method makes it possible to first reset the entire device through a Reset Pin, reset the device again with SCRST, and guarantee both the state of the device and how it will be configured after reset.

Chapter 10
Specifications

This chapter specifies the physical and electrical characteristics of the CWDSP1650 DSP core. It contains the following sections:

- Section 10.1, "Physical Specifications"
- Section 10.2, "AC Timing Diagrams"

10.1 Physical Specifications

Table 10.1 lists the dimensions of the CWDSP1650 core in LSI Logic's G10 technology.

Table 10.1 CWDSP1650 Physical Layout Size

Core	Technology	Width	Height	Total Area[1]
CWDSP1650	G10	2.0 mm	2.5 mm	5.0 mm^2

[1] This figure excludes power rings.

10.2 AC Timing Diagrams

The CWDSP1650 can use various clocks for different operations. The Clock Control Unit (CCU) selects the appropriate clock for the current transaction needs. Specific information about the CCU and the clocks it generates can be found in Section 6.7, "Clock Control Unit (CCU)."

The input *setup* time is defined from the signal valid to the rising edge of the clock and the input *hold* time is defined from the rising edge of the clock to the signal valid. For input setup times, the driver must drive the signal valid before any receivers need it. For input hold times, the driver must hold the signal valid longer than needed by any receiver.

The output *Valid* and *Invalid delay* times are defined from the rising edge of the clock to the signal valid.

The section is further divided into the following subsections:

- Section 10.2.1, “OCEM Registers”
- Section 10.2.2, “Data and Program Memory”
- Section 10.2.3, “User-Defined Registers”

10.2.1 OCEM Registers

This section illustrates waveforms for reading and writing the OCEM registers. The OCEM registers are memory-mapped to location 0xF7F0 to 0xF7FF in the data memory space and can be accessed with or without wait states. The OCEM provides a service interface to mimic a static RAM interface that allows the OCEM to interface with the core through a bus interface unit (BIU) as a memory device.

Figure 10.1 and Figure 10.2 show timing diagrams for writing and reading the OCEM. Table 10.2 lists the related timing values from these figures.

Figure 10.1 Writing the OCEM

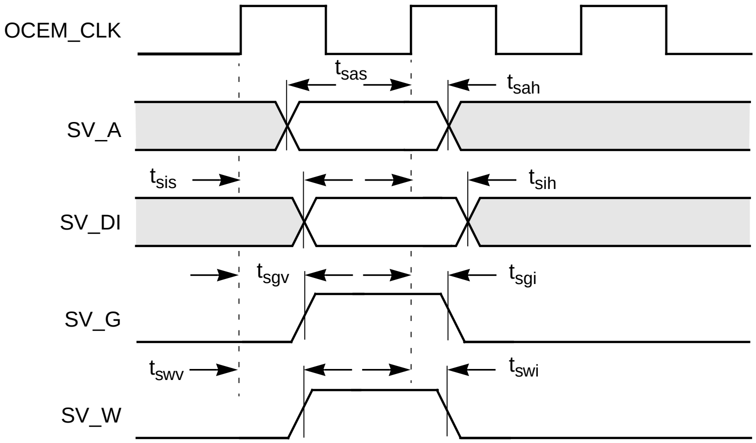


Figure 10.2 Reading the OCEM

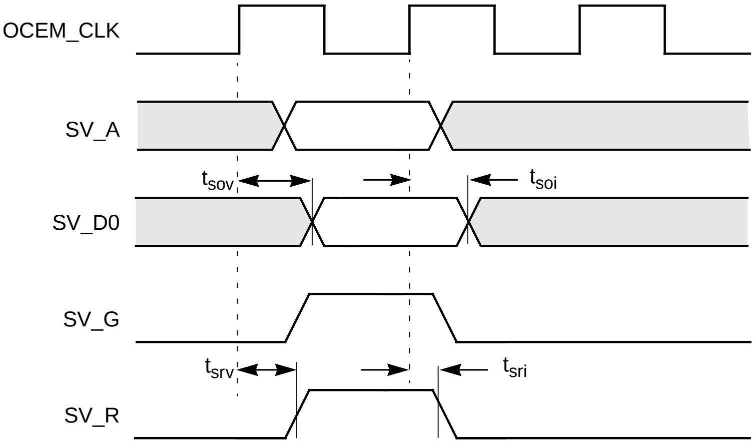


Table 10.2 OCEM Access Timing Constraints

Parameter	Timing Constraint	Min (ns)	Max (ns)	Remark
t_{sas}	SV_A Setup	24	–	Required by OCEM
t_{sah}	SV_A Hold	–	1	Required by OCEM
t_{sis}	SV_DI Setup	22	–	Required by OCEM
t_{sih}	SV_DI Hold	–	1	Required by OCEM
t_{sov}	SV_DO Valid	2	–	–
t_{soi}	SV_DO Invalid	–	1.1	–
t_{sgv}	SV_G Valid	–	1.6	Required by OCEM
t_{sgi}	SV_G Invalid	1	–	Required by OCEM
t_{srv}	SV_R Valid	–	1.6	Required by OCEM
t_{sri}	SV_R Invalid	1	–	Required by OCEM
t_{swv}	SV_W Valid	–	1.8	Required by OCEM
t_{swi}	SV_W Invalid	1.3	–	Required by OCEM

10.2.2 Data and Program Memory

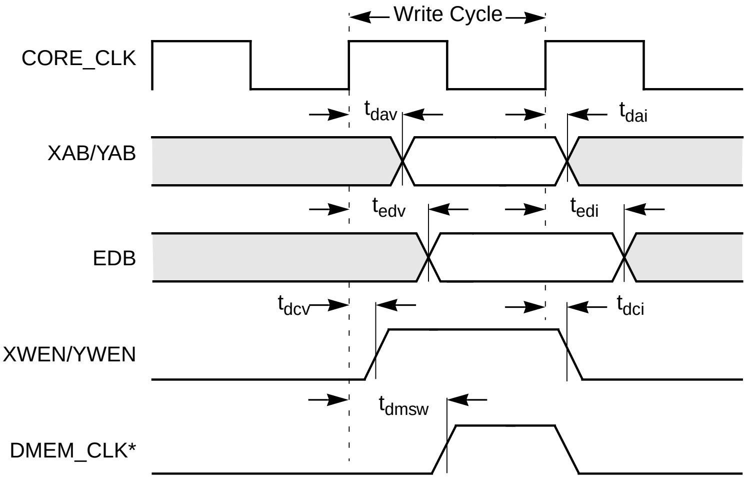
The figures in this section show the optional memory clocks, required for synchronous memory only. Typically, a CCU generates these memory clocks to optimize performance in any application. The core facilitates optimal data memory clocking with the signal RAM_WT. RAM_WT is an early indicator of a write transaction with the data memory. In designs where the data memory is in the critical path, RAM_WT can be used to turn off the data memory clock early in a read transaction cycle.

> Note: To interface with asynchronous memory, external strobing logic might be required.

The reminder of this section contains the following timing figures and tables:

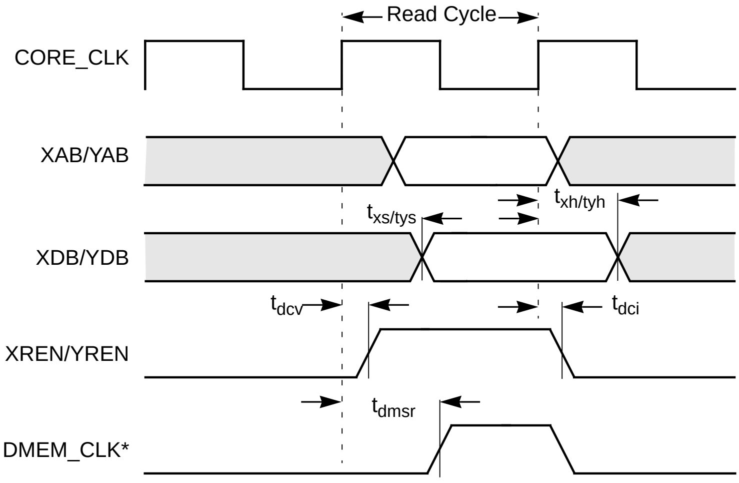
- Figure 10.3, “Data Memory Write Access”
- Figure 10.4, “Data Memory Read Access”
- Figure 10.5, “Program Memory Write Access”
- Figure 10.6, “Program Memory Read Access”
- Figure 10.7, “Memory Access with Wait State”
- Table 10.3, “Memory Interface Timing Constraints”

Figure 10.3 Data Memory Write Access



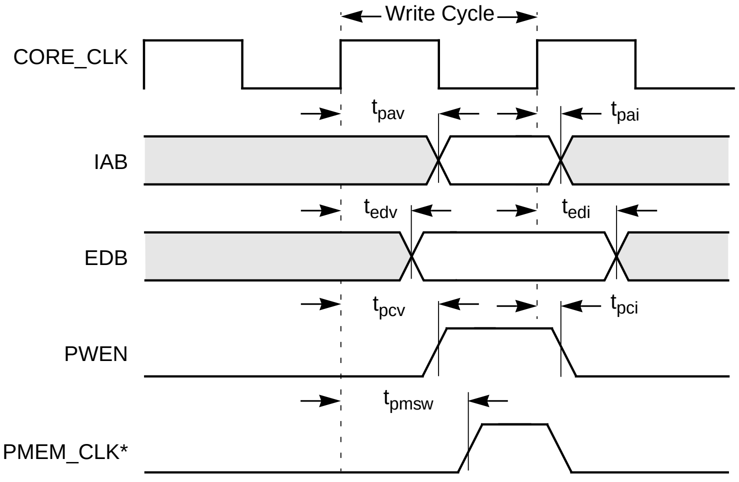
* Required for synchronous memory only.

Figure 10.4 Data Memory Read Access



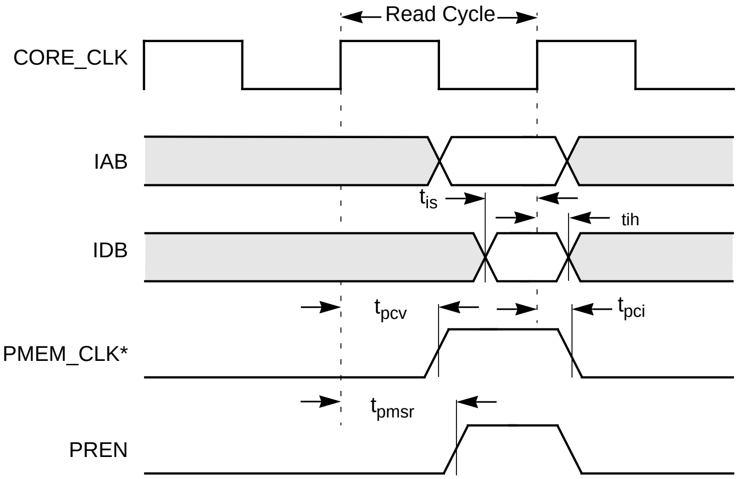
* Required for synchronous memory only.

Figure 10.5 Program Memory Write Access



* Required for synchronous memory only.

Figure 10.6 Program Memory Read Access



* Required for synchronous memory only.

Figure 10.7 shows the waveforms for a data memory read with one wait state. Because the data read transaction has one wait state, WAIT_CTL is held asserted for one cycle. The address bus XAB and the memory read enable XREN are kept stable during the wait cycle. The machine cycle is effectively extended to two MCLK cycles. The core reads the data from EDB at the end of the second cycle.

Figure 10.7 Memory Access with Wait State

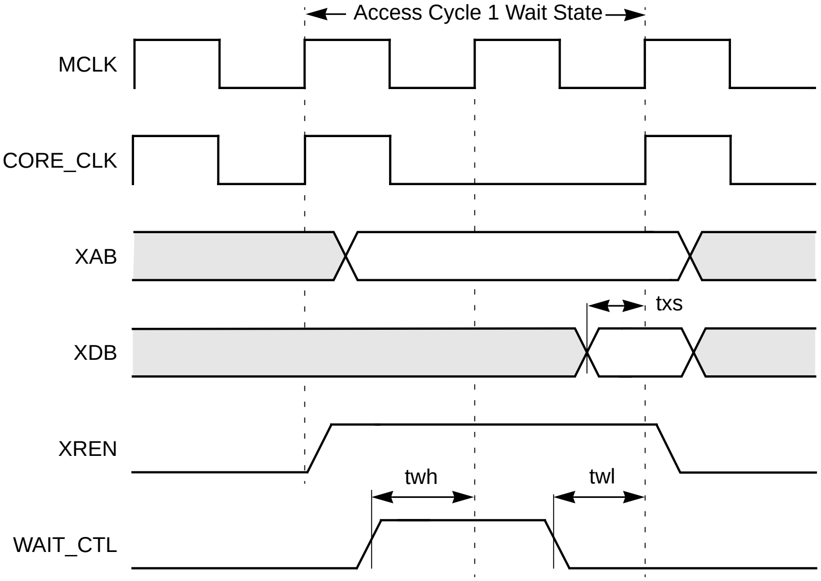


Table 10.3 Memory Interface Timing Constraints

Parameter	Timing Constraint	Min (ns)	Max (ns)	Remark
t_{dav}	Data Address Valid	–	3.8	–
t_{dai}	Data Address Invalid	0	–	–
t_{edv}	External Data Valid	–	8.6	–
t_{edi}	External Data Invalid	0	–	–
t_{dcv}	Data Controls Valid	–	4.1	–
t_{dci}	Data Controls Invalid	0	–	–
t_{xs}	X Data Setup	9.7	–	Required by core.
t_{xh}	X Data Hold	2	–	Required by core.
t_{ys}	Y Data Setup	9.7	–	Required by core.
t_{yh}	Y Data Hold	2	–	Required by core.

(Sheet 1 of 2)

Table 10.3 Memory Interface Timing Constraints (Cont.)

Parameter	Timing Constraint	Min (ns)	Max (ns)	Remark
t_{pav}	Program Address Valid	–	10.2	–
t_{pai}	Program Address Invalid	0	–	–
t_{pcv}	Program Controls Valid	–	11.6	–
t_{pci}	Program Controls Invalid	0	–	Required by core.
t_{is}	IDB (Program Data) Setup	1	–	Required by core.
t_{ih}	IDB (Program Data) Hold	2	–	Required by core.
t_{dmsw}	DMEM_CLK Write Skew[1]	See Note		
t_{dmsr}	DMEM_CLK Read Skew[2]	See Note		
t_{pmsw}	PMEM_CLK Write Skew[3]	See Note		
t_{pmsr}	PMEM_CLK Read Skew[4]	See Note		
t_{wh}	WAIT_CTL High Setup	18.6	–	Required by core.
t_{wl}	WAIT_CTL Low Setup	20.8	–	Required by core.

(Sheet 2 of 2)

[1] max(t_{dav}, t_{edv}, t_{dcv}) + memory setup
[2] cycle time - t_{xs}/t_{ys} - 1 ns
[3] max(t_{pav}, t_{edv}, t_{pcv},) + memory setup
[4] cycle time - $t_{is - 1 ns}$

10.2.3 User-Defined Registers

Figure 10.8 shows the waveforms for both a user-defined register read and a user-defined register write. Table 10.4 lists the timing constraints shown in this figure.

Figure 10.8 User-Defined Register Access

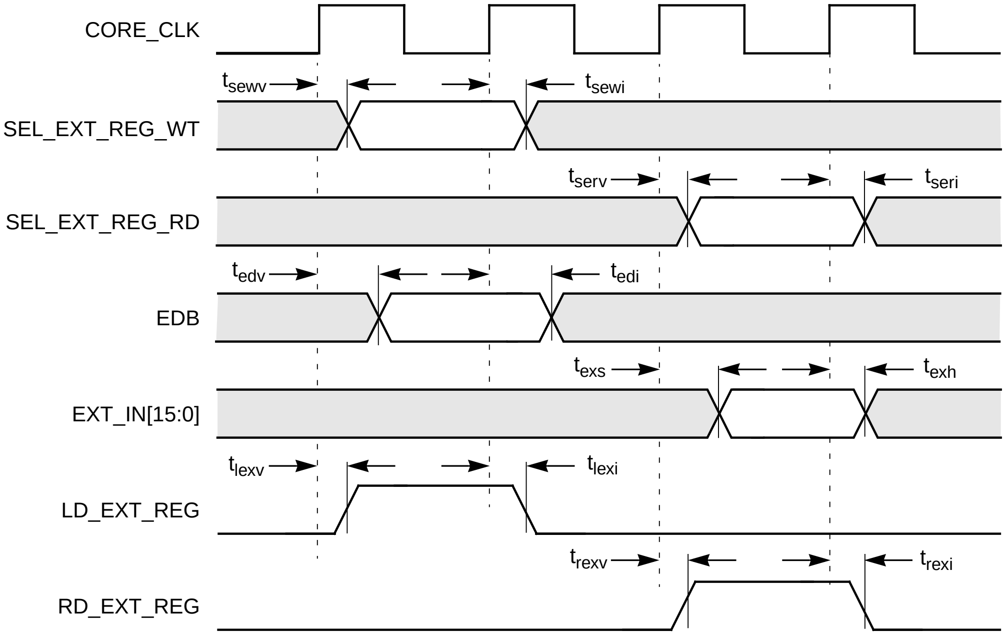


Table 10.4 User-Defined Register Interface Timing Values

Parameter	Timing Constraint	Min (ns)	Max (ns)	Remarks
t_{sewv}	SEL_EXT_REG_WT Valid	–	1.3	–
t_{sewi}	SEL_EXT_REG_WT Invalid	1.3	–	–
t_{serv}	SEL_EXT_REG_RD Valid	–	1.3	–
t_{seri}	SEL_EXT_REG_RD Invalid	1.3	–	–
t_{edv}	External Data Valid	–	8.6	–
t_{edi}	External Data Invalid	0	–	–
t_{exs}	EXT_IN[15:0] Setup	14	–	Required by core.
t_{exh}	EXT_IN[15:0] Hold	2	–	Required by core.
t_{lexv}	LD_EXT_REG Valid	–	1.5	–
t_{lexi}	LD_EXT_REG Invalid	1.4	–	–
t_{rexv}	RD_EXT_REG Valid	–	1.4	–
t_{rexi}	RD_EXT_REG Invalid	1.4	–	–

Appendix A CWDSP1650 Register Summary

This appendix contains a quick description of all the CWDSP1650 registers. Table A.1 lists the CWDSP1650 registers, any abbreviations used in this manual, and a page number for the description of each.

Table A.1 CWDSP1650 Registers

Functional Block	Register Name	Abbreviations	Reference Page
CBU Registers	Ax Accumulators	(AxE, AxH, or AxL)	4-3
	Bx Accumulators	(BxE, BxH, or BxL)	4-5
	X Register	–	4-6
	Y Register	–	4-6
	P Register	–	4-6
	Interrupt Context Swithing Registers	–	4-7
	Shift Value Register	SV	4-9
DAAU Registers	Address Registers 0-5	R0-R5	4-10
	Configuration Registers	CFGI, CFGJ	4-11
	Base Register	RB	4-11
	Stack Pointer Register	SP	4-12
	Alternative Bank Registers	R0B, R1B, R4B, CFGIB	4-12
	Min/Max Pointer Latching Register	MIXP	4-13
(Sheet 1 of 2)			

Table A.1 CWDSP1650 Registers (Cont.)

Functional Block	Register Name	Abbreviations	Reference Page
PCU Registers	Data Value Match Register	DVM	4-14
	Internal Configuration Register	ICR	4-14
	Program Counter	PC	4-15
	Loop Counter	LC	4-15
Status Registers	Status Register 0	ST0	4-16
	Status Register 1	ST1	4-18
	Status Register 2	ST2	4-19
CCU Register	CCU Register	–	6-19
OCEM Registers	Status 0 Register	–	8-4
	Status 1 Register	–	8-6
	Mode Register	–	8-7
	Data Address Breakpoint	–	8-9
	Data Address Mask	–	8-9
	Program Address Breakpoint Counter 3	–	8-10
	Program Address Breakpoint Counter 2	–	8-10
	Program Address Breakpoint Counter 1	–	8-10
	Program Address Breakpoint 3	–	8-10
	Program Address Breakpoint 2	–	8-10
	Program Address Breakpoint 1	–	8-10
	Program Flow Trace Register	–	8-10
ScanICE Register	Scan Control Register	–	9-7
(Sheet 2 of 2)			

Customer Feedback

We would appreciate your feedback on this document. Please copy the following page, add your comments, and fax it to us at the number shown.

If appropriate, please also fax copies of any marked-up pages from this document.

Important: Please include your name, phone number, fax number, and company address so that we may contact you directly for clarification or additional information.

Thank you for your help in improving the quality of our documents.

Reader’s Comments

Fax your comments to:

LSI Logic Corporation
Technical Publications
M/S E-198
Fax: 408.433.4333

Please tell us how you rate this document: *CWDSP1650 DSP Core Technical Manual.* Place a check mark in the appropriate blank for each category.

	Excellent	**Good**	**Average**	**Fair**	**Poor**
Completeness of information	____	____	____	____	____
Clarity of information	____	____	____	____	____
Ease of finding information	____	____	____	____	____
Technical content	____	____	____	____	____
Usefulness of examples and illustrations	____	____	____	____	____
Overall manual	____	____	____	____	____

What could we do to improve this document?

____

____

____

____

If you found errors in this document, please specify the error and page number. If appropriate, please fax a marked-up copy of the page(s).

____

____

____

Please complete the information below so that we may contact you directly for clarification or additional information.

Name ____________________ Date __________

Telephone ____________________ Fax ____________________

Title ____________________

Department ____________________ Mail Stop __________

Company Name ____________________

Street ____________________

City, State, Zip ____________________

U.S. Distributors by State

H. H. Hamilton Hallmark
W. E. Wyle Electronics

Alabama
Huntsville
H. H. Tel: 205.837.8700
W. E. Tel: 800.964.9953

Alaska
Anchorage
H. H. Tel: 800.332.8638
W. E. Tel: 907.257.8016

Arizona
Phoenix
H. H. Tel: 602.736.7000
W. E. Tel: 800.528.4040
Tucson
H. H. Tel: 520.742.0515

Arkansas
H. H. Tel: 800.327.9989

California
Irvine
H. H. Tel: 714.789.4100
W. E. Tel: 800.626.9953
Los Angeles
H. H. Tel: 818.594.0404
W. E. Tel: 800.288.9953
Sacramento
H. H. Tel: 916.632.4500
W. E. Tel: 800.627.9953
San Diego
H. H. Tel: 619.571.7540
W. E. Tel: 800.829.9953
San Jose
H. H. Tel: 408.435.3500
Santa Clara
W. E. Tel: 800.866.9953
Woodland Hills
H. H. Tel: 818.594.0404

Colorado
Denver
H. H. Tel: 303.790.1662
W. E. Tel: 800.933.9953

Connecticut
Cheshire
H. H. Tel: 203.271.5700
Wallingford
W. E. Tel: 800.605.9953

Delaware
North/South
H. H. Tel: 800.526.4812
Tel: 800.638.5988

Florida
Fort Lauderdale
H. H. Tel: 305.484.5482
W. E. Tel: 800.568.9953
Orlando
H. H. Tel: 407.657.3300
W. E. Tel: 407.740.7450
N. Florida
W. E. Tel: 800.395.9953
St. Petersburg
H. H. Tel: 813.507.5000

Georgia
Atlanta
H. H. Tel: 770.623.4400
W. E. Tel: 800.876.9953

Hawaii
H. H. Tel: 800.851.2282

Idaho
H. H. Tel: 801.266.2022

Illinois
North/South
H. H. Tel: 847.797.7300
Tel: 314.291.5350
Chicago
W. E. Tel: 800.853.9953

Indiana
Indianapolis
H. H. Tel: 317.575.3500
W. E. Tel: 317.581.6152

Iowa
Cedar Rapids
H. H. Tel: 319.393.0033

Kansas
Kansas City
H. H. Tel: 913.663.7900

Kentucky
Central/Northern/ Western
H. H. Tel: 800.984.9503
Tel: 800.767.0329
Tel: 800.829.0146

Louisiana
North/South
H. H. Tel: 800.231.0253
Tel: 800.231.5575

Maine
H. H. Tel: 800.272.9255

Maryland
Baltimore
H. H. Tel: 410.720.3400
W. E. Tel: 800.863.9953

Massachusetts
Boston
H. H. Tel: 508.532.9808
W. E. Tel: 800.444.9953
Marlborough
W. E. Tel: 508.480.9900

Michigan
Detroit
H. H. Tel: 313.416.5800
Grandville
H. H. Tel: 616.531.0345

Minnesota
Minneapolis
H. H. Tel: 612.881.2600
W. E. Tel: 800.860.9953

Mississippi
H. H. Tel: 800.633.2918

Missouri
St. Louis
H. H. Tel: 314.291.5350

Montana
H. H. Tel: 800.526.1741

Nebraska
H. H. Tel: 800.332.4375

Nevada
Las Vegas
H. H. Tel: 800.528.8471
W. E. Tel: 702.765.4040

New Hampshire
H. H. Tel: 800.272.7117

New Jersey
North/South
H. H. Tel: 201.515.1641
Tel: 609.222.6400
Pine Brook
W. E. Tel: 800.862.9953

New Mexico
Albuquerque
H. H. Tel: 505.293.5119

New York
Long Island
H. H. Tel: 516.434.7400
W. E. Tel: 800.861.9953
Rochester
H. H. Tel: 716.475.9130
W. E. Tel: 800.319.9953
Syracuse
H. H. Tel: 315.453.4000

North Carolina
Raleigh
H. H. Tel: 919.872.0712
W. E. Tel: 800.560.9953

North Dakota
H. H. Tel: 800.829.0116

Ohio
Cleveland
H. H. Tel: 216.498.1100
W. E. Tel: 800.763.9953
Dayton
H. H. Tel: 614.888.3313
W. E. Tel: 800.763.9953

Oklahoma
Tulsa
H. H. Tel: 918.459.6000

Oregon
Portland
H. H. Tel: 503.526.6200
W. E. Tel: 800.879.9953

Pennsylvania
Pittsburgh
H. H. Tel: 412.281.4150
Philadelphia
H. H. Tel: 800.526.4812
W. E. Tel: 800.871.9953

Rhode Island
H. H. 800.272.9255

South Carolina
H. H. Tel: 919.872.0712

South Dakota
H. H. Tel: 800.829.0116

Tennessee
East/West
H. H. Tel: 800.241.8182
Tel: 800.633.2918

Texas
Austin
H. H. Tel: 512.219.3700
W. E. Tel: 800.365.9953
Dallas
H. H. Tel: 214.553.4300
W. E. Tel: 800.955.9953
El Paso
H. H. Tel: 800.526.9238
Houston
H. H. Tel: 713.781.6100
W. E. Tel: 800.888.9953
Rio Grande Valley
H. H. Tel: 210.412.2047

Utah
Draper
W. E. Tel: 800.414.4144
Salt Lake City
H. H. Tel: 801.365.3800
W. E. Tel: 800.477.9953

Vermont
H. H. Tel: 800.272.9255

Virginia
H. H. Tel: 800.638.5988

Washington
Seattle
H. H. Tel: 206.882.7000
W. E. Tel: 800.248.9953

Wisconsin
Milwaukee
H. H. Tel: 414.513.1500
W. E. Tel: 800.867.9953

Wyoming
H. H. Tel: 800.332.9326

Sales Offices and Design Resource Centers

LSI Logic Corporation
Corporate Headquarters
Tel: 408.433.8000
Fax: 408.433.8989

NORTH AMERICA

California
Irvine
◆ Tel: 714.553.5600
Fax: 714.474.8101

San Diego
Tel: 619.613.8300
Fax: 619.613.8350

Silicon Valley
Sales Office
Tel: 408.433.8000
Fax: 408.954.3353
Design Center
◆ Tel: 408.433.8000
Fax: 408.433.7695

Colorado
Boulder
Tel: 303.447.3800
Fax: 303.541.0641

Florida
Boca Raton
Tel: 561.989.3236
Fax: 561.989.3237

Illinois
Schaumburg
◆ Tel: 847.995.1600
Fax: 847.995.1622

Kentucky
Bowling Green
Tel: 502.793.0010
Fax: 502.793.0040

Maryland
Bethesda
Tel: 301.897.5800
Fax: 301.897.8389

Massachusetts
Waltham
◆ Tel: 617.890.0180
Fax: 617.890.6158

Minnesota
Minneapolis
◆ Tel: 612.921.8300
Fax: 612.921.8399

New Jersey
Edison
◆ Tel: 732.549.4500
Fax: 732.549.4802

New York
New York
Tel: 716.223.8820
Fax: 716.223.8822

North Carolina
Raleigh
Tel: 919.783.8833
Fax: 919.783.8909

Oregon
Beaverton
Tel: 503.645.0589
Fax: 503.645.6612

Texas
Austin
Tel: 512.388.7294
Fax: 512.388.4171

Dallas
◆ Tel: 972.788.2966
Fax: 972.233.9234

Houston
Tel: 281.379.7800
Fax: 281.379.7818

Washington
Issaquah
Tel: 425.837.1733
Fax: 425.837.1734

Canada
Ontario
Ottawa
◆ Tel: 613.592.1263
Fax: 613.592.3253

Toronto
◆ Tel: 416.620.7400
Fax: 416.620.5005

Quebec
Montreal
◆ Tel: 514.694.2417
Fax: 514.694.2699

INTERNATIONAL

Australia
Reptechnic Pty Ltd
New South Wales
◆ Tel: 612.9953.9844
Fax: 612.9953.9683

Denmark
LSI Logic Development Centre
Ballerup
Tel: 45.44.86.55.55
Fax: 45.44.86.55.56

France
LSI Logic S.A.
Immeuble Europa
Paris
◆ Tel: 33.1.34.63.13.13
Fax: 33.1.34.63.13.19

Germany
LSI Logic GmbH
Munich
◆ Tel: 49.89.4.58.33.0
Fax: 49.89.4.58.33.108

Stuttgart
Tel: 49.711.13.96.90
Fax: 49.711.86.61.428

Hong Kong
AVT Industrial Ltd
Hong Kong
Tel: 852.2428.0008
Fax: 852.2401.2105

India
LogiCAD India Private Ltd
Bangalore
◆ Tel: 91.80.526.2500
Fax: 91.80.338.6591

Israel
LSI Logic
Ramat Hasharon
◆ Tel: 972.3.5.403741
Fax: 972.3.5.403747

Netanya
◆ Tel: 972.9.657190
Fax: 972.9.657194

Italy
LSI Logic S.P.A.
Milano
◆ Tel: 39.39.687371
Fax: 39.39.6057867

Japan
LSI Logic K.K.
Tokyo
◆ Tel: 81.3.5463.7821
Fax: 81.3.5463.7820

Osaka
◆ Tel: 81.6.947.5281
Fax: 81.6.947.5287

Korea
LSI Logic Corporation of Korea Ltd.
Seoul
◆ Tel: 82.2.528.3400
Fax: 82.2.528.2250

Singapore
LSI Logic Pte Ltd
Singapore
◆ Tel: 65.334.9061
Fax: 65.334.4749

Spain
LSI Logic S.A.
Madrid
◆ Tel: 34.1.556.07.09
Fax: 34.1.556.75.65

Sweden
LSI Logic AB
Stockholm
◆ Tel: 46.8.444.15.00
Fax: 46.8.750.66.47

Switzerland
LSI Logic Sulzer AG
Brugg/Biel
Tel: 41.32.536363
Fax: 41.32.536367

Taiwan
LSI Logic Asia-Pacific
Taipei
◆ Tel: 886.2.718.7828
Fax: 886.2.718.8869

Cheng Fong Technology Corporation
Tel: 886.2.910.1180
Fax: 886.2.910.1175

Jeilin Technology Corporation, Ltd.
Tel: 886.2.248.4828
Fax: 886.2.242.4397

Lumax International Corporation, Ltd
Tel: 886.2.788.3656
Fax: 886.2.788.3568

Macro-Vision Technology Inc.
Tel: 886.2.698.3350
Fax: 886.2.698.3348

United Kingdom
LSI Logic Europe Ltd
Bracknell
◆ Tel: 44.1344.426544
Fax: 44.1344.481039

◆ Sales Offices with Design Resource Centers