LSI LOGIC

MiniRISC™ CW400x Building Blocks Technical Manual

Document DB14-000022-00, First Edition (September 1996)
This document describes Revision A of LSI Logic Corporation's MiniRISC™ CW400x Building Blocks and will remain the official reference source for all revisions/releases of this product until rescinded by an update.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 (or 415-940-6877 outside the U.S. and Canada) and ask for Department JDS; or visit us at http://www.lsilogic.com.

Preface

This book is the primary reference and Technical Manual for the MiniRISC™ CW400x Building Blocks. It contains a complete functional and operational description of the Building Blocks. For layout guidelines, see Chapter 8 of the *MiniRISC CW400x Microprocessor Core Technical Manual*. For complete physical and electrical specifications, see the *MiniRISC CW400x Building Blocks Technical Manual Specifications Addendum*.

Audience

This document assumes that you have some familiarity with microprocessors and related support devices. This book is written for:

- ♦ Engineers and managers who are evaluating the processor for possible use in a system
- ♦ Engineers who are designing the processor into a system

Organization

This document has the following chapters and appendixes:

- ♦ Chapter 1, **Introduction**
- ♦ Chapter 2, **Adding a Coprocessor**
- ♦ Chapter 3, **Multiply/Divide Unit (MDU)**
- ♦ Chapter 4, **Memory Management Unit (MMU)**
- ♦ Chapter 5, **Basic BIU and Cache Controller (BBCC)**
- ♦ Chapter 6, **Adding or Removing Write Buffers**
- ♦ Chapter 7, **Timer**
- ♦ Chapter 8, **Debugger (DBX)**

Related Publications

MiniRISC™ CW400x Building Blocks Technical Manual Specifications Addendum, Order No. C14031.A1

MiniRISC™ CW400x Microprocessor Core Technical Manual, Order No. C14030

MiniRISC™ CW400x Microprocessor Core Technical Manual CW4001 and CW4002 Specifications Addendum, Order No. C14030.AD1

MiniRISC™ CW4001 Microprocessor Core Preliminary Datasheet, Order No. C15001

MiniRISC™ MR4001 Microprocessor Lead Vehicle Preliminary Datasheet, Order No. C15005

MiniRISC™ MR4002 Microprocessor Reference Device Preliminary Datasheet, Order No. C15008

Conventions Used in This Manual

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

The following signal naming conventions are used throughout this manual:

- A level-significant signal that is true or valid when the signal is LOW always has an overbar ($\overline{\quad}$) over its name.
- An edge-significant signal that initiates actions on a HIGH-to-LOW transition always has an overbar ($\overline{\quad}$) over its name.

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” before the number—for example, 0x32CF. Binary numbers are indicated by a subscripted “2” following the number—for example, $0011.0010.1100.1111_2$.

Contents

Figures

Chapter 1 Introduction

This chapter introduces LSI Logic's MiniRISC CW400x building blocks. Chapter 2 describes how to attach a coprocessor to the CW400x Core, while Chapters 3 through 8 describe the building blocks in detail. For building block Methodologies and Layout Guidelines, see Chapter 8 of the *MiniRISC CW400x Microprocessor Core Technical Manual*.

This chapter contains the following sections:

1.1 System Overview

The MiniRISC CW400x Microprocessor Core family, components of the LSI Logic CoreWare® Library, are exceptionally compact, high-performance microprocessors compatible with the MIPS R4000 MIPS-II Instruction Set (for details see Chapter 4 of the *MiniRISC CW400x Micro-processor Core Technical Manual*). The CW400x can be easily designed into a wide range of products. The CW400x can be combined with industry standard cores and proprietary functional building blocks to create a completely customized embedded system on a chip. LSI Logic currently provides the following optional building blocks:

- ♦ Multiply/Divide Unit (MDU)
- ♦ Memory Management Unit (MMU)
- ♦ Basic Bus Interface Unit and Cache Controller (BBCC)
- ♦ Write Buffers
- ♦ Timer
- ♦ Debugger (DBX)

System designers can use these building blocks (unmodified or modified) and/or add their own customized logic to the CW400x Core.

LSI Logic also provides the following external modules (for more information, see Chapter 6 of the *MiniRISC CW400x Microprocessor Core Technical Manual*):

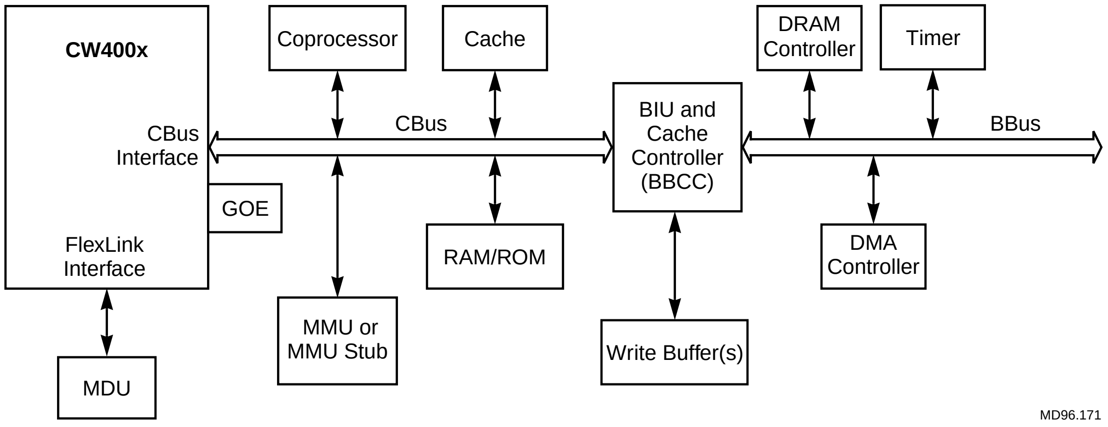
- Global Output Enable Module (GOE)
- MMU Stub (to be used if there is no MMU in the system)

The CW400x has been optimized for low-power and cost-sensitive applications such as portable telecommunications, digital cameras, and consumer multimedia systems.

The CW400x's FlexLink Interface allows customer-specific microprocessor instructions. The core implements a simple three-stage pipeline and provides a single cache/memory interface for both instructions and data. The core implements full scan to achieve greater than 99% fault coverage.

Figure 1.1 shows how the CW400x Microprocessor Core interfaces with system logic in a typical customer design.

Figure 1.1
CW400x in a Typical System



1.2 CW400x Features Summary

The CW400x has the following features:

- MIPS-II Instruction Set Compatible
- Configurable, compact, modular design and unified bus architecture
- Simple three-stage pipeline: Fetch, Execute, and Writeback
- WAITI (Wait for Interrupt) Instruction for power savings
- Powerful FlexLink Interface allows customer-specific microprocessor instructions
- High-performance Coprocessor Interface for user-definable coprocessors and high-performance hardware Floating Point Unit
- 32-bit memory and cache interfaces
- Optional building blocks: Timer, MMU, MDU, BBCC
- 3.3-volt operation
- Implementation of full scan to achieve 99% fault coverage
- 85-MHz worst case commercial maximum clock rate using high-performance 0.35-micron process
- 85 MIPS peak, 64 MIPS sustained with standard compiled MIPS code at 85 MHz
- Performance and software development, VHDL, Verilog, and gate-level, timing-accurate models available
- Compatible with the full range of MIPS, third party software development, and system verification environment tools
- Fully testable in embedded ASIC designs
- MR4001 Lead Vehicle chip available with cache, MMU, and MDU

1.3 MiniRISC Product Family

The MiniRISC product family has all the necessary tools to develop a system on a chip, including:

- LSI Logic’s MiniSIM™ architectural simulator
- Verilog and VHDL models
- A system verification environment
- A PROM monitor
- Third party software support
- A core bond-out chip for emulation

1.4 CoreWare Program

Through the CoreWare program, LSI Logic lets customers combine the CW400x Microprocessor Core with other cores on a single chip to create products uniquely suited to specific applications. This approach – combining high-performance building blocks, sophisticated design software, and expert support – provides unparalleled design flexibility and lets designers create high-quality, leading-edge products for a wide range of markets.

The CoreWare program consists of three main elements: a library of cores, a design development and simulation package, and expert applications support. The CoreWare library contains a wide range of complex cores based on accepted and emerging industry standards, including high-speed interconnection, digital video, DSP, and others. LSI Logic provides a complete framework for device and system development and simulation. LSI Logic's advanced ASIC technologies consistently produce Right-First-Time™ silicon. LSI Logic's in-house experts provide design support from system architecture definition through chip layout and test vector generation.

1.4.1 CoreWare Building Blocks

The CoreWare building blocks include elements based on the LSI Logic high-performance standard products as well as other, industry-standard products. The CoreWare building blocks, which include embedded MiniRISC MIPS processors, bus interface controllers, and a family of floating-point processors, are fully supported library elements for use in the LSI Logic hardware development environment. The building blocks include gate-level simulation models with timing information, so designers can accurately simulate device performance and trade off various implementation options. In addition to gate-level simulation models, the building blocks also include behavioral simulation models.

1.4.2 Design Environment

The new ASIC families are supported by LSI Logic's comprehensive system-on-a-chip design methodology. This design methodology uses both internally developed and industry-standard tools integrated with the LSI ToolKit. LSI ToolKit is a system of software and libraries that lets engineers use third-party software to access LSI Logic's technology. Designers can select from a suite of industry standard simulators, synthesizers, timing analyzers, and test tools that are seamlessly integrated into a common environment for verification and sign-off.

1.4.3 Expert Support

LSI Logic’s in-house experts support the CoreWare program with high-level design experience in a wide variety of application areas. These experts provide design support from system architecture definition through chip layout and test vector generation. They help determine how many functions to integrate on a single chip, trading off functionality versus cost to find the most cost-effective solution.

Chapter 2
Adding a Coprocessor

This chapter explains how to integrate a customer-defined coprocessor with the MiniRISC CW400x Microprocessor Core. An example of a customer-defined coprocessor might be a Floating Point Unit or Graphics Processor.

This chapter contains the following sections:

2.1 Overview

A coprocessor is a user-defined, external sub-module to the CW400x. Since a coprocessor is not a stand-alone unit, it can either pass data through the CW400x, or pass it directly to, and from, external memory.

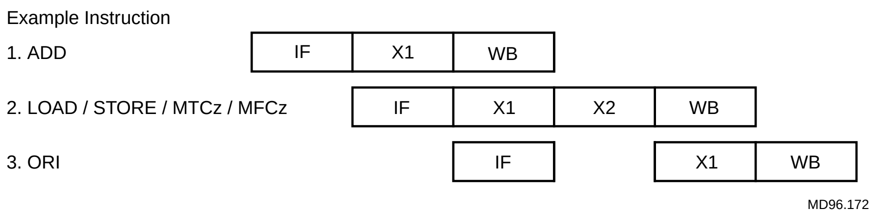
A CW400x coprocessor can:

♦ Read the data bus for instructions
♦ Read/write to external memory (Load/Store)
♦ Read/write to CW400x registers (MTCz/MFCz)
♦ Stall and interrupt the CW400x

The system designer must adhere to four guidelines to correctly implement an interface between the CW400x Microprocessor Core and a coprocessor.

1. The coprocessor must be compatible with the CW400x three-stage pipeline. Figure 2.1 shows a typical pipeline flow of instruction fetches and data transactions. For more information, see Section 2.3, "Pipeline Architecture," of the *MiniRISC CW400x Microprocessor Core Technical Manual*. The coprocessor should execute instructions in lock-step with the CW400x (whatever instructions are in the Instruction Fetch (IF)/Execute (X)/Writeback (WB) Stages in a specific cycle in the CW400x should be the same ones in the coprocessor's IF/X/WB Stages).

Figure 2.1
Typical Pipeline
Flow



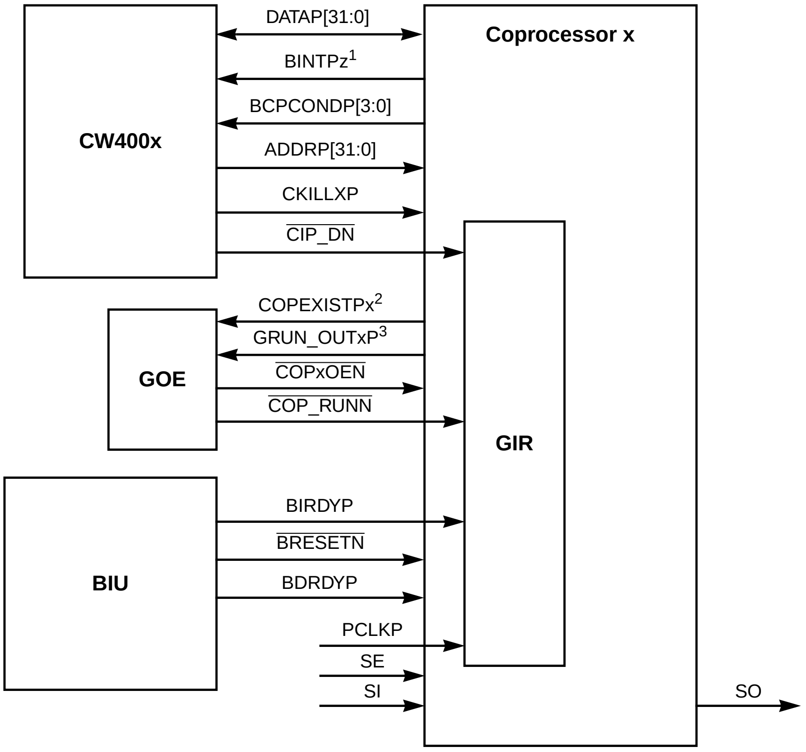
2. The system designer must implement a defined set of signals between the CW400x and the coprocessor. These signals allow the coprocessor to perform the four transaction types listed above.
3. The coprocessor must conform to specific signal protocol when performing the four transactions listed above.
4. The user must include two modules, in addition to the coprocessor, as part of the CW400x-coprocessor interface:
   – GOE (Global Output Enable Module), which is external to the coprocessor
   – GIR (Global Instruction Register Module), which is internal to the coprocessor

For more information on the GOE, see Chapter 6 of the *MiniRISC CW400x Microprocessor Core Technical Manual*. For more information on the GIR, see Section 2.9, "Global Instruction Register Module (GIR)."

2.2 Connection Block Diagram

Figure 2.2 shows how the coprocessor connects to the CW400x Microprocessor Core and a BIU. Notice the required interface modules, the GOE and the GIR, are included. LSI Logic recommends that the GIR be integrated into the coprocessor, and that the GOE remain external to the coprocessor.

Figure 2.2 Coprocessor in a CW400x System



1. z = 3, 4, or 5. Coprocessor 1 connects to BINTP3, Coprocessor 2 connects to BINTP4, Coprocessor 3 connects to BINTP5.
2. x = 1, 2 or 3.
3. x = 1 or 2.

MD96.173

2.3 Signals

This section describes the signals that comprise the bit-level interface of a coprocessor. Tables 2.1 through 2.3 summarize the coprocessor signals. Detailed descriptions follow the tables.

The signals are described in alphabetical order by mnemonic. Each signal definition contains the mnemonic and the full signal name. The mnemonics for active LOW signals end with an “N” and have an overbar over their names.

In the descriptions that follow, “assert” means to drive TRUE or active and “deassert” means to drive FALSE or inactive.

Table 2.1
Coprocessor Input Signals Summary

Input	Source	Description
ADDRP[31:0]	CW400x	Address Bus
$\overline{\text{BRESETN}}$	BIU	Coprocessor Reset
BDRDYP	BIU	Load Word Ready - Data on Data Bus
BIRDYP	BIU	Instruction Ready
$\overline{\text{CIP\_DN}}$	CW400x	CW400x Instruction/Data Indication
CKILLXP	CW400x	Cancel Instruction in Execute Stage
$\overline{\text{COPxOEN}}$	GOE	Coprocessor Enable (x =1, 2, or 3)
$\overline{\text{COP\_RUNN}}$	GOE	Coprocessor Run[1]
PCLKP	System Logic	System Clock
SE	System Logic	Scan Enable
SI	Scan Chain	Scan Data In

1. Since the GOE does not have a $\overline{\text{COP\_RUNN}}$ signal, the user must connect $\overline{\text{COP\_RUNN}}$ to either $\overline{\text{CRUN\_INN}}$, $\overline{\text{BRUN\_INN}}$, or $\overline{\text{MRUN\_INN}}$. LSI Logic recommends using the least loaded of the three.

Table 2.2
Coprocessor Output Signals Summary

Output	Destination	Description
BCPCONDP[3:0]	CW400x	Coprocessor Condition
BINTPz	CW400x	Interrupts[1] (z = 0, 1, 2, 3, 4, or 5)
COPEXISTPx	GOE	Coprocessor Exists (x = 1, 2, or 3)
GRUN_OUTxP	GOE	Coprocessor Run (x = 1 or 2)
SO	Scan Chain	Scan Data Out

1. Coprocessor 1 connects to BINTP3, Coprocessor 2 connects to BINTP4, Coprocessor 3 connects to BINTP5 to maintain MIPS compatibility.

Table 2.3
Coprocessor Bidirectional Signals Summary

Bidirectional	Connect	Description
DATAP[31:0]	CW400x	Data Bus

ADDRP[31:0]	**Address Bus** The CW400x drives these signals with the instruction or data address.	**Input**
BCPCONDP[3:0]	**Coprocessor Condition** These signals inform the CW400x of the corresponding Coprocessor Condition. BCPCONDP[3:0] correspond to Coprocessors 3, 2, 1, 0. The CW400x tests these signals during the Execute Stage of BCzF, BCzFL, BCzT, and BCzTL instructions.	**Output**
BDRDYP	**Load Word Ready - Data on Data Bus** The BIU asserts this signal to inform the coprocessor that DATAP[31:0] contains valid data for a data fetch.	**Input**
BINTPz	**Interrupt (z = 0, 1, 2, 3, 4, or 5)** The coprocessor asserts this signal to cause the CW400x to take an Interrupt Exception when interrupts are enabled. To maintain MIPS compatibility, Coprocessor 1 connects to BINTP3, Coprocessor 2 connects to BINTP4, Coprocessor 3 connects to BINTP5.	**Output**
BIRDYP	**Instruction Ready** The BIU asserts this signal to inform the coprocessor that DATAP[31:0] contains valid data for an instruction fetch.	**Input**
$\overline{\text{BRESETN}}$	**Coprocessor Reset** The BIU asserts this signal to reset the coprocessor. (If the BBCC is present, the BBCC $\overline{\text{BCPURESETN}}$ output drives this input.)	**Input**
$\overline{\text{CIP\_DN}}$	**CW400x Instruction/Data Indication** This signal qualifies the type of memory fetch. The CW400x drives this signal HIGH to indicate that it is performing an instruction fetch. The CW400x drives this signal LOW to indicate that it is performing a data fetch.	**Input**
CKILLXP	**Cancel Instruction in Execute Stage** After the CW400x kills an instruction in the Execute Stage, it asserts this signal to cause the coprocessor to also kill the instruction in the Execute Stage.	**Input**
COPEXISTPx	**Coprocessor Exists (x = 1, 2, or 3)** The coprocessor asserts this signal to inform the GOE that it is Coprocessor x.	**Output**

$\overline{\text{COP\_RUNN}}$ **Coprocessor Run** **Input**
The GOE asserts this signal to inform the coprocessor that it can continue running. The GOE deasserts this signal to stall the coprocessor.

$\overline{\text{COPxOEN}}$ **Coprocessor Enable (x = 1, 2, or 3)** **Input**
This signal from the GOE indicates which Coprocessor (1, 2, or 3) drives the data bus.

DATAP[31:0] **Data Bus** **Bidirectional**
These signals transfer data to, and from, the CW400x.

GRUN_OUTxP
Coprocessor Run (x = 1 or 2) **Output**
This output is an input to the GOE. Deasserting this signal LOW stalls the CW400x.

PCLKP **System Clock** **Input**
This signal is the global clock input.

SE **Scan Enable** **Input**
Asserting this signal enables the scan chain.

SI **Scan Data In** **Input**
This signal is the scan data input.

SO **Scan Data Out** **Output**
This signal is the scan data output.

2.4 Instructions

Table 2.4 summarizes the predefined instructions that the CW400x supports for Coprocessors 1 through 3. For more detailed descriptions, see Section 4.10, “Coprocessor Instructions,” of the *MiniRISC CW400x Microprocessor Core Technical Manual*.

The user can define new coprocessor instructions, as long as they adhere to the opcode bit encoding defined in Chapter 4 of the *MiniRISC CW400x Microprocessor Core Technical Manual*.

Table 2.4
Coprocessor
Instructions

Instruction	Description
BCzT	Branch on Coprocessor z True[1]
BCzF	Branch on Coprocessor z False
BCzTL	Branch on Coprocessor z True Likely
BCzFL	Branch on Coprocessor z False Likely
CTCz	Move Control to Coprocessor z
CFCz	Move Control from Coprocessor z
LWCz	Load Word to Coprocessor z
MTCz	Move to Coprocessor z
MFCz	Move From Coprocessor z
SWCz	Store Word from Coprocessor z

1. z = 1, 2, or 3.

2.5 Read/Write Transactions

This section explains how the CW400x sends and receives data to, and from, an attached coprocessor. It also provides a functional description and waveform for each coprocessor transaction type. The coprocessor must decode its own instructions off the data bus, DATAP[31:0], and know when to read, and write, DATAP[31:0].

There are several guidelines to observe, and several signals to monitor, when performing these transactions. Control signals are valid during run cycles ($\overline{\text{COP\_RUNN}}$ asserted), and pipe stages are extended by stalls. The GOE asserts the Run/Stall Signal, $\overline{\text{COP\_RUNN}}$, for every bus cycle, including the first cycle of the X2 Stage. (For a detailed description of the pipeline stages and Run/Stall Cycles, see Chapter 2 of the *MiniRISC CW400x Microprocessor Core Technical Manual*.)

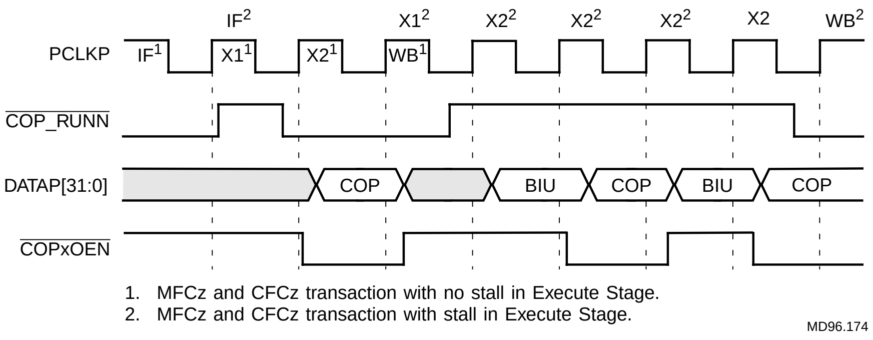
The coprocessor should also monitor the CKILLXP signal to determine when to invalidate an instruction. The coprocessor should use CKILLXP to prevent writes (MTCz/CTCz/LWCz) to coprocessor registers, to prevent altering the coprocessor state. Using CKILLXP properly in load invalidation is crucial to maintaining coprocessor integrity. If CKILLXP is asserted in either the X1 or X2 Stage, the load is not scheduled, and is, thus, invalidated. If the CKILLXP occurs in the X1 Stage, the X2 Stage will not occur. However, once the load passes the X2 Stage, it has been scheduled and cannot be invalidated.

2.5.1 CW400x Register Writes from Coprocessor (CFCz, MFCz)

The data for CFCz or MFCz should be available on the data bus during the second Execute (X2) Stage of the instruction whenever $\overline{\text{COPxOEN}}$ is asserted. If the data is not ready to be driven onto the data bus, the coprocessor should stall the CW400x by deasserting GRUN_OUTxP.

When the coprocessor does drive the data bus ($\overline{\text{COPxOEN}}$ asserted), the data should be valid on the rising edge of the clock. This cycle (X2 Stage) is a CW400x Register write, so the CW400x latches the data from the data bus. The coprocessor must monitor the appropriate $\overline{\text{COPxOEN}}$ to know when to drive the data bus.

Figure 2.3 CFCz, MFCz Waveforms


IF2
X12
X22
X22
X22
X2
WB2
PCLKP
IF1
X11
X21
WB1
COP_RUNN
DATAP[31:0]
COP
BIU
COP
BIU
COP
COPxOEN
1. MFCz and CFCz transaction with no stall in Execute Stage.
2. MFCz and CFCz transaction with stall in Execute Stage.
MD96.174

2.5.2 Coprocessor Register Writes from Memory (LWCz)

The CW400x Coprocessor Interface supports two mechanisms for reads from external memory (loads): load scheduling and non-load scheduling. The CW400x executes instructions while it is waiting for the load data from external memory, until a data dependency occurs, then it stalls. The coprocessor should emulate this behavior if it supports load scheduling, that is, it should continue to execute instructions and stall only if there is a data dependency. Assertion of BDRDYP informs the coprocessor that the LWCz has fetched the data, and the data is ready on the data bus. The coprocessor should immediately write the data on the bus back to the Coprocessor Register File to preserve the pipeline.

If the coprocessor does not support load-scheduling, it should signal a stall in the Execute Stage by deasserting GRUN_OUTxP in the X2 Stage. The CW400x waits until the load data is ready (the BIU asserts BDRDYP and the coprocessor asserts GRUN_OUTxP) before resuming execution. When BDRDYP is asserted, the coprocessor should latch the load data from the data bus and write it in to its register file in the next run cycle (WB).

Figure 2.4 shows load scheduling LWCz waveforms. Figure 2.5 shows non-load scheduling LWCz waveforms.

Figure 2.4
Load Scheduling LWCz Waveforms

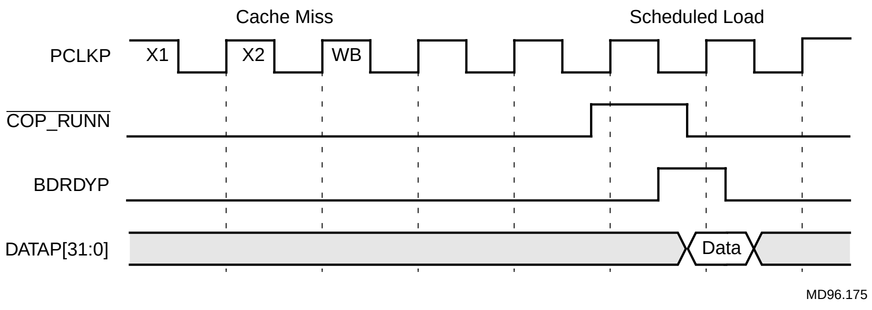
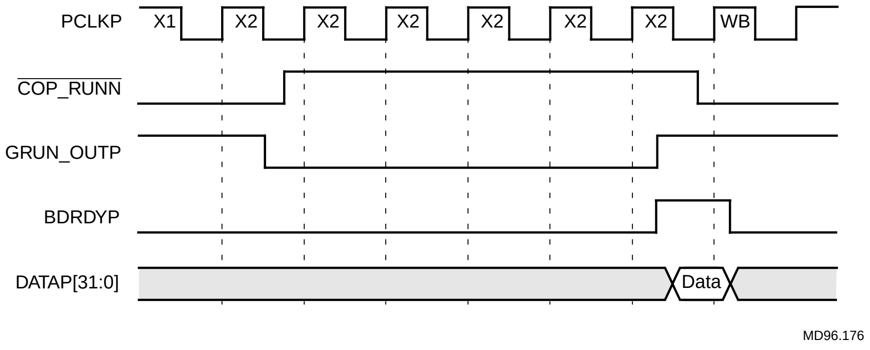


Figure 2.5
Non-Load Scheduling LWCz Waveforms

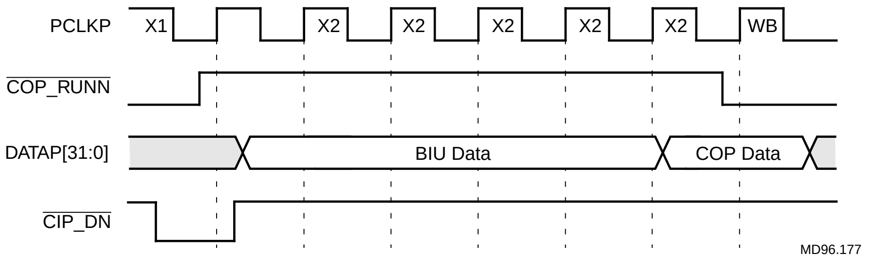


2.5.3 Coprocessor Register Writes from CW400x (CTCz, MTCz)

The data for MTCz or CTCz is always ready on the data bus during the last cycle of the instruction's X2 Stage. Thus, the data latched in the final cycle of the X2 Stage is the valid data. The data asserted in the previous cycles in the X2 Stage should be disregarded. If the coprocessor requires more than one cycle to latch the data, or is not ready to latch the data, it should deassert GRUN_OUTxP to stall the CW400x.

Figure 2.6 shows MTCz/CTCz Waveforms.

Figure 2.6
MTCz/CTCz Waveforms



2.5.4 Memory Writes from Coprocessor to External Memory (SWCz)

The data for the write to memory should be ready in the X2 Stage of the store (SWCz). If it is not, the coprocessor should stall the pipeline by deasserting GRUN_OUTxP. The coprocessor should always drive the data bus with the data for the store when the appropriate $\overline{\text{COPxOEN}}$ is asserted during the X2 Stage.

See Figure 2.3 for the waveform. Substitute SWCz for MTCz/CTCz as the instruction being executed.

2.6 Condition Bits

The CW400x supports the Branch on Coprocessor Condition for Coprocessors 0 through 3. The instructions BCzT/L and BCzF/L branch to Coprocessor z depending on the state of the Condition Bit. The coprocessor can set these bits in any way. The Coprocessor Usability Bit must be set in the CW400x Status Register for the coprocessor instructions relating to the Condition Bits to function properly.

2.7 Interrupt Protocol

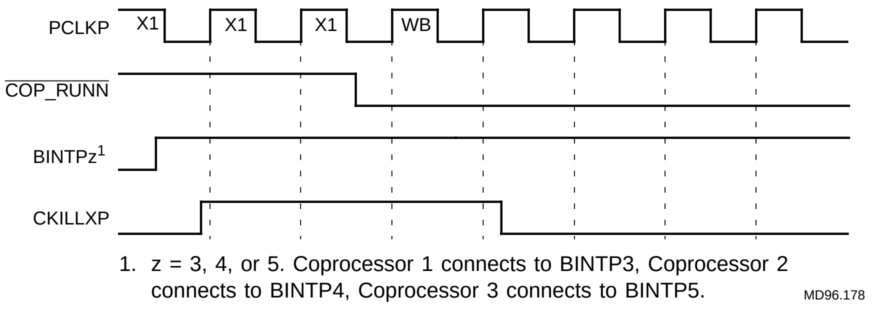
The coprocessor can request an interrupt by asserting one of the BINTP[5:0] Inputs to the CW400x, causing an interrupt exception. The interrupt from the coprocessor should be asserted until the CW400x writes a Clear Interrupt value to the coprocessor's Control Register.

Even though asserting one of the BINTP[5:0] Inputs for one run cycle causes the CW400x to take an exception, to be compatible with the MIPS architecture, the coprocessor must hold the interrupt HIGH until it is cleared by the CW400x.

The coprocessor should signal the interrupt to the CW400x in the Execute Stage of the instruction causing the interrupt. The CW400x asserts CKILLXP in the same run cycle, informing the coprocessor to cancel the instruction in the Execute Stage.

Figure 2.7 shows a waveform of a coprocessor sending an interrupt. Note that CKILLXP is asserted during the X Stage of two instructions. The first instruction is the one that caused the interrupt. The second is the one following the interrupt. For more details regarding the instruction cancellation mechanism, see Chapter 5, in the *MiniRISC CW400x Microprocessor Core Technical Manual*.

Figure 2.7 Coprocessor Sending Interrupt



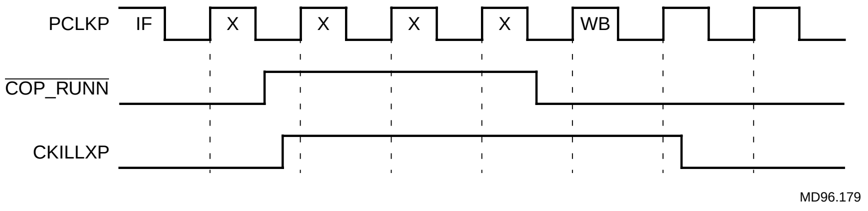
1. z = 3, 4, or 5. Coprocessor 1 connects to BINTP3, Coprocessor 2 connects to BINTP4, Coprocessor 3 connects to BINTP5.

2.8 Instruction Cancellation

When an exception occurs, the CW400x indicates to the coprocessor which instructions to cancel by asserting CKILLXP at the appropriate time. The duration of the instruction cancellation signal (CKILLXP) depends on when the exception was signalled (in the IF, X, or WB Stage).

The example in Figure 2.8 shows an exception signalled in the Execute Stage of the exceptional instruction. The CW400x asserts CKILLXP for two run cycles to cancel the Execute Stage of the exceptional instruction and the following instruction in the pipeline, which is currently in the IF Stage.

Figure 2.8 Instruction Cancellation



2.9 Global Instruction Register Module (GIR)

All coprocessors are required to include a Global Instruction Register (GIR) as the interface between the coprocessor decode and the data bus (see Figure 2.9). This module allows the coprocessor to latch the correct instruction off the data bus for a decode. A master-slave type circuit is used in the instruction bus latching. The instruction is latched in the master at the rising edge of the clock cycle when BIRDYP is HIGH. The instruction is then latched into the slave and made available to the decode at the rising edge of the clock cycle in the next run cycle.

Figure 2.10 shows a typical transaction. The instruction is on the data bus at Point A, and is latched in by the master flip-flop. It is gated by the slave only in the instruction's X Stage, Point B. The slave gate logic is complicated because load scheduling must be taken into account (data from a scheduled load can be asserted on the bus at any time).

Figure 2.9
Global Instruction
Register Logic

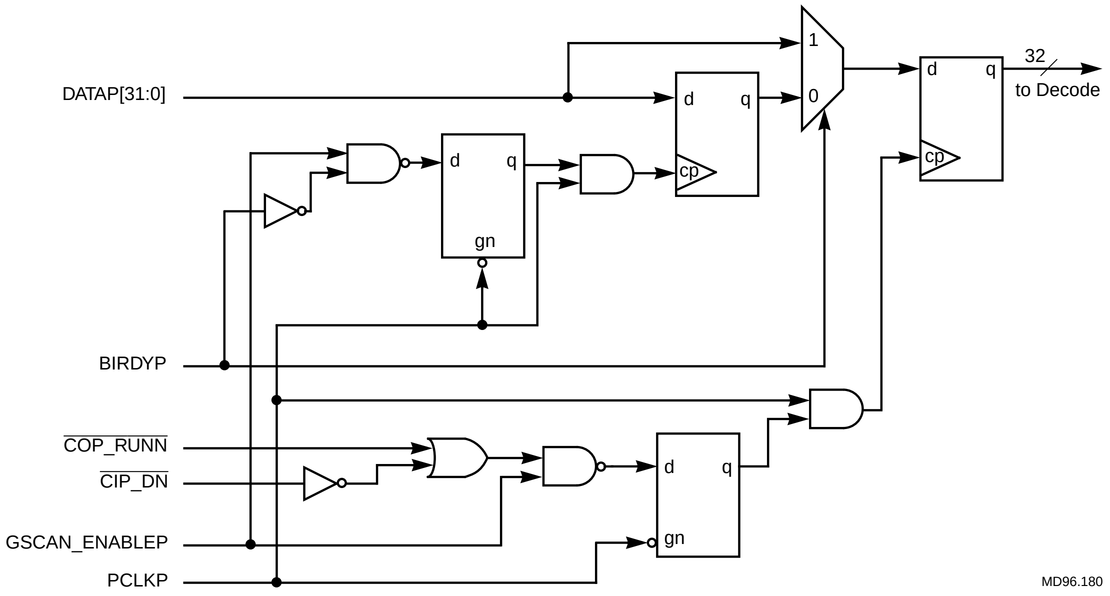
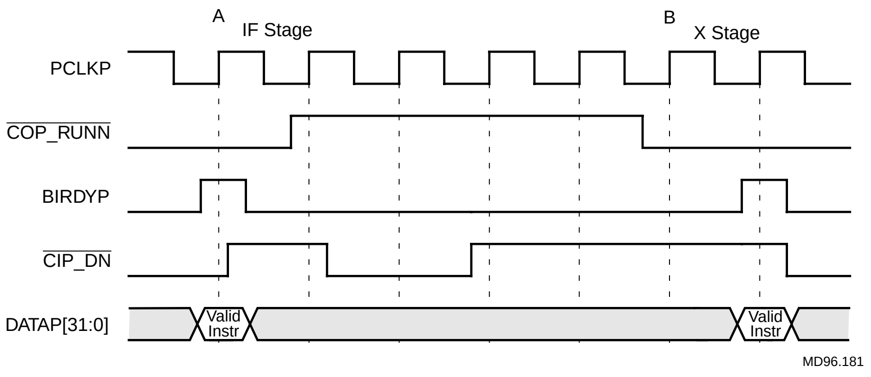


Figure 2.10
Instruction
Grabbing



Chapter 3 Multiply/Divide Unit (MDU)

This chapter describes the MiniRISC Multiply/Divide Unit (MDU) building block for the CW400x.

This chapter contains the following sections:

3.1 Overview

The Multiply/Divide Unit (MDU) is a high-performance arithmetic engine, closely coupled to the MiniRISC CW400x Microprocessor Core through the FlexLink Interface. It supports the following arithmetic functions:

- 32-bit x 32-bit signed and unsigned integer multiplication, with 2-cycle latency
- 32-bit x 32-bit signed and unsigned integer multiplication-accumulation, with a 2-cycle latency and a throughput of one multiply-accumulate per cycle
- 32-bit signed and unsigned division, with a 34-cycle latency for the quotient and a 35-cycle latency for the remainder
- MFHI, MFLO, MTHI, MTLO instructions for moving data between the CW400x and the MDU

The MDU generates hardware interlocks if the CW400x tries to read from the MDU's HI or LO Registers before it has written the results from any previous MDU operations to the HI and LO Registers.

Multiplication is a full 32-bit by 32-bit operation. The MDU takes the operands from CRSP[31:0] and CRTP[31:0], multiplies them, and writes the higher 32 bits of the result into the HI Register, and the lower 32 bits into the LO Register. It takes two cycles to complete a multiplication.

Multiplication-accumulation is similar to multiplication, except, instead of writing the result back to the HI/LO Registers, the MDU adds (MADD) or subtracts (MSUB) the multiplication result from the previous result in the HI/LO Registers. A multiplication-accumulation takes two cycles to complete. The MDU supports one instruction per cycle throughput.

For division, the MDU takes CRSP[31:0] as the dividend and CRTP[31:0] as the divisor. After performing the division, the MDU puts the quotient into the LO Register, and the remainder into the HI Register. The latency for the quotient is 34 cycles, and for the remainder it is 35 cycles.

Based on the latency and throughput of the various operations, the MDU does not generate a hardware interlock for the following pieces of code:

```
MULT(U) or MADD(U) or MSUB(U)
NOP
MFHI or MFLO
------------------------------
MULT(U) or MADD(U) or MSUB(U)
MADD(U) or MSUB(U)[1]
MADD(U) or MSUB(U)
NOP
MFHI or MFLO
------------------------------
DIV(U)
33 NOP's
MFLO
MFHI
```

No overflow exceptions occur for any multiplications, multiplication-accumulations, or divisions.

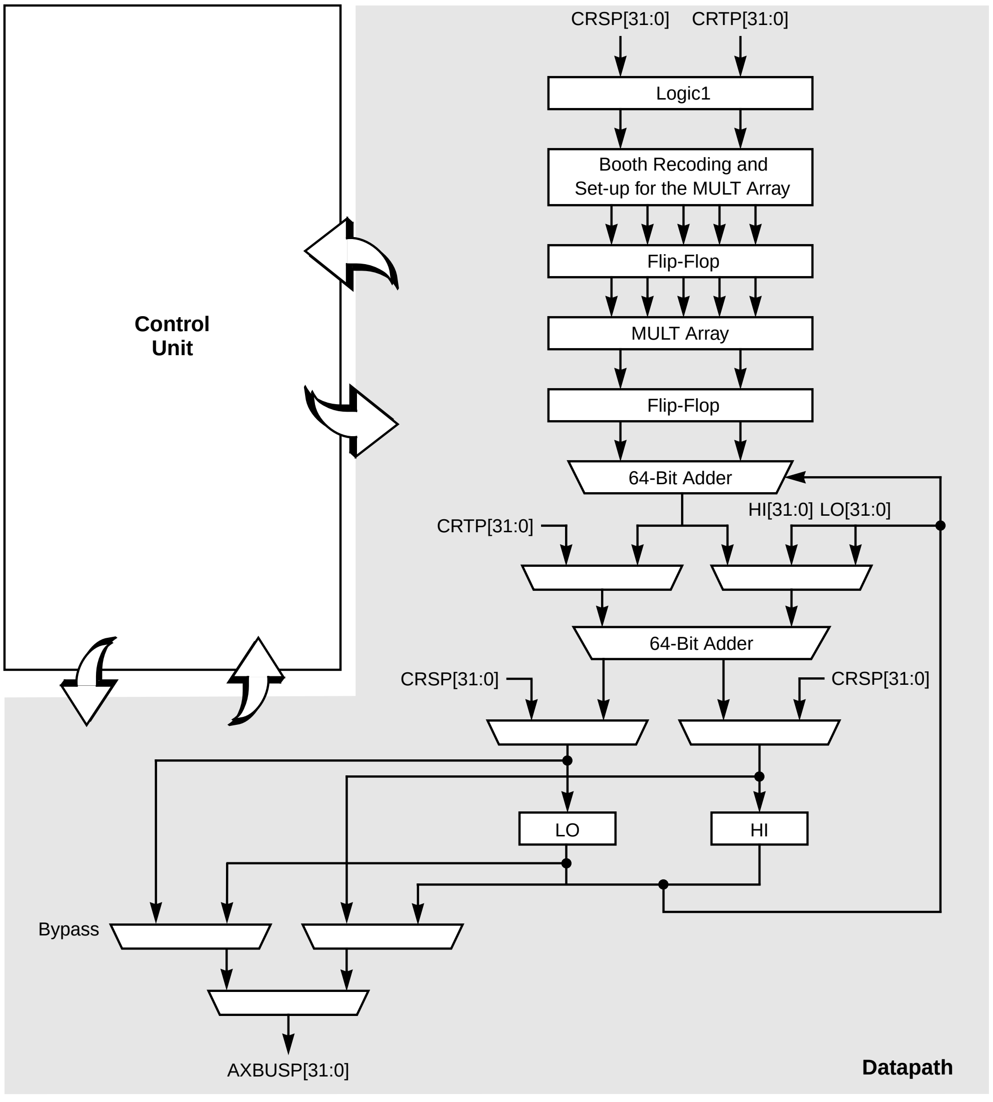
The MDU has no hardware detection for division-by-zero, which takes 34/35 cycles to complete, and causes the HI/LO Registers to contain undefined values.

1. Note that MADD or MSUB adds its result to that of the immediately preceding instruction even though the preceding MDU Instruction has not completed when this MADD or MSUB is executed.

3.2 Architecture

Logically, the MDU consists of two parts: the Control Unit, and the Datapath (which has a unified Multiply and Divide Unit). This configuration allows the multiplier to share resources with the divider. Figure 3.1 illustrates the high-level structure of the MDU.

Figure 3.1
MDU Architecture



1. Power-saving logic which blocks the operands for non-MDU instructions.

MD96.17

The Control Unit decodes the instructions from CIR_TOPP[5:0] and CIR_BOTP[5:0], and asserts ASELP if it decodes valid MDU Instructions. It also generates interlocks between the MULT/DIV/MADD/MSUB and the MFHI/LO Instructions if needed, and forwards the multiply/divide result to AXBUSP[31:0].

The multiply operation starts in the Execute Stage. To minimize power dissipation, the MDU loads operands off CRSP[31:0] and CRTP[31:0] only after decoding a valid MDU Instruction. The Multiply Unit uses 3-bit Booth recoding to encode the 32-bit multiplier. A multiply array sums partial products to produce a 64-bit sum, and a carry bit. In multiplication-accumulation, the result of the multiply operation is added to or subtracted from the previous result, contained in the HI/LO registers. The lower 32 bits of the result are loaded into the LO Register, and the higher 32 bits are loaded into the HI Register.

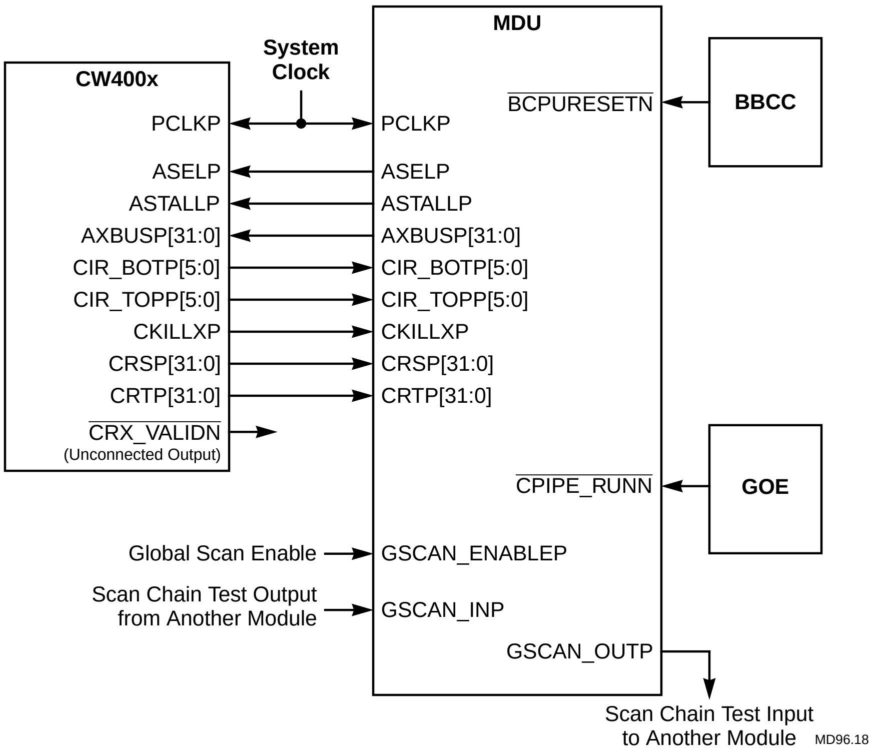
The Divide Unit implements a nonrestoring algorithm, and the operation is set off in Phase 1 of the Execute Stage. When the operation is completed, the MDU loads the quotient into the LO Register and the remainder into the HI Register.

Since it takes two cycles to complete a multiplication or a multiplication-accumulation, and 34/35 cycles to complete a division, the Control Unit stalls the CW400x (by asserting ASTALLP), if there is an attempt to read the HI/LO Registers before an outstanding operation is finished.

3.3 Connection Block Diagram

Figure 3.2 shows how to attach the MDU to the CW400x, the building blocks, and system logic.

Figure 3.2
Attaching the MDU



3.4 Signals

This section describes the signals that comprise the bit-level interface of the MDU. Tables 3.1 and 3.2 summarize the MDU signals. Detailed descriptions follow the tables.

The signals are described in alphabetical order by mnemonic. Each signal definition contains the mnemonic and the full signal name. The mnemonics for active LOW signals end with an “N” and have an overbar over their names; the mnemonics for active HIGH signals end in a “P.”

In the descriptions that follow, “assert” means to drive TRUE or active and “deassert” means to drive FALSE or inactive.

Note that the $\overline{\text{CRX\_VALIDN}}$ CW400x FlexLink Interface signal does not connect to the MDU. This signal indicates when CRSP[31:0] and CRTP[31:0] are valid during stall cycles, and is intended for the implementation of arithmetic operations that write results directly to the CW400x registers. Since all the MDU operations write results back to the HI/LO Registers, and an MFHI/LO instruction is needed to move results back to the CW400x registers, the MDU does not use $\overline{\text{CRX\_VALIDN}}$.

Table 3.1
MDU Input Signals Summary

Input	Source	Definition
$\overline{\text{BCPURESETN}}$	BBCC	Global Reset
CIR_BOTP[5:0]	CW400x	Instruction Register Bottom Six Bits
CIR_TOPP[5:0]	CW400x	Instruction Register Top Six Bits
CKILLXP	CW400x	Kill Instruction in Execute Stage
$\overline{\text{CPIPE\_RUNN}}$	GOE	CW400x Pipe Run
CRSP[31:0]	CW400x	CW400x Source Register (`rs`) Bus
CRTP[31:0]	CW400x	CW400x Source Register (`rt`) Bus
GSCAN_ENABLEP	System Logic	Scan Test Mode Enable
GSCAN_INP	System Logic	Scan Test Input
PCLKP	System Logic	System Clock

Table 3.2
MDU Output Signals Summary

Output	Destination	Definition
ASELP	CW400x	MDU Select
ASTALLP	CW400x	MDU Stall Request
AXBUSP[31:0]	CW400x	MDU Result Bus
GSCAN_OUTP	System Logic	Scan Test Output

ASELP **MDU Select** **Output**

The MDU asserts this signal to inform the CW400x that the current instruction is an MDU Instruction. This prevents the CW400x from signalling a Reserved Instruction Exception.

ASTALLP **MDU Stall Request** **Output**

The MDU asserts this signal to request a stall of the pipeline. The MDU asserts ASTALLP if it discovers any data dependencies that prevent it from executing the upcoming MDU Instruction. The CW400x might override this stall if CKILLXP is also asserted, because the CW400x kills the upcoming MDU Instruction.

AXBUSP[31:0]

MDU Result Bus **Output**

The MDU puts the result from the HI or LO Registers onto this bus.

$\overline{\textbf{BCPURESETN}}$

Global Reset **Input**

Asserting this signal resets the MDU, and kills any outstanding MDU Instructions. The contents inside the HI and LO Registers become unknown.

CIR_BOTP[5:0]

Bottom Six Bits of Instruction Register **Input**

These signals from the CW400x contain the bottom six bits of the Instruction Register. These signals allow the MDU to decode its own instructions.

CIR_TOPP[5:0]

Top Six Bits of Instruction Register **Input**

These signals from the CW400x contain the top six bits of the Instruction Register. These signals allow the MDU to decode its own instructions.

CKILLXP **Instruction Killed in Execute Stage** **Input**

The CW400x asserts this signal to request that the MDU kill the MDU Instruction that is in the Execute Stage.

$\overline{\textbf{CPIPE\_RUNN}}$ **CW400x Pipeline Run Indicator** **Input**

The GOE Module asserts this signal to inform the MDU that the core is in a Pipeline Run Cycle. The GOE deasserts this signal to inform the MDU that the core is in a Pipeline Stall Cycle. For more information on the GOE, see the *MiniRISC CW400x Microprocessor Core Technical Manual*.

CRSP[31:0] **CW400x Source Register (rs) Bus** **Input**

These signals contain the `rs` Operand of the current instruction from the CW400x.

CRTP[31:0] **CW400x Source Register (`rt`) Bus** **Input**
These signals contain the `rt` Operand of the current instruction from the CW400x.

GSCAN_ENABLEP
Scan Test Mode Enable **Input**
Asserting this signal enables scan testing.

GSCAN_INP **Scan Test Input** **Input**
Another module drives this signal with the scan test input.

GSCAN_OUTP
Scan Test Output **Output**
The MDU drives this signal with the scan test output.

PCLKP **System Clock** **Input**
This signal is the global clock input.

3.5 Instructions

All MDU Instructions are in one of the three formats shown in Figures 3.3 through 3.5.

Table 3.3 lists all of the MDU Instructions and their corresponding opcode bits. Table 3.4 summarizes and describes the Multiply/Divide Instructions.

Figure 3.3
MFHI, MFLO

31 — 16	15 — 11	10 — 6	5 — 0
0000000000000000_2	rd Address	00000_2	MDU Opcode

Figure 3.4
MTHI, MTLO

31 — 26	25 — 21	20 — 6	5 — 0
000000_2	rs Address	000000000000000_2	MDU Opcode

Figure 3.5
MULT(U), DIV(U),
MADD(U), MSUB(U)

31 — 26	25 — 21	20 — 16	15 — 6	5 — 0
000000_2	rs Address	rt Address	0000000000_2	MDU Opcode

Table 3.3
MDU Instructions

Instruction	Description	Opcode
DIV	Divide Signed Numbers	011010
DIVU	Divide Unsigned Numbers	011011
MADD[1]	Multiply, and Add the Result to HI/LO Registers	011100
MADDU[1]	Unsigned Multiply, and add result to HI/LO Registers	011101
MFLO	Move from LO Register	010010
MFHI	Move from HI Register	010000
MSUB[1]	Multiply, then Subtract the Result from HI/LO Registers	011110
MSUBU[1]	Unsigned Multiply, then Subtract the Result from HI/LO Registers	011111
MTHI	Move to HI Register	010001
MTLO	Move to LO Register	010011
MULT	Multiply Signed Numbers	011000
MULTU	Multiply Unsigned Numbers	011001

1. MR400x-specific instruction.

Table 3.4
Multiply/Divide
Instruction Summary[1]

Instruction	Format and Description
Multiply and Add	`MADD rs, rt` Multiplies the contents of Registers `rs` and `rt` as two's complement values. Adds the 64-bit result to Special Registers HI and LO[2].
Multiply and Add Unsigned	`MADDU rs, rt` Multiplies the contents of Registers `rs` and `rt` as unsigned values. Adds the 64-bit result to Special Registers HI and LO[2].
Multiply and Subtract	`MSUB rs, rt` Multiplies the contents of Registers `rs` and `rt` as two's complement values. Subtracts the 64-bit result from Special Registers HI and LO[2].
Multiply and Subtract Unsigned	`MSUBU rs, rt` Multiplies the contents of Registers `rs` and `rt` as unsigned values. Subtracts the 64-bit result from Special Registers HI and LO[2].
Multiply	`MULT rs, rt` Multiplies the contents of Registers `rs` and `rt` as two's complement values. Stores the 64-bit result into Special Registers HI and LO[2].
Multiply Unsigned	`MULTU rs, rt` Multiplies the contents of Registers `rs` and `rt` as unsigned values. Stores the 64-bit results into Special Registers HI and LO[2].
Divide	`DIV rs, rt` Divides the content of Register `rs` by the content of Register `rt` as two's complement values. Stores the 32-bit quotient into Special Register LO, and the 32-bit remainder into Special Register HI.
Divide Unsigned	`DIVU rs, rt` Divides the content of Register `rs` by the content of Register `rt` as unsigned values. Stores the 32-bit quotient into Special Register LO, and the 32-bit remainder into Special Register HI.
Move From HI Register	`MFHI rd` Moves the content of Special Register HI into Register `rd`.
Move From LO Register	`MFLO rd` Moves the content of Special Register LO into Register `rd`.
Move To HI Register	`MTHI rd` Moves the content of Register `rd` into Special Register HI.
Move To LO Register	`MTLO rd` Moves the content of Register `rd` into Special Register LO.

1. These instructions require the addition of a Multiply/Divide Unit.
2. The HI and LO Registers are used as one 64-bit register.

Table 3.5 shows the execution time of multiply and divide instructions when using the MDU.

Table 3.5
Execution Time of Multiply/Divide Instructions Using MDU

Multiply Operands	Number of Cycles
MULT	2
MADD/MSUB[1]	2
DIV	34 (quotient) 35 (remainder)

1. With the results accumulating in the HI and LO Registers. The throughput is one instruction per cycle.

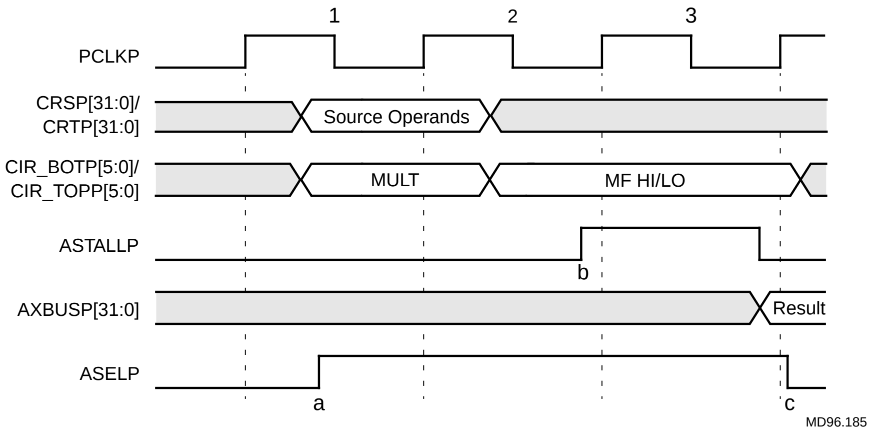
3.6 Operation

This section explains MDU operation and contains waveforms for each operation.

3.6.1 MULT Followed by MFHI/LO

Figure 3.6 shows a MULT followed by a MFHI/LO.

Figure 3.6
MULT Followed by MFHI/LO



a. The MDU asserts ASELP as soon as it decodes a valid instruction to prevent the CW400x from generating a Reserved Instruction Exception.

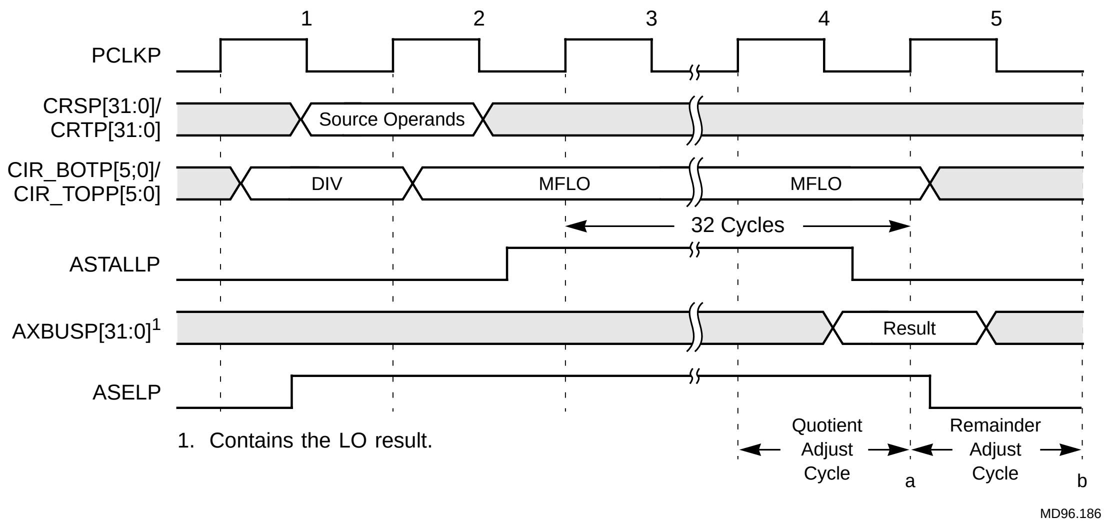
b. The MDU asserts ASTALLP for interlock since the CW400x tries to read the result (MFHI/LO) before it is ready.

c. Both the HI and LO results are ready. The MDU writes them back into the HI/LO Registers at this rising clock edge. At the same time, the MDU forwards them to AXBUSP[31:0] so that the CW400x can latch the result back to its register file.

3.6.2 DIV Followed by a MFLO

Figure 3.7 shows a DIV followed by a MFLO.

Figure 3.7
DIV Followed by MFLO



1. Contains the LO result.

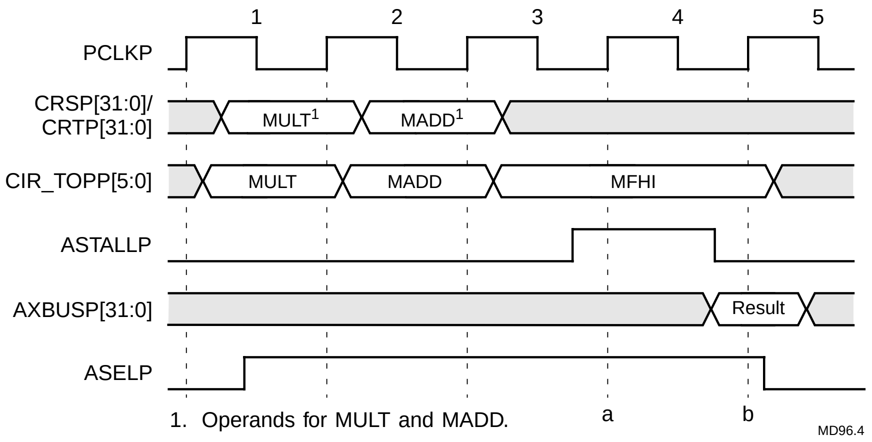
a. The quotient result is ready, and the MDU writes it back into the LO Register at this rising clock edge. At the same time, the MDU also forwards the quotient to AXBUSP[31:0] so that the CW400x can latch the result back to its register file.

b. The remainder result is ready, and the MDU writes it back to the HI Register at this rising clock edge.

3.6.3 MULT Followed by MADD/MSUB

Figure 3.8 shows a MULT followed by a MADD/MSUB.

Figure 3.8
MULT Followed by MADD/MSUB



1. Operands for MULT and MADD.

a. The MDU writes the result of MULT back into the HI/LO Registers.

b. The MDU writes the result of MADD back into the HI/LO Registers, and the HI Result is also forwarded to AXBUSP[31:0] at the same rising clock edge.

Note that sequences like the following do not require MDU stalls:

```
MULT
MADD
MSUB
MADD
MADD
```

The MDU accumulates each MADD and MSUB multiplication result (added to, or subtracted from, the result of the preceding instruction) instead of overwriting it, even if the preceding instruction is not finished when the MADD or MSUB is executed.

3.6.4 DIV Followed by MADD/MSUB

If there are any unkilled MADD or MSUB Instructions issued during an outstanding DIV, the MDU does not generate any interlocks. The MDU preempts the DIV in the middle of its operation, executes the MADD or MSUB, and then adds or subtracts the division result to whatever values the HI/LO Registers contain when the DIV is preempted. Therefore, the HI/LO Results are unknown.

3.6.5 Destructive MDU Instructions

Any MULT, MULTU, DIV, DIVU, MTHI, or MTLO kill outstanding MULT(U), DIV(U), MADD(U), or MSUB(U) Instructions. Therefore, for a sequence of instructions like the one below, MFHI gets the result of the MTHI, so the content of the LO Register is undefined.

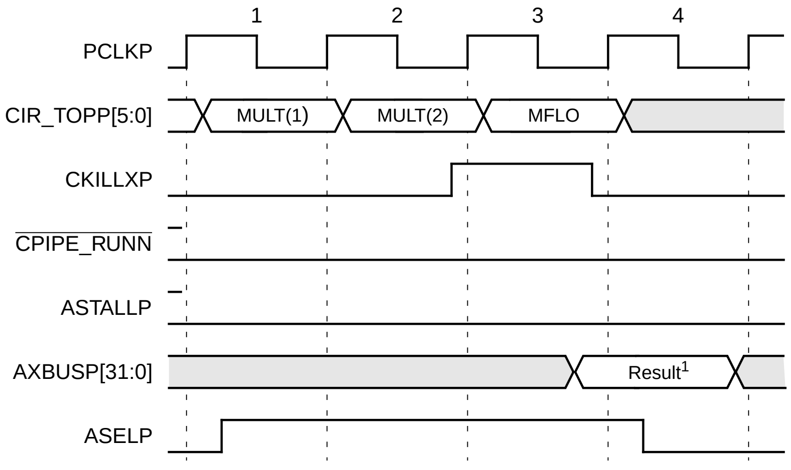
```
MULT
MULT
MTHI
MFHI
```

The same happens to the HI Register during an MTLO.

3.6.6 Effect of CKILLXP on MDU Operations

Figure 3.9 shows the effect of CKILLXP on a MULT. The same waveform applies to MADD, MSUB, and DIV. CKILLXP kills the MULT(2) Instruction which is in its Execute Stage. MULT(1) finishes gracefully, and the MFLO gets the LO Result of MULT(1). The MDU does not generate any interlock stalls.

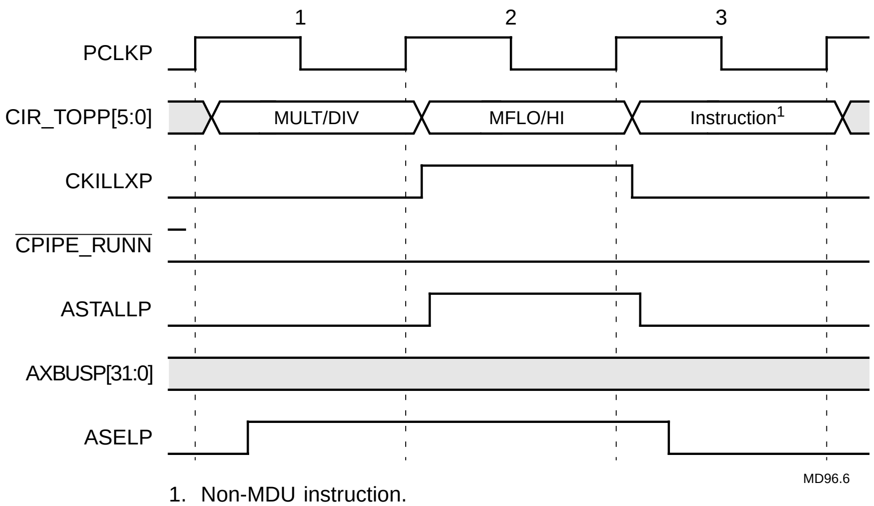
Figure 3.9 Effect of CKILLXP on a MULT Operation



1. LO result of MULT(1).

Figure 3.10 shows the effect of CKILLXP on a MFHI or MFLO.

Figure 3.10 Effect of CKILLXP on a MFLO/HI Operation



1. Non-MDU instruction.

Cycle 1: A MULT or DIV is issued.

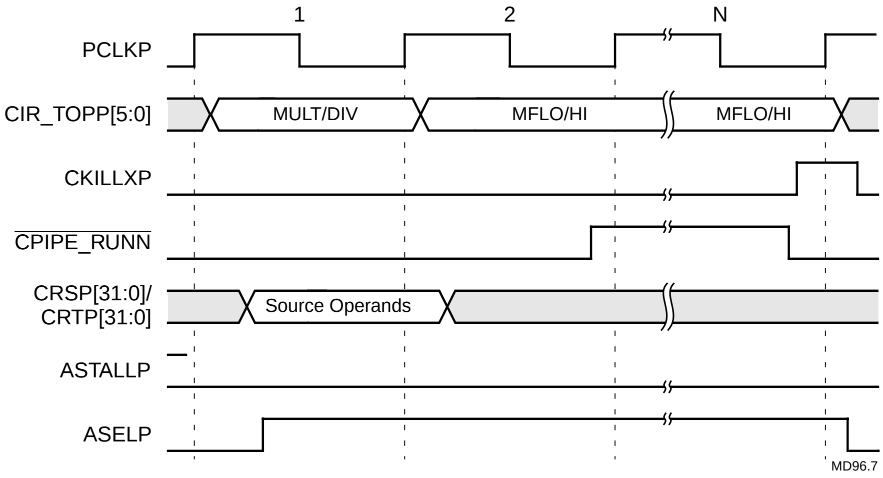
Cycle 2: A MFLO or MFHI is issued. The assertion of CKILLXP kills the MFLO/HI; however, because of the critical timing, ASTALLP remains asserted. The CW400x detects the assertion of CKILLXP, and ignores ASTALLP. $\overline{\text{CPIPE\_RUNN}}$ does not go HIGH even though ASTALLP is asserted. This example assumes that there are no other stalls (for example, stalls initiated by the CW400x itself or the coprocessors); otherwise, $\overline{\text{CPIPE\_RUNN}}$ might still go HIGH due to other stalls.

Cycle 3: The CW400x moves on to the next instruction, which is not an MDU Instruction; thus, ASTALLP and ASELP are both deasserted.

3.6.7 Effect of $\overline{\text{CPIPE\_RUNN}}$ on MDU Operations

Figure 3.11 shows the effect of $\overline{\text{CPIPE\_RUNN}}$ on an MDU Operation. CKILLXP kills the MULT(2) Instruction, which is in its Execute Stage. MULT(1) finishes correctly, and the MFLO gets the LO Result of MULT(1). The MDU does not generate interlock stalls.

Figure 3.11
Effect of $\overline{\text{CPIPE\_RUNN}}$ on a MFLO/HI Operation



Cycle 1: A MULT or DIV is issued.

Cycle 2: Another MULT or DIV is issued. $\overline{\text{CPIPE\_RUNN}}$ goes HIGH by the end of Cycle 2. This extends the Execute Stage of the second MULT or DIV, and prevents the instruction from being executed.

Cycle N: $\overline{\text{CPIPE\_RUNN}}$ goes LOW, and the MULT/DIV which has been held in its Execute Stage, proceeds again. However, CKILLXP is asserted by the end of the cycle, which kills this second MULT or DIV. The HI/LO Registers still hold the results of the first MULT or DIV.

In conclusion, both CKILLXP and $\overline{\text{CPIPE\_RUNN}}$ have effects only on the MDU Instruction in its Execute Stage. $\overline{\text{CPIPE\_RUNN}}$ can extend an instruction's Execute Stage, and CKILLXP can kill an instruction in its Execute Stage if it is asserted in the last Execute Cycle of that instruction. After an instruction has passed beyond its Execute Stage, neither input signals have any effect on the operation of that instruction. For example, if CKILLXP or $\overline{\text{CPIPE\_RUNN}}$ goes HIGH in the twentieth cycle of a DIV operation, the DIV keeps going without stalling or being killed.

3.6.8 Effect of $\overline{\text{BCPURESETN}}$

The MDU acknowledges the assertion of $\overline{\text{BCPURESETN}}$ within one clock cycle. However, since the CW400x requires a minimum of two clock cycles to reset correctly, LSI Logic recommends asserting $\overline{\text{BCPURESETN}}$ for at least two clock cycles.

Asserting $\overline{\text{BCPURESETN}}$:

- ♦ Causes outstanding MDU Instructions to stall (not finish)
- ♦ Resets the state machine for the divide operation
- ♦ Deasserts ASTALLP if it has been asserted
- ♦ Causes the contents in HI/LO Registers to be undefined

Chapter 4
Memory Management Unit (MMU)

This chapter describes the Memory Management Unit (MMU) building block for the CW400x. For information on the MMU Stub, the logic required if the system has no MMU, see Chapter 6 of the *MiniRISC CW400x Microprocessor Core Technical Manual*.

This chapter contains the following sections:

4.1 Overview

The MiniRISC Memory Management Unit (MMU) translates virtual addresses from the MiniRISC CW400x Core into physical addresses. The MMU is MIPS-compatible and similar in design and function to the MIPS R3000 MMU.

The eight-entry fully-associative Translation Lookaside Buffer (TLB) is a cache of Page Table Entries (PTE) for the operating system. The Page Table contains the information for mapping processes from virtual to physical addresses. Caching PTEs minimizes the size of the Page Table structure implemented in hardware. The TLB is implemented as an 8-entry, 20-bit data Content-Addressable Memory (CAM) coupled with a 23-bit RAM array custom-designed for the MMU.

4.2 Function and Operation

The MMU takes virtual addresses from the MiniRISC CW400x Core and translates them to physical addresses. Figure 4.1 illustrates how the MMU performs this translation by replacing the 20 most significant bits (Bits [31:12]) of the Virtual Address (called the Virtual Page Number, or VPN) with 20 bits from the Page Table (called the Page Table Entry, or PTE). The MMU reads the PTE from the TLB.

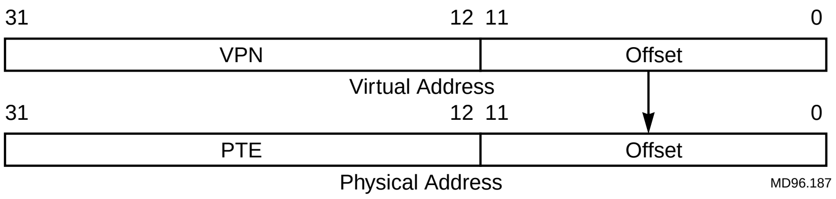


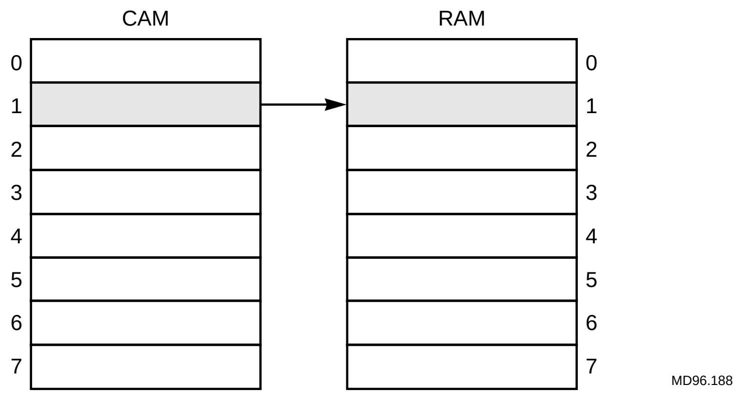
Figure 4.1
Virtual to Physical Address Mapping

Before the PTE can replace the VPN, the MMU uses the TLB Virtual Page Number CAM (TLB, VPN, CAM) to determine which of the PTEs stored in the TLB to use.

The TLB is a 8 x 20 array of CAM cells that can be read, and written, and used in Match Cycles. In a Match Cycle, the CW400x passes a VPN from the Address Bus, ADDRP[19:0], to the CAM array, which activates a match line for the entry in the TLB that matches. Once this match line is driven, the CAM notes a match. The match line drives the word line in the adjoining 8 x 23 PTE RAM, which in turn, drives the 23-bit PTE RAM word out of the RAM array and onto the MMU Address Bus, MADDROUTP[19:0].

Figure 4.2 shows an example where the address in CAM Entry 1 matches the virtual address. The MMU drives RAM Entry 1 as the physical address.

Figure 4.2
CAM Entry
Matching Virtual
Address



The 20-bit PTE/VPN size allows for a 12-bit offset field, which allows for up to 4 Kbytes of cache with a physical address. The 23-bit word consists of the PTE, the Valid Bit, the Dirty Bit and the No-cache Bit. The Valid Bit set to one indicates that the TLB entry is valid. The Dirty Bit set to one indicates that the page is writable. The No-cache Bit set to one indicates that the entry is not cacheable.

In addition to the soft-mapping made possible by the TLB, the MMU performs hard-mapping. To perform hard mapping, the MMU implements the R3000 *kseg0/kseg1* Memory Map. The CW400x generates addresses that are within *kseg0* or *kseg1*. The MMU maps these addresses immediately to the lowest 512 Mbytes of physical space, not actually using the TLB for this map function. If the CW400x address is in *kseg1*, the MMU signals that the access is not to be cached.

For further details, see the MMU Stub section in Chapter 6 of the *MiniRISC CW400x Microprocessor Core Technical Manual*.

Under the following conditions, the MMU does not softmap the VPN to a PTE:

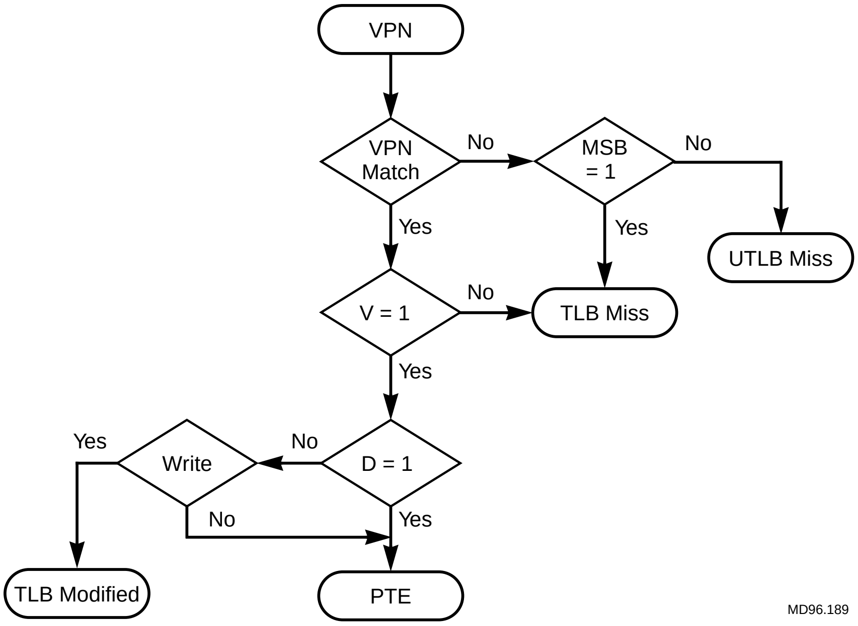
- ♦ The MMUENP input is inactive, which disables the MMU.
- ♦ The two MSBs of the address are 10_2, which indicates operation in *kseg1* or *kseg0* (defined in the MIPS architecture to be unmapped segments in kernel space). These segments are physically mapped to the address where the two MSBs are 00_2 by the MMU Stub.
- ♦ The current cycle is not a Bus Run Cycle.

There are several conditions in which, although an exception has been generated, the MMU performs a translation on the available data. The exception handler software should prevent any erroneous memory accesses based on this falsely translated address. These conditions are:

- There is no VPN match, which causes an exception. A reference to kernel space causes a TLB Miss Exception. A reference to user space causes a UTLB Miss Exception.
- The VPN match points to a PTE RAM entry with the Valid Bit LOW. This condition indicates that the page is invalid, and causes a TLB Miss Exception to the CW400x.
- The Dirty Bit is not set for the current page. This condition occurs only if the entry was valid; it indicates that an attempt was made to write to memory that is protected (clean), which causes a TLB Modified Exception to the CW400x. This condition occurs only for a store cycle.

Figure 4.3 illustrates the conditions that cause TLB Exceptions.

Figure 4.3
TLB Miss
Exception
Conditions



If there is a TLB hit, and the entry is valid and dirty (the Valid and Dirty Bits are set), the VPN in the MMU replaces the address with the PTE.

If there is a TLB Modified Exception or a TLB Miss Exception because the entry was invalid, the MMU encodes the eight match lines and writes them to the CW400x Index Register. If the TLB Miss Exception or UTLB Miss Exception was simply due to no VPN match, the MMU writes the value of the Random Register to the Index Register, and sets the P Bit. Setting the P Bit suggests that the entry needs to be replaced; however, the operating system does not need to use the P Bit.

4.3 MMU Modules

The following modules are part of the MMU:

- TLB
- MMU Stub
- GIR

4.3.1 TLB

The TLB is made of a 8 x 20 CAM, tightly coupled with a 8 x 23 RAM. The CAM match lines drive the word lines in the RAM. The entire TLB is a fully-customized, special high-density structure capable of writes and reads to the CAM and RAM portions. It also performs Match Cycles. The match lines are driven out so that the values can be encoded.

The CW400x must load the Index Register (IR) before writing to or reading from the Hi (CAM) or Lo (RAM) words in the TLB. The IR drives the address for a write or read to the TLB. Writes to the TLB occur at the end of the first X2 Cycle, and because of this, the MMU generates a stall to hold the CW400x in X2 for one more cycle, so it can complete the write. Thus, every TLB write requires two X2 Cycles. TLB reads require only one X2 Cycle.

The TLB also detects the TLB Shutdown condition (multiple matching entries), and shuts down the read, to prevent permanent hardware damage.

4.3.2 MMU Stub

The stand-alone MMU Stub Module is also a part of the MMU. This module registers the address for every Bus Run Cycle and also translates the CW400x address, if in *kseg0* or *kseg1*, to the least significant 512 Mbytes of physical memory. In addition, it indicates that the address received by the MMU Stub was in *kseg0* or *kseg1*, and has been hard-mapped, rather than mapped through the TLB (see Section 6.2, “MMU Stub,” of the *MiniRISC CW400x Microprocessor Core Technical Manual*).

4.3.3 GIR

The Global Instruction Register (GIR) is the specified interface to the CW400x Data Bus, DATAP[31:0]. It guarantees that instructions are registered, since they are to be fetched at the end of the IF Stage of the instruction. The GIR guarantees that a valid instruction is loaded during the run cycle in which the IF Stage occurred. Thus, designers do not need to design logic to:

- Detect stalls from other instructions
- Distinguish between data and instructions on the unified bus (DATAP[31:0])

When the instruction is passed out of the GIR, it has reached the X Stage.

For more information on the GIR, see Section 2.9, “Global Instruction Register Module (GIR).”

4.4 Signals

This section describes the signals that comprise the bit-level interface of the MMU. Tables 4.1 through 4.3 summarize the MMU signals. Detailed descriptions follow the tables.

The signals are described in alphabetical order by mnemonic. Each signal definition contains the mnemonic and the full signal name. The mnemonics for active LOW signals end with an “N” and have an overbar over their names.

In the descriptions that follow, “assert” means to drive TRUE or active and “deassert” means to drive FALSE or inactive.

Table 4.1
MMU Input
Signals Summary

Input	Source	Definition
ADDRP[19:0]	CW400x	CW400x Virtual Address Bus
$\overline{\text{BBUS\_STEALN}}$	BIU	BBus Bus Steal
BIRDYP	BIU	BBus Instruction Data Ready
$\overline{\text{BSYSRESETN}}$	BIU	Reset
CADDR_ERRORP	CW400x	CW400x Memory Address Error
$\overline{\text{CIP\_DN}}$	CW400x	CW400x Instruction/Data Indication

(Sheet 1 of 2)

Table 4.1 (Cont.)
MMU Input
Signals Summary

Input	Source	Definition
CKILLXP	CW400x	CW400x Instruction Killed in Execute Stage
CMEM_FETCHP	CW400x	CW400x Memory Fetch Request
CSTOREP	CW400x	CW400x Store to Memory Request
MMUENP	System Logic	MMU Enable
$\overline{\text{MRUN\_INN}}$	GOE	System Running
PCLKP	System Logic	System Clock
(Sheet 2 of 2)		

Table 4.2
MMU Output
Signals Summary

Output	Destination	Definition
MADDROUTP[19:0]	BIU	CW400x Physical Address Bus
MNOCACHEP	BIU	MMU Non-Cacheable Page
MRUN_OUTP	GOE	MMU Running
MTLBMISSEXCP	CW400x	MMU TLB Miss Exception
MTLBMODEXCP	CW400x	MMU TLB Modified Exception
MUTLBMISSEXCP	CW400x	MMU User TLB Miss Exception

Table 4.3
MMU Bidirectional
Signals Summary

Bidirectional	Connect	Description
DATAP[31:0]	CW400X	CW400x Data Bus

ADDRP[19:0] **CW400x Virtual Address Bus** **Input**
These signals are the 20 MSBs of the CW400x Address Bus (the Virtual Page Number).

$\overline{\text{BBUS\_STEALN}}$
BBus Bus Steal **Input**
The BIU asserts this signal to inform the MMU that the BIU has stolen the BBus, and the MMU should not drive the BBus or write the value on the BBus into a register (because the CW400x is not controlling the bus).

BIRDYP **BBus Instruction Data Ready** **Input**
The BIU asserts this signal to inform the MMU that DATAP[31:0] contains valid data for an instruction fetch. This is a control signal to the GIR. For more information on the GIR, see Section 2.9, "Global Instruction Register Module (GIR)."

BSYSRESETN

Reset **Input**

The BIU asserts this signal to inform the MMU that the system is in reset state, and to reset the MMU. This signal must be valid for at least one clock cycle, although LSI Logic recommends that for future design flexibility, this signal be valid for at least three clock cycles.

CADDR_ERRORP

CW400x Memory Address Error **Input**

The CW400x asserts this signal to inform the MMU that a memory transaction address error has occurred. The CW400x can assert this signal in either the IF or the X2 Stage. This information is necessary to store the Bad Virtual Address in the BadVA Register.

$\overline{\text{CIP\_DN}}$ **CW400x Instruction/Data Indication** **Input**

The CW400x drives this signal LOW to inform the MMU that the decoded MMU MTC0/MFC0 Instruction has entered the X2 Stage, and that the MMU (if $\overline{\text{BBUS\_STEALN}}$ is not asserted) can either write a register or drive data on DATAP[31:0].

This signal qualifies the type of memory fetch when a memory fetch is indicated by CMEM_FETCHP. The CW400x drives this signal HIGH to indicate that it is performing an instruction fetch. The CW400x drives this signal LOW to indicate that it is performing a data fetch.

CKILLXP **CW400x Instruction Killed in Execute Stage** **Input**

The CW400x asserts this signal to inform the MMU that the instruction currently executing in the X Stage (X1 or X2) has been cancelled due to an exception.

Asserting this signal causes the MMU to ignore:

♦ any MTC0 Instruction currently in the X1 or X2 Stage before the write occurs to the register,

♦ or any MFC0 Instruction in the X1 Stage, before the read data is driven on the bus.

If this signal is not asserted by the end of the X1 Stage, the MMU drives the data during the X2 Stage of an MFC0 Instruction.

CMEM_FETCHP	**CW400x Memory Fetch Request**	**Input**	The CW400x asserts this signal to inform the MMU that the CW400x is now ready to load data from memory. This indicates that the MMU is now able to drive MADDROUTP[19:0] in a translation. The CW400x asserts this signal in the X1 Stage. It is valid at the rising edge of the clock in the X2 Stage.
CSTOREP	**CW400x Store to Memory Request**	**Input**	The CW400x asserts this signal to inform the MMU that the CW400x is now ready to store data to memory. The MMU can now drive MADDROUTP[19:0] in a translation. The CW400x asserts this signal in the X1 Stage. It is valid at the rising edge of the clock in the X2 Stage.
DATAP[31:0]	**CW400x Data Bus**	**Bidirectional**	These signals transfer data to and from the CW400x.
MADDROUTP[19:0]	**CW400x Physical Address Bus**	**Output**	The MMU drives the 20 MSBs of the translated CW400x Address Bus (the Page Table Entry) onto these signals. When the MMU is not translating, these signals are a registered version of ADDRP[19:0].
MMUENP	**MMU Enable**	**Input**	Asserting this signal enables the MMU. It should be a bit in a register elsewhere in the design, or it should be hardwired HIGH.
MNOCACHEP	**MMU Non-Cacheable Page**	**Output**	The MMU asserts this signal to indicate that the MMU is preventing data from being stored into, or read from, the cache.
$\overline{\textbf{MRUN\_INN}}$	**System Running**	**Input**	The GOE asserts this signal LOW to inform the MMU that the system is running. The GOE deasserts this signal HIGH to inform the MMU that the system is stalled. The cause of this stall is unknown to the MMU.
MRUN_OUTP	**MMU Running**	**Output**	The MMU asserts this signal HIGH to indicate that it is running. The MMU deasserts this signal LOW in the first X2 Cycle of a MTC0 EntryHi/EntryLo Instruction and

deasserts it one cycle later, indicating that the MMU requires a stall cycle.

MTLBMISSEXCP

MMU TLB Miss Exception **Output**
The MMU asserts this signal to indicate that it has detected an MMU TLB Miss Exception condition.

MTLBMODEXCP

MMU TLB Modified Exception **Output**
The MMU asserts this signal to indicate that it has detected a MMU TLB Modified Exception condition.

MUTLBMISSEXCP

MMU User TLB Miss Exception **Output**
The MMU asserts this signal to indicate that it has detected a MMU User TLB Miss Exception condition.

PCLKP **System Clock** **Input**
This signal is the global clock input. It is used to clock elements in the CW400x Interface.

4.5 TLB Registers

This section describes:

- TLB Exception Processing Registers
- Other TLB Registers

4.5.1 TLB Exception Processing Registers

Table 4.4 lists the TLB Exception Processing Registers and their addresses.

Table 4.4 TLB Exception Processing Register Addresses

Address	Register Name
0	Index
1	Random
4	Context (Read Only)
8	Bad Virtual Address (Read Only)

4.5.1.1 Index Register (R0)

The Index Register contains the matching TLB entry for any TLB-generated exception. If the exception was caused by no matching entries in the TLB, the Index Register contains a random number from the Random Register. This number is the address of a possible TLB entry to be replaced.

The MiniRISC MMU Probe Bit differs in function from the R3000's. The MMU sets the Probe Bit when it writes the value from the Random Register to the Index Register, not when executing the TLBP Instruction, as in the R3000. The MMU sets the Index Register Probe Bit only on a TLB Miss. A TLB hit-based exception does not cause the MMU to set the Probe Bit.

When attempting to write to the Index Register, the CW400x stalls and drives the data bus in the X2 Stage of the pipeline. The CW400x writes the register at the end of the X2 Stage. Index Register reads occur during the X2 Stage when the CW400x has stalled and allowed the MMU to drive the data bus.

The Index Register contains:

- ♦ the address of the TLB entry to be accessed in a MTC0 EntryHi, MTC0 EntryLo, MFC0 EntryHi, or MFC0 EntryLo,
- ♦ the address of the matching TLB entry on any TLB-generated exception,
- ♦ or a random number from the Random Register that is the address of a TLB entry that can be replaced when there are no TLB matches.

Figure 4.4 shows the format of the Index Register. Upon reset, the content of this register is undefined.

Figure 4.4
Index Register

31	30 … 11	10 … 8	7 … 0
P	Reserved	Index	Reserved

P **Probe** **31**
The MMU sets this bit to indicate that no TLB entry matched the Virtual Page Number and that the Index Field contains a random entry address to be used for replacement.

Reserved	**Reserved Bits** These bits are not writable, and read as zero.	**[30:11], [7:0]**
Index	**Index Bits** The MMU uses these bits to:	**[10:8]**

- ♦ Indirectly index the address of the TLB entry of an upcoming TLB access by a CW400x MTC0/MFC0 Instruction
- ♦ Indicate the value of a matching TLB entry when that entry caused a TLB Exception
- ♦ Hold the random number from the Random Register after a TLB Miss

4.5.1.2 Random Register (R1)

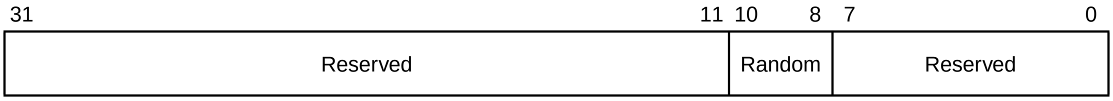
The MMU Random Register is smaller than the R3000 Random Register because the MMU TLB is smaller (8 entries, instead of 64). Therefore, the Random Field in the register only requires three bits, instead of six. Entries can be made safe, or reserved, by writing a value to the Random Field. For example, writing a five to the Random Register, makes Entries 0 through 4 safe.

The Random Register provides the address of a Random Entry in the TLB that can be replaced when a TLB miss occurs. The Random Register Module consists of a three-bit counter and a three-bit register that is the output of the counter. The CW400x can load the register using the MTC0 Random Instruction, and read the counter using the MFC0 Random Instruction. The MMU loads the counter value into the register and counts continually every run cycle, as long as the MMU is enabled. When the counter reaches seven, the MMU reloads it with the value in the register. Loading a value into the random register allows the CW400x to reserve certain entries in the TLB.

When the CW400x attempts to write to the Random Register, the CW400x stalls and drives the data bus in the X2 Stage of the pipeline. The CW400x writes the register at the end of the X2 Stage. Random Register reads occur during the X2 Stage when the CW400x has stalled and allowed the MMU to drive the data bus.

Figure 4.5 shows the format of the Random Register. Upon reset, the content of this register is undefined.

Figure 4.5
Random Register



Reserved **Reserved Bits** **[31:11], [7:0]**
These bits are not writable, and read as zero.

Random **Random Register Counter** **[10:8]**
These bits contain the Random Register count. A write to this register initializes the counter. A read from this register reads from the counter.

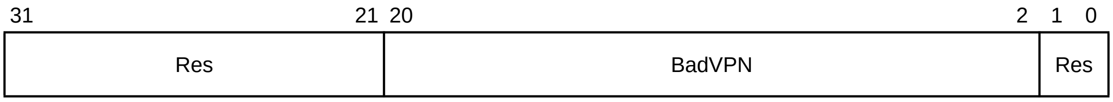
4.5.1.3 Context Register (R4)

LSI Logic has included this register to maintain R3000 compatibility. It provides a register and format useful for TLB Exception Handlers. However, the MiniRISC MMU does not implement the full R3000 Context Register. LSI Logic has removed the Physical Table Entry (PTE) Base Field. The BadVPN Field is in the same location as in the R3000, Bits [20:2].

The CW400x cannot write to this register. When the CW400x reads from this register, the MMU drives Bits [30:12] of the BadVA Register onto the data bus.

Figure 4.6 shows the format of the Context Register. Upon reset, the content of this register is undefined.

Figure 4.6
Context Register



Res **Reserved Bits** **[30:21], [1:0]**
These bits are not writable, and read as zero.

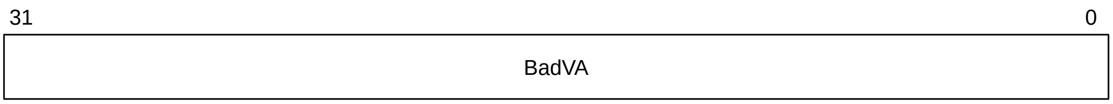
BadVPN **Bad Virtual Page Number** **[20:2]**
This field holds the virtual page number from the BadVA Register, which contains the virtual page number that caused the last Address Error or MMU Exception.

4.5.1.4 Bad Virtual Address (BadVA) Register (R8)

The Bad Virtual Address (BadVA) Register is a read-only register that saves the bad virtual address associated with an illegal access (an address exception, either an address error or a TLB Exception). This register saves only addresses for addressing errors (CW400x Cause Register Exception Code AdEL, AdES, TLBMOD, TLBL, or TLBS), not bus errors.

Figure 4.7 shows the format of the Bad Virtual Address Register. Upon reset, the content of this register is undefined.

Figure 4.7
Bad Virtual Address
Register



BadVA **Bad Virtual Address** **[31:0]**
This field stores the virtual address that was the cause of either an Address Error or an MMU Exception.

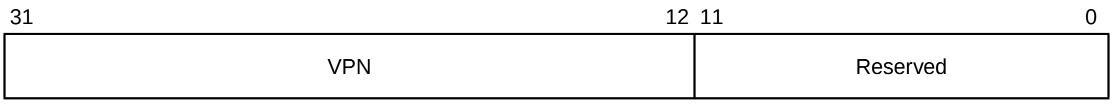
4.5.2 Other TLB Registers

The EntryHi and EntryLo Registers are the TLB entries.

4.5.2.1 TLB EntryHi Register (R10)

This register refers to the CAM entries.

Figure 4.8
TLB EntryHi Register



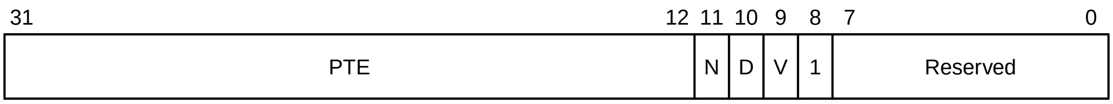
VPN **Virtual Page Number** **[31:12]**
This 20-bit field contains the Virtual Page Number for an entry in the TLB. This is the value compared with the 20 MSBs of the address from the CW400x Microprocessor (CAM) to detect a TLB hit.

Reserved **Reserved Bits** **[11:0]**
These bits are not writable, and read as zero.

4.5.2.2 TLB EntryLo Register (R2)

This register refers to the RAM entries.

Figure 4.9
TLB EntryLo Register



PTE — **Page Table Entry** **[31:12]**
This 20-bit field contains the Page Table Entry for each entry in the TLB. This is the value that replaces the 20 MSBs of the address from the CW400x Microprocessor to generate a physical address.

N — **Non-Cacheable** **11**
This bit set to one, indicates that caching this page is not allowed.

D — **Dirty** **10**
This bit set to one, indicates that this line is dirty and is writable. Writing to a page with this bit cleared to zero causes a TLB Modified Exception.

V — **Valid** **9**
This bit set to one, indicates that this line is valid. An access to a line with this bit cleared to zero causes a TLB Load or TLB Store Miss Exception.

1 — **One** **8**
This bit is hardwired to one for MIPS compatibility.

Reserved — **Reserved Bits** **[7:0]**
These bits are not writable, and read as zero.

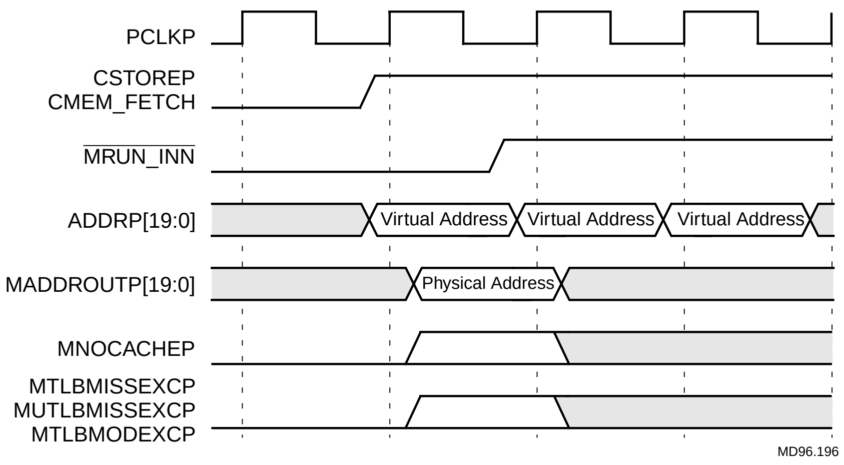
4.6 Address Translation Functional Waveform

Figure 4.10 shows the timing for a simple MMU address translation. A Match Cycle can occur every cycle, and therefore is independent of the instruction pipeline. A translation occurs for each Bus Run Cycle. The rising edge of the clock starts the Match Cycle. If there is a match, the MMU drives data onto MADDROUTP[19:0]. The MNOCACHEP signal is directly driven from the RAM contents, or from a *kseg1* hard translation. Asserting MNOCACHEP causes the BIU to invalidate any cache access for the current transaction. The MMU determines the type of TLB Exception from the values in the CAM array. These values are available after the rising clock edge.

Figure 4.10 is also valid for nontranslated addresses. If the translation does not occur, the MMU registers the address and drives it onto MADDROUTP[19:0] after the clock edge, just as it drives a translated address.

For a detailed description of MMU register reading and writing, see Section 2.5, "Read/Write Transactions."

Figure 4.10
MMU Address Translation



4.7 TLB Exceptions

The following exceptions are valid for the MiniRISC MMU. None of them are maskable. For more information on exceptions, see Chapter 5 of the *MiniRISC CW400x Microprocessor Core Technical Manual*.

Three types of TLB exceptions that can occur in the MiniRISC MMU:

- If the input Virtual Page Number (VPN) does not match the VPN of any TLB entry, it causes a:
  - UTLB Miss, for *kuseg*
  - TLB Miss, for *kseg2*
- If an entry matches, but the Valid Bit is not set, it causes a TLB Miss.
- If the dirty bit in a matching TLB entry is not set and the access is a write, it causes a TLB Modified Exception.

For more information on *kuseg* and *kseg2*, see Section 6.2 of the *MiniRISC CW400x Microprocessor Core Technical Manual*.

4.7.1 TLB Miss Exception

4.7.1.1 Cause

A TLB Miss Exception occurs when the MMU does not map a virtual address reference to memory and the most significant bit of the address is one, or when a virtual address reference to memory matches an invalid TLB Entry (the Valid Bit in the matching EntryLo Register is zero).

4.7.1.2 Handling

The CW400x branches to the General Exception Vector (0x80000080 or 0xBFC00180) and sets the TLBL or TLBS Code in the Cause Register ExcCode Field to indicate whether the miss was due to an instruction fetch, a load operation (TLBL), or a store operation (TLBS).

The EPC Register points to the instruction that caused the exception, unless the instruction is in a branch delay slot and the branch is taken. In that case, the EPC Register points to the branch instruction that preceded the exceptional instruction, and the CW400x sets the BD Bit of the Cause Register.

The MMU saves the KUp, IEp, KUc, and IEc Bits of the Status Register into the KUo, IEo, KUp, and IEp Bits, respectively, and clears the KUc and IEc Bits.

When this exception occurs, the BadVA and Context Registers contain the virtual address that failed address translation. If the exception was caused by an invalid entry, the Index Register contains the address of the invalid entry. If the exception was caused by an unmapped reference, the Index Register contains the value from the Random Register, and the P Bit is set. The value in the Random Register is the address of a suggested entry to be replaced.

4.7.1.3 Servicing

The Index Register refers to the invalid or suggested entry to be replaced. The operating system should load the entry's EntryLo Register with the appropriate PTE that contains the physical page frame and access control bits.

4.7.2 TLB Modified Exception

4.7.2.1 Cause

A TLB Modified Exception occurs when a store operation's virtual address reference to memory matches a TLB entry that is marked valid but not dirty (the Valid Bit is set, but the Dirty Bit is not set for the matching entry).

4.7.2.2 Handling

The CW400x branches to the General Exception Vector (0x80000080 or 0xBFC00180) and sets the TLBMOD Code in the Cause Register ExcCode Field.

The EPC Register points to the instruction that caused the exception, unless the instruction is in a branch delay slot and the branch is taken. In that case, the EPC Register points to the branch instruction that preceded the exceptional instruction, and the CW400x sets the BD Bit of the Cause Register.

The MMU saves the KUp, IEp, KUc, and IEc Bits of the Status Register into the KUo, IEo, KUp, and IEp Bits, respectively, and clears the KUc and IEc Bits.

When this exception occurs, the BadVA and Context Registers contain the virtual address that failed address translation.

4.7.2.3 Servicing

The BadVA Register and Index Register hold the address and index of the failing virtual address and entry. The operating system should transfer control to the appropriate system routine.

4.7.3 UTLB Miss Exception

4.7.3.1 Cause

A virtual address reference to unmapped user memory space (the most significant bit in the address is zero) causes a UTLB Miss Exception.

4.7.3.2 Handling

The CW400x branches to the UTLB Miss Exception Vector (0x80000000 or 0xBFC00100) and sets the TLBL or TLBS Code in the Cause Register

ExcCode Field to indicate whether the miss was due to an instruction fetch or load operation (TLBL) or a store operation (TLBS).

The EPC Register points to the instruction that caused the exception, unless the instruction is in a branch delay slot and the branch is taken. In that case, the EPC Register points to the branch instruction that preceded the instruction that caused the exception, and the CW400x sets the BD Bit of the Cause Register.

The MMU saves the KUp, IEp, KUc, and IEc Bits of the Status Register into the KUo, IEo, KUp, and IEp Bits, respectively, and clears the KUc and IEc Bits.

When this exception occurs, the BadVA and Context Registers contain the virtual address that failed address translation. The Index Register contains the value from the Random Register and the P Bit is set, indicating no matching TLB entries.

4.7.3.3 Servicing

The value in the Index Register refers to a suggested entry to be replaced. The operating system should load the entry's EntryLo Register with the appropriate PTE that contains the physical page frame and access control bits. The operating system should load the respective EntryHi with the Virtual Page Number.

4.8 Differences from the R3000 MMU

This section explains the differences between the MiniRISC MMU Building Block and the MIPS R3000 MMU.

4.8.1 Writing and Reading MMU Registers

Each register in the MiniRISC MMU is part of the CP0 Register Set, but the MMU does not have access to the internal CW400x data bus. Therefore, the CW400x must stall at least one cycle to allow the external data bus (DATAP[31:0]) to be driven with the data for the MTC0 or MFC0 Instruction. The MMU ignores the address bus (ADDRP[31:0]) during an MTC0/MFC0 Instruction, since the instruction itself contains all information necessary for the write.

In contrast to the R3000, the MiniRISC MMU allows direct TLB access. This makes the TLBP, TLBR, TLBWI, and TLBWR Instructions obsolete. The MTC0/MFC0 EntryHi/EntryLo Instructions directly access the data in the CAM/RAM instead of requiring an MTC0 EntryHi/TLBWI-type instruction combination. This difference makes the MMU design simpler and smaller, as well as more testable. To access a TLB entry, software must load the Index Register with the value to be accessed, then perform a MTC0/MFC0 EntryHi/EntryLo Instruction.

4.8.2 Unique Features of the MiniRISC MMU

Other differences between the MiniRISC MMU and a MIPS R3000 MMU are:

- The MMU only implements an 8-entry TLB, there are no PID Fields and no G Bit. To maintain a compatibility with MIPS code, the G Bit is set HIGH for MFC0 EntryLo Instructions. All TLB entries are assumed to be global.
- The Context Register only implements the BadVPN Field, not the PTE Base Field.

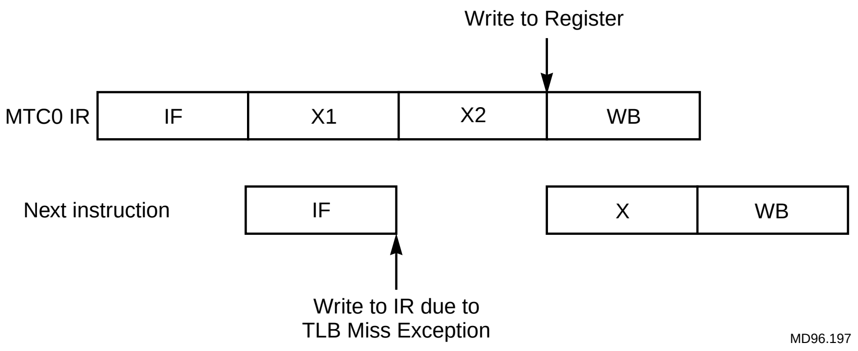
4.9 Operation Peculiarities and Details

This section describes conditions in which the pipeline, or events, cause unexpected results unless these conditions are prevented.

4.9.1 MTC0 X2/IF TLB Miss Peculiarities

This section explains scenarios in which an exception in the IF Stage of an instruction following a MTC0/MFC0 Instruction (whose action occurs one cycle later, in the X2 Stage) causes an unexpected result. Figure 4.11 illustrates the pipeline stages for each instruction when this situation occurs.

Figure 4.11 MTC0 Inconsistency Pipeline



- Index Register (IR)

  If a write to the IR is followed by an instruction that causes a TLB Miss Exception due to a faulty Instruction Fetch (IF), the MMU generates the exception at the end of the IF Stage. The write to the IR from the MTC0 IR Instruction does not occur until the end of the X2 Stage, which causes the MMU to write this value over the value generated by the IF Exception. This causes an incorrect value to be passed to the operating system. In addition, if the MTC0 IR, MFC0 IR sequence is followed by an instruction generating a TLB Miss Condition in the IF Stage, the IF causes the MMU to overwrite the IR with the value causing the exception, before the MFC0 Instruction can do the IR read. This overwrite causes the register to have an unexpected value.

- Translation Lookaside Buffer (TLB)

  The state of the TLB must be preserved so the CW400x can examine it. If, however, an MTC0 EntryHi or MTC0 EntryLo Instruction occurs, followed by an IF TLB Miss, the instruction might replace the entry in the TLB that caused the IF Exception.

- ♦ Random Register (RR)

  The MMU might produce an unexpected result when using the RR to protect low address entries in the TLB. If the MTC0 Instruction is an MTC0 RR, increasing the value held in the RR to protect more addresses (say from 1 to 3), and the instruction following creates an IF TLB Miss, the RR address written to the IR might be in the region of the TLB that the previous MTC0 RR is attempting to protect (1 or 2). In this case, if the value in the IR is read and used as a replacement location for a new TLB entry, the address might be inside the newly protected area. To prevent this condition, ensure that a MCT0 RR Instruction that increases the protected area is not followed by an instruction that can cause a TLB Miss. The converse situation (decreasing the protected area) can cause a nonoptimal page replacement scheme for the missed page, causing fewer locations than possible to be considered.

Programmers should avoid creating software that executes an MTC0/MFC0 Instruction before any instruction that might cause an IF Exception. Software should only perform TLB maintenance in non-mapped space. If this is not possible, the programmer must ensure that the page containing the instruction that follows the MTC0 Instruction is valid, and not on a page that is not loaded into the TLB.

There is another case where a previous instruction's write to the MMU registers overwrites the current instruction's writes. However, in this case, the overwrite does not cause an error, so nothing special need be done. This situation occurs, for example, when the instruction labeled MTC0 IR in Figure 4.11 is a Store or Load Instruction with a TLB Miss Exception in the X2 Stage, followed by any instruction generating a TLB Miss in the IF Stage. The write to the IR in the X2 Stage would, in this case, be caused by an exception, not an MTC0 Instruction. The exception from the store or load should be taken. In this case, the correct exception overwrites the write to the IR because of the second exception.

4.9.2 Exceptions Between MTC0 EntryHi/EntryLo Instructions

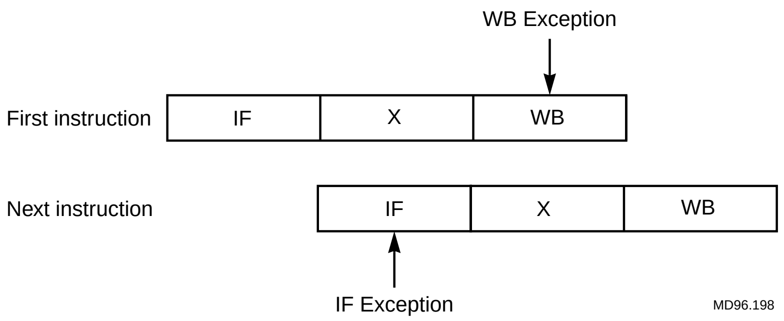
Unlike the R3000, the MiniRISC MMU TLB EntryHi and EntryLo Registers are actually located in the TLB. Because of this difference, the TLB Entry is not updated in one cycle, as in the R3000. If an exception occurs between a MTC0 EntryHi and an MTC0 EntryLo Instruction, the MMU TLB enters an intermediate state, only partially updated. (In the R3000, these instructions only write to separate registers that are then loaded simultaneously into the TLB with a TLBW Instruction. This causes the entire entry to be written at once, removing the possibility of an intermediate state stored in the TLB if an exception occurs.)

Because of this condition, it is necessary to prevent exceptions and interrupts from occurring between MTC0 EntryHi and EntryLo Instructions (in either order) to maintain R3000-compatibility.

4.9.3 Register Consistency in the Pipeline

It is possible to corrupt all of the MMU registers unexpectedly when a WB Exception in one instruction is followed by an IF Exception (an address error, or MMU exception, that causes a write to the MMU registers) in the following instruction (see Figure 4.12).

Figure 4.12
WB Exception Followed by an IF Exception



Since WB Exceptions do not write the MMU registers, and it is not known until after the IF Exception occurs that the registers should not be written, this condition unexpectedly corrupts the MMU registers. Therefore, if the value in the registers is important, it is important to ensure that this condition does not occur. In general, if these registers are important, LSI Logic recommends that the user prevent the generation of an IF or X2 Exception in the subsequent code (usually the exception handler).

This condition also occurs during simultaneous exceptions. The CP0 prioritizes exceptions based on the type of exception and which stage of the pipeline it is in. Therefore, it is possible that the MMU exception (or address error) will cause a lower priority exception than the exception that is actually recognized. However, this exception writes the MMU registers, causing the MMU registers to be in an unexpected state.

4.9.4 TLB Initialization

Initializing the TLB requires at least one NOP Cycle with a valid clock signal, and that all control inputs be inactive. The MMU generates this required NOP when executing an MT/MFC0 to those MMU registers not in the TLB.

To properly and quickly initialize the TLB and prevent an intermediate state, assert $\overline{\text{BSYSRESETN}}$ for at least one, preferably three clock cycles. Asserting $\overline{\text{BSYSRESETN}}$ forces all control inputs LOW and allows the gated clock to clock the TLB during a reset cycle.

Chapter 5
Basic BIU and Cache Controller (BBCC)

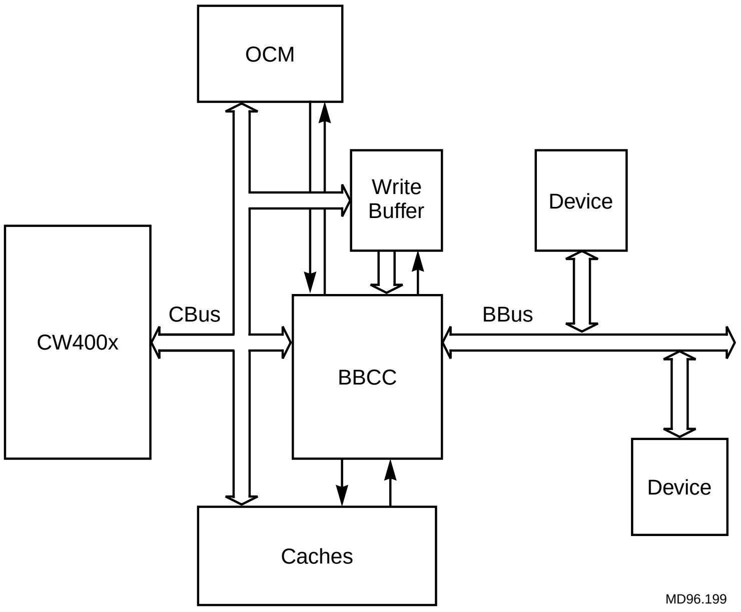
This chapter describes the Basic BIU and Cache Controller (BBCC) building block for the CW400x Core. For information on how to add or remove a write buffer, see Chapter 6.

This chapter contains the following sections:

5.1 Overview

The Basic BIU and Cache Controller (BBCC) is a generic bus interface unit and cache controller for use with the CW400x Microprocessor Core. It serves as the CW400x’s interface to on-chip memory (OCM), the cache RAMs, and other devices. Figure 5.1 shows a block diagram of a system using the CW400x and the BBCC.

Figure 5.1
CW400x System
with the BBCC



5.2 Features

The BBCC supports:

- ♦ Two-way set associative or direct-mapped Instruction-cache (I-cache)
- ♦ Direct-mapped Data-cache (D-cache)
- ♦ Cache sizes of 1 to 32 Kbytes (for D-cache and each I-cache set)
- ♦ Locking of I-cache Set 0 lines
- ♦ Snooping on I-cache and/or D-cache
- ♦ Software Cache Test Mode
- ♦ Hardware Cache Test Mode
- ♦ Minimum cache miss penalty of two clock cycles
- ♦ 1-8 deep write buffer (one internal entry, with support for seven external write buffer entries)
- ♦ Load Scheduling
- ♦ Instruction Streaming
- ♦ Data Streaming

- Read Priority
- Block Fetching
- Burst Writes
- Reset Control
- System Configuration Register

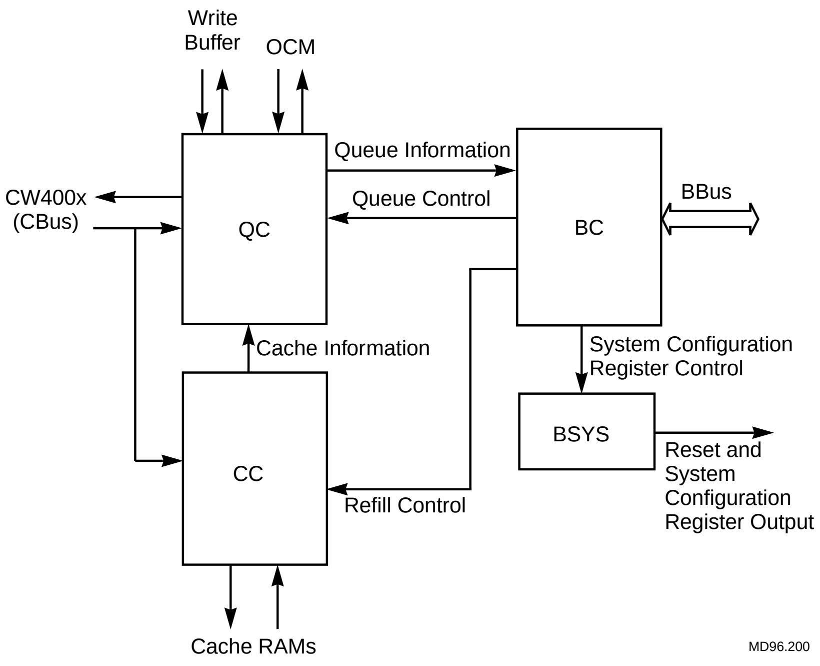
5.3 Functional Description

The BBCC consists of four modules:

- Cache Controller (CC)
- Queue Controller (QC)
- BBus Controller (BC)
- System Configuration Module (BSYS)

Figure 5.2 shows the communication between the modules and which modules control the external interfaces.

Figure 5.2
BBCC Internal Block Diagram



5.3.1 Cache Controller (CC)

The Cache Controller (CC) controls the system instruction and data caches. It identifies instruction and data transactions, checks if the appropriate line is in cache, and (if the line is in cache) performs the requested transaction. The CC also informs the QC of the status of the transaction (cache hit/miss and other cache-related information), and allows the BC to read and write the cache and Tag RAMs.

The CC has the following features:

♦ **Cache Configuration:** The CC allows for the I-cache to be configured to be direct-mapped or two-way set associative. The D-cache is always direct-mapped.

♦ **Instruction Set 0 Line-locking:** Each line of I-cache Set 0 can be locked (Lock Bit) to guarantee that the contents of the selected lines remain in cache.

♦ **Flexible Cache Size:** The CC supports maximum cache sizes of 64 Kbytes for I-cache (32 Kbytes for each set) and 32 Kbytes for D-cache. The minimum supported size is 1 Kbyte for I-cache (2 Kbytes for two-way set associative) and 1 Kbyte for D-cache. Smaller I-caches can be supported with a small amount of additional logic.

♦ **Software Cache Test Mode:** The CC allows the CW400x to directly write/read to/from the cache and Tag RAMs.

♦ **Direct Cache Access:** The CC allows the BC to directly access the cache and Tag RAMs. This mechanism can be used to perform Hardware Test of the cache RAMs.

♦ **Snooping:** The CC provides logic to check the tags and invalidate lines in the D-cache and I-cache. If snooping is performed on only D-cache or I-cache, each snoop requires two clock cycles. If both D-cache and I-cache snooping are performed, each snoop requires four clock cycles.

♦ **Integrated Instruction-cache Set 0 and Data-cache:** I-cache Set 0 and D-cache share the same physical data and Tag RAMs.

5.3.2 Queue Controller (QC)

The Queue Controller (QC) orders memory requests to the main memory. It consists of three queues:

♦ Instruction Fetch Queue (one entry)

- Data Fetch Queue (one entry)
- Data Store Queue (one internal entry, with the ability to add up to seven additional write buffer entries external to the BBCC)

The QC monitors the CW400x memory transaction signals, then enters requests into the appropriate queues. The QC arbitrates requests in the queue and generates transaction requests to the BC. The QC also issues the Ready Signals to the CW400x, and when necessary, stalls the system.

The QC has the following features:

- **Burst Writes:** When consecutive stores are to the same page, and the BBus device is able to handle burst writes, the QC gives priority to stores.
- **Read Priority:** If configured to do so, the QC gives data fetches higher priority than stores, when possible.
- **OCM Control:** The QC issues the write enable and output enable signals to the OCM.
- **Variable Write Buffer Depth:** One store queue entry is internal to the BBCC, and up to seven additional write buffer entries can be added externally.

5.3.3 BBus Controller (BC)

The BBus is the Basic Bus, which serves as a generic interface to memory, DMA controllers, DRAM controllers, and other devices. Its characteristics are:

- 32-bit bus with separate address and data buses
- Zero to n wait states (one-cycle transactions are possible)
- Burst transactions (BBCC supports bursts of 1, 2, 4, and 8 words)
- Back to back transactions
- Maximum bandwidth of 200 Mbytes/s at 50 MHz
- Bus error and bus retry reporting
- Multiple bus masters possible

For more information on the BBus, see Section 5.5.2, “Basic Bus (BBus).”

The BBus Controller (BC) handles the BBus. The BC handles requests by the CW400x, and uses a direct cache access interface to refill the cache with instructions or data. The BC also uses a direct cache access interface to refill the caches during Hardware Cache Test Mode.

The BC has two modes: Master and Slave. When the BC is master, there are two types of requests: memory reads and memory writes. When the BC is a slave (when an external device is granted the bus and starts a BBus transaction), the BC can perform a direct cache access operation, or snoop on the BBus transaction.

The BC has the following features:

- Block Fetching Support
- System Configuration Control
- Snooping Support

5.3.4 System Configuration Module (BSYS)

The System Configuration Module issues reset signals to the system, and contains the System Configuration Register. The BC controls the System Configuration Register. When the CW400x loads or stores to the address 0xBFFF0000, the transaction accesses the System Configuration Register rather than memory. All the bits of the System Configuration Register are output by the BBCC on BS_CONFIGP[31:0]. Some of the bits are predefined; those that are not predefined are available for use by the system designer. Figure 5.3 shows the System Configuration Register.

Figure 5.3
System Configuration Register

31	30	29 14	13	12 10	9 8	7	6 5	4	3 2	1	0
A	W	A	E	PS	CM	R	DRS	D	IRS	1E	IE

A **Available** **31, [29:14]**
These bits are currently unassigned and are available.

W **Write Buffer Enable** **30**
Setting this bit, enables the Write Buffer.

E **Data Error** **13**
The BBCC sets this bit when a Bus Error occurs during a data load/store. Writing a zero to this bit, clears it.

PS **Page Size** **[12:10]**

Writing these bits, informs the BBCC of the page size so it can determine whether stores are burst writes (consecutive stores to the same page).

PS	Page Size	PS	Page Size
000	16 words	100	256 words
001	32 words	101	512 words
010	64 words	110	1024 words
011	128 words	111	2048 words

CM **Cache Mode** **[9:8]**

Writing these bits, sets the Cache Mode.

CM	Cache Mode
00	Normal
01	I-Cache Software Test - Data RAM
10	I-Cache Software Test - Tag RAM
11	D-Cache Software Test - Tag RAM

R **Read Priority Enable** **7**

Setting this bit, causes the BBCC to give loads higher priority than stores, when possible.

DRS **D-Cache Block Refill Size** **[6:5]**

Writing these bits, sets the D-cache Block Refill Size.

DRS	Refill Size
00	1 word
01	2 words
10	4 words
11	8 words

D **D-Cache Enable** **4**

Setting this bit, enables the Data Cache.

IRS **I-Cache Block Refill Size** **[3:2]**

Writing these bits, sets the I-cache Block Refill Size.

IRS	Refill Size
00	1 word
01	2 words
10	4 words
11	8 words

1E **I-Cache Set 1 Enable** **1**
Setting this bit, enables the Instruction Cache Set 1.

IE **I-Cache Enable** **0**
Setting this bit, enables the Instruction Cache.

5.4 Signals

This section describes the signals that comprise the bit-level interface of the BBCC. Tables 5.1 through 5.3 summarize the BBCC signals. Detailed descriptions follow the tables.

The signals are described in alphabetical order by mnemonic. Each signal definition contains the mnemonic and the full signal name. The mnemonics for active LOW signals have an overbar over their names.

In the descriptions that follow, “assert” means to drive TRUE or active and “deassert” means to drive FALSE or inactive.

Table 5.1
BBCC Input Signals Summary

Input	Source	Description
ADDRP[14:2]	CW400x	CW400x Address Bus
BADDRPI[31:2]	BBus	BBus Address Input Bus
BCACHE_SELP	BBus	Hardware Cache Test Mode
BDATAPI[31:0]	BBus	BBus Data Input Bus
BDSNOOP	User-Specified Value	Data Snoop Enable
$\overline{\text{BERRORN}}$	BBus	BBus Bus Error
$\overline{\text{BGNTN}}$	BBus	BBus Bus Grant
$\overline{\text{BIP\_DNI}}$	BBus	BBus Instruction/Data Input Indicator
BISETP	BBus	BBus Instruction Cache Set
BISNOOP	User-Specified Value	Instruction Snoop Enable
$\overline{\text{BIUOEN}}$	Global Output Enable	BBCC Data Output Enable

(Sheet 1 of 3)

Table 5.1 (Cont.)
BBCC Input Signals
Summary

Input	Source	Description
$\overline{\text{BLKGNTN}}$	BBus	BBus Block Transaction Grant
BOCMEXISTP	User-Specified Value	On-Chip Memory (OCM) Exists Indicator
BOCMSELP	OCM	OCM Memory Transaction
$\overline{\text{BRDYNI}}$	BBus	BBus Ready Input
$\overline{\text{BRESETN}}$	System Logic/Reset Module	Reset
$\overline{\text{BRETRYN}}$	BBus	BBus Bus Retry
$\overline{\text{BRUN\_INN}}$	Global Output Enable	Run Enable
$\overline{\text{BSTARTNI}}$	BBus	BBus Transaction Start Input
$\overline{\text{BTAGTESTN}}$	BBus	BBus Tag RAM Transaction
$\overline{\text{BTXNI}}$	BBus	BBus Transaction Indicator Input
$\overline{\text{BWBURST\_REQN}}$	BBus	BBus Burst Write Request
$\overline{\text{BWRNI}}$	BBus	BBus Write Transaction Indicator Input
CBYTEP[3:0]	CW400x	CBus Byte Enables
$\overline{\text{CIP\_DN}}$	CW400x	Instruction/Data Indicator
CKILLMEMP	CW400x	Kill Memory Transaction Request
CMEM_FETCHP	CW400x	Fetch Indicator
CSTOREP	CW400x	Store Indicator
IDLCKP	I-Set 0/D Tag RAM	Lock Bit
IDMATCHP	I-Set 0/D Tag Match Logic	Tag Match
IDVLDP[3:0]	I-Set 0/D Tag RAM	Valid Bits
I1MATCHP	I-Set 1 Tag Match Logic	Tag Match
I1VLDP[3:0]	I-Set 1 Tag RAM	Valid Bits
MADDROUTP[31:2]	MMU	Mapped Address
MEARLYKS1P	MMU	MMU Early *kseg1* Indicator
MNOCACHEP	MMU	Mapped Address Not Cacheable
PCLKP	System Logic	System Clock
SE	System Logic	Scan Enable
SI	Scan Chain	Scan Data In
TST	System Logic	Test Enable
WB_ADDRP[31:2]	Write Buffer	Write Buffer Address
WB_ARRIVEBFLDP	Write Buffer	Write Buffer Store Arrived Before Load
WB_BYTEP[3:0]	Write Buffer	Write Buffer Byte Enables

(Sheet 2 of 3)

Table 5.1 (Cont.)
BBCC Input Signals
Summary

Input	Source	Description
WB_CFGP	Write Buffer	Write Buffer Store to System Configuration Register
WB_DATAP[31:0]	Write Buffer	Write Buffer Data
WB_FULLP	Write Buffer	Write Buffer Full
WB_STPNDP	Write Buffer	Write Buffer Store Pending
WB_VWBFLDP	Write Buffer	Write Buffer Valid Write Before Load
(Sheet 3 of 3)		

Table 5.2
BBCC Output Signals
Summary

Output	Destination	Description
BADDRPO[31:2]	BBus	BBus Address Output Bus
$\overline{\text{BB\_SLVDOEN}}$	Global Output Enable	Slave Data Output Enable
$\overline{\text{BBUS\_STEALN}}$	Various, External Logic	CBus Bus Steal Indicator
$\overline{\text{BCPURESETN}}$	CW400x	CW400x Reset
BDATAPO[31:0]	BBus	BBus Data Output Bus
BDOEP	BBus	BBCC Data Output Enable Request
BDRDYP	CW400x	Load Data Ready
BIBERRORP	CW400x	Instruction Bus Error
$\overline{\text{BIP\_DNO}}$	BBus	BBus Instruction/Data Output Indicator
BIRDYP	CW400x	Instruction Data Ready
$\overline{\text{BLKREQN}}$	BBus	BBus Block Transaction Request
BMCNTLOEP	BBus	BBCC Master Control Output Enable Request
BQ_OCMOEP	OCM	OCM Output Enable
$\overline{\text{BQ\_OCMWEN}}$	OCM	OCM Write Enable
$\overline{\text{BRDYNO}}$	BBus	BBus Ready Output
$\overline{\text{BREQN}}$	BBus	BBus Bus Request
BRUN_OUTP	Global Output Enable	Run Enable Output
BSCNTLOEP	BBus	BBus Slave Control Output Enable Request
BS_CONFIGP[31:0]	Various, External Logic	System Configuration Register
BSNOOPWAITP	BBus	BBus Snoop Wait

(Sheet 1 of 3)

Table 5.2 (Cont.)
BBCC Output Signals
Summary

Output	Destination	Description
$\overline{\text{BSTARTNO}}$	BBus	BBus Transaction Start Output
$\overline{\text{BSYSRESETN}}$	Various, External Logic	Reset
$\overline{\text{BTXNO}}$	BBus	BBus Transaction Indicator Output
$\overline{\text{BWBURST\_GNTN}}$	BBus	BBus Burst Write Grant
$\overline{\text{BWRNO}}$	BBus	BBus Write Transaction Indicator Output
BW_ARRIVEBFLDP	Write Buffer	Store Arrived Before Load
BW_CFGSELP	Write Buffer	Store to System Configuration Register
BW_DFDONEP	Write Buffer	Data Fetch Done
BW_DFQADDRP[3:0]	Write Buffer	Data Fetch Queue Address Bits
BW_DFQUPDATEP	Write Buffer	Data Fetch Queue Update
BW_RDSTQP	Write Buffer	Read Store Queue
BW_STPNDP	Write Buffer	Store Pending
BW_WRSTQP	Write Buffer	Write Store Queue
$\overline{\text{BYTENO}}$[3:0]	BBus	BBus Byte Enables Output
BZ_IDDCLKP	I-Set 0/D Data RAM	I-Cache Set 0/D-Cache Data RAM Clock
BZ_IDDOEP	I-Set 0/D Data RAM	I-Cache Set 0/D-Cache Data RAM Output Enable
BZ_IDDWEP[3:0]	I-Set 0/D Data RAM	I-Cache Set 0/D-Cache Data RAM Write Enables
BZ_IDTCLKP	I-Set 1 Tag RAM	I-Cache Set 0/D-Cache Tag RAM Clock
BZ_IDTWEP[5:0]	I-Set 1 Tag RAM	I-Cache Set 0/D-Cache Tag RAM Write Enables
$\overline{\text{BZ\_IDT\_OEN}}$	External 3-state Gates	I-Cache Set 0/D-Cache Tag RAM Output Enable
BZ_INDEXP[12:0]	Cache RAMs	Cache RAM Index
$\overline{\text{BZ\_IP\_DN}}$	Cache RAMs	Instruction/Data Cache Select
$\overline{\text{BZ\_IP\_DN\_L}}$	Tag Match Logic	Instruction/Data Cache Select, Registered
BZ_I1DCLKP	I-Set 1 Data RAM	I-Cache Set 1 Data RAM Clock
BZ_I1DOEP	I-Set 1 Data RAM	I-Cache Set 1 Data RAM Output Enable
BZ_I1DWEP	I-Set 1 Data RAM	I-Cache Set 1 Data RAM Write Enable
BZ_I1TCLKP	I-Set 0/Data Tag RAM	I-Cache Set 1 Tag RAM Clock
BZ_I1TWEP[4:0]	I-Set 0/Data Tag RAM	I-Cache Set 1 Tag RAM Write Enables

(Sheet 2 of 3)

Table 5.2 (Cont.)
BBCC Output Signals
Summary

Output	Destination	Description
$\overline{\text{BZ\_I1T\_OEN}}$	External 3-state Gates	I-Cache Set 1 Tag RAM Output Enable
BZ_LOCKP	I-Set 0/Data Tag RAM	Cache RAM Lock Bit
BZ_TAGP[21:0]	Cache Tag RAMs	Cache RAM Tag
BZ_TAG4MATCHP[21:0]	Tag Match Logic	Tag for Tag Match
BZ_VALIDP[3:0]	Cache Tag RAMs	Cache RAM Valid Bits
SO	Scan Chain	Scan Data Out

(Sheet 3 of 3)

Table 5.3
BBCC Bidirectional
Signals Summary

Bidirectional	Connect	Description
DATAP[31:0]	Various, External Logic	Data Bus

ADDRP[14:2] **CW400x Address Bus** **Input**
The CW400x drives the lower bits of the unmapped address for CBus transactions onto these inputs. These signals are valid in the clock cycle before the run cycle.

BADDRPI[31:2]
BBus Address Input Bus **Input**
When the BBCC is a slave on the BBus, the BBus master drives the address bus for BBus transactions onto these inputs.

BADDRPO[31:2]
BBus Address Output Bus **Output**
When the BBCC is a master on the BBus, the BBCC drives the address for BBus transactions onto these outputs.

$\overline{\text{BB\_SLVDOEN}}$ **Slave Data Output** **Output**
This signal is valid when the BBCC asserts $\overline{\text{BBUS\_STEALN}}$. The BBCC asserts this signal to the Global Output Enable Module (GOE) to indicate that in the next clock cycle, one of the cache RAMs will drive DATAP[31:0].

$\overline{\text{BBUS\_STEALN}}$
CBus Bus Steal **Output**
The BBCC asserts this signal to indicate that it will control driving data on DATAP[31:0] during the next clock cycle.

BCACHE_SELP
Hardware Cache Test Mode **Input**
This signal connects to the BBus. Asserting this signal causes the BBCC to interpret BBus transactions as Hardware Cache Mode transactions.

$\overline{\text{BCPURESETN}}$
CW400x Reset **Output**
The BBCC asserts this signal to reset the CW400x.

BDATAPI[31:0]
BBus Data Input Bus **Input**
The BBus device drives the input data for BBus transactions onto these signals.

BDATAPO[31:0]	**BBus Data Output Bus** The BBCC drives the output data for BBus transactions onto these signals.	**Output**
BDOEP	**BBus Data Output Enable Request** This signal is provided for systems with a 3-state BBus design. The BBCC asserts this signal to inform the BBus output enable logic that the BBCC wants to place BDATAPO[31:0] on the appropriate 3-state bus.	**Output**
BDRDYP	**Load Data Ready** The BBCC asserts this signal to inform the CW400x that DATAP[31:0] contains valid data for a data fetch.	**Output**
BDSNOOP	**Data Snoop Enable** Asserting this signal causes the BBCC to snoop on the D-cache when writes are performed by an external master on the BBus.	**Input**
$\overline{\textbf{BERRORN}}$	**BBus Bus Error** Asserting this signal informs the BBCC that the current BBus transaction terminated with an error.	**Input**
$\overline{\textbf{BGNTN}}$	**BBus Bus Grant** Asserting this signal informs the BBCC that it has been granted mastership of the BBus.	**Input**
BIBERRORP	**Instruction Bus Error** The BBCC asserts this signal to inform the CW400x that the current instruction fetch terminated with an error.	**Output**
$\textbf{BIP\_}\overline{\textbf{DNI}}$	**BBus Instruction/Data Input Indicator** This signal is used during Hardware Cache Test Mode. The BBus drives this signal HIGH to inform the BBCC that the I-cache is being accessed. The BBus drives this signal LOW to inform the BBCC that the D-cache is being accessed.	**Input**
$\textbf{BIP\_}\overline{\textbf{DNO}}$	**BBus Instruction/Data Output Indicator** The BBCC drives this signal HIGH to indicate that an instruction transaction is being performed on the BBus. The BBCC drives this signal LOW to indicate that an data transaction is being performed on the BBus.	**Output**

BIRDYP **Instruction Data Ready** **Output**
The BBCC asserts this signal to inform the CW400x that DATAP[31:0] contains valid data for an instruction fetch.

BISETP **BBus Instruction Cache Set** **Input**
This signal is used during Hardware Cache Test Mode. The BBus drives this signal HIGH to inform the BBCC that the I-cache Set 1 is being accessed. The BBus drives this signal LOW to inform the BBCC that the I-cache Set 0 is being accessed.

BISNOOP **Instruction Snoop Enable** **Input**
Asserting this signal causes the BBCC to snoop on the I-cache when writes are performed by an external master on the BBus.

$\overline{\text{BIUOEN}}$ **BBCC Data Output Enable** **Input**
The Global Output Enable Module (GOE) asserts this signal to cause the BBCC to drive DATAP[31:0].

$\overline{\text{BLKGNTN}}$ **BBus Block Transaction Grant** **Input**
Asserting this signal informs the BBCC that it has been granted permission to perform a block transaction on the BBus.

$\overline{\text{BLKREQN}}$ **BBus Block Transaction Request** **Output**
The BBCC asserts this signal to request a block transaction on the BBus.

BMCNTLOEP **BBCC Master Control Output Enable Request** **Output**
This signal, which connects to the BBus, is provided for systems with a 3-state BBus design. The BBCC asserts this signal to inform the BBus output enable logic that the BBCC is attempting to place BADDRPO[31:2], $\overline{\text{BIP\_DNO}}$, $\overline{\text{BSTARTNO}}$, $\overline{\text{BTXNO}}$, and $\overline{\text{BWRNO}}$ on their appropriate 3-state buses.

BOCMEXISTP **On-Chip Memory (OCM) Exists Indicator** **Input**
Asserting this signal informs the BBCC that the system has OCM.

BOCMSELP **OCM Memory Transaction** **Input**
Asserting this signal informs the BBCC that the current CBus transaction is for the OCM.

BQ_OCMOEP	**OCM Output Enable** The BBCC asserts this signal to cause the OCM to drive its data onto DATAP[31:0].	**Output**
$\overline{\text{BQ\_OCMWEN}}$	**OCM Write Enable** This output connects to the OCM. The BBCC asserts this signal to enable writes to the OCM.	**Output**
$\overline{\text{BRDYNI}}$	**BBus Ready Input** Asserting this signal informs the BBCC that the BBus transaction will finish at the end of the current clock cycle.	**Input**
$\overline{\text{BRDYNO}}$	**BBus Ready Output** The BBCC asserts this signal to indicate that the BBus transaction will finish at the end of the current clock cycle.	**Output**
$\overline{\text{BREQN}}$	**BBus Bus Request** The BBCC asserts this signal to request mastership of the BBus.	**Output**
$\overline{\text{BRESETN}}$	**Reset** Asserting this signal causes the BBCC to reset and generate the $\overline{\text{BCPURESETN}}$ and $\overline{\text{BSYSRESETN}}$ signals.	**Input**
$\overline{\text{BRETRYN}}$	**BBus Bus Retry** Asserting this signal informs the BBCC that it should retry the current BBus transaction.	**Input**
$\overline{\text{BRUN\_INN}}$	**Run Enable** The GOE asserts this signal to inform the BBCC that the next clock cycle will be a run cycle.	**Input**
BRUN_OUTP	**Run Enable Output** The BBCC asserts this signal to the GOE to enable the system to run. The BBCC deasserts this signal to the GOE to cause the system to stall.	**Output**
BSCNTLOEP	**BBus Slave Control Output Enable Request** This signal is provided for systems with a 3-state BBus design. The BBCC asserts this signal to inform the BBus output enable logic that the BBCC wants to place the $\overline{\text{BRDYNO}}$ signal on the appropriate 3-state bus.	**Output**

BS_CONFIGP[31:0]

System Configuration Register **Output**
The BBCC places the value of the System Configuration Register on these signals.

BSNOOPWAITP

BBus Snoop Wait **Output**
The BBCC asserts this signal to inform other devices attached to the BBus that they should not start another BBus transaction because the BBCC is performing both I-cache and D-cache snooping on a BBus transaction.

$\overline{\text{BSTARTNI}}$ **BBus Transaction Start Input** **Input**
Asserting this signal informs the BBCC that a BBus transaction is starting.

$\overline{\text{BSTARTNO}}$ **BBus Transaction Start Output** **Output**
The BBCC asserts this signal to indicate when it starts a BBus transaction.

$\overline{\text{BSYSRESETN}}$

Reset **Output**
The BBCC asserts this signal to reset the system.

$\overline{\text{BTAGTESTN}}$ **BBus Tag RAM Transaction** **Input**
This signal is used during Hardware Cache Test Mode. The BBus drives this signal LOW to inform the BBCC that the Tag RAM is being accessed. The BBus drives this signal HIGH to inform the BBCC that the Data RAM is being accessed.

$\overline{\text{BTXNI}}$ **BBus Transaction Indicator Input** **Input**
Asserting this signal informs the BBCC that a BBus transaction is in progress.

$\overline{\text{BTXNO}}$ **BBus Transaction Indicator Output** **Output**
The BBCC asserts this signal to indicate that a BBus transaction is in progress.

$\overline{\text{BWBURST\_GNTN}}$

BBus Burst Write Grant **Output**
The BBCC asserts this signal to indicate that it will perform a burst write.

$\overline{\text{BWBURST\_REQN}}$
BBus Burst Write Request **Input**
Asserting this signal requests that the BBCC perform a burst write.

$\overline{\text{BWRNI}}$ **BBus Write Transaction Indicator Input** **Input**
Asserting this signal informs the BBCC that the current BBus transaction is a write.

$\overline{\text{BWRNO}}$ **BBus Write Transaction Indicator Output** **Output**
The BBCC asserts this signal to indicate that the current BBus transaction is a write.

BW_ARRIVEBFLDP
Store Arrived Before Load **Output**
The BBCC asserts this signal to inform the Write Buffer that the current CBus store transaction is occurring while the Data Fetch Queue is empty.

BW_CFGSELP
Store to System Configuration Register **Output**
The BBCC asserts this signal to inform the Write Buffer that the current CBus store is to the System Configuration Register.

BW_DFDONEP
Data Fetch Done **Output**
The BBCC asserts this signal to inform the Write Buffer that a data fetch transaction has just completed.

BW_DFQADDRP[3:0]
Data Fetch Queue Address Bits **Output**
These signals are a few bits of the address from the Data Fetch Queue. The Write Buffer uses these bits to detect load/store dependencies.

BW_DFQUPDATEP
Data Fetch Queue Update **Output**
The BBCC asserts this signal to inform the Write Buffer that the Data Fetch Queue is being updated.

BW_RDSTQP **Read Store Queue** **Output**
The BBCC asserts this signal to initiate a read operation to the Write Buffer.

BW_STPNDP Store Pending Output

The BBCC asserts this signal to inform the Write Buffer that a store is pending in the Store Queue.

BW_WRSTQP Write Store Queue Output

The BBCC asserts this signal to initiate a write operation to the Write Buffer.

$\overline{\text{BYTENO}}$[3:0] BBus Byte Enables Output Output

These signals are the byte enables from the BBCC. The BBCC asserts the byte enables HIGH to inform the BBus device that the corresponding bytes are valid on BDATAPO[31:0].

The following table shows the correspondence between byte enables and the data bus bytes.

Byte Enable	Corresponding BDATAPO[31:0] Byte
$\overline{\text{BYTENO3}}$	[31:24]
$\overline{\text{BYTENO2}}$	[23:16]
$\overline{\text{BYTENO1}}$	[15:8]
$\overline{\text{BYTENO0}}$	[7:0]

BZ_IDDCLKP I-Cache Set 0/D-Cache Data RAM Clock Output

This output is connected to the I-cache Set 0/D-cache Data RAM clock input.

BZ_IDDOEP I-Cache Set 0/D-Cache Data RAM Output Enable Output

The BBCC asserts this signal to enable the I-cache Set 0/D-cache Data RAM to drive data onto DATAP[31:0].

BZ_IDDWEP[3:0]

I-Cache Set 0/D-Cache Data RAM Write Enables Output

The BBCC asserts these signals to enable writes to the I-cache Set 0/D-cache Data RAM from DATAP[31:0].

The following table shows the correspondence between write enables and the data bus bytes.

Write Enable	Corresponding DATAP[31:0] Byte
BZ_IDDWEP3	[31:24]
BZ_IDDWEP2	[23:16]
BZ_IDDWEP1	[15:8]
BZ_IDDWEP0	[7:0]

BZ_IDTCLKP **I-Cache Set 0/D-Cache Tag RAM Clock** **Output**
This output is connected to the I-cache Set 0/D-cache Tag RAM clock input.

BZ_IDTWEP[5:0]
I-Cache Set 0/D-Cache Tag RAM Write Enables **Output**
The BBCC asserts these signals to enable writes to the I-cache Set 0/D-cache Tag RAM. Figure 5.16 on page 5-48 shows the correspondence between byte enables and what is enabled.

$\overline{\text{BZ\_IDT\_OEN}}$ **I-Cache Set 0/D-Cache Tag RAM Output Enable** **Output**
This output is an input to a set of external 3-state gates. The BBCC asserts this signal to enable the 3-state gates to drive data from the I-cache Set 0/D-cache Tag RAM onto DATAP[31:0].

BZ_INDEXP[12:0]
Cache RAM Index **Output**
These signals are connected to the cache RAM address inputs.

$\overline{\text{BZ\_IP\_DN}}$ **Instruction/Data Cache Select** **Output**
The BBCC drives this signal HIGH to read/write from/to the I-cache, and drives this signal LOW to read/write from/to the D-cache.

$\overline{\text{BZ\_IP\_DN\_L}}$ **Instruction/Data Cache Select, Registered** **Output**
This output is an input to the tag match logic. This signal is the same as $\overline{\text{BZ\_IP\_DN}}$, but delayed by one clock cycle.

BZ_I1DCLKP **I-Cache Set 1 Data RAM Clock** **Output**
This output is connected to the I-cache Set 1 Data RAM clock input.

BZ_I1DOEP **I-Cache Set 1 Data RAM Output Enable** **Output**
The BBCC asserts this signal to enable the I-cache Set 1 Data RAM to drive data onto DATAP[31:0].

BZ_I1DWEP[3:0]
I-Cache Set 1 Data RAM Write Enable **Output**
The BBCC asserts this signal to enable writes to the I-cache Set 1 Data RAM.

BZ_I1TCLKP **I-Cache Set 1 Tag RAM Clock** **Output**
This output is the I-cache Set 1 Tag RAM clock input.

BZ_I1TWEP[4:0]
I-Cache Set 1 Tag RAM Write Enables **Output**
The BBCC asserts these signals to enable writes to the I-cache Set 1 Tag RAM. Figure 5.18, on page 5-48, shows the correspondence between byte enables and what is enabled.

$\overline{\textbf{BZ\_I1T\_OEN}}$ **I-Cache Set 1 Tag RAM Output Enable** **Output**
This output is an input to a set of external 3-state gates. The BBCC asserts this signal to enable the 3-state gates to drive data from the I-cache Set 1 Tag RAM onto DATAP[31:0].

BZ_LOCKP **Cache RAM Lock Bit** **Output**
This output is connected to the Lock Bit input to the I-Set 0/D Tag RAM. The BBCC asserts this signal to lock the I-Set 0 Tag RAM line.

BZ_TAGP[21:0]
Cache RAM Tag **Output**
These signals contain the tag which the BBCC writes to the cache Tag RAMs.

BZ_TAG4MATCHP[21:0]
Tag for Tag Match **Output**
This signal is an input to the tag match logic. These signals contain the tag which is compared against the tag in the Tag RAMs to determine if there is a cache hit or miss.

BZ_VALIDP[3:0]
Cache RAM Valid Bits **Output**
These signals are the Valid Bits which are written to the cache Tag RAMs.

CBYTEP[3:0] **CBus Byte Enables** **Input**
These signals are the byte enables from the CW400x. The CW400x asserts the byte enables HIGH to inform the BBCC that the corresponding bytes are valid on DATAP[31:0].

The following table shows the correspondence between byte enables and the data bus bytes.

Byte Enable	Corresponding DATAP[31:0] Byte
CBYTEP3	[31:24]
CBYTEP2	[23:16]
CBYTEP1	[15:8]
CBYTEP0	[7:0]

$\overline{\text{CIP\_DN}}$ **Instruction/Data Indicator** **Input**
The CW400x drives this signal HIGH to indicate that it is performing an instruction fetch. The CW400x drives this signal LOW to indicate that it is performing a data fetch or store.

CKILLMEMP **Kill Memory Transaction Request** **Input**
The CW400x asserts this signal to request that the BBCC kill the current CBus transaction. CKILLMEMP is valid only during run cycles.

CMEM_FETCHP
Fetch Indictor **Input**
The CW400x asserts this signal to inform the BBCC that the current CBus transaction is a fetch.

CSTOREP **Store Indictor** **Input**
The CW400x asserts this signal to indicate that the current CBus transaction is a store.

DATAP[31:0] **Data Bus** **Bidirectional**
These signals are the CBus data bus.

IDLCKP **Lock Bit** **Input**
This signal is the Lock Bit from the I-cache Set 0/D-cache Tag RAM.

IDMATCHP **Tag Match** **Input**
The I-Set 0/D Tag Match Logic asserts this signal to inform the BBCC that the tag from the I-cache Set 0/ D-cache Tag RAM matched the appropriate bits of BZ_TAG4MATCHP[21:0].

IDVLDP[3:0] **Valid Bits** **Input**
These signals are the Valid Bits from the I-cache Set 0/ D-cache Tag RAM.

I1MATCHP	**Tag Match** The I-Set 1 Tag Match Logic asserts this signal to inform the BBCC that the tag from the I-cache Set 1 Tag RAM matched the appropriate bits of BZ_TAG4MATCHP[21:0].	**Input**
I1VLDP[3:0]	**Valid Bits** These signals are the Valid Bits from the I-cache Set 1 Tag RAM.	**Input**
MADDROUTP[31:2]	**Mapped Address** These signals are the mapped CBus address from the MMU or MMU Stub.	**Input**
MEARLYKS1P	**MMU Early *kseg1* Indicator** The MMU or MMU Stub asserts this signal to inform the BBCC that the CBus unmapped (virtual) address is in *kseg1* (non-cacheable address space).	**Input**
MNOCACHEP	**Mapped Address Not Cacheable** The MMU or MMU Stub asserts this signal to inform the BBCC that the current CBus transaction is non-cacheable.	**Input**
PCLKP	**System Clock** This signal is the global clock input.	**Input**
SE	**Scan Enable** Asserting this signal enables the scan chain.	**Input**
SI	**Scan Data In** This signal is the scan data input.	**Input**
SO	**Scan Data Out** This signal is the scan data output.	**Output**
TST	**Test Enable** Asserting this signal puts the BBCC in Test Mode for scan.	**Input**
WB_ADDRP[31:2]	**Write Buffer Address** These signals are the address of the earliest store transaction held in the Write Buffer.	**Input**

WB_ARRIVEBFLDP

Write Buffer Store Arrived Before Load **Input**

The Write Buffer asserts this signal to inform the BBCC that the earliest store transaction held in the Write Buffer was started while the Data Fetch Queue was empty.

WB_BYTEP[3:0]

Write Buffer Byte Enables **Input**

The Write Buffer drives these signals, which are the byte enables of the earliest store transaction held in the Write Buffer.

WB_CFGP **Write Buffer Store to Configuration Register** **Input**

The Write Buffer asserts this signal to inform the BBCC that the earliest store transaction held in the Write Buffer is to the System Configuration Register.

WB_DATAP[31:0]

Write Buffer Data **Input**

These signals are the data of the earliest store transaction held in the Write Buffer.

WB_FULLP **Write Buffer Full** **Input**

The Write Buffer asserts this signal to inform the BBCC that the Write Buffer is full.

WB_STPNDP **Write Buffer Store Pending** **Input**

The Write Buffer asserts this signal to inform the BBCC that the Write Buffer contains a valid store transaction.

WB_VWBFLDP

Write Buffer Valid Write Before Load **Input**

The Write Buffer asserts this signal to inform the BBCC that the Write Buffer contains a valid store transaction that should have a higher priority than data fetch transactions.

5.5 Interfaces

This section describes the BBCC interfaces to the following:

- CBus
- Basic Bus (BBus)
- Caches
- On-Chip Memory (OCM)
- Write Buffer

5.5.1 CBus

5.5.1.1 CBus Transactions

This section describes the use of the CBus signals.

CBus transactions occur during *run cycles*. The Global Output Enable Module (GOE, see Chapter 6 of the *MiniRISC CW400x Microprocessor Core Technical Manual*) asserts $\overline{\text{BRUN\_INN}}$ to inform the BBCC that the following clock cycle will be a run cycle. When a clock cycle is not a run cycle ($\overline{\text{BRUN\_INN}}$ deasserted), the system stalls (a stall cycle).

The CW400x $\overline{\text{CIP\_DN}}$, CMEM_FETCHP, and CSTOREP signals are valid in the clock cycle before the run cycle, and indicate what type of transaction will occur during the run cycle. The signals are decoded as shown in Table 5.4.

Table 5.4
Transaction Type
Signal Decoding

$\overline{\text{BRUN\_INN}}$	$\overline{\text{CIP\_DN}}$	CMEM_FETCHP	CSTOREP	Transaction
1	-	-	-	No Operation
0	1	1	0	Instruction Fetch
0	0	1	0	Data Fetch
0	0	0	1	Store
0	0	0	0	Coprocessor

All other combinations do not occur, so designers can simplify the decoding to that in Table 5.5.

Table 5.5
Transaction Type Signal
Decoding Simplified

$\overline{\text{BRUN\_INN}}$	$\overline{\text{CIP\_DN}}$	CMEM_FETCHP	CSTOREP	Transaction
1	-	-	-	No Operation
0	1	-	-	Instruction Fetch
0	0	1	-	Data Fetch
0	-	-	1	Store
0	-	0	0	Coprocessor

The CW400x drives the lower bits of the unmapped address for CBus transactions onto ADDRP[14:2]. These signals are valid in the clock cycle before the run cycle.

MADDROUTP[31:2] are the mapped CBus transaction address from the MMU or MMU Stub. These signals are valid during the run cycle. The MMU or MMU Stub holds these signals stable until the next run cycle.

If ADDRP[31:29] = 101_2, the MMU or MMU Stub asserts MEARLYKS1P to inform the BBCC that the CBus unmapped (virtual) address is in *kseg1* (non-cacheable address space).

The MMU or MMU Stub assert MNOCACHEP to indicate that MADDROUTP[31:2] is a non-cacheable address, either because the address is in *kseg1* or because the MMU is setup to make the address non-cacheable.

The CW400x asserts CKILLMEMP to request that the BBCC kill the memory transaction. CKILLMEMP is valid during run cycles.

The CW400x drives CBYTEP[3:0] to indicate which bytes it is requesting to read or write. They are only valid for data transactions. CBYTEP[3:0] is not used for instruction fetches since instruction fetches are always word fetches. CBYTEP[3:0] holds its value during stall cycles.

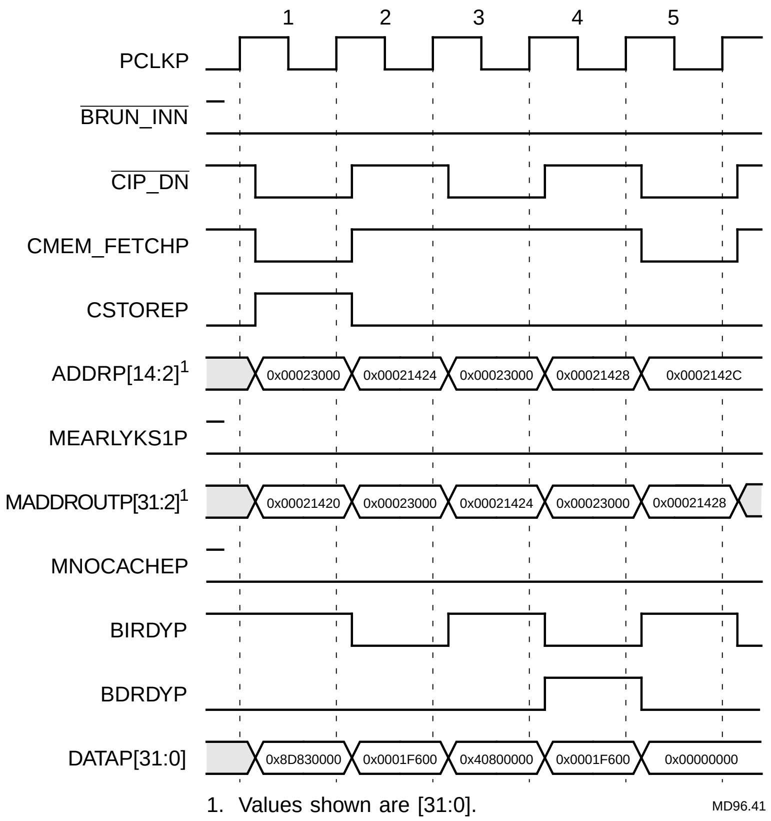
CBus fetch transactions are normally completed when the BBCC asserts BIRDYP (instruction ready) or BDRDYP (data ready) to the CW400x. At this time, the data for the fetch is on DATAP[31:0]. For instruction fetches, BIRDYP must be received before the CW400x pipeline can proceed. For data fetches, the CW400x pipeline can proceed without receiving a BDRDYP assertion, as long as no dependency occurs with the data fetch, and no other data fetch transaction needs to be started. Only one data fetch can be outstanding at any time. For stores, the CW400x pipeline can proceed unless the Write Buffer is full, in which case the BBCC stalls the CW400x pipeline.

The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ to indicate that the next clock cycle will be bus-stolen. A clock cycle is bus-stolen when the BBCC assumes control of DATAP[31:0] so it can place data on it for an instruction or data fetch. When the BBCC places the data for an instruction or data fetch on DATAP[31:0], it asserts BIRDYP or BDRDYP. For block fetches, the BBCC asserts $\overline{\text{BBUS\_STEALN}}$ for each word, in order to write it to the cache. When the data for a cache refill is also the data required by the

CW400x for an instruction or data fetch, the BBCC asserts BIRDYP or BDRDYP, and is said to be *streaming* the data to the CW400x.

Figure 5.4 shows examples of some CBus transactions.

Figure 5.4
CBus Transactions



Cycle 1: The CBus signals indicate that the next cycle will be a store to address 0x00023000. The data for the store (0x8D830000) is on DATAP[31:0].

Cycle 2: MADDROUTP[31:2] contains the mapped address (in this case it is the same as the unmapped address) for the store. The store is to a cacheable address, as shown by MNOCACHEP. The next cycle will be an instruction fetch from address 0x00021424.

Cycle 3: MADDROUTP[31:2] contains the mapped address for the instruction fetch. BIRDYP indicates that the data for the instruction fetch is on DATAP[31:0]. The next cycle will be a data fetch from address 0x00023000.

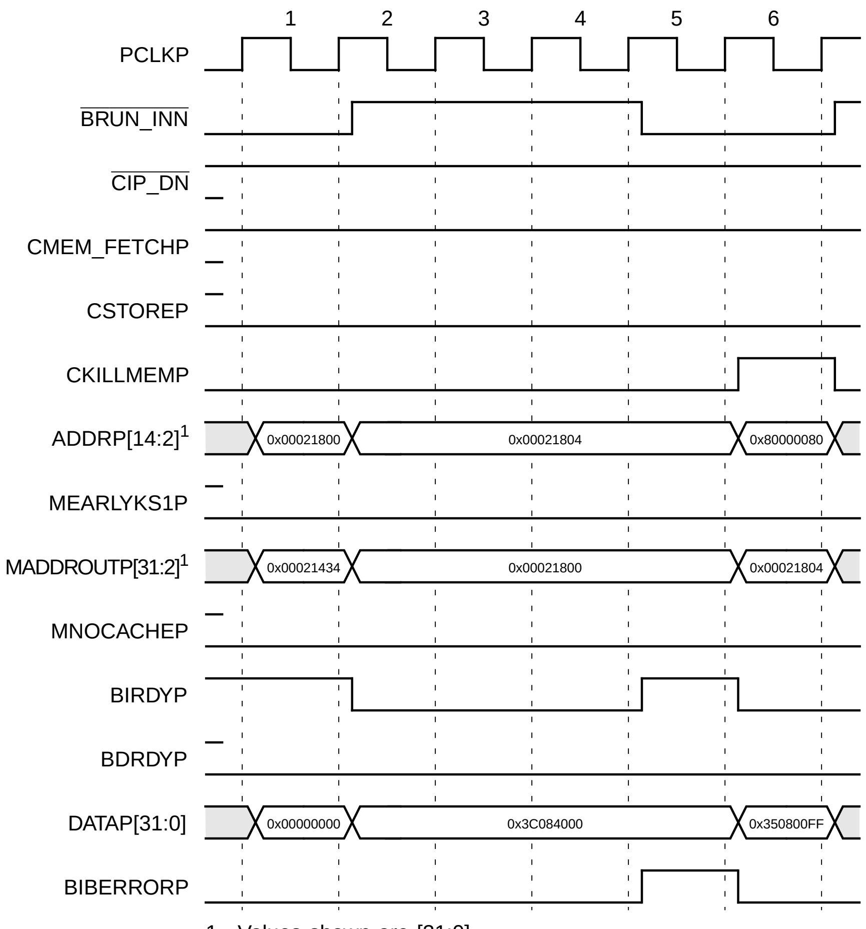
Cycle 4: MADDROUTP[31:2] contains the mapped address for the data fetch. BDRDYP indicates that the data for the data fetch is on DATAP[31:0]. The next cycle will be an instruction fetch from address 0x00021428.

Cycle 5: MADDROUTP[31:2] contains the mapped address for the instruction fetch. BIRDYP indicates that the data for the instruction fetch is on DATAP[31:0]. The next cycle will be a move-from-coprocessor or a move-to-coprocessor.

5.5.1.2 Bus Error During an Instruction Fetch

When a transaction terminates unsuccessfully, the BBCC reports a bus error to the CW400x. Instruction fetch bus errors are handled differently than data transaction bus errors. On an instruction fetch, the CW400x must wait for BIRDYP to be asserted before the CW400x pipeline can proceed. Therefore, bus errors that are signaled for instruction fetches occur during the instruction fetch. The BBCC asserts BIBERRORP and BIRDYP to signal a bus error to the CW400x. Figure 5.5 shows a Bus Error during an Instruction Fetch.

Figure 5.5
Bus Error During
Instruction Fetch



1. Values shown are [31:0].

Cycle 1: CBus signals indicate that the next cycle will be an instruction fetch from address 0x00021800.

Cycle 2: MADDROUTP[31:2] contains the mapped address for the instruction fetch. The GOE deasserts $\overline{\text{BRUN\_INN}}$ to indicate that the CW400x will stall in the next clock cycle.

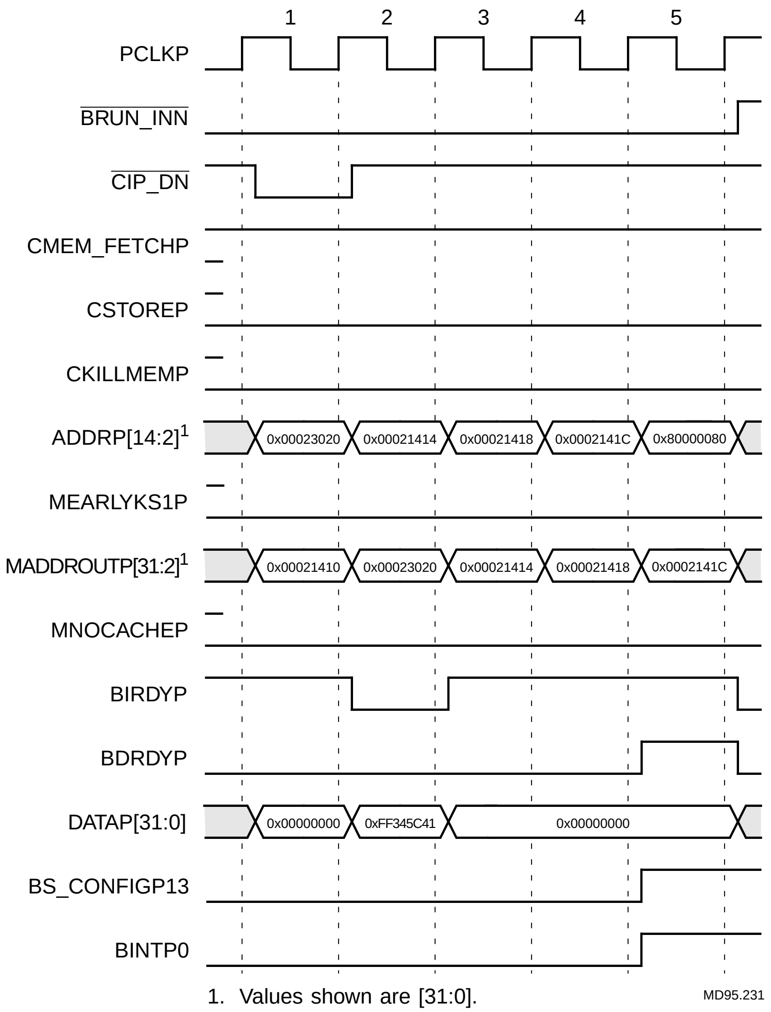
Cycle 5: BIRDYP and BIBERRORP are asserted, signaling a bus error. The next cycle will be an instruction fetch from address 0x00021804.

Cycle 6: The instruction fetch from address 0x00021804 is killed by the assertion of CKILLMEMP. The next cycle will be an instruction fetch from the exception handler (address 0x80000080).

5.5.1.3 Bus Error During a Data Transaction

For data transactions, bus errors might not occur during the data transaction, since the pipeline can proceed before the data transaction is complete. In this case, the BBCC signals a bus error by setting System Configuration Register Bit 13 (E Bit). This bit is reset by writing a zero to it. This signal can be used to detect the bus error by tying it to one of the CW400x interrupts. The CW400x CP0 Status Register should be setup to detect this interrupt, or the bus error will not be detected. Figure 5.6 shows a Bus Error during a Data Transaction.

Figure 5.6
Bus Error During
Data Transaction



Cycle 1:	The CBus signals indicate that the next cycle will be a data fetch from address 0x00023020.
Cycle 2-4:	The CW400x continues to fetch instructions.
Cycle 5:	The BBCC asserts BDRDYP. The BBCC asserts BS_CONFIGP13, which is tied to Interrupt 0 of the CW400x, and hold it. The next cycle will be an instruction fetch from the exception handler (address 0x80000080).

For more details about the CBus Interface, see the *MiniRISC CW400x Microprocessor Core Technical Manual*.

5.5.2 Basic Bus (BBus)

The BBus has a simple, generic bus protocol to interface to BBus devices. The BBCC contains two sets of inputs/outputs to interface to the BBus, one set for when the BBCC is the master on the BBus, the other for when the BBCC is a slave on the BBus.

5.5.2.1 Single Transactions

This section describes the use of the main signals on the BBus for simple transactions with the BBCC as a bus master.

The BBCC asserts $\overline{\text{BSTARTNO}}$ at the beginning of transactions for one clock cycle, then deasserts it until the beginning of a new transaction. The BBCC asserts $\overline{\text{BTXNO}}$ at the beginning of a transactions, and continues to assert it for the duration of the transaction. $\overline{\text{BTXNO}}$ might stay asserted between back-to-back BBus transactions.

The BBCC asserts $\overline{\text{BWRNO}}$ to indicate the transaction is a write. It deasserts this signal to indicate the transaction is a read. The BBCC holds this signal stable for the duration of the transaction. The BBCC drives $\overline{\text{BIP\_DNO}}$ LOW to indicate the transaction is a data transaction. It drives this signal HIGH to indicate the transaction is an instruction fetch. The BBCC also holds this signal stable for the duration of the transaction.

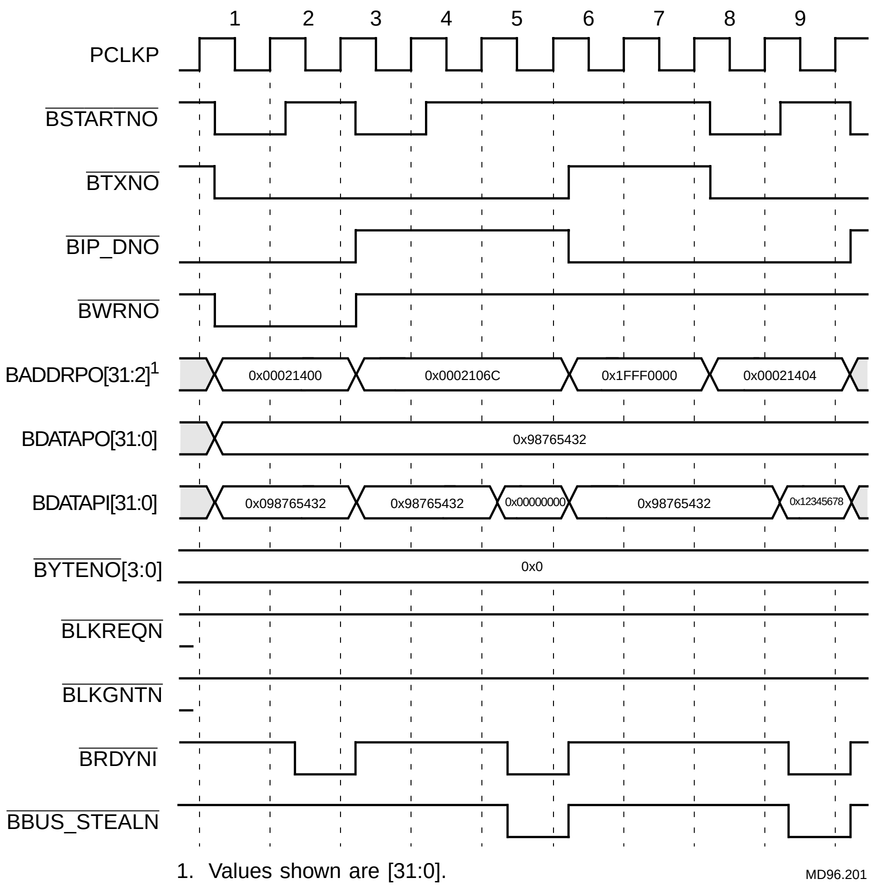
The BBCC puts the transaction address on BADDRPO[31:2] and the data to be stored on BDATAPO[31:0]. The BBus device drives data on BDATAPI[31:0] for instruction and data fetches. The BBCC also drives $\overline{\text{BYTENO}}$[3:0] to indicate which bytes are being fetched or stored. The BBCC asserts all the byte enables for instruction fetches and cacheable data fetches; otherwise it asserts only the byte enables for the bytes which the CW400x requested to read or write.

The BBus device asserts $\overline{\text{BRDYNI}}$ when it places data on BDATAPI[31:0] (for fetches) or when the BBus device has received the data from BDATAPO[31:0] (for stores). The BBCC supports one-cycle transactions, so $\overline{\text{BRDYNI}}$ can be asserted in the first clock cycle (the same clock cycle that $\overline{\text{BSTARTNO}}$ is asserted).

The BBus device asserts $\overline{\text{BERRORN}}$ when the transaction terminates in an error. The BBus device asserts $\overline{\text{BRETRYN}}$ to request that the BBCC do the transaction over again.

Figure 5.7 shows some BBus transactions.

Figure 5.7
BBus Transactions



1. Values shown are [31:0].

Cycle 1: The BBCC asserts $\overline{\text{BSTARTNO}}$ and $\overline{\text{BTXNO}}$ LOW to indicate the beginning of a BBus transaction, $\overline{\text{BIP\_DNO}}$ LOW to indicate that it is a data transaction, and $\overline{\text{BWRNO}}$ LOW to indicate that it is a write transaction. BADDRPO[31:2] shows that the transaction address is 0x00021400. Since this is a store operation, BDATAPO[31:0] contains the data to be stored. $\overline{\text{BYTENO}}$[3:0] indicates that all four bytes will be written.

Cycle 2: The BBCC deasserts $\overline{\text{BSTARTNO}}$. The BBus device asserts $\overline{\text{BRDYNI}}$ to indicate that it has received the data from BDATAPO[31:0]. The transaction is terminated at the end of this clock cycle, when the BBCC detects the $\overline{\text{BRDYNI}}$ assertion.

Cycle 3: The previous BBCC transaction is finished, and the BBCC asserts $\overline{\text{BSTARTNO}}$ to start a new transaction. This transaction is a back-to-back with the previous store, so the $\overline{\text{BTXNO}}$ stays asserted. $\overline{\text{BIP\_DNO}}$ and $\overline{\text{BWRNO}}$ indicate that this is an instruction fetch from address 0x0002106C.

Cycle 4: The BBCC deasserts $\overline{\text{BSTARTNO}}$, and waits for the assertion of $\overline{\text{BRDYNI}}$.

Cycle 5: The BBus device asserts $\overline{\text{BRDYNI}}$, and drives the data on BDATAPI[31:0]. The transaction is terminated at the end of this clock cycle, at which time the BBCC registers the data from BDATAPI[31:0]. It asserts $\overline{\text{BBUS\_STEALN}}$ so that it can place the data on DATAP[31:0] in the next clock cycle.

Cycle 8: The BBCC starts a data fetch transaction from address 0x00021404.

Cycle 9: The BBus device asserts $\overline{\text{BRDYNI}}$ and places the data on BDATAPI[31:0]. The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ so it can put the data on DATAP[31:0] in the next clock cycle.

5.5.2.2 Block Fetching

The BBCC executes a block fetch transaction to refill the caches. The transaction must be a cacheable transaction, and the System Configuration Register must be written so that the block size is more than one word. In the System Configuration Register, the block size can be set to 1, 2, 4, or 8 words.

For a four-word block size, the BBCC stops the refill once the line has all the words valid. For example, if Words 2 and 3 of a line are valid, and a cache miss occurs on Word 0, the BBCC does a block fetch for only

Words 0 and 1. For block sizes of 1, 2, or 8, the BBCC tries to fetch 1, 2, or 8 words, respectively.

The BBCC also stops block fetches if a subsequent store occurs within the same eight-word block as a block fetch. Stopping block fetches prevents a block fetch from overwriting a later value in the cache. The BBCC stops block fetches if a new store operation address matches the refill address in Bits [9:5]. In this case, the BBCC continues to expect the currently requested word, but does not write that word to the cache when it is received.

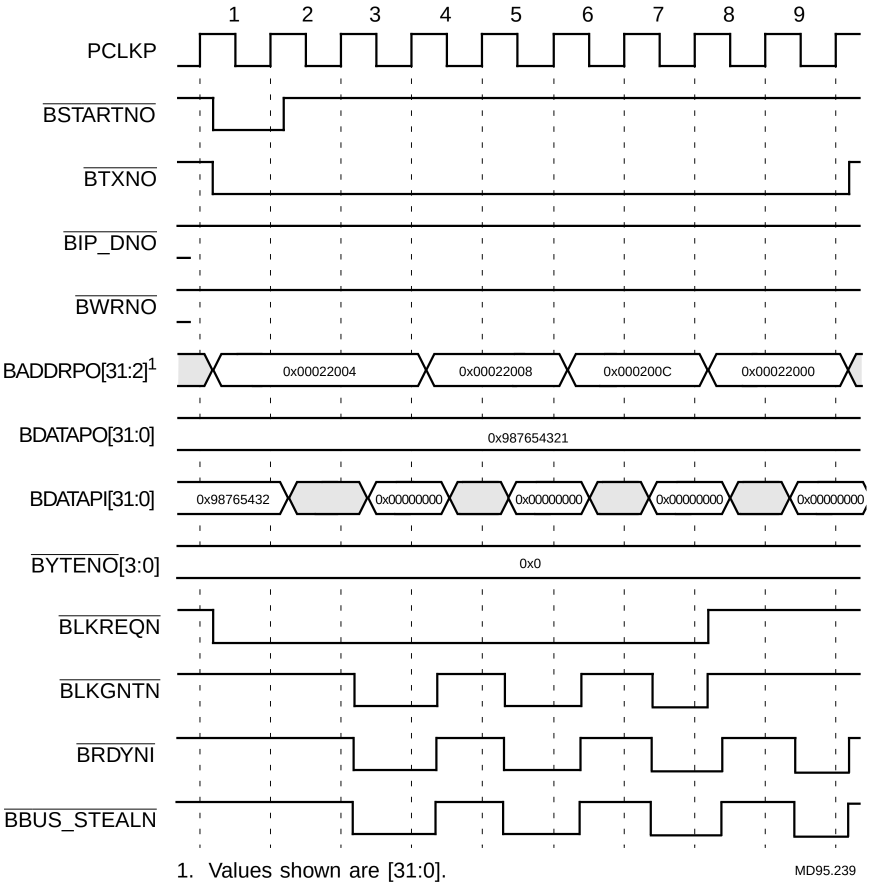
The BBCC starts block fetches with the word which was fetched by the CW400x. For example, if the block size is eight words, and the fetch was for 0x0002101C, the order of the block fetch is: 0x0002101C, 0x00021000, 0x00021004, 0x00021008, 0x0002100C, 0x00021010, 0x00021014, 0x00021018.

When the BBCC is able to do a block fetch, it asserts $\overline{\text{BLKREQN}}$ with the start signal, $\overline{\text{BSTARTNO}}$. The BBus device asserts $\overline{\text{BLKGNTN}}$ to acknowledge the request for a block fetch. If the BBus device does not assert $\overline{\text{BLKGNTN}}$, the block transfer ends. $\overline{\text{BLKGNTN}}$ is sampled by the BBCC when $\overline{\text{BRDYNI}}$ is asserted.

If the BBus device asserts $\overline{\text{BERRORN}}$ or $\overline{\text{BRETRYN}}$, the BBCC ends the block fetch. The BBCC signals a bus error to the CW400x if the BBus device asserts $\overline{\text{BERRORN}}$ while the BBCC is fetching the first word of the block fetch. If the BBus device asserts $\overline{\text{BERRORN}}$ while the BBCC is fetching a subsequent word of the block fetch, the BBCC ends the block fetch, but does not report the bus error to the CW400x. The BBCC retries a transaction if the BBus device asserts $\overline{\text{BRETRYN}}$ while the BBCC is fetching the first word of the block fetch. If the BBus device asserts $\overline{\text{BRETRYN}}$ while the BBCC is fetching a subsequent word of the block fetch, the BBCC ends the block fetch, but does not retry the transaction.

Figure 5.8 shows an example of a block fetch with a four-word block size.

Figure 5.8
Block Fetch with Four-Word Block Size



Cycle 1: The BBCC asserts the $\overline{\text{BSTARTNO}}$ and $\overline{\text{BTXNO}}$ signals to start an instruction fetch from address 0x00022004 (BADDRPO[31:0]). The BBCC asserts $\overline{\text{BLKREQN}}$ to indicated that it is able to perform a block fetch.

Cycle 3: The BBus device asserts $\overline{\text{BRDYNI}}$ to indicate that it has placed the data on BDATAPI[31:0]. It also asserts $\overline{\text{BLKGNTN}}$ to indicate that it can perform a block fetch.

Cycle 4: The BBCC places the address 0x00022008 on BADDRPO[31:2] for the next fetch of the block fetch.

Cycle 5: The BBus device asserts $\overline{\text{BRDYNI}}$ and $\overline{\text{BLKGNTN}}$ to continue the block fetch.

Cycle 6:	The BBCC places the address 0x0002200C on BADDRPO[31:2] for the next fetch of the block fetch.
Cycle 7:	The BBus device asserts $\overline{\text{BRDYNI}}$ and $\overline{\text{BLKGNTN}}$ to continue the block fetch.
Cycle 8:	The BBCC places the address 0x00022000 on BADDRPO[31:2] for the next fetch of the block fetch. Since the block size is four, the address wrapped to the first word of the line. The BBCC deasserts $\overline{\text{BLKREQN}}$ to indicate that this is the last word of the block fetch.
Cycle 9:	The BBus device asserts $\overline{\text{BRDYNI}}$ for the last fetch.

5.5.2.3 Burst Writes

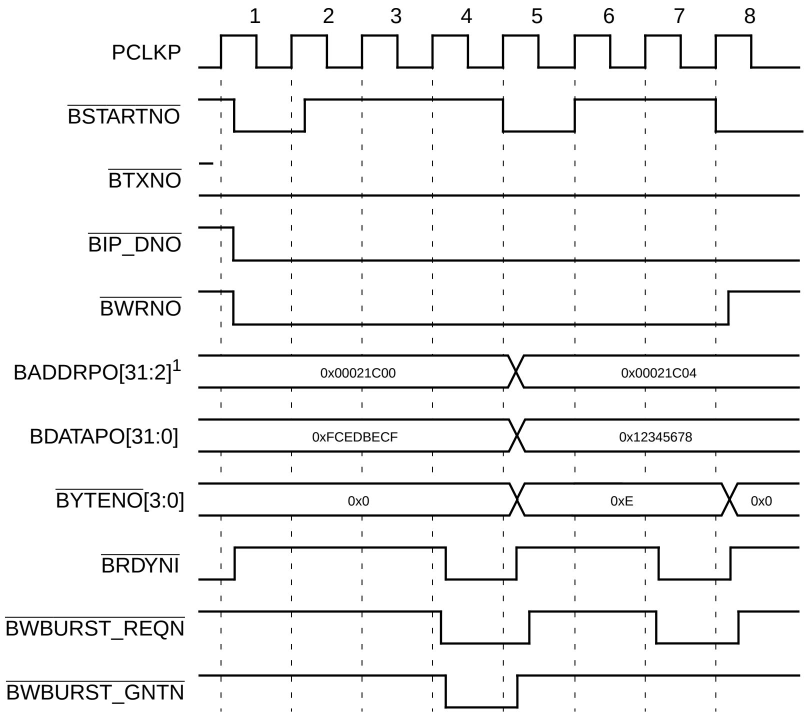
Burst writes are writes that occur back-to-back to the same page. They are different from block fetches in that each burst write transaction is separate. The BBCC still asserts $\overline{\text{BSTARTNO}}$ at the beginning of the subsequent writes. The BBCC performs burst writes when:

- ♦ a BBus device requests it by asserting $\overline{\text{BWBURST\_REQN}}$,
- ♦ the BBCC is currently doing a store transaction,
- ♦ and the following data transaction is a store residing in the Write Buffer which stores to the same page (page size is defined in the System Configuration Register) as the current store.

When these conditions are met, the BBCC asserts $\overline{\text{BWBURST\_GNTN}}$ to indicate that it will perform a burst write. Although the $\overline{\text{BWBURST\_REQN}}$ is sampled at all times, and the BBCC asserts/deasserts $\overline{\text{BWBURST\_GNTN}}$ based on the state of $\overline{\text{BWBURST\_REQN}}$, $\overline{\text{BWBURST\_REQN}}$ is only meaningful when the $\overline{\text{BRDYNI}}$ is asserted by the BBus device, because that is the clock cycle in which the next BBus transaction is determined.

Figure 5.9 shows a series of burst writes.

Figure 5.9
Series of Burst Writes



1. Values shown are [31:0].

Cycle 1: The BBCC starts a word store to address 0x00021C00 (BADDRPO[31:0]).

Cycle 4: The BBus device asserts $\overline{\text{BRDYNI}}$ to indicate that it has received the data from BDATAPO[31:0]. It also asserts $\overline{\text{BWBURST\_REQN}}$ to request a burst write. The BBCC responds by asserting $\overline{\text{BWBURST\_GNTN}}$, indicating that the next transaction will be a store to the same page.

Cycle 5: The BBCC starts a new store to the same page. The address in this case is 0x00021C04 (BADDRPO[31:0]). $\overline{\text{BYTENO}}$[3:0] indicates that only Byte 0 will be stored.

Cycle 7: The BBus device again asserts $\overline{\text{BRDYNI}}$ and $\overline{\text{BWBURST\_REQN}}$. However, the BBCC does not assert $\overline{\text{BWBURST\_GNTN}}$ in response.

5.5.2.4 Bus Mastership

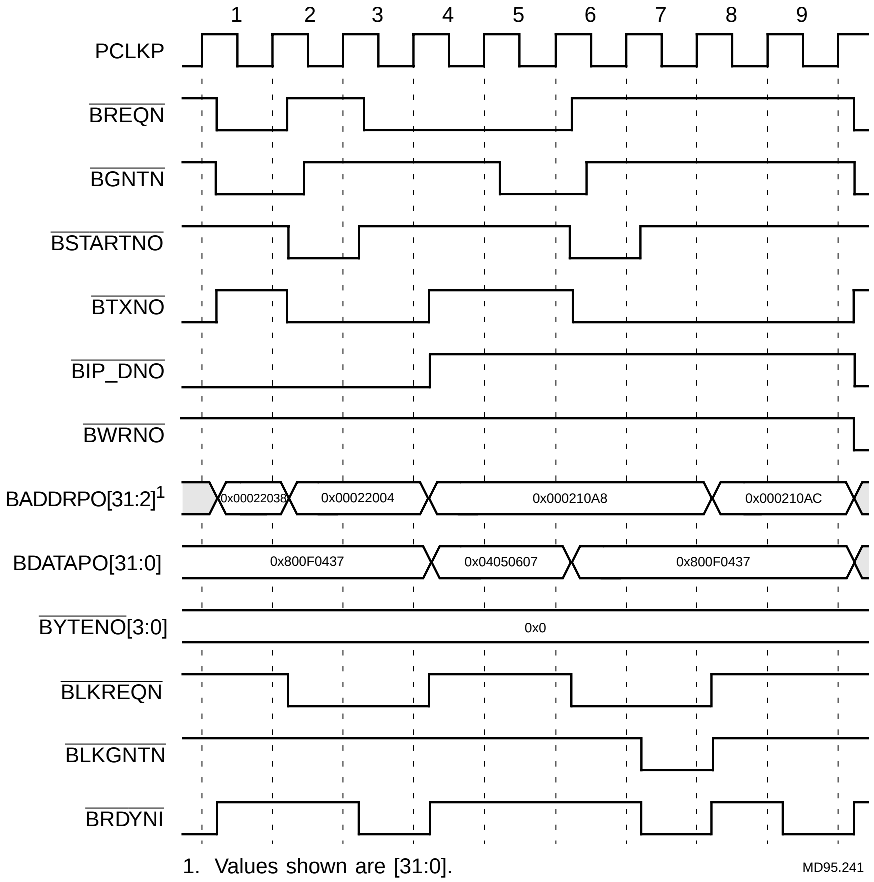
The BBus can have multiple bus masters, with only one bus master taking control of the bus at a time. The arbitration of bus ownership can occur while a transaction is in progress, because the transfer of ownership from one master to another does not affect the transaction in progress. Bus masters should not begin transactions until $\overline{\text{BTXNO}}$ is deasserted, so there are no conflicts on the bus.

$\overline{\text{BREQN}}$ and $\overline{\text{BGNTN}}$ control ownership of the bus. The BBCC asserts $\overline{\text{BREQN}}$ when it does not have ownership of the bus, but needs control of the bus. The BBCC also asserts $\overline{\text{BREQN}}$ any time that it has two or more transactions queued. When $\overline{\text{BGNTN}}$ is asserted, the BBCC has ownership of the bus, and can start a BBus transaction after $\overline{\text{BTXNI}}$ has been deasserted.

When the BBCC requests the BBus, the BBCC normally starts a transaction when $\overline{\text{BGNTN}}$ is asserted and $\overline{\text{BTXNI}}$ is deasserted. An exception is when the bus request is for the System Configuration Register. In this case, the BBCC performs an internal transaction which is not visible on the BBus. The BBCC deasserts $\overline{\text{BREQN}}$ when the System Configuration Register transaction is completed.

Figure 5.10 shows examples of bus arbitration.

Figure 5.10 Examples of Bus Arbitration



Cycle 1: The BBCC asserts $\overline{\text{BREQN}}$ to indicate that it needs the BBus. The external arbiter asserts $\overline{\text{BGNTN}}$ in reply to give ownership of the bus to the BBCC.

Cycle 2: The BBCC detects $\overline{\text{BGNTN}}$ asserted and begins a data fetch BBus transaction. The BBCC asserts $\overline{\text{BLKREQN}}$ to indicate a block fetch. Because there was only one transaction queued, the BBCC deasserts $\overline{\text{BREQN}}$.

Cycle 3: The BBus device asserts $\overline{\text{BRDYNI}}$, but not $\overline{\text{BLKGNTN}}$. The transaction is complete when the BBCC detects the $\overline{\text{BRDYNI}}$ assertion at the end of this clock cycle. Another transaction was queued, so the BBCC asserts $\overline{\text{BREQN}}$ again.

Cycle 4: $\overline{\text{BGNTN}}$ is not asserted, so no transaction occurs.

Cycle 5: The external arbiter asserts $\overline{\text{BGNTN}}$, giving ownership of the bus to the BBCC.

Cycle 6: The BBCC begins an instruction fetch transaction, requesting a block fetch. It deasserts the $\overline{\text{BREQN}}$, since it no longer needs the bus.

Cycle 7: The BBus device asserts $\overline{\text{BRDYNI}}$ and $\overline{\text{BLKGNTN}}$. The transaction continues.

Cycle 8: The BBCC deasserts $\overline{\text{BLKREQN}}$, indicating that this is the last fetch of the block fetch.

Cycle 9: The BBus device asserts $\overline{\text{BRDYNI}}$, and the transaction completes at the end of this clock cycle.

5.5.2.5 Hardware Cache Test

Hardware Cache Test Mode allows the caches to be accessed through the BBus. When the BBCC is not the master on the BBus, and BCACHE_SELP is asserted, the BBCC is in Cache Test Mode. BCACHE_SELP acts as a select to the BBCC. When BCACHE_SELP is asserted and a transaction is started on the BBus, the BBCC assumes it is the target of the transaction; when BCACHE_SELP is not asserted, the BBCC assumes that the transaction is to another device on the BBus.

Hardware Cache Test Mode transactions are all two-cycle transactions. The cache transaction is determined by the inputs:

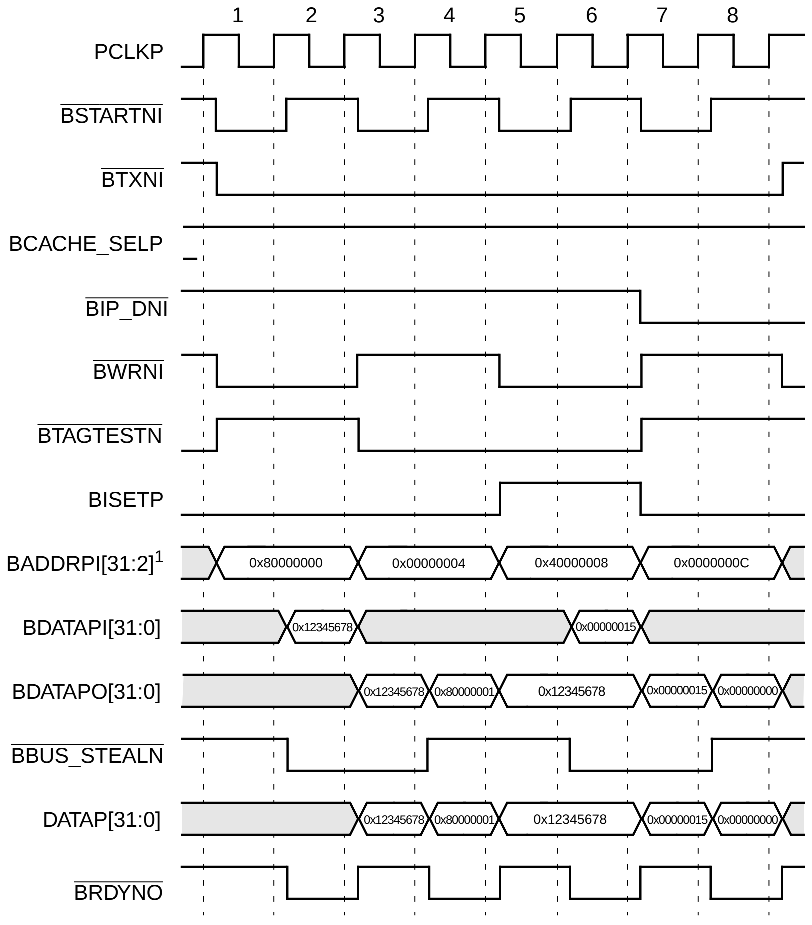
- $\overline{\text{BIP\_DNI}}$ (instruction or data cache)
- $\overline{\text{BWRNI}}$ (read or write transaction)
- $\overline{\text{BTAGTESTN}}$ (tag portion or data portion of cache)
- BISETP (which instruction set)

For hardware test reads, the BBCC asserts $\overline{\text{BBUS\_STEALN}}$ in the first clock cycle, to perform the read from the designated RAM at the end of the first clock cycle. In the second clock cycle, the data is available from the RAM, and is returned on BDATAPO[31:0]. For hardware test writes, the BBCC asserts $\overline{\text{BBUS\_STEALN}}$ in the second clock cycle, and performs the write at the end of the second clock cycle.

Reads from the data portion of the caches are straightforward; the data from the cache is simply returned on BDATAPO[31:0]. Write transactions to the data portion of the cache cause the BBCC to do a *write-first* type transaction. The BBCC writes the data and tag, sets the corresponding Valid Bit for the word, and clears the other Valid Bits in the line. Reads from the tag portion of the cache have the same format as in Software Cache Test Mode; the tag is in the upper portion of the bus, and the Lock and Valid Bits are in the lower portion of the bus. Writes to the tag portion of the cache are similar to software test writes; the tag portion of BADDRPI[31:2] is written as the tag, and the lower five bits of BDATAPI[31:0] are used as the Lock and Valid Bits.

Figure 5.11 shows some hardware cache test transactions.

Figure 5.11 Hardware Cache Test Transactions



1. Values shown are [31:0].

MD95.242

Cycle 1: The BBus device initiates a transaction to the BBCC (BCACHE_SELP asserted). The transaction is a write to the Data RAM of I-cache Set 0 at address 0x80000000. This corresponds to Line 0 of the cache Tag RAM, and Word 0 of the Cache Data RAM.

Cycle 2: The data for the write should be on BDATAPI[31:0] during this clock cycle. The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ to steal DATAP[31:0] for the write, and asserts $\overline{\text{BRDYNO}}$ to signal the transaction is complete.

Cycle 3: DATAP[31:0] is driven with the data to be written to the RAM. The write is performed at the beginning of the clock cycle. A new hardware cache test transaction is started. This time it is a read from the Tag RAM of the I-cache Set 0, at address 0x00000004. This corresponds to Line 0 of the cache Tag RAM. The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ to perform the read from the RAM.

Cycle 4: The BBCC puts the data from the read onto BDATAPO[31:0]. The BBCC asserts $\overline{\text{BRDYNO}}$. The data reflects what was written in the Tag RAM from the write to the Data RAM, which uses a write-first type of transaction. The upper bits contain the tag from the write address, and the appropriate Valid Bit is set.

Cycle 5: The BBus device starts a write to the Tag RAM of the I-cache Set 1. The address 0x40000008 corresponds to Line 0 of the Tag RAM.

Cycle 6: The data written to the Tag RAM is the tag portion of BADDRPI[31:2] and the lower five bits of BDATAPI[31:0]. In this case, the Lock Bit and two of the Valid Bits are set. The BBCC asserts $\overline{\text{BRDYNO}}$.

Cycle 7: The BBus device starts a read from the Data RAM of the D-cache.

Cycle 8: The BBCC returns data on BDATAPO[31:0], and asserts $\overline{\text{BRDYNO}}$.

5.5.2.6 Snooping

Snooping occurs when:

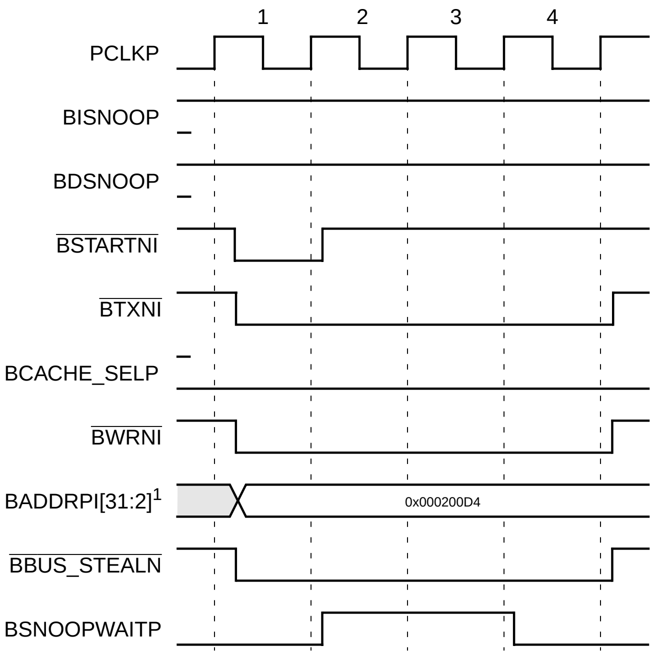
- ♦ BISNOOP and/or BDSNOOP signal are asserted,
- ♦ the BBCC is not a master on the BBus,

- BCACHE_SELP is not asserted,
- and a BBus write transaction occurs.

When these conditions are met, the BBCC does a tag comparison on the appropriate line(s) in the caches, and if the tags match, it invalidates the cache line(s). Snooping the I-cache requires two clock cycles (for either direct-mapped or two-way set associative) and snooping the D-cache requires two clock cycles. If both I-cache and D-cache snooping are enabled (BISNOOP and BDSNOOP set), then snooping requires four clock cycles. To help assure that the BBus device does not perform write transactions more frequently than it is possible to snoop on, the BBCC asserts BSNOOPWAITP when both I-cache and D-cache snooping are enabled.

Figure 5.12 shows I-cache and D-cache snooping.

Figure 5.12
I-Cache and
D-Cache Snooping



1. Values shown are [31:0].

Cycle 1: The BBus device begins a write transaction. The BBus device does not assert BCACHE_SELP, and BISNOOP and BDSNOOP are asserted, so the BBCC snoops on address 0x000200D4 (BADDRPI[31:2]). The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ to read the appropriate tag from the D-cache.

When both D-cache snooping and I-cache snooping are enabled, the BBCC performs the D-cache snooping first.

Cycle 2: The tag match logic performs the D-cache tag comparison in this clock cycle. If the tag in the Tag RAM matched the BADDRPI Tag, the BBCC turns off (clears) the Valid Bits for that line. The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ so that it can perform the Tag RAM write, if necessary. The BBCC asserts BSNOOPWAITP, indicating that it will perform both D-cache and I-cache snooping.

Cycle 3: The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ so it can read the appropriate tag from the I-cache (both tags if two-way set associative I-cache).

Cycle 4: The tag comparison is done in this clock cycle. The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ so that it can write the Valid Bits if necessary. The BBCC deasserts BSNOOPWAITP since this is the last clock cycle.

5.5.2.7 Bidirectional BBus

The BBus inputs and outputs can be made into a bidirectional bus by adding 3-state gates, and using the BMCNTLOEP, BDOEP, and BSCNTLOEP BBCC outputs. When BMCNTLOEP is asserted, $\overline{\text{BSTARTNO}}$, $\overline{\text{BTXNO}}$, $\overline{\text{BWRNO}}$, $\overline{\text{BIP\_DNO}}$, and BADDRPO[31:2] from the BBCC should be driven onto their 3-state buses. When BDOEP is asserted, BDATAPO[31:0] from the BBCC should be driven onto its 3-state bus. When BSCNTLOEP is asserted, the $\overline{\text{BRDYNO}}$ from the BBCC should be driven onto its 3-state bus.

For all 3-state buses, it is important that one and only one driver be driving the bus at all times. This involves assuring that there is a default driver when no driver needs to drive the bus, and that there is a resolution to having multiple drivers trying to drive the bus at the same time.

The following example shows how to create a default driver module for the BBus Data Bus. If the BBus Data Bus is connected to the BBCC and two other devices (Device X and Device Z, for example), the BBCC can be made the default driver by implementing Table 5.6 in logic.

Table 5.6
Example Data Output Enable Logic

Inputs			Outputs		
BDOEP	XDOEP	ZDOEP	BBCC_DOEP	DEVX_DOEP	DEVZ_DOEP
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	0	0
1	1	0	1	0	0
1	1	1	1	0	0

BDOEP **BBCC BBus Data Output Enable Request** **Input**
The BBCC asserts this signal to inform the BBus output enable logic that the BBCC wants to place its data (BDATAPO[31:0]) on the 3-state bus.

XDOEP **Device X BBus Data Output Enable Request** **Input**
Device X asserts this signal to inform the BBus output enable logic that Device X wants to place its data (XDATAPO[31:0]) on the 3-state bus.

ZDOEP **Device Z BBus Data Output Enable Request** **Input**
Device Z asserts this signal to inform the BBus output enable logic that Device Z wants to place its data (ZDATAPO[31:0]) on the 3-state bus.

BBCC_DOEP **BBCC Data Output Enable** **Output**
The BBus output enable logic asserts this signal to cause the set of 3-state gates to place BDATAPO[31:0] on the 3-state bus.

DEVX_DOEP **Device X Data Output Enable** **Output**
The BBus output enable logic asserts this signal to cause the set of 3-state gates to place XDATAPO[31:0] on the 3-state bus.

DEVZ_DOEP **Device Z Data Output Enable** **Output**
The BBus output enable logic asserts this signal to cause the set of 3-state gates to place ZDATAPO[31:0] on the 3-state bus.

Figure 5.13 shows the gates for Table 5.6 logic. The buffer gates are added to make the delay through the module approximately equal for all inputs to outputs. The equal delay causes the switching of the 3-state gates to occur nearly simultaneously. The duplicate outputs are included because the data bus is 32-bits wide, and having multiple drivers improves the driving of the 32 3-state gates (each output can drive 16 3-state gates). Figure 5.14 shows how the default driver logic should be hooked up in the system.

Figure 5.13
Default Driver
Logic

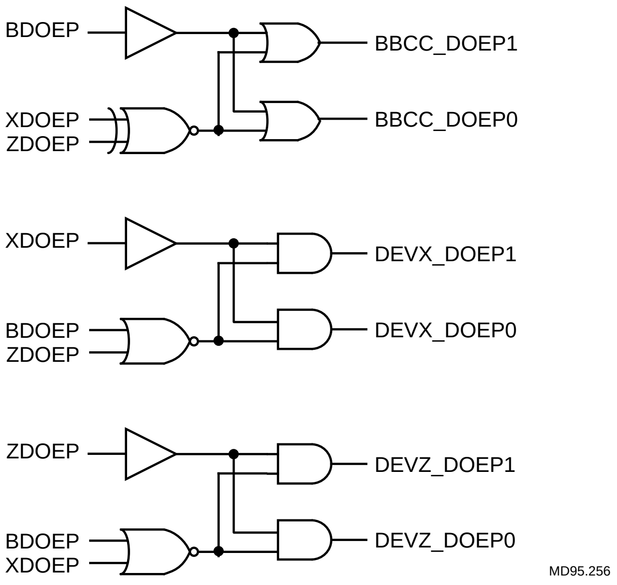
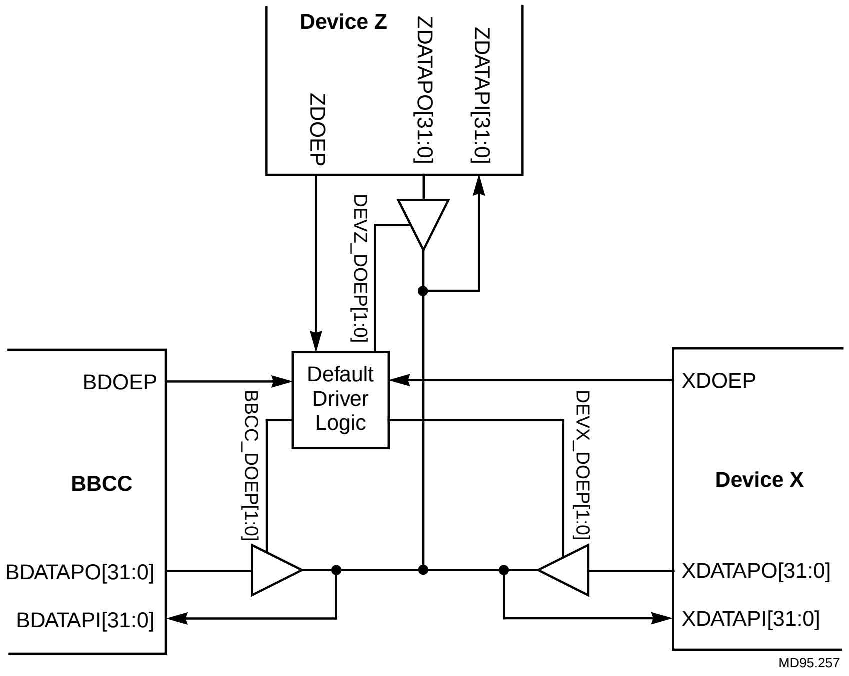


Figure 5.14
Default Driver
Logic in System



5.5.3 Caches

The BBCC interfaces to either two or four cache RAMs. A system with a two-way set associative I-cache requires four RAMs. Otherwise, only two RAMs are required. All of the signals to the RAMs (clock, write enables, output enables, data) come from the BBCC except for DATAP[31:0]. The outputs of the RAMs go to DATAP[31:0], the BBCC, and to logic that determines if there was a tag match.

Because the caches use the unmapped address to access the cache RAMs, if an MMU is in the system, the size of the caches must be smaller than the MMU page size. If the MMU Stub is installed, rather than the MMU, this limitation does not apply.

The Instruction address space and Data address space are assumed to be nonoverlapping. When the CW400x performs a store to an address, no checking is performed to determine if that address is contained in the I-cache.

Figures 5.15 through 5.18 show the contents of each RAM for a 1 Kbyte cache, and which portions of the RAM each write enable controls.

Figure 5.15
I-Cache Set 0/D-Cache
Data RAM

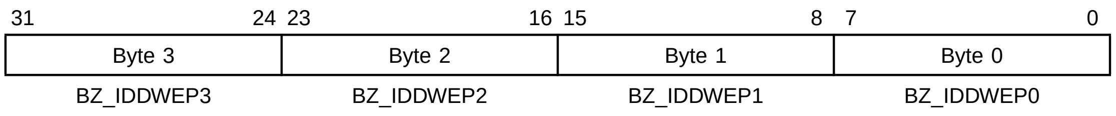


Figure 5.16
I-Cache Set 0/D-Cache
Tag RAM

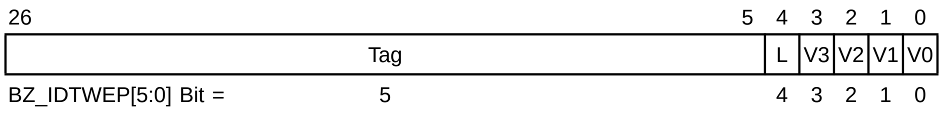


Figure 5.17
I-Cache Set 1 Data
RAM

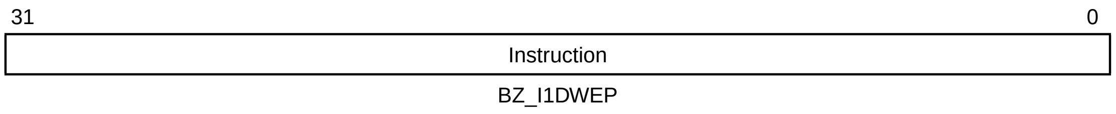
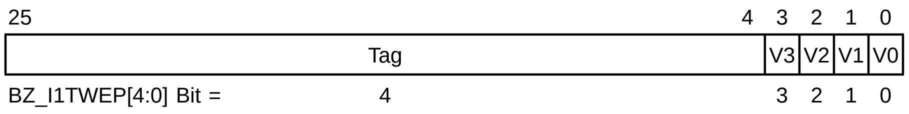


Figure 5.18
I-Cache Set 1 Tag RAM



There are two operating modes for the CW400x interface to the caches: Normal and Software Cache Test. The System Configuration Register determines in which mode the BBCC operates.

There are three modes for the BBus Interfaces to the caches: Normal, Hardware Test, and Snooping. These modes are determined by the transactions on the BBus.

5.5.3.1 Normal Instruction Cache Transactions

When an instruction fetch is initiated by the CW400x, the BBCC does a tag lookup in the first non-bus-stolen cycle. In this cycle, the BBCC looks at whether there was a tag match (IDMATCHP and I1MATCHP) and the

Valid Bits (IDVLDP[3:0] and I1VLDP[3:0]) to determine whether the instruction is in the I-cache. In the same cycle, the I-cache places its data onto DATAP[31:0]. If the instruction is in the I-cache (a cache hit), the BBCC asserts the BIRDYP to the CW400x, and the CW400x uses the data on DATAP[31:0]. If the instruction is not in the I-cache, the BBCC does not assert BIRDYP, and the CW400x stalls until the instruction is fetched from the external memory.

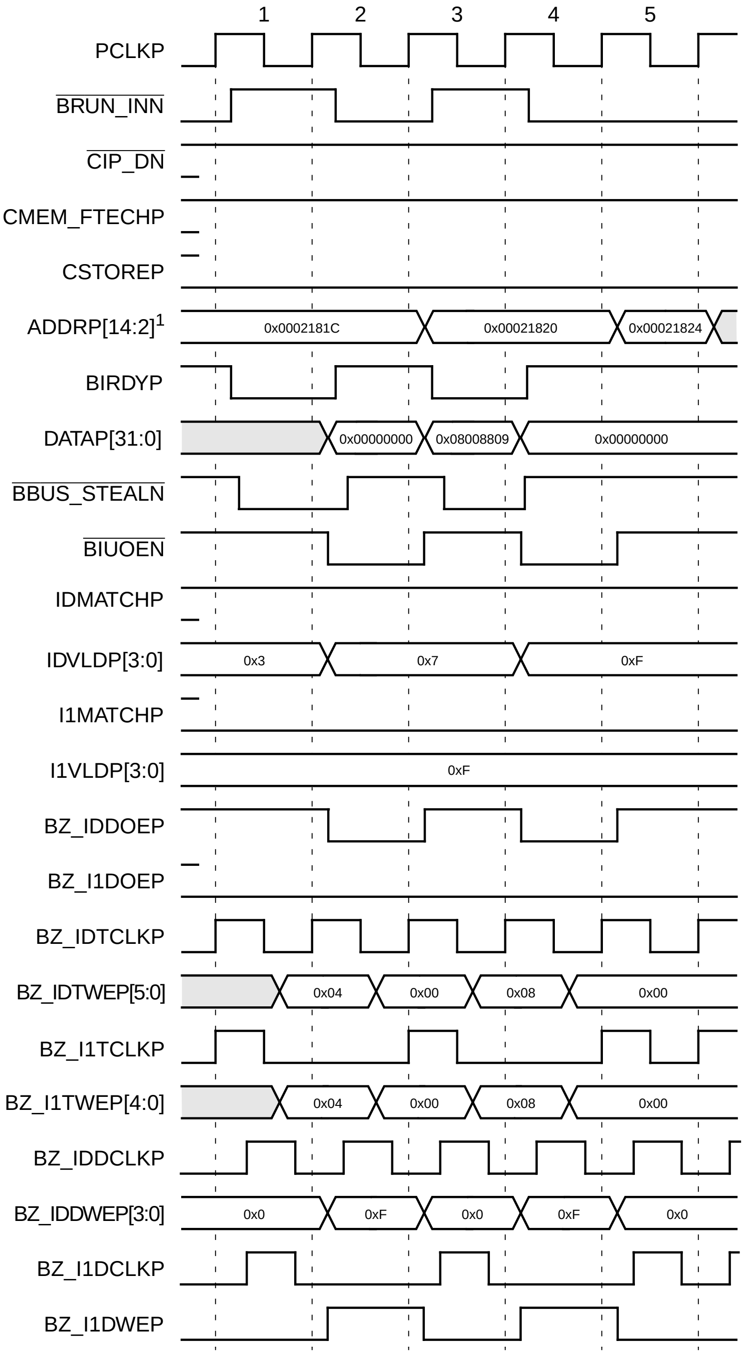
The I-cache can be direct-mapped or two-way set associative. For two-way set associative I-cache, when a tag miss occurs and both lines contain valid instructions, the BBCC replaces one of the lines. At each tag miss when both sets contain valid instructions, the BBCC alternates between which set it replaces. The first time it replaces Set 0, the second time Set 1, then Set 0, then Set 1, and so on.

Each line of I-cache Set 0 can be locked (Lock Bit) to guarantee that the contents of the selected lines remain in cache. Locking is useful for keeping certain code in the cache all the time, such as an exception handler, or some other time-critical code. Setting the Lock Bit in the Tag RAM during Software Cache Test Mode locks a line in the I-cache. The Lock Bit can only be modified while in Software Cache Test Mode or Hardware Cache Test Mode.

The clocks to the Data RAMs of the caches are delayed clocks, so that stores can happen in the first cycle of the store.

Figure 5.19 shows some normal I-cache transactions.

Figure 5.19
Normal I-Cache
Transactions



1. Values shown are [31:0].

Cycle 2:	The next cycle will be an instruction fetch from address 0x0002181C.
Cycle 3:	The tag look-up is done in this cycle, since the cycle is not bus-stolen. IDMATCHP HIGH indicates that the current address matched the tag in Instruction Set 0. However, the Valid Bit corresponding to the fetched word, IDVLDP3, is zero. This indicates a tag miss. BIRDYP is not asserted, and the CW400x stalls. Even though there was a cache miss, the caches still drive DATAP[31:0] (in this case it is Set 0; indicated by BZ_IDDOEP HIGH). Since the BIRDYP was not asserted, the CW400x ignores the value on DATAP[31:0]. The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ to show that the next cycle will be bus-stolen.
Cycle 4:	The GOE asserts $\overline{\text{BIUOEN}}$, and the BBCC drives DATAP[31:0] with the instruction for the instruction fetch during this bus-stolen cycle. The BBCC asserts BIRDYP. At the beginning of this cycle, the BBCC also writes the data into Set 0 of the I-cache (note BZ_IDDWEP[3:0] and BZ_IDDCLKP), and sets the appropriate Valid Bit in the Set 0 Tag RAM (note BZ_IDTWEP[5:0] and BZ_IDTCLKP). The next cycle is an instruction fetch from address 0x00021820.
Cycle 5:	The tag look-up is done in this cycle, since the cycle is not bus-stolen. The tag matched Set 0, and the appropriate Valid Bit was set. These conditions indicate a I-cache hit in Set 0. The Set 0 RAM drives DATAP[31:0], and the BBCC asserts BIRDYP to inform the CW400x that the instruction is on DATAP[31:0].

5.5.3.2 Normal Data Cache Transactions

D-cache fetches are similar to I-cache fetches, except the D-cache is always direct-mapped, and the lines are not lockable.

Stores to D-cache occur if the store is in cacheable address space. The write to the Data RAM occurs at the beginning of the first non-bus-stolen cycle, using a delayed clock. Also at the beginning of the clock cycle, if the store is a full-word store, the BBCC writes the appropriate Valid Bit to the Tag RAM. If the store is a partial store, then no Valid Bits are written. In this same cycle, the tag match is done. Based on the tag match and the Valid Bits, the BBCC might stall the CW400x to update the Tag RAM.

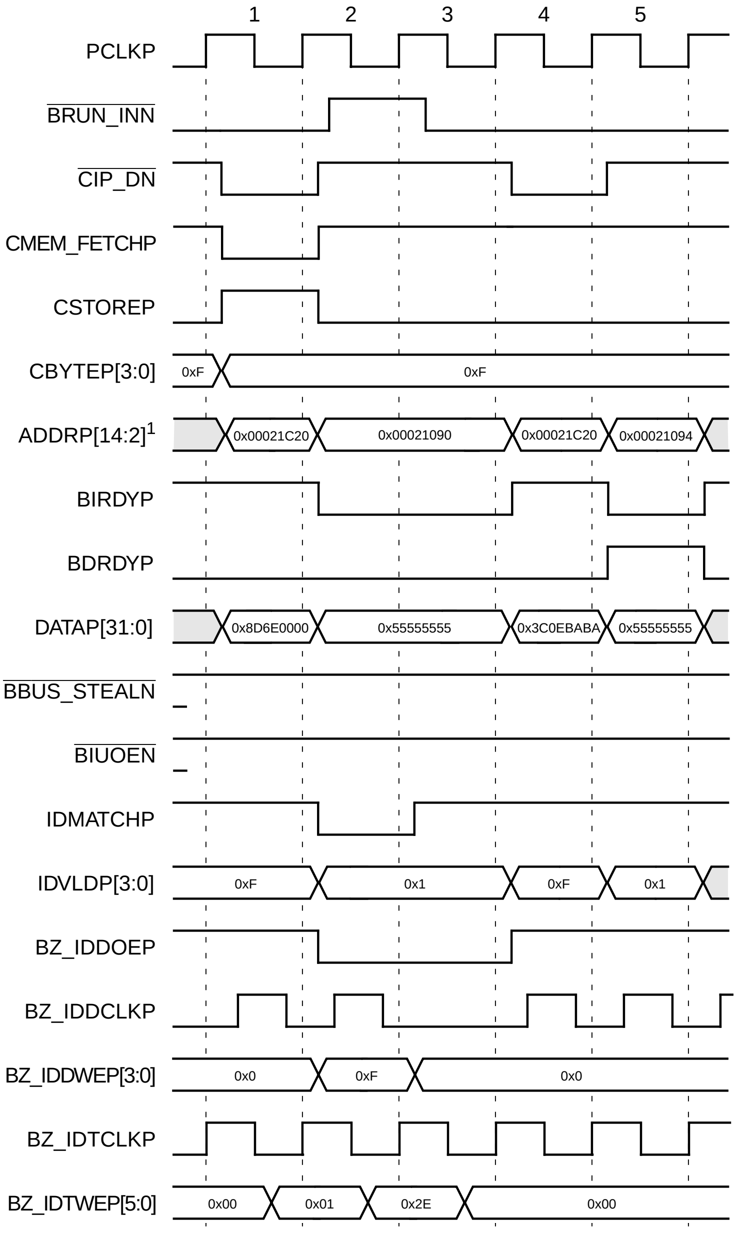
This stalling is necessary for the following cases:

- Full word store, no tag match: allocate line (update tag, clear other Valid Bits)
- Partial store, no tag match, word was valid: invalidate word
- CKILLMEMP or MNOCACHEP asserted, word was valid or full word store: invalidate word

If the Tag RAM needs to be updated, it is updated at the beginning of the next non-bus-stolen cycle.

Figure 5.20 shows some D-cache transactions.

Figure 5.20
D-Cache
Transactions



1. Values shown are [31:0].

Cycle 1: The next cycle will be a store to address 0x00021C20.

Cycle 2: Since the cycle is not bus-stolen ($\overline{\text{BBUS\_STEALN}}$ was not asserted in the previous cycle), the data is written to the D-cache RAM, using CBYTEP[3:0] to control which bytes are written. In this case, the transaction is a full-word store, and so the BBCC writes the corresponding Valid Bit. Both of these transactions occur at the beginning of the cycle; the Valid Bit is written on the normal rising edge, and the data is written with a delayed clock.

During this cycle, the BBCC checks whether the tag matches and whether the appropriate Valid Bit is on. The Valid Bit is set since it was just written, and the tag does not match (note IDMATCHP). These conditions indicate a cache miss, and so the BBCC allocates the line for the new data by updating the tag, and clearing the appropriate Valid Bits. Since a Tag RAM fix-up cycle is needed, the BBCC stalls the CW400x by deasserting BRUN_OUTP, which causes $\overline{\text{BRUN\_INN}}$ to be deasserted.

Cycle 3: Since this cycle is not bus-stolen, the BBCC updates the D-cache Tag RAM. The BBCC writes the Tag and the appropriate Valid Bits. If the cycle had been $\overline{\text{BUS\_STOLEN}}$, the BBCC would have continued stalling, waiting for a non-bus-stolen cycle. The next cycle will be an instruction fetch from address 0x00021090.

Cycle 4: This cycle is non-bus-stolen, and the instruction is in I-cache Set 0. The BBCC asserts BIRDYP and puts the instruction onto DATAP[31:0] by asserting BZ_IDDOEP. The next cycle will be a data fetch from address 0x00021C20.

Cycle 5: This cycle is non-bus-stolen, and the data is in the D-cache. The BBCC asserts BDRDYP and puts the data onto DATAP[31:0] by asserting BZ_IDDOEP. The next cycle will be an instruction fetch from address 0x00021094.

5.5.3.3 Software Cache Test Mode

Software Cache Test Mode allows the CW400x to write and read to cache RAMs that it would not normally have direct access to: the I-cache Data RAMs and the I-cache and D-cache Tag RAMs. This mode is useful for initializing the cache RAMs on reset, or for locking code into the I-cache Set 0.

When the System Configuration Register is written such that CM Bits (BS_CONFIGP[9:8]) indicate that the BBCC is in Software Cache Test Mode, the BBCC interprets loads and stores differently. All loads come from the appropriate cache RAM, unless the address is in the *kseg1* address space. All stores are written to the appropriate cache RAM if they are not in *kseg1*. When BS_CONFIGP[9:8] indicate that the BBCC is in Software Cache Test Mode for the I-cache, 1E Bit (BS_CONFIGP1) in the System Configuration Register specifies the I-cache set.

During Software Cache Test Mode, the MMU should be disabled, and the software test code should be in *kseg1*. The D Bit (BS_CONFIGP4) in the System Configuration Register must be set for Software Cache Test Mode, regardless of whether D-cache exists. Asserting CKILLMEMP causes unexpected results (sometimes the transaction occurs, and other times it does not), so CKILLMEMP must not be asserted during Software Cache Test Mode.

Table 5.7 shows the settings of the System Configuration Register for the various Software Cache Test Modes.

Table 5.7 System Configuration Register Settings for Software Cache Test Mode

Cache Mode BS_CONFIGP[9:8]	**D-Cache Enable BS_CONFIGP4**	**I-Cache Set 1 Enable BS_CONFIGP1**	**Software Cache Test Mode**
01	1	0	I-Cache Set 0 Data
01	1	1	I-Cache Set 1 Data
10	1	0	I-Cache Set 0 Tag
10	1	1	I-Cache Set 1 Tag
11	1	-	D-Cache Tag

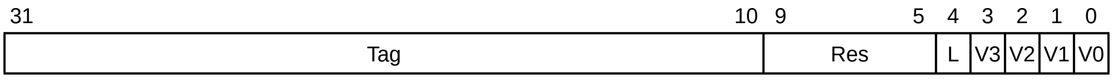
The address for the RAMs is the normally used index, and is dependent on the size of the caches. For the Data RAMs, the lower two bits of the address are ignored. All loads and stores are interpreted as full word transactions. For the Tag RAMs, the lower four bits are ignored. Loads and stores to the Tag RAMs are line operations.

During stores to the Data RAM of the I-cache, the data is on DATAP[31:0]. During stores to the Tag RAM of the I-cache or D-cache, the tag originates from the upper bits of MADDROUTP[31:2] (the tag normally used for the cache) and is placed on BZ_TAGP[21:0]. The Lock Bit originates from DATAP4, and the Valid Bits originate from DATAP[3:0]. The BBCC places these bits on BZ_LOCKP and BZ_VALIDP[3:0] to write them to the RAMs. Because the data on DATAP[31:0] is not available at the rising edge of the

Tag RAM clock, writing to the Tag RAM requires two clock cycles. The BBCC stalls the CW400x for one cycle by deasserting BRUN_OUTP, and writes the data on a subsequent non-bus-stolen cycle.

Fetches from the I-cache Data RAM are straightforward. The RAM places the 32 bits of its data on DATAP[31:0]. Figure 5.21 shows data returned from reading the Tag RAM of a 1 Kbyte cache while in Software Cache Test Mode.

Figure 5.21
Tag RAM Read Data



The following is sample code to lock six instructions in the I-cache.

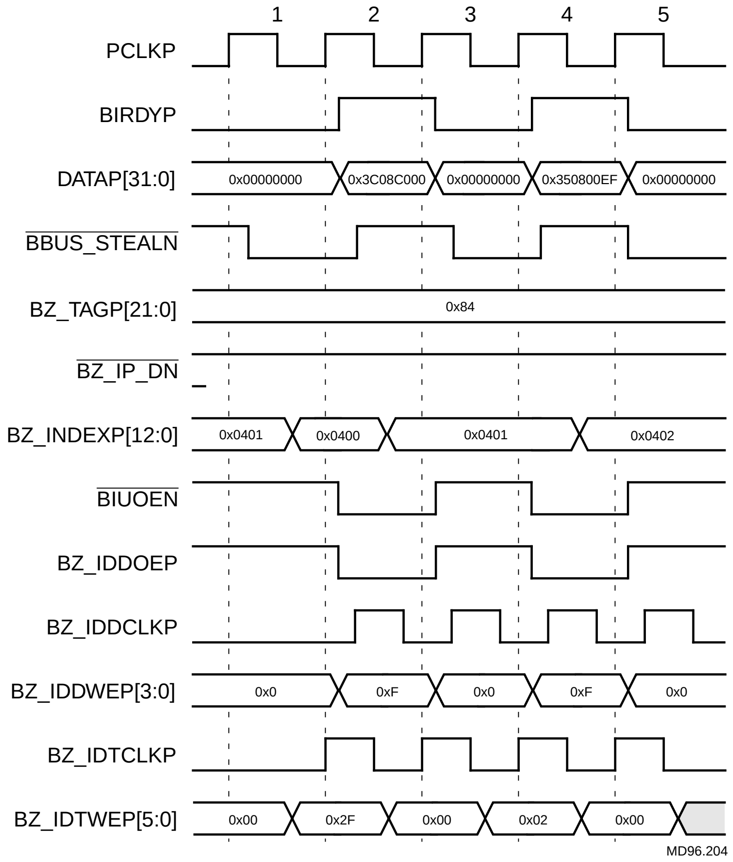
```
SETUP_T0:
    li   t0, 0x000002fd # sw test to write tags to Set 0
    li   t1, 0xbfff0000 # address of Configuration Register
    sw   t0, 0(t1)      # store to Configuration Register
    lw   r0, 0(t1)      # to flush Write Buffer
    addi r0, r0, 1      # force dependency on load

    li   t0, 0x80000000 # tag
    li   t1, 0x1f       # lock, all words valid
    li   t2, 0x13       # lock, two words valid

    sw   t1, 0x80(t0)   # line 0
    sw   t2, 0x90(t0)   # line 1

SETUP_D0:
    li   t0, 0x000001fd # sw test to write data to Set 0
    li   t1, 0xbfff0000 # address of Configuration Register
    sw   t0, 0(t1)      # store to Configuration Register
    lw   r0, 0(t1)      # to flush Write Buffer
    addi r0, r0, 1      # force dependency on load

    li   t0, 0x80000000 # tag doesn’t really matter
    li   t1, 0x3c0eb000 # lui t6, 0xb000
    li   t2, 0x8dcf0000 # lw t7, 0(t6)
    li   t3, 0x21ef0001 # addi t7, t7,1
    li   t4, 0xadcf0000 # sw t7, 0(t6)
    li   t5, 0x03e00008 # jr ra
    li   t6, 0x00000000 # nop

    sw   t1, 0x80(t0)   # Instruction 0
    sw   t2, 0x84(t0)   # Instruction 1
    sw   t3, 0x88(t0)   # Instruction 2
    sw   t4, 0x8c(t0)   # Instruction 3
    sw   t5, 0x90(t0)   # Instruction 4
    sw   t6, 0x94(t0)   # Instruction 5
```

5.5.3.4 BBus Normal Cache Interface

When the BBCC is a master on the BBus, it places the data for cacheable transactions in the cache. There are two types of cache transactions: *write-first* and *write-subsequent*. The transaction can be to the I-cache Set 0, the I-cache Set 1, or the D-cache. Each transaction generates a bus steal, so that the BBCC can write the cache RAMs. The write-first type transaction writes the data (always a full word) to the appropriate Data RAM, updates the tag with the tag of the BBus transaction address, sets the Valid Bit for the word which is written, and clears all the other Valid Bits. The write-subsequent type transaction writes the data to the appropriate Data RAM, and sets the corresponding Valid Bit.

Figure 5.22 shows some normal BBus cache transactions.

Figure 5.22
Normal BBus
Cache
Transactions



Cycle 1: The BBCC asserts $\overline{\text{BBUS\_STEALN}}$, which indicates that in the next cycle it will drive DATAP[31:0].

Cycle 2: At the beginning of the cycle, the BBCC writes the I-cache Set 0 Tag RAM with a write-first type of operation; it writes the tag (corresponding to address 0x00021000 for a 1-Kbyte cache), sets the appropriate Valid Bit, and resets the other Valid Bits. Also at the beginning of the cycle, the BBCC writes the instruction on DATAP[31:0] into the I-cache Set 0 Data RAM, using the delayed clock. This instruction is also the instruction being fetched, so the BBCC asserts BIRDYP.

Cycle 3: The BBCC asserts $\overline{\text{BBUS\_STEALN}}$ to bus steal the next cycle.

Cycle 4: The BBCC again writes the instruction to the I-cache Set 0 Data RAM. This time, however, the BBCC writes only one Valid Bit to the Tag RAM. This operation is a write-subsequent type. The BBCC asserts BIRDYP again, to indicate that the instruction on DATAP[31:0] is the fetched instruction. This example demonstrates streaming; the BBCC transmits the instruction to the CW400x at the same time as the instruction is written to the cache.

5.5.3.5 BBus Hardware Test Cache Interface

The Hardware Cache Test Mode cache interface is similar to the Normal Mode cache interface. The write-first and write-subsequent type of operations are used to write to the Data RAMs. In addition, the BBus transaction might indicate that the BBCC should read the Data RAMs, or write/read the Tag RAMs.

Hardware Cache Test Mode transactions are all similar. The BBCC asserts $\overline{\text{BBUS\_STEALN}}$, and performs the appropriate RAM transaction. For Data RAM reads, the 32-bit data is read from the RAM and output enabled onto DATAP[31:0]. For Tag RAM reads, the format is the same as that of Software Cache Test Mode; the tag is on the upper bits of DATAP[31:0], the Lock Bit is DATAP4, and the Valid Bits are DATAP[3:0]. For Tag RAM writes, the tag portion of the address on BADDRPI[31:2] is placed on BZ_TAGP[21:0], and the BBCC writes these signals to the RAMs. The Lock and Valid Bits are from BDATAPI[4:0], and are placed on BZ_LOCKP and BZ_VALIDP[3:0] to be written to the RAMs.

5.5.3.6 Snooping

The BBCC monitors transactions on the BBus. When another device on the BBus writes to memory, the BBCC generates a snoop transaction. Snooping requires two cycles for the I-cache, and two cycles for the D-cache. The BBCC uses the inputs BISNOOP and BDSNOOP to determine whether or not to snoop on the caches. If both BISNOOP and BDSNOOP are asserted, snooping takes four clock cycles. Each cycle, the BBCC asserts $\overline{\text{BBUS\_STEALN}}$.

For each snoop operation, in the first cycle the BBCC compares BZ_TAG4MATCHP[21:0] (which contains the value from the tag portion of BADDRPI[31:2]) and the tag in the Tag RAM. During the second cycle of the snoop, if there is a tag match, the BBCC invalidates the line.

5.5.4 On-Chip Memory (OCM)

On-chip memory (OCM) resides on the CBus, and is accessed in one clock cycle. Asserting BOCMEXISTP informs the BBCC that OCM exists. Transactions to the OCM do not cause BBus transactions; the transactions are only to the OCM, not to external memory. OCM can be used as a data scratchpad or as a boot ROM. The address space for the OCM must be in *kseg1* (non-cacheable space). The OCM uses both ADDRP[31:2] and MADDROUTP[31:2] to address the memory. For reads, the OCM uses ADDRP[31:2]; for writes, it uses MADDROUTP[31:2]. The OCM must select which address to use. The OCM must also assert BOCMSELP to the BBCC when MADDROUTP[31:2] matches the OCM address space. The BBCC provides the output enable and write enable to the OCM. If OCM is installed, the OCM is output enabled onto DATAP[31:0] for every non-bus-stolen *kseg1* fetch, regardless of whether the address matched the OCM or not.

The OCM uses the unmapped address to access the RAM. Thus, if an MMU is in the system, the OCM must be smaller in size than the MMU page size. If the MMU Stub is installed, rather than the MMU, this limitation does not apply.

Reads from the OCM take place at the beginning of the run cycle, using ADDRP[31:2] as the address. If the first run cycle is bus-stolen, then the OCM must do the read over, this time using MADDROUTP[31:2] as the address. Alternatively, a gated clock can be used for the OCM, which would cause the read to be only done once. The data from the OCM is output enabled onto DATAP[31:0] during the first non-bus-stolen cycle. At this time, the BBCC asserts BIRDYP or BDRDYP to the CW400x.

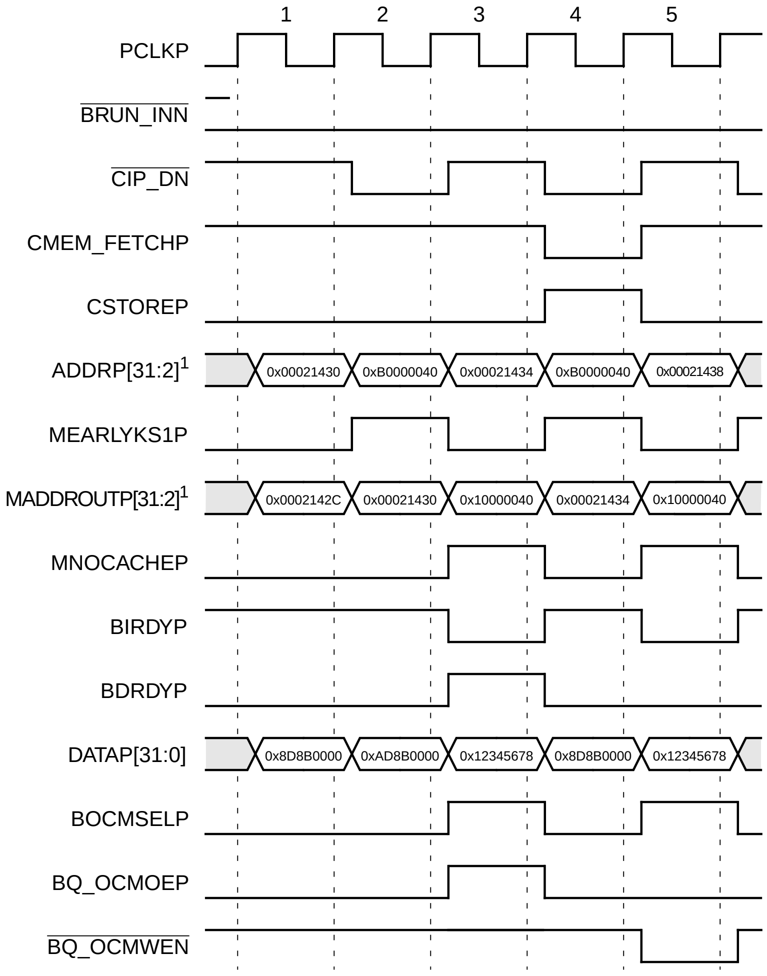
Writes to the OCM take place at the end of the run cycle. This enables the CW400x to have time to place the store data on DATAP[31:0] before the write occurs. Writes occur in the first non-bus-stolen cycle after the run cycle. Because writes occur at the end of the cycle, instead of the beginning, it is not possible to write data to the OCM while executing instructions out of the OCM (since the write to the OCM is followed by an instruction fetch, which occurs at the beginning of the cycle). One way to work around this limitation, at the expense of some performance, is to force the system to stall whenever a write to OCM takes place. This stalling is easily done by ANDing BRUN_OUTP with $\overline{\text{BQ\_OCMWEN}}$, and using this modified BRUN_OUTP signal as an input to the GOE Module.

The address match logic for BOCMSELP should be based on MADDROUTP[31:2] and a registered version of MEARLYKS1P. The registered MEARLYKS1P signal is needed to remember whether the transaction was in the *kseg1* address space; since MADDROUTP[31:2] is mapped, the information needs to be held separately. The range for the address match is determined by the size of the OCM.

Using a gated clock for the OCM saves power. If the operating frequency is slow, ADDRP[31:2] can be decoded, and the OCM can be clocked only when ADDRP[31:2] matches the OCM address space. This method might be too slow, since ADDRP[31:2] is one of the later-arriving signals. Power can still be saved by clocking the OCM only on *kseg1* run cycles.

Figure 5.23 shows some OCM transactions. In this example, the OCM is 1 Kbyte, and has an address space from 0xB000000 to 0xB00003FF.

Figure 5.23
OCM Transactions



1. Values shown are [31:0].

MD95.232

Cycle 1: The CBus signals indicate that the next cycle will be an instruction fetch from address 0x00021430.

Cycle 2: MADDROUTP[31:2] contains the mapped address for the instruction fetch. The next cycle will be a data fetch from 0xB0000040, which is an OCM address.

Cycle 3: MADDROUTP[31:2] contains the mapped address for the data fetch. The OCM asserts BOCMSELP to indicate that the transaction is for the OCM. The BBCC asserts the output enable to the OCM, and BDRDYP to the CW400x. The next cycle will be an instruction fetch from 0x00021434.

Cycle 4: The next cycle will be a store to address 0xB0000040, an OCM address.

Cycle 5: The OCM asserts BOCMSELP to indicate that the transaction is for the OCM. The BBCC asserts the write enable to the OCM, and the store occurs at the end of this cycle.

The following is sample Verilog code for a 1-Kbyte OCM. It uses the *mg922 RAM* and a gated clock.

```
module cw400x_ocm (pclkp, runn, ocmoep, ocmwen, mearlyks1p,
                  addrp, upper_maddroutp, maddroutp,
                  ocmselp, datap );

//****************** PORT DECLARATIONS ********************
input
        pclkp, runn, ocmoep, ocmwen, mearlyks1p;

input [29:10]
        upper_maddroutp;

input [9:2]
        addrp, maddroutp;

output
        ocmselp;

inout [31:0]
        datap;
//*************** END OF PORT DECLARATIONS ****************
//********************** PARAMETERS ***********************
parameter
start_adrs = 19’h40000;       // address is 0xB0000000
//****************** END OF PARAMETERS ********************
//************************ WIRES **************************
wire
        gclkp;

wire [9:2]
        adrsp;
//********************* END OF WIRES **********************
//************************* REGS **************************
reg
        gatep, kseg1p;
//********************* END OF REGS ***********************
```

```
//******************* INFER REGISTERS *******************
always @ (posedge pclkp)
   kseg1p <= runn ? kseg1p : mearlyks1p;

always @ (pclkp or mearlyks1p or runn or ocmwen)
        if (pclkp == 1'b0)
                gatep <= ((mearlyks1p & ~runn) | ~ocmwen);

assign gclkp = pclkp & gatep;
//*************** END OF INFER REGISTERS ****************
assign adrsp = ocmwen ? addrp : maddroutp;
assign ocmselp = kseg1p & (upper_maddroutp == start_adrs);
//***************** RAM INSTANTIATION *******************
rr32x256_922 ocm_rami(
        .a(adrsp[9:2]),
        .clk(gclkp),
        .oe(ocmoep),
        .we(~ocmwen),
        .di(datap[31:0]),
        .do(datap[31:0]) );

endmodule
```

5.5.5 Write Buffer

The Write Buffer is an extension to the Store Queue in the BBCC. It is a FIFO for store transactions. When stores are received from the CW400x, the store is either placed in the Store Queue in the BBCC, or in one of the entries of the Write Buffer. The address (from MADDROUTP[31:2]), the data (from DATAP[31:0]), and the byte enables (from CBYTEP[3:0]) are held until the store can be done. Additional information about the store is held in the Store Queue/Write Buffer, such as whether the store was to the System Configuration Register (BW_CFGSELP), and whether the store arrived while the data fetch queue was empty (BW_ARRIVEBFLDP). The Store Queue/Write Buffer entries are loaded with the store information at the end of the run cycle (unless it is a bus-stolen cycle, in which case the entry is loaded after the bus is no longer stolen).

When a store transaction is completed, the store transactions in the Store Queue and Write Buffer are passed up the FIFO queue. The BBCC signals starting with “WB_” are inputs to the BBCC from the Write Buffer. The WB_ARRIVEBFLDP, WB_VWBFLDP, WB_STPNDP, and WB_FULLP signals are queue management information. The WB_ADDRP[31:2] and WB_DATAP[31:0] signals are put in the BBCC Store Queue. The BBCC signals starting with “BW_” are outputs from the BBCC to the Write Buffer.

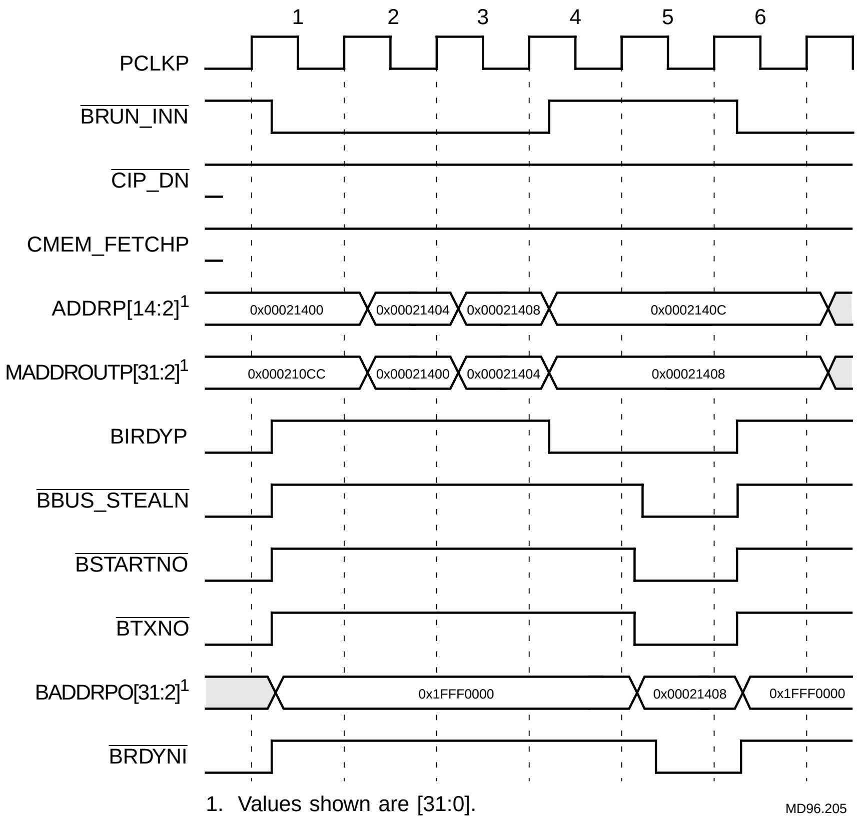
Flushing of the Write Buffer is based on Bits [8:5] of the address. When Read Priority is on, the Load Queue is given priority over the Store Queue. However, if the load is preceded by a store to the same block, then the store must be done before the load for proper operation. Each entry for the Store Queue/Write Buffer compares Bits [8:5] of its address to Bits [8:5] of the load transaction’s address. If there is a match, and the store transaction preceded the load transaction, then the Store Queue has priority over the Load Queue.

For information on how to attach additional Write Buffers, see Chapter 6.

5.6 Cache-Miss Penalty, BBus Latency

The minimum cache-miss penalty for the BBCC is two cycles. The minimum cache-miss penalty occurs when the BBus device can support a one-cycle transaction, as shown in Figure 5.24.

Figure 5.24 Cache-Miss Penalty, BBus Latency



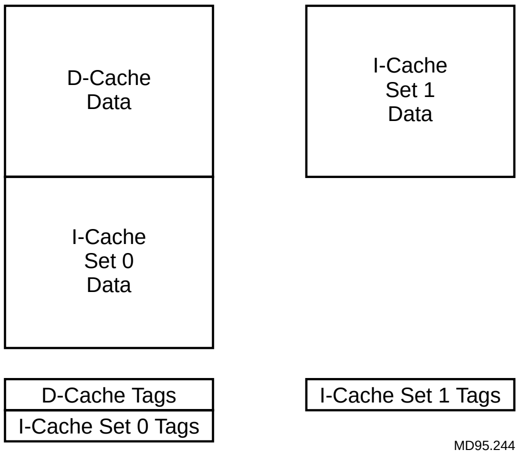
Cycle 1:	The previous instruction fetch has been received (the BBCC asserted BIRDYP), and so the CW400x begins a new instruction fetch from address 0x00021400.
Cycle 2:	The instruction is in the cache, and the BBCC asserts BIRDYP. The CW400x begins a new instruction fetch from address 0x00021404.
Cycle 3:	The instruction is in the cache, the BBCC asserts BIRDYP. The CW400x begins a new instruction fetch from address 0x00021408.
Cycle 4:	The instruction is not in the cache, so the BBCC does not assert BIRDYP. The BBCC must generate a BBus transaction to fetch the instruction. The CW400x stalls, waiting for the instruction to be returned.
Cycle 5:	The BBCC starts a BBus transaction to address 0x00021408. The BBCC asserts $\overline{\text{BSTARTNO}}$ and $\overline{\text{BTXNO}}$, and places the address on BADDRPO[31:2]. In this case, the BBus device supports a one-cycle BBus transaction, and so the BBus device asserts $\overline{\text{BRDYNI}}$ in this cycle (in response to $\overline{\text{BSTARTNO}}$ being asserted), and the BBCC asserts $\overline{\text{BBUS\_STEALN}}$ so that it can put the instruction on DATAP[31:0].
Cycle 6:	The BBCC asserts BIRDYP to indicate that the instruction is on DATAP[31:0]. The CW400x begins a new instruction fetch.

As seen in Figure 5.24, the latency to start a BBus transaction is one cycle after the cache-lookup cycle. This accounts for the first cache-miss penalty cycle. Also, the latency from the assertion of $\overline{\text{BRDYNI}}$ to the assertion of BIRDYP is one cycle. This accounts for the second cache-miss penalty cycle.

5.7 Adding Cache

The BBCC interfaces to four cache RAMs if the system has a two-way set associative I-cache, otherwise it interfaces to two cache RAMs. Figure 5.25 shows the cache RAMs for a system. The D-cache and I-cache Set 0 share the same RAMs. However, the D-cache and I-cache Set 0 are distinct portions of the RAM and act as separate caches, not a unified cache. $\overline{\text{BZ\_IP\_DN}}$, which is used as the highest bit of the address, determines which portion of the RAM to access.

Figure 5.25
Cache RAMs for a System



The Data RAMs of the caches are word-wide (32-bits wide), and the lowest address bit is Bit 2 of the address (BZ_INDEXP0). Each location of the Tag RAMs is associated with a cache line, and the lowest address bit is Bit 4 of the address (BZ_INDEXP2).

5.7.1 RAM Sizes

Tables 5.8 through 5.12 show the RAM sizes needed for various cache configurations. Note that although some configurations call for RAMs greater than 4 Kbytes deep, the 500 Kbyte memory compiler currently only supports mg922 RAMs up to 4 Kbytes in depth. Shaded entries represent RAM configurations that are not supported with the current 500 Kbyte mg922 RAM compiler.

The caches use the unmapped address to access the cache RAMs. Thus, if an MMU is in the system, the size of the caches must be smaller than the MMU page size. If the MMU Stub is installed, rather than the MMU, this limitation does not apply.

Table 5.8
Direct Mapped I-Cache

	1K	**2K**	**4K**	**8K**	**16K**	**32K**
Tag	64 x 27	128 x 26	256 x 25	512 x 24	1K x 23	2K x 22
Data	256 x 32	512 x 32	1K x 32	2K x 32	4K x 32	8K x 32

Table 5.9
Two-Way Set
Associative I-Cache

	1K	**2K**	**4K**	**8K**	**16K**	**32K**
Tag	64 x 27 64 x 26	128 x 26 128 x 25	256 x 25 256 x 24	512 x 24 512 x 23	1K x 23 1K x 22	2K x 22 2K x 21
Data	256 x 32 256 x 32	512 x 32 512 x 32	1K x 32 1K x 32	2K x 32 2K x 32	4K x 32 4K x 32	8K x 32 8K x 32

Table 5.10
D-Cache

	1K	**2K**	**4K**	**8K**	**16K**	**32K**
Tag	64 x 27	128 x 26	256 x 25	512 x 24	1K x 23	2K x 22
Data	256 x 32	512 x 32	1K x 32	2K x 32	4K x 32	8K x 32

Table 5.11
Direct-Mapped I-Cache
with D-Cache

	I-Cache	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**
1K D-Cache	Tag	128 x 27	192 x 27	320 x 27	576 x 27	1088 x 27	2112 x 27
	Data	512 x 32	768 x 32	1280 x 32	2304 x 32	4352 x 32	8448 x 32
2K D-Cache	Tag	192 x 27	256 x 26	384 x 26	640 x 26	1152 x 26	2176 x 26
	Data	768 x 32	1K x 32	1536 x 32	2560 x 32	4608 x 32	8704 x 32
4K D-Cache	Tag	320 x 27	384 x 26	512 x 25	768 x 25	1280 x 25	2304 x 25
	Data	1280 x 32	1536 x 32	2K x 32	3K x 32	5K x 32	9K x 32
8K D-Cache	Tag	576 x 27	640 x 26	768 x 25	1K x 24	1536 x 24	2560 x 24
	Data	2304 x 32	2560 x 32	3K x 32	4K x 32	6K x 32	10K x 32
16K D-Cache	Tag	1088 x 27	1152 x 26	1280 x 25	1536 x 24	2K x 23	3K x 23
	Data	4352 x 32	4608 x 32	5K x 32	6K x 32	8K x 32	12K x 32
32K D-Cache	Tag	2112 x 27	2176 x 26	2304 x 25	2560 x 24	3K x 23	4K x 22
	Data	8448 x 32	8704 x 32	9K x 32	10K x 32	12K x 32	16K x 32

Table 5.12
Two-Way Set
Associative I-Cache
with D-Cache

	I-Cache	1K	2K	4K	8K	16K	32K
1K D-Cache	Tag	128 x 27 64 x 26	192 x 27 128 x 25	320 x 27 256 x 24	576 x 27 512 x 23	1088 x 27 1K x 22	2112 x 27 2K x 21
	Data	512 x 32 256 x 32	768 x 32 512 x 32	1280 x 32 1K x 32	2304 x 32 2K x 32	4352 x 32 4K x 32	8448 x 32 8K x 32
2K D-Cache	Tag	192 x 27 64 x 26	256 x 26 128 x 25	384 x 26 256 x 24	640 x 26 512 x 23	1152 x 26 1K x 22	2176 x 26 2K x 21
	Data	768 x 32 256 x 32	1K x 32 512 x 32	1536 x 32 1K x 32	2560 x 32 2K x 32	4608 x 32 4K x 32	8704 x 32 8K x 32
4K D-Cache	Tag	320 x 27 64 x 26	384 x 26 128 x 25	512 x 25 256 x 24	768 x 25 512 x 23	1280 x 25 1K x 22	2304 x 25 2K x 21
	Data	1280 x 32 256 x 32	1536 x 32 512 x 32	2K x 32 1K x 32	3K x 32 2K x 32	5K x 32 4K x 32	9K x 32 8K x 32
8K D-Cache	Tag	576 x 27 64 x 26	640 x 26 128 x 25	768 x 25 256 x 24	1K x 24 512 x 23	1536 x 24 1K x 22	2560 x 24 2K x 21
	Data	2304 x 32 256 x 32	2560 x 32 512 x 32	3K x 32 1K x 32	4K x 32 2K x 32	6K x 32 4K x 32	10K x 32 8K x 32
16K D-Cache	Tag	1088 x 27 64 x 26	1152 x 26 128 x 25	1280 x 25 256 x 24	1536 x 24 512 x 23	2K x 23 1K x 22	3K x 23 2K x 21
	Data	4352 x 32 256 x 32	4608 x 32 512 x 32	5K x 32 1K x 32	6K x 32 2K x 32	8K x 32 4K x 32	12K x 32 8K x 32
32K D-Cache	Tag	2112 x 27 64 x 26	2176 x 26 128 x 25	2304 x 25 256 x 24	2560 x 24 512 x 23	3K x 23 1K x 22	4K x 22 2K x 21
	Data	8448 x 32 256 x 32	8704 x 32 512 x 32	9K x 32 1K x 32	10K x 32 2K x 32	12K x 32 4K x 32	16K x 32 8K x 32

5.7.2 Examples

This subsection describes four example RAM configurations:

- Example 1: 1-Kbyte D-Cache and 2-Kbyte Two-Way Set Associative I-Cache (1 Kbyte Each Set)
- Example 2: 4-Kbyte D-Cache and 2-Kbyte Two-Way Set Associative I-Cache (1 Kbyte Each Set)
- Example 3: 2-Kbyte D-Cache and 4-Kbyte Direct Mapped I-Cache
- Example 4: 8-Kbyte Non-Partitioned Cache

5.7.2.1 Example 1: 1-Kbyte D-Cache and 2-Kbyte Two-Way Set Associative I-Cache (1 Kbyte Each Set)

Tables 5.13 through 5.16 show the signal connections for the RAMs.

Table 5.13
D-Cache/I-Cache
Set 0 Data RAM

RAM Port	BBCC Connection	Function
CLK	BZ_IDDCLKP	Clock
A8	$\overline{\mathrm{BZ\_IP\_DN}}$	Select D-Cache Or I-Cache Portion
A[7:0]	BZ_INDEXP[7:0]	Address
WE[31:24]	BZ_IDDWEP3	Byte 3 Write Enable
WE[23:16]	BZ_IDDWEP2	Byte 2 Write Enable
WE[15:8]	BZ_IDDWEP1	Byte 1 Write Enable
WE[7:0]	BZ_IDDWEP0	Byte 0 Write Enable
OE[31:0]	BZ_IDDOEP	Output Enable
DI[31:0]	DATAP[31:0]	Data In
DO[31:0]	DATAP[31:0]	Data Out

Table 5.14
D-Cache/I-Cache
Set 0 Tag RAM

RAM Port	BBCC Connection	Function
CLK	BZ_IDTCLKP	Clock
A6	$\overline{\mathrm{BZ\_IP\_DN}}$	Select D-Cache or I-Cache Portion
A[5:0]	BZ_INDEXP[7:2]	Address
WE[26:5]	BZ_IDTWEP5	Tag Write Enable
WE4	BZ_IDTWEP4	Lock Bit Write Enable
WE[3:0]	BZ_IDTWEP[3:0]	Valid Bits Write Enables
OE[26:0]	Tied HIGH	Output Enable
DI[26:5]	BZ_TAGP[21:0]	Tag
DI4	BZ_LOCKP	Lock Bit
DI[3:0]	BZ_VALIDP[3:0]	Valid Bits
DO[26:5]	IDTAGP[21:0]	Tag (to Match Logic)
DO4	IDLCKP	Lock Bit
DO[3:0]	IDVLDP[3:0]	Valid Bits

Table 5.15
I-Cache Set 1
Data RAM

RAM Port	BBCC Connection	Function
CLK	BZ_I1DCLKP	Clock
A[7:0]	BZ_INDEXP[7:0]	Address
WE	BZ_I1DWEP	Write Enable
OE	BZ_I1DOEP	Output Enable
DI[31:0]	DATAP[31:0]	Data In
DO[31:0]	DATAP[31:0]	Data Out

Table 5.16
I-Cache Set 1
Tag RAM

RAM Port	BBCC Connection	Function
CLK	BZ_I1TCLKP	Clock
A[5:0]	BZ_INDEXP[7:2]	Address
WE[25:4]	BZ_I1TWEP4	Tag Write Enable
WE[3:0]	BZ_I1TWEP[3:0]	Valid Bits Write Enables
OE[25:0]	Tied HIGH	Output Enable
DI[25:4]	BZ_TAGP[21:0]	Tag
DI[3:0]	BZ_VALIDP[3:0]	Valid Bits
DO[25:4]	I1TAGP[21:0]	Tag (to Match Logic)
DO[3:0]	I1VLDP[3:0]	Valid Bits

In addition to hooking up the RAMs, the system designer must code the tag match logic and the logic to put the tag on DATAP[31:0] (for Software Cache Test Mode and Hardware Cache Test Mode). In Verilog code, the tag match logic is:

```
assign idmatchp = (bz_tag4matchp[21:0] == idtagp[21:0]);
assign i1matchp = (bz_tag4matchp[21:0] == i1tagp[21:0]);
```

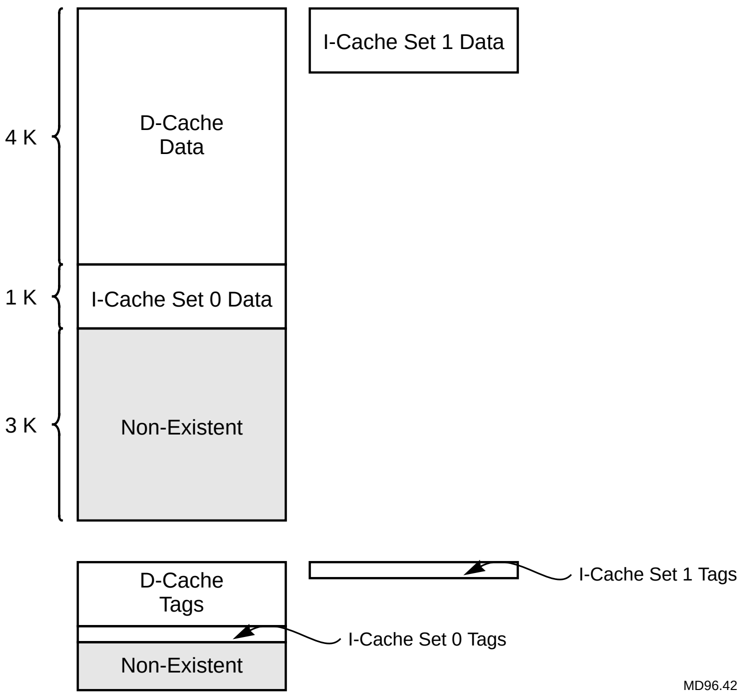
The Verilog code for the 3-state gates to place the Tag RAM outputs onto DATAP[31:0] is:

```
assign datap = bz_idt_oen ? 32’hz : {idtagp[21:0], 5’h0,
   idlckp, idvldp[3:0]};
assign datap = bz_i1t_oen ? 32’hz : {i1tagp[21:0], 6’h0,
   i1vldp[3:0]};
```

5.7.2.2 Example 2: 4-Kbyte D-Cache and 2-Kbyte Two-Way Set Associative I-Cache (1 Kbyte Each Set)

In this example, the combined D-cache and I-cache Set 0 RAM sizes are irregular; they are not a power-of-two size. In this case, the address and tag must be manipulated. The address to the RAM is limited when the I-cache is accessed. Also, the tag for the D-cache is smaller, so some bits of the tag are forced to zero when it is written to the RAM. Figure 5.26 shows the RAM configuration.

Figure 5.26 Example 2 RAM Configuration



Tables 5.17 through 5.20 show the signal connections for the RAMs.

Table 5.17
D-Cache/I-Cache
Set 0 Data RAM

RAM Port	BBCC Connection	Function
CLK	BZ_IDDCLKP	Clock
A10	$\overline{\text{BZ\_IP\_DN}}$	Select D-Cache or I-Cache Portion
A[9:8]	BZ_INDEXP[9:8] &[1] {~$\overline{\text{BZ\_IP\_DN}}$, ~$\overline{\text{BZ\_IP\_DN}}$}[2]	Address High Bits (Force to Zero for I-Cache)
A[7:0]	BZ_INDEXP[7:0]	Address Low Bits
WE[31:24]	BZ_IDDWEP3	Byte 3 Write Enable
WE[23:16]	BZ_IDDWEP2	Byte 2 Write Enable
WE[15:8]	BZ_IDDWEP1	Byte 1 Write Enable
WE[7:0]	BZ_IDDWEP0	Byte 0 Write Enable
OE[31:0]	BZ_IDDOEP	Output Enable
DI[31:0]	DATAP[31:0]	Data In
DO[31:0]	DATAP[31:0]	Data Out

1. & means logical AND.
2. ~ means that the signal is connected through an inverter.

Table 5.18
D-Cache/I-Cache
Set 0 Tag RAM

RAM Port	BBCC Connection	Function
CLK	BZ_IDTCLKP	Clock
A8	$\overline{\text{BZ\_IP\_DN}}$	Select D-Cache or I-Cache Portion
A[7:6]	BZ_INDEXP[9:8] &[1] {~$\overline{\text{BZ\_IP\_DN}}$, ~$\overline{\text{BZ\_IP\_DN}}$}[2]	Address High Bits (Force to Zero for I-Cache)
A[5:0]	BZ_INDEXP[7:2]	Address Low Bits
WE[26:5]	BZ_IDTWEP5	Tag Write Enable
WE4	BZ_IDTWEP4	Lock Bit Write Enable
WE[3:0]	BZ_IDTWEP[3:0]	Valid Bits Write Enables
OE[26:0]	Tied HIGH	Output Enable
DI[26:7]	BZ_TAGP[21:2]	Tag High Bits
DI[6:5]	BZ_TAGP[1:0] &[1] {$\overline{\text{BZ\_IP\_DN}}$, $\overline{\text{BZ\_IP\_DN}}$}	Tag Low Bits (Force to Zero for D-Cache)
DI4	BZ_LOCKP	Lock Bit
DI[3:0]	BZ_VALIDP[3:0]	Valid Bits
DO[26:5]	IDTAGP[21:0]	Tag (to Match Logic)
DO4	IDLCKP	Lock Bit
DO[3:0]	IDVLDP[3:0]	Valid Bits

1. & means logical AND.
2. ~ means that the signal is connected through an inverter.

Table 5.19
I-Cache Set 1
Data RAM

RAM Port	BBCC Connection	Function
CLK	BZ_I1DCLKP	Clock
A[7:0]	BZ_INDEXP[7:0]	Address
WE	BZ_I1DWEP	Write Enable
OE	BZ_I1DOEP	Output Enable
DI[31:0]	DATAP[31:0]	Data In
DO[31:0]	DATAP[31:0]	Data Out

Table 5.20
I-Cache Set 1
Tag RAM

RAM Port	BBCC Connection	Function
CLK	BZ_I1TCLKP	Clock
A[5:0]	BZ_INDEXP[7:2]	Address
WE[25:4]	BZ_I1TWEP4	Tag Write Enable
WE[3:0]	BZ_I1TWEP[3:0]	Valid Bits Write Enables
OE[25:0]	Tied HIGH	Output Enable
DI[25:4]	BZ_TAGP[21:0]	Tag
DI[3:0]	BZ_VALIDP[3:0]	Valid Bits
DO[25:4]	I1TAGP[21:0]	Tag (to Match Logic)
DO[3:0]	I1VLDP[3:0]	Valid Bits

The match logic is the same as in Example 1, except that the tag for the D-cache is smaller. One way to handle this is to force some of the bits of BZ_TAG4MATCHP[21:0] to zero when the D-cache is accessed. The $\overline{\text{BZ\_IP\_DN\_L}}$ signal, a registered version of $\overline{\text{BZ\_IP\_DN}}$, is provided for ANDing with the bits to force them to zero.

```
assign new_tag4matchp = {bz_tag4matchp[21:2],
   bz_tag4matchp[1:0] & {bz_ip_dn_l, bz_ip_dn_l}};
assign idmatchp = (new_tag4matchp[21:0] == idtagp[21:0]);
assign i1matchp = (bz_tag4matchp[21:0] == i1tagp[21:0]);
```

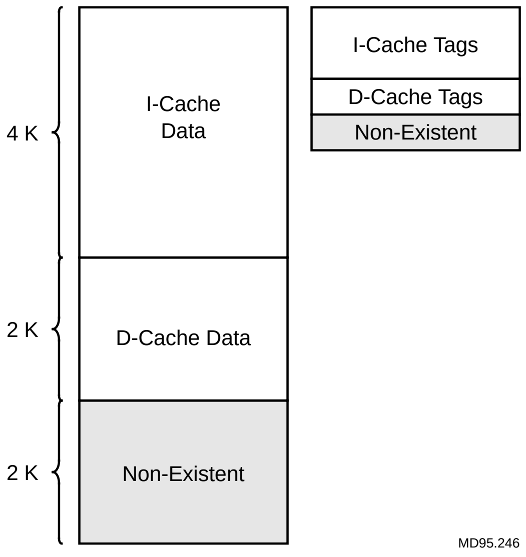
The Verilog code for the 3-state gates to place the Tag RAM outputs onto DATAP[31:0] is the same as in Example 1:

```
assign datap = bz_idt_oen ? 32'hz : {idtagp[21:0], 5'h0,
   idlckp,idvldp[3:0]};
assign datap = bz_i1t_oen ? 32'hz : {i1tagp[21:0], 6'h0,
   i1vldp[3:0]};
```

5.7.2.3 Example 3: 2-Kbyte D-Cache and 4-Kbyte Direct Mapped I-Cache

In this example, the I-cache is larger than the D-cache, and should be placed in the lower-addressed portion of the RAM. Again, the address and tag must be manipulated. The address to the RAM is limited when the D-cache is accessed. Also, the tag for the I-cache is smaller, so the same bits of the tag are forced to zero when it is written to the RAM. Figure 5.27 shows the RAM configuration.

Figure 5.27
Example 3 RAM Configuration



Tables 5.21 and 5.22 show the signal connections for the RAMs.

Table 5.21
D-Cache/I-Cache Data RAM

RAM Port	BBCC Connection	Function
CLK	BZ_IDDCLKP	Clock
A10	~$\overline{\text{BZ\_IP\_DN}}$[1]	Select D-Cache or I-Cache Portion
A9	BZ_INDEXP9 &[2] $\overline{\text{BZ\_IP\_DN}}$	Address High Bit (Zero for D-Cache)
A[8:0]	BZ_INDEXP[8:0]	Address Low Bits
WE[31:24]	BZ_IDDWEP3	Byte 3 Write Enable
WE[23:16]	BZ_IDDWEP2	Byte 2 Write Enable
WE[15:8]	BZ_IDDWEP1	Byte 1 Write Enable

(Sheet 1 of 2)

Table 5.21 (Cont.)
D-Cache/I-Cache
Data RAM

RAM Port	BBCC Connection	Function
WE[7:0]	BZ_IDDWEP0	Byte 0 Write Enable
OE[31:0]	BZ_IDDOEP	Output Enable
DI[31:0]	DATAP[31:0]	Data In
DO[31:0]	DATAP[31:0]	Data Out
(Sheet 2 of 2)		

1. ~ means that the signal is connected through an inverter.
2. & means logical AND.

Table 5.22
D-Cache/I-Cache
Set 0 Tag RAM

RAM Port	BBCC Connection	Function
CLK	BZ_IDTCLKP	Clock
A8	~$\overline{\text{BZ\_IP\_DN}}$[1]	Select D-Cache or I-Cache Portion
A7	BZ_INDEXP9 &[2] $\overline{\text{BZ\_IP\_DN}}$	Address High Bit (Zero for D-Cache)
A[6:0]	BZ_INDEXP[8:2]	Address Low Bits
WE[25:5]	BZ_IDTWEP5	Tag Write Enable
WE4	BZ_IDTWEP4	Lock Bit Write Enable
WE[3:0]	BZ_IDTWEP[3:0]	Valid Bits Write Enables
OE[25:0]	Tied HIGH	Output Enable
DI[25:6]	BZ_TAGP[21:2]	Tag High Bits
DI5	BZ_TAGP1 &[2] ~$\overline{\text{BZ\_IP\_DN}}$[1]	Tag Low Bit (Zero for I-Cache)
DI4	BZ_LOCKP	Lock Bit
DI[3:0]	BZ_VALIDP[3:0]	Valid Bits
DO[25:5]	IDTAGP[21:1]	Tag (to Match Logic)
DO4	IDLCKP	Lock Bit
DO[3:0]	IDVLDP[3:0]	Valid Bits

1. ~ means that the signal is connected through an inverter.
2. & means logical AND.

The match logic is the same as in Examples 1 and 2, except that the tag for the I-cache is smaller. One way to handle this is to force some of the bits of BZ_TAG4MATCHP[21:0] to zero when the I-cache is accessed. The $\overline{\text{BZ\_IP\_DN\_L}}$ signal, a registered version of $\overline{\text{BZ\_IP\_DN}}$, is provided for ANDing with the bits to force them to zero.

```
assign new_tag4matchp = {bz_tag4matchp[21:2],
   bz_tag4matchp[1] & ~bz_ip_dn_l};
```

```
assign idmatchp = (new_tag4matchp[21:1] == idtagp[21:1]);
assign i1matchp = 1'b0;
```

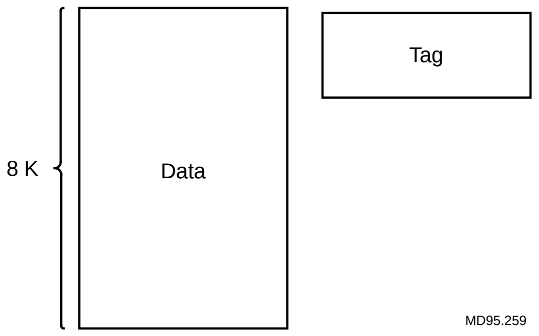
The Verilog code that causes the 3-state gates to place the Tag RAM outputs onto DATAP[31:0] is:

```
datap = bz_idt_oen ? 32'hz : {idtagp[21:1], 6'h0, idlckp,
   idvldp[3:0]};
```

5.7.2.4 Example 4: 8-Kbyte Non-Partitioned Cache

In this example, the cache is not partitioned, and can be used as either D-cache or I-cache, depending on the settings of the Cache Enable Bits in the System Configuration Register. If the D-cache Enable Bit is on, and the I-cache Enable Bit is not, then the cache is D-cache. If the I-cache enable bit is on, and the D-cache enable bit is not, then the cache is I-cache. If both the D-cache and I-cache enable bits are on, then the cache acts as a unified cache. However, when the cache is unified, the Lock Bits should not be used, since the data fetches and stores ignore the Lock Bit. Figure 5.28 shows the RAM configuration.

Figure 5.28
Example 4 RAM
Configuration



Tables 5.23 and 5.24 show the signal connections for the RAMs.

Table 5.23
Cache Data RAM

RAM Port	BBCC Connection	Function
CLK	BZ_IDDCLKP	Clock
A[10:0]	BZ_INDEXP[10:0]	Address
WE[31:24]	BZ_IDDWEP3	Byte 3 Write Enable
WE[23:16]	BZ_IDDWEP2	Byte 2 Write Enable
WE[15:8]	BZ_IDDWEP1	Byte 1 Write Enable
WE[7:0]	BZ_IDDWEP0	Byte 0 Write Enable
OE[31:0]	BZ_IDDOEP	Output Enable
DI[31:0]	DATAP[31:0]	Data In
DO[31:0]	DATAP[31:0]	Data Out

Table 5.24
D-Cache/I-Cache Set 0 Tag RAM

RAM Port	BBCC Connection	Function
CLK	BZ_IDTCLKP	Clock
A[8:0]	BZ_INDEXP[10:2]	Address
WE[23:5]	BZ_IDTWEP5	Tag Write Enable
WE4	BZ_IDTWEP4	Lock Bit Write Enable
WE[3:0]	BZ_IDTWEP[3:0]	Valid Bits Write Enables
OE[23:0]	Tied HIGH	Output Enable
DI[23:5]	BZ_TAGP[21:3]	Tag
DI4	BZ_LOCKP	Lock Bit
DI[3:0]	BZ_VALIDP[3:0]	Valid Bits
DO[23:5]	IDTAGP[21:3]	Tag (to Match Logic)
DO4	IDLCKP	Lock Bit
DO[3:0]	IDVLDP[3:0]	Valid Bits

The match logic is:

```
assign idmatchp = (tag4matchp[21:3] == idtagp[21:3]);
assign i1matchp = 1’b0;
```

The Verilog code for the 3-state gates to place the Tag RAM outputs onto DATAP[31:0] is:

```
datap = bz_idt_oen ? 32’hz : {idtagp[21:3], 8’h0, idlckp,
   idvldp[3:0]};
```

5.7.3 Adding Smaller Caches

Systems that require I-caches smaller than 1 Kbyte require the minor necessary logic that allows the BBCC to work with I-caches as small as 32 bytes. This logic generates additional Cache RAM Tag bits and appends these bits to BZ_TAGP[21:0]. It also generates additional bits of the tag for tag match and appends these bits to BZ_TAG4MATCHP[21:0].

5.7.3.1 Cache RAM Tag

To support smaller I-caches, additional low order bits must be appended to BZ_TAGP[21:0]. The additional bits of the Cache RAM Tag should be selected from either MADDROUTP[9:5] or the appropriate BBus address. The additional bits of the Cache RAM Tag must also be put through a transparent-low latch to provide sufficient hold time for the cache RAMs. The Verilog code for the Cache RAM Tag additional logic follows:

```
assign bbus_tagp[9:5] = bmcntloep ? baddrpo[9:5] :
   baddrpi[9:5];
assign bz_tagp_d[9:5] = bbus_stealn ? maddroutp[9:5] :
   bbus_tagp[9:5];
always @ (clock or bz_tagp_d)
   if (clock == 1’b0) low_bz_tagp[4:0] <= bz_tagp_d[9:5];
```

5.7.3.2 Tag for Tag Match

To support smaller I-caches, additional low order bits must be appended to BZ_TAG4MATCHP[21:0]. The tag for tag match additional logic is only needed if the system requires the BBCC to snoop. If the system requires snooping, the tag for tag match additional logic supplies lower order bits to be appended to BZ_TAG4MATCHP[21:0]. If the system does not require snooping, the appropriate bits of MADDROUTP[9:5] supply the lower order bits to be appended to BZ_TAG4MATCHP[21:0] (for example, “`assign low_bz_tag4matchp[4:0] = maddroutp[9:5]`”). The Verilog code for the tag for tag match additional logic follows:

```
always @ (posedge clock)
   bbus_stolen <= bbus_stealn;
always @ (posedge clock)
   bbus_tagp_l[9:5] <= bbus_tagp[9:5];
assign low_bz_tag4matchp[4:0] =
   bbus_stolen ? maddroutp[9:5] : bbus_tagp_l[9:5];
```

5.8 BBus Arbitration

BBus arbitration occurs independently of transactions on the BBus. While a device is currently doing a transaction, another device can be granted mastership of the bus.

The criteria the BBCC uses to determine whether it is able to start a BBus transaction are listed below. Other BBus devices might use similar criteria.

- In the previous cycle, $\overline{\text{BGNTN}}$ was asserted.
- In the previous cycle, $\overline{\text{BTXNI}}$ was deasserted (no other device is on the bus) or $\overline{\text{BTXNO}}$ was asserted (the BBCC was doing a transaction. If a new transaction is started, $\overline{\text{BTXNO}}$ remains asserted, and the BBCC can maintain control of the bus).

Many types of BBus arbiters can be designed. Figure 5.29 shows a block diagram of an example arbiter. Figure 5.30 shows a state-transition diagram of an example arbiter.

The rules and assumptions for the arbiter are:

- $\overline{\text{BTXN}}$ is a bidirectional bus that is driven by both the BBCC and another BBus device called Device X.
- When neither the BBCC nor Device X request the BBus, the bus grant is given to the BBCC.
- Once the BBCC or the other BBus device is given the bus grant, it maintains the grant until the bus request is deasserted or a transaction is started.
- The bus requests and $\overline{\text{BTXN}}$ are fast signals (signals that come early in the cycle).
- The bus grant signals do not need much setup time.

Figure 5.29
Block Diagram of Example Arbiter

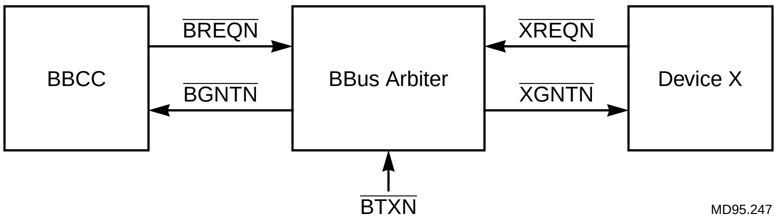
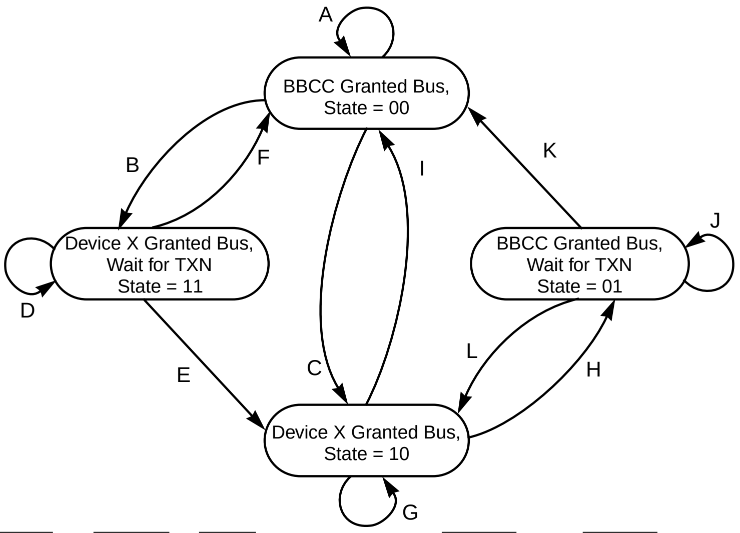


Figure 5.30
Example BBus
Arbiter State
Diagram



A:	$\overline{\text{XREQN}} \mid (\sim\overline{\text{BREQN}} \ \& \ \overline{\text{BTXN}})$	:	$\overline{\text{BGNTN}} = 0$	$\overline{\text{XGNTN}} = 1$
B:	$\sim\overline{\text{XREQN}} \ \& \ \sim\overline{\text{BTXN}}$	:	$\overline{\text{BGNTN}} = 1$	$\overline{\text{XGNTN}} = 0$
C:	$\overline{\text{BREQN}} \ \& \ \sim\overline{\text{XREQN}} \ \& \ \overline{\text{BTXN}}$	:	$\overline{\text{BGNTN}} = 1$	$\overline{\text{XGNTN}} = 0$
D:	$\sim\overline{\text{XREQN}} \ \& \ \sim\overline{\text{BTXN}}$	:	$\overline{\text{BGNTN}} = 1$	$\overline{\text{XGNTN}} = 0$
E:	$\sim\overline{\text{XREQN}} \ \& \ \overline{\text{BTXN}}$	:	$\overline{\text{BGNTN}} = 1$	$\overline{\text{XGNTN}} = 0$
F:	$\overline{\text{XREQN}}$:	:	$\overline{\text{BGNTN}} = 0$	$\overline{\text{XGNTN}} = 1$
G:	$(\overline{\text{BREQN}} \ \& \ \sim\overline{\text{XREQN}}) \mid (\sim\overline{\text{XREQN}} \ \& \ \overline{\text{BTXN}})$	:	$\overline{\text{BGNTN}} = 1$	$\overline{\text{XGNTN}} = 0$
H:	$(\sim\overline{\text{BREQN}} \ \& \ \sim\overline{\text{BTXN}}) \mid (\overline{\text{XREQN}} \ \& \ \sim\overline{\text{BTXN}})$	:	$\overline{\text{BGNTN}} = 0$	$\overline{\text{XGNTN}} = 1$
I:	$\overline{\text{XREQN}} \ \& \ \overline{\text{BTXN}}$	:	$\overline{\text{BGNTN}} = 0$	$\overline{\text{XGNTN}} = 1$
J:	$(\sim\overline{\text{BREQN}} \ \& \ \sim\overline{\text{BTXN}}) \mid (\overline{\text{XREQN}} \ \& \ \sim\overline{\text{BTXN}})$	:	$\overline{\text{BGNTN}} = 0$	$\overline{\text{XGNTN}} = 1$
K:	$(\sim\overline{\text{BREQN}} \ \& \ \overline{\text{BTXN}}) \mid (\overline{\text{XREQN}} \ \& \ \overline{\text{BTXN}})$	:	$\overline{\text{BGNTN}} = 0$	$\overline{\text{XGNTN}} = 1$
L:	$\overline{\text{BREQN}} \ \& \ \sim\overline{\text{XREQN}}$	:	$\overline{\text{BGNTN}} = 1$	$\overline{\text{XGNTN}} = 0$

1. ~ means that the signal is connected through an inverter.
2. & means logical AND.
3. | means logical OR.

MD96.206

State ‘00’ is when the BBCC is granted the bus. The arbiter is in State ‘00’ when neither the BBCC nor Device X are requesting the bus or performing a transaction. The arbiter stays in this state until Device X requests the bus and either the BBCC is not requesting the bus, or the BBCC starts a transaction.

State ‘11’ is a transition state. The bus is granted to Device X, but the BBCC is performing a transaction.

State ‘10’ is when Device X is granted the bus. The arbiter stays in this state until the BBCC requests the bus and either Device X is not requesting the bus, or Device X starts a transaction. The arbiter also leaves this state if neither Device X nor the BBCC are requesting the bus.

State '01' is a transition state. The bus is granted to the BBCC, but Device X is performing a transaction.

The Verilog code for the state machine is:

```
module arbiter ( pclkp, breqn, xreqn, btxn, bgntn, xgntn );
input pclkp, breqn, xreqn, btxn;
output bgntn, xgntn;

reg bgntn, wait_for_txn;
reg [1:0] state;

always @ (posedge pclkp)
   state <= {bgntn, wait_for_txn};

assign xgntn = ~bgntn;

always @ ( state or breqn or xreqn or btxn ) begin
   case (state)
        2'b00:
                begin
                     bgntn = ~(xreqn | (~breqn & btxn));
                     wait_for_txn = ~xreqn & ~btxn;
                end
        2'b11:
                begin
                     bgntn = ~xreqn;
                     wait_for_txn = ~xreqn & ~btxn;
                end
        2'b10:
                begin
                     bgntn = ~(xreqn | (~breqn & ~btxn));
                     wait_for_txn = (~breqn & ~btxn) |
                          (xreqn & ~btxn);
                end
        2'b01:
                begin
                     bgntn = ~(~breqn | xreqn);
                     wait_for_txn = (~breqn & ~btxn) |
                          (xreqn & ~btxn);
                end
   endcase
end

endmodule
```

5.9 Timing Considerations

This section describes timing considerations when designing with the BBCC.

5.9.1 Cache Data RAM Clocks

The clocks to the Cache Data RAMs are clocked with delayed clocks (BZ_IDDCLKP, BZ_I1DCLKP). The system designer might need to add additional delay to these clocks, depending on the loading of DATAP[31:0] and the placement of the RAMs. Figure 5.31 is a conceptual drawing of the CW400x, the BBCC, and the Cache Data RAMs. The data being written to the RAMs can be from the CW400x (for stores) or from the BBCC (for cache refills).

Figure 5.31
Conceptual
System Diagram

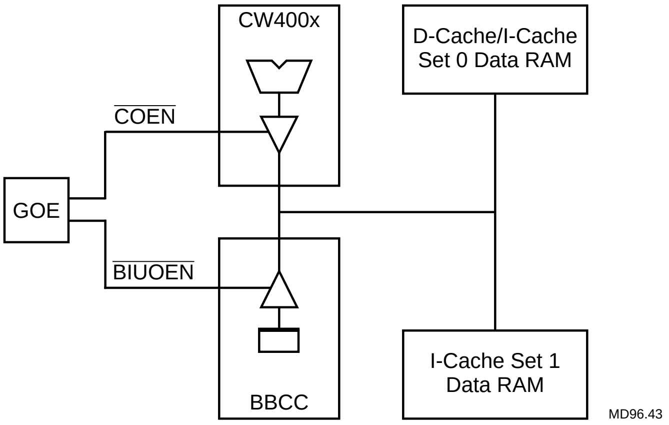
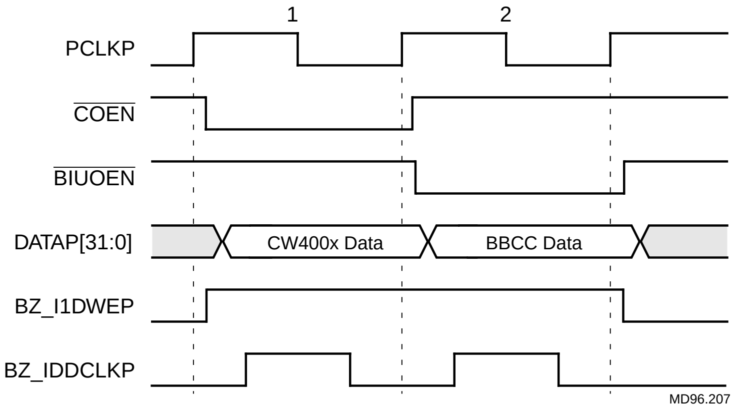


Figure 5.32 shows waveforms for writes to the D-cache/I-cache Set 0 Data RAM. Writes to the I-cache Set 1 Data RAM are similar. The purpose of the delayed clock to the RAM is to allow data setup time to the RAMs.

Figure 5.32
Writes to the
D-Cache/I-Cache
Set 0 Data RAM



Cycle 1: The CW400x performs a store to the RAM. There are two paths that matter in this case. The first path is the maximum delay from the GOE Module's $\overline{\text{COEN}}$ Output Enable (to put the data from the CW400x onto DATAP[31:0]) to the cache RAM (a zero-cycle path). The second path is the data from the CW400x (a one-cycle path). The clock to the RAM should be delayed so that DATAP[31:0] has enough setup time before the RAM is clocked. Usually, the longer path is from $\overline{\text{COEN}}$.

Cycle 2: The BBCC performs a cache refill, and the $\overline{\text{BIUOEN}}$ assertion enables the output data onto DATAP[31:0]. The data originates from a register in the BBCC called (in the Verilog code) DATAREG. In this case, there are also two paths that matter. The first path is the maximum delay from the GOE Module's $\overline{\text{BIUOEN}}$ Output Enable (to put the data from the BBCC onto DATAP[31:0]) (a zero-cycle path). The second path is the maximum delay from the DATAREG Register (also a zero-cycle path). The clock to the RAM should be delayed enough so that DATAP[31:0] has enough setup time before the RAM is clocked.

In static timing analysis, check the Cache Data RAMs setup time constraint. The CW400x data path should be considered a one-cycle path, and the paths from $\overline{\text{COEN}}$, $\overline{\text{BIUOEN}}$, and DATAREG should be considered zero-cycle paths. Check the setup time constraint at the best and worst timing conditions (BCCOM and WCCOM). If necessary, add delay gates to BZ_IDDCLKP and BZ_I1DCLKP. BZ_IDDCLKP and BZ_I1DCLKP should not be delayed too much, however, since delays on these signals cause RAM reads to occur later in the clock cycle.

5.9.2 Cache Data RAM Address

Another timing constraint is the address bus to the Cache Data RAMs, BZ_INDEXP[12:0] and $\overline{\text{BZ\_IP\_DN}}$. Since the clock signals to these RAMs are delayed, it is especially important to check the address hold time requirement of these RAMs. Figure 5.33 shows some transactions to the RAMs with the clock at a 50% duty cycle and Figure 5.34 shows some transactions to the RAMs with the clock at a 30% duty cycle.

The duty cycle is important since $\overline{\text{BZ\_IP\_DN}}$ and BZ_INDEXP[12:0] come from transparent-low latches. These signals change at the falling edge of PCLKP. At a 50% duty cycle, there is plenty of hold time on the address to the RAM. At a 30% duty cycle, there is less hold time on the address since the duty cycle is smaller.

The address to the Cache Data RAMs might need to be delayed, depending on the:

- ♦ Required minimum duty cycle
- ♦ Amount of time BZ_IDDCLKP and BZ_I1DCLKP are delayed
- ♦ Clock frequency

If the address to the Cache Data RAMs must be delayed, add delay gates to the $\overline{\text{BZ\_IP\_DN}}$ and BZ_INDEXP[12:0] signals to the Cache Data RAMs, but not to the Cache Tag RAMs.

Figure 5.33
RAM Transactions (Clock with a 50% Duty Cycle)

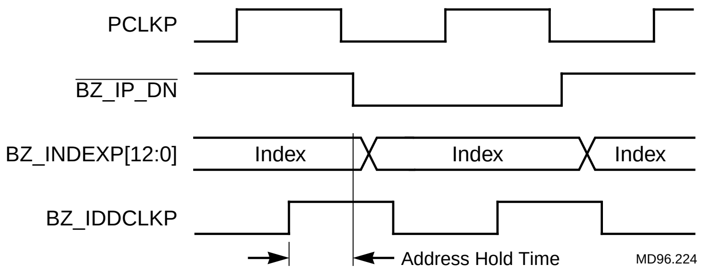
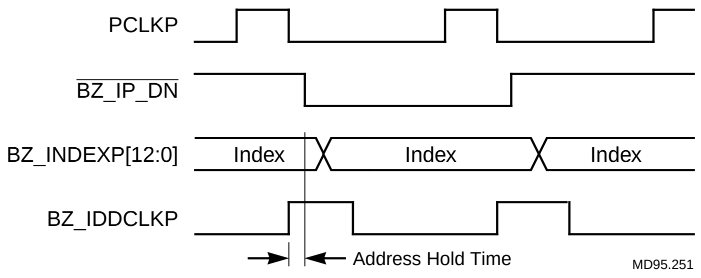


Figure 5.34
RAM Transactions
(Clock with a 30%
Duty Cycle)



5.9.3 Tag Match Logic

If a real MMU (not the MMU Stub) is in the system, and the system has a two-way set associative I-cache, MADDROUTP[31:2] (from the MMU through the tag match logic) might be the critical path. If this is the case, and if the BBCC does not need to snoop, then this path can be reduced by replacing BZ_TAG4MATCHP[21:0] with MADDROUTP[31:10]. This replacement reduces the delay from the MMU to the tag match logic.

Chapter 6 Adding or Removing Write Buffers

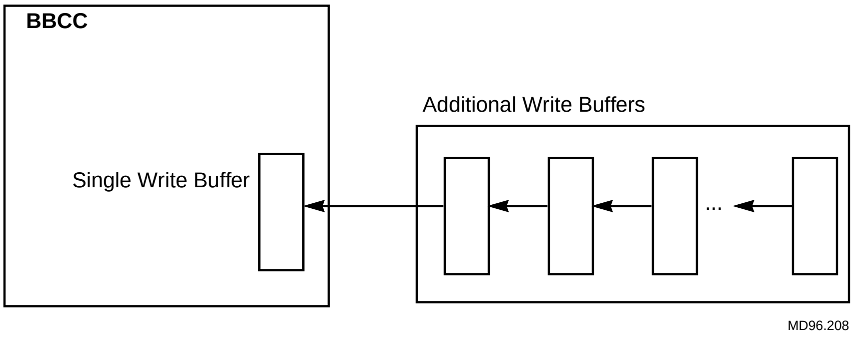
This chapter explains how to add write buffers to the BBCC by modifying the LSI Logic `cw400x_wb3` Module. It also explains how to remove them.

This chapter contains the following sections:

6.1 Overview

The BBCC contains a single write buffer. It can support up to seven additional write buffers outside the BBCC, attached as shown in Figure 6.1. These additional buffers form a first-in-first-out (FIFO) queue; the write buffer inside the BBCC is at the head of this queue.

Figure 6.1 Typical Write Buffer Configuration



6.2 Signals

This section describes the signals that comprise the bit-level interface of a write buffer (WB). Tables 6.1 and 6.2 summarize the write buffer signals. Detailed descriptions follow the tables.

The signals are described in alphabetical order by mnemonic. Each signal definition contains the mnemonic and the full signal name. The mnemonics for active LOW signals end in an “N” and have an overbar over their names.

In the descriptions that follow, “assert” means to drive TRUE or active and “deassert” means to drive FALSE or inactive.

Table 6.1
Write Buffer Input Signals Summary

Input	Source	Description
BHD_ADDRP[31:2]	Next Transaction WB’s WB1_ADDRP[31:2], Ground if last WB	Store Address
BHD_ARRIVEBFLDP[1]	Next Transaction WB’s WB1_ARRIVEBFLDP, Ground if last WB	Arrive Before Load
BHD_BYTEP[3:0]	Next Transaction WB’s WB1_BYTEP[3:0], Ground if last WB	Byte Enables
BHD_CFGP	Next Transaction WB’s WB1_CFGP, Ground if last WB	Store to Configuration Register
BHD_DATAP[31:0]	Next Transaction WB’s WB1_DATAP[31:0], Ground if last WB	Store Data
BHD_STPNDP	Next Transaction WB’s WB1_STPNDP, Ground if last WB	Store Pending
BQ_ARRIVEBFLDP[1]	BBCC	Arrive Before Load[1]
BQ_CFGSELP	BBCC	Store to Configuration Register
BQD_DFQADDRP[3:0][1]	BBCC	Data Fetch Address
BQ_DFQUPDATEP	BBCC	Update Data Fetch Queue
BQO_DFDONEP	BBCC	Data Fetch Done
BQ_RDSTQP	BBCC	Read Store Queue
BQ_WRSTQP	BBCC	Write Store Queue
$\overline{\text{BRESETN}}$	System Logic/Reset Module	Reset
CBYTEP[3:0]	CW400x	CW400x Byte Enables
DATAP[31:0]	CW400x	Cw400x Store Data

(Sheet 1 of 2)

Table 6.1 (Cont.)
Write Buffer Input
Signals Summary

Input	Source	Description
MADDROUTP[31:2]	CBus	CBus Store Address
PCLKP	System Logic	System Clock
PREV_STPNDP	Previous Transaction WB's WB1_STPNDP or BBCC's BW_STPNDP	Store Pending
SE	System Logic	Scan Enable
SI	Previous Transaction WB or BBCC's SO	Scan Data In
(Sheet 2 of 2)		

1. Needed to implement read priority.

Table 6.2
Write Buffer Output
Signals Summary

Output	Destination	Description
SO	Next Transaction WB	Scan Data Out
WB1_ADDRP[31:2]	Previous Transaction WB's BHD_ADDRP[31:2], BBCC's WB_ADDRP[31:2] if First External WB	Store Address
WB1_ARRIVEBFLDP	Previous Transaction WB's BHD_ARRIVEBFLDP, BBCC's WB_ARRIVEBFLDP if First External WB	Arrive Before Load
WB1_BYTEP[3:0]	Previous Transaction WB's BHD_BYTEP[3:0], BBCC's WB_BYTEP[3:0] if First External WB	Byte Enables
WB1_CFGP	Previous Transaction WB's BHD_CFGP, BBCC's WB_CFGP if First External WB	Configuration Register Select
WB1_DATAP[31:0]	Previous Transaction WB's BHD_DATAP[31:0], BBCC's WB_DATAP[31:0] if First External WB	Store Data
WB1_STPNDP	Previous Transaction WB's BHD_STPNDP, BBCC's WB_STPNDP if First External WB	Store Pending
WB1_VWBFLDP	OR-Gated to BBCC	Valid Write Before Load

BHD_ADDRP[31:2]

Store Address **Input**

The current WB receives the Store Address from the next transaction WB on these signals when data moves up the FIFO queue. If the current WB is the last WB in the queue, these signals are connected to ground.

BHD_ARRIVEBFLDP

Arrive Before Load **Input**

The current WB receives the Arrive Before Load signal from the next transaction WB on this signal when data moves up the FIFO queue. If the current WB is the last WB in the queue, this signal is connected to ground.

BHD_BYTEP[3:0]

Byte Enables **Input**

The current WB receives the Byte Enables from the next transaction WB on these signals when data moves up the FIFO queue. If the current WB is the last WB in the queue, these signals are connected to ground.

BHD_CFGP **Store to Configuration Register** **Input**

The current WB receives the Configuration Register Select signal from the next transaction WB on this signal when data moves up the FIFO queue. If the current WB is the last WB in the queue, this signal is connected to ground.

BHD_DATAP[31:0]

Store Data **Input**

The current WB receives the Store Data from the next transaction WB on these signals when data moves up the FIFO queue. If the current WB is the last WB in the queue, these signals are connected to ground.

BHD_STPNDP **Store Pending** **Input**

The current WB receives the Store Pending signal from the next transaction WB on this signal when data moves up the FIFO queue. If the current WB is the last WB in the queue, this signal is connected to ground.

BQ_ARRIVEBFLDP

Store Arrived Before Load **Input**

The BBCC BW_ARRIVEBFLDP output connects to this input. The BBCC asserts this signal to inform the WB that the current CBus store transaction is occurring while the Data Fetch Queue is empty. This signal is needed to implement read priority.

BQ_CFGSELP **Store to System Configuration Register** **Input**

The BBCC BW_CFGSELP output connects to this input. The BBCC asserts this signal to inform the WB that the current CBus store is to the System Configuration Register.

BQD_DFQADDRP[3:0]

Data Fetch Queue Address **Input**

The BBCC BW_DFQADDRP[3:0] outputs connect to these inputs. These signals are a few bits of the address from the Data Fetch Queue. The WB uses these bits to detect load/store dependencies. This signal is needed to implement read priority.

BQ_DFQUPDATEP

Data Fetch Queue Update **Input**

The BBCC BW_DFQUPDATEP output connects to this input. The BBCC asserts this signal to inform the WB that the Data Fetch Queue is being updated.

BQO_DFDONEP

Data Fetch Done **Input**

The BBCC BW_DFDONEP output connects to this input. The BBCC asserts this signal to inform the WB that a data fetch transaction just completed.

BQ_RDSTQP **Read Store Queue** **Input**

The BBCC BW_RDSTQP output connects to this input. The BBCC asserts this signal to initiate a read operation to the WB.

BQ_WRSTQP **Write Store Queue** **Input**

The BBCC BW_WRSTQP output connects to this input. The BBCC asserts this signal to initiate a write operation to the WB.

$\overline{\text{BRESETN}}$ **Reset** **Input**

Asserting this signal resets the WB.

CBYTEP[3:0] **Byte Enables** **Input**

These signals from the CW400x inform the WB (when asserted HIGH) which corresponding bytes are valid on DATAP[31:0].

The following table shows the correspondence between byte enables and the data bus bytes.

Byte Enable	Corresponding DATAP[31:0] Byte
CBYTEP3	[31:24]
CBYTEP2	[23:16]
CBYTEP1	[15:8]
CBYTEP0	[7:0]

DATAP[31:0]	**CW400x Data Bus** These signals transfer data to, and from, the CW400x.	**Bidirectional**
MADDROUTP[31:2]	**Mapped Address** These signals are the mapped CBus address from the MMU or MMU Stub.	**Input**
PCLKP	**System Clock** This signal is the global clock input.	**Input**
PREV_STPNDP	**Store Pending** When the previous transaction WB (which might be in the BBCC) asserts this signal (by passing along WB1_STPNDP), it informs the current WB that the previous transaction WB contains a valid store transaction.	**Input**
SE	**Scan Enable** Asserting this signal enables the scan chain.	**Input**
SI	**Scan Data In** This signal is the scan data input.	**Input**
SO	**Scan Data Out** This signal is the scan data output.	**Output**
WB1_ADDRP[31:2]	**Store Address** The current WB uses these signals to pass on its Store Address to the previous transaction WB (which might be in the BBCC).	**Output**
WB1_ARRIVEBFLDP	**Arrive Before Load** The current WB uses this signal to pass on its Arrive Before Load signal to the previous transaction WB (which might be in the BBCC). If this signal is asserted, it informs the previous transaction WB in the queue (which might be in the BBCC) that the store transaction held in the current WB was started while the Data Fetch Queue was empty.	**Output**

WB1_BYTEP[3:0]

Byte Enables to Previous Transaction WB **Output**

The current WB uses these signals to pass on its Byte Enables to the previous transaction WB (which might be in the BBCC).

WB1_CFGP **Configuration Register Select** **Output**

The current WB uses this signal to pass on its Configuration Register Select signal to the previous transaction WB (which might be in the BBCC).

WB1_DATAP[31:0]

Store Data to Previous Transaction WB **Output**

The current WB uses these signals to pass on its store data to the previous transaction WB (which might be in the BBCC).

WB1_STPNDP **Store Pending** **Output**

The current WB uses this signal to pass on its Store Pending signal to the previous transaction WB (which might be in the BBCC).

If this signal is asserted, it informs the previous transaction WB and the next transaction WB (or, if there is no next transaction WB, then the BBCC) that the current WB contains a valid store transaction.

If the current WB is the last WB in the queue, this signal asserted also informs the BBCC that the WB is Full (it is connected to the BBCC WB_FULLP signal).

WB1_VWBFLDP

Valid Write Before Load **Output**

The current WB asserts this signal to inform the BBCC that the current WB contains a valid store transaction has higher priority than data fetch transactions. All the WB's WB1_VWBFLDP outputs are logically ORed to provide the BBCC's BW_VWBFLDP input.

6.3 Basic Operation of a Write Buffer

The three store pending bits (PREV_STPNDP, BHD_STPNDP, and WB1_STPNDP) and the Read/Write Indicators (BQ_RDSTQP and BQ_WRSTQP) determine the source of the WB input.

During a read operation (BQ_RDSTQP HIGH), the BBCC takes data from the top of the FIFO queue, and subsequent data moves up the queue. The current WB takes its input from the next WB in queue (the next transaction WB).

During a write operation (BQ_WRSTQP HIGH), if PREV_STPNDP is HIGH and WB1_STPNDP is LOW, the current WB is the first nonempty buffer in the queue; therefore it takes its input from the CW400x. Otherwise, the current WB holds its current data.

Table 6.3 shows how BHD_STPNDP and WB1_STPNDP control the current WB when a read and a write operation happen at the same time.

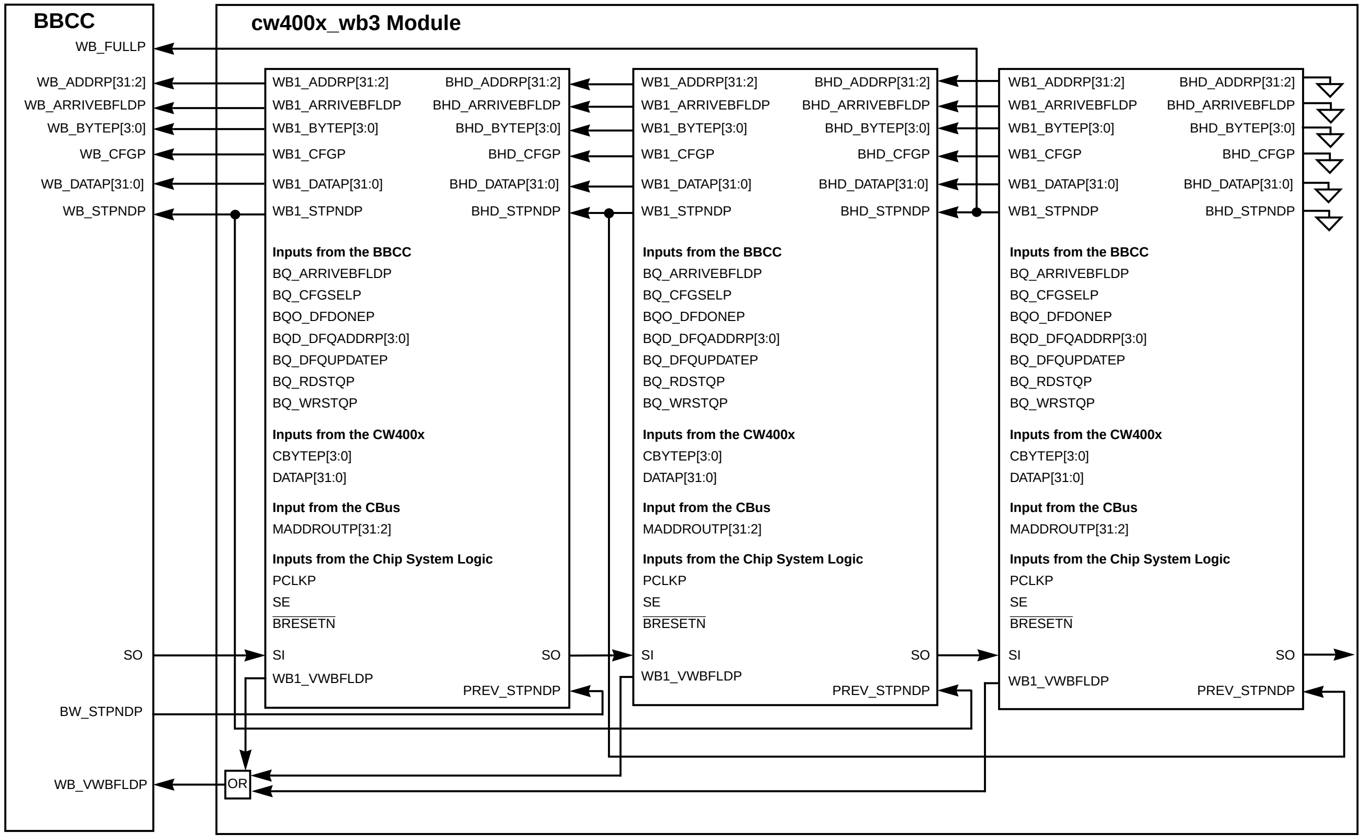
Table 6.3
Write Buffer Operation for Reads and Writes

BHD_STPNDP	WB1_STPNDP	Results
x	0	Does Not Clock in New Data
0	1	Clocks in Data from the CW400x
1	1	Clocks in Data from the Next Write Buffer

6.4 Adding a Write Buffer

Figure 6.2 shows a block diagram of the write buffer connections within the `cw400x_wb3` Module.

Figure 6.2
Write Buffer Connection Diagram



The previous transaction WB is to the left of the current WB, and the next transaction WB is to the right of the current WB.

MD96.209

6.4.1 Connect the Inputs

Connect the input signals as follows:

- ♦ Inputs from the next transaction write buffer (BHD_*) - tie to ground
- ♦ Inputs from the CW400x, CBus, System Logic, and BBCC - these signals are direct inputs to all of the write buffers (see Figure 6.2).
- ♦ Inputs from the previous transaction write buffer - connect the previous transaction WB's WB1_STPNDP output to the PREV_STPNDP input. If the previous transaction write buffer is inside the BBCC, then connect the BBCC BW_STPNDP output to the PREV_STPNDP input.
- ♦ Scan Input - Connect the previous transaction WB's Scan Out signal, SO, to the Scan In signal, SI.

6.4.2 Connect the Outputs

Connect the output signals as follows:

- ♦ Outputs to the previous transaction write buffer inputs - Connect the outputs with the corresponding signal names to the previous transaction write buffer's inputs (WB1_* to BHD_*). For example, connect the WB1_STPNDP output to the BHD_STPNDP input of the previous transaction write buffer.
- ♦ Valid Write Before Load Global Output signal - If any of the WB1_VWBFLDP signals inside the Store Queue are asserted, the write buffer should be flushed before the Queue Controller lets the data fetch onto the bus. Therefore, OR all the WB's WB1_VWBFLDP signals to generate WB_VWBFLDP. Then connect WB1_VWBFLDP to the BBCC's WB_VWBFDP input. The BBCC passes this signal on to the Queue Controller inside the BBCC.
- ♦ WB1_STPNDP signal - If there is a valid store operation in the last write buffer, the write buffer asserts WB1_STPNDP, which indicates that the Store Queue is full. Therefore, connect the BBCC WB_FULLP signal to the WB1_STPNDP signal of the last write buffer. Assertion of WB_FULLP informs the Queue Controller that the Store Queue is full and therefore the Queue Controller should stall the CW400x if there are any more data stores issued.
- ♦ Scan Out - Connect the Scan Out signal, SO, to the next transaction WB's Scan In signal, SI.

6.5 Removing a Write Buffer

Removing a write buffer is the reverse of adding a write buffer. Disconnect the inputs and outputs. Ensure that the last (previous) WB's BHD_* signals are connected to ground, and that the WB1_STPNDP signal is connected to the BBCC's WB_FULLP input (see Figure 6.2).

Chapter 7 Timer

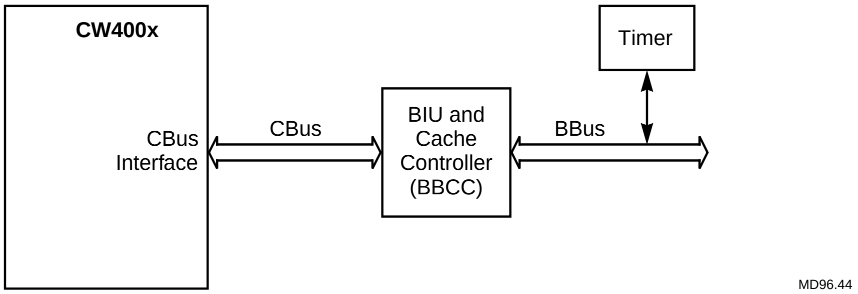
This chapter describes the Timer building block for the CW400x.

It contains the following sections:

7.1 Overview

The Timer contains two general-purpose programmable timers (Timer 0 and Timer 1) that can be used to control special functions. Figure 7.1 shows a block diagram of a system using the CW400x and the Timer.

Figure 7.1 CW400x System with the Timer



7.2 Features

The Timer supports the following features:

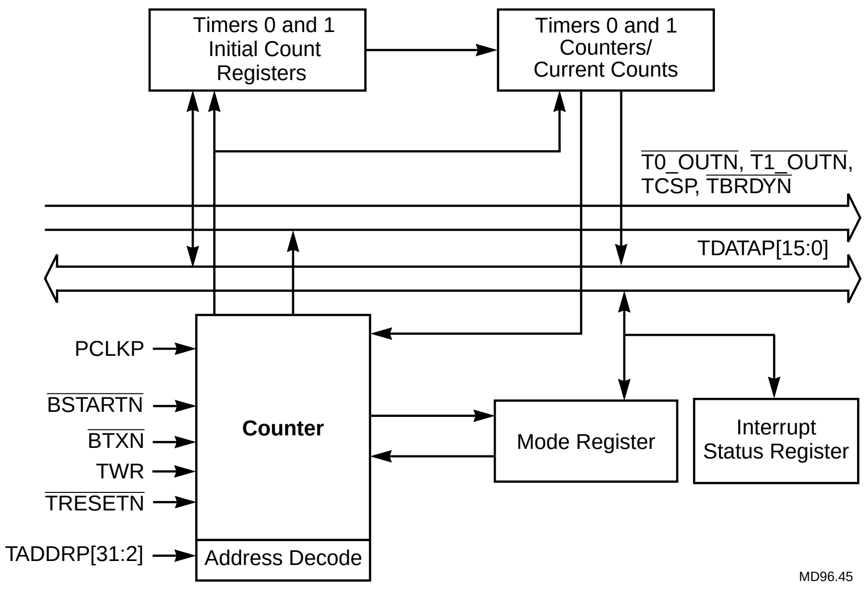
- Two independent general-purpose 16-bit down counters with programmable initial count values

- Programmable output modes (Toggle or Pulse)
- Special modes: Bus Watch Dog (Timer 1), Interrupt (Timer 0)
- External logic half-speed mode

7.3 Functional Description

Figure 7.2 shows an internal block diagram of the Timer building block.

Figure 7.2
Timer Internal
Block Diagram


Timers 0 and 1 Initial Count Registers
Timers 0 and 1 Counters/ Current Counts
T0_OUTN, T1_OUTN, TCSP, TBRDYN
TDATAP[15:0]
PCLKP
BSTARTN
BTXN
TWR
TRESETN
TADDRP[31:2]
Counter
Address Decode
Mode Register
Interrupt Status Register
MD96.45

The Timer decodes the BBus transaction address (see Table 7.4) from the BBCC and asserts a select signal to indicate that the BBCC has selected the Timer.

Each timer is a 16-bit down counter (a 32-bit data bus with the high 16 bits tied to 0) with an Initial Count Register and a Current Count Register. The Mode Register defines the operation of both counters. The Interrupt Status Register enables the Timer 0's Interrupt Mode and provides the interrupt output. Section 7.5, "Registers," describes the Mode and Interrupt Status Registers in detail.

7.4 Signals

This section describes the signals that comprise the bit-level interface of the Timer. Tables 7.1 through 7.3 summarize the Timer signals. Detailed descriptions follow the tables.

The signals are described in alphabetical order by mnemonic. Each signal definition contains the mnemonic and the full signal name. The mnemonics for signals that are active LOW end with “N” and have an overbar over their names.

In the descriptions that follow, “assert” means to drive TRUE or active and “deassert” means to drive FALSE or inactive.

Table 7.1
Timer Input Signals Summary

Input	Source	Definition
BRDYNODRIVE	BBus Controller	$\overline{\text{TBRDYN}}$ 3-State Control
$\overline{\text{BSTARTN}}$	BBCC	BBus Transaction Start
$\overline{\text{BTXN}}$	BBCC	BBus Transaction Active
CTESTP	External Logic	Running Cache Test, Disable Timer
DATANODRIVE	BBus Controller	Data Bus 3-State Control
HALFP	External Logic	Running at Half CW400x Speed Indicator
HCLKP	External Logic	Half-Speed Clock
PCLKP	External Logic	System Clock
SE	External Logic	Scan Test Mode Enable
SI	External Logic	Scan Test Input
TADDRP[31:2]	BBCC	BBus Address Bus
$\overline{\text{TRESETN}}$	External Logic	Reset
TWR	BBCC	BBCC Read/Write Indicator

Table 7.2
Timer Output Signals
Summary

Output	Destination	Definition
SO	External Logic	Scan Test Output
$\overline{\text{TBERRORN}}$	External Logic	Timer 1 Counted Down to Zero Bus Error
$\overline{\text{TBRDYN}}$	BBCC	Timer Data to BBCC Ready
TCSP	External Logic	Timer Module Selected
TOUTENP	BBus Controller	Data Bus Output Enable Request
TRDYENP	BBus Controller	Data Ready Signal Output Enable Request
$\overline{\text{T0\_INTN}}$	External Logic	Timer 0 Interrupt
$\overline{\text{T0\_OUTN}}$	External Logic	Timer 0 Output for General Purpose Counting
$\overline{\text{T1\_OUTN}}$	External Logic	Timer 1 Output for General Purpose Counting

Table 7.3
BBCC Bidirectional
Signals Summary

Bidirectional	Connect	Description
TDATAP[31:0]	BBCC	BBus Data Bus

BRDYNODRIVE

$\overline{\text{TBRDYN}}$ 3-State Control **Input**
The BBus controller asserts this signal to enable the Timer to assert $\overline{\text{TBRDYN}}$.

$\overline{\text{BSTARTN}}$ **BBus Transaction Start** **Input**
The BBCC asserts this signal to inform the Timer that a BBus transaction is starting.

$\overline{\text{BTXN}}$ **BBus Transaction Active** **Input**
The BBCC asserts this signal to inform the Timer that a BBus transaction is in progress.

CTESTP **Running Cache Test, Disable Timer** **Input**
Asserting this signal during a hardware cache test disables the Timer.

DATANODRIVE

Data Bus 3-State Control **Input**
Asserting this signal allows the Timer to output data onto the data bus, TDATAP[31:0].

HALFP	**Running at Half CW400x Speed Indicator** Asserting this signal informs the Timer that external logic is running at half of the CW400x clock speed.	**Input**
HCLKP	**Half-Speed Clock** This signal is the half-speed clock input.	**Input**
PCLKP	**System Clock** This signal is the global clock input.	**Input**
SE	**Scan Enable** Asserting this signal enables the scan chain.	**Input**
SI	**Scan Data In** This signal is the scan data input.	**Input**
SO	**Scan Data Out** This signal is the scan data output.	**Output**
TADDRP[31:2]	**BBus Address Bus** The BBCC drives these signals with the address for the current BBus transaction.	**Input**
$\overline{\textbf{TBERRORN}}$	**Timer 1 Counted Down to Zero Bus Error** The Timer asserts this signal to indicate that a Bus Error occurred because Timer 1 counted down to zero, only if the Mode Register is set to be Watch Dog Mode.	**Output**
$\overline{\textbf{TBRDYN}}$	**Timer Data to BBCC Ready** The Timer asserts this signal to indicate that data is ready on TDATAP[31:0] (only if BRDYNODRIVE is also HIGH).	**Output**
TCSP	**Timer Module Selected** The Timer asserts this signal to indicate that it has been selected.	**Output**
TDATAP[31:0]	**BBus Data Bus** These signals transfer data between the Timer and the BBCC.	**Bidirectional**
TOUTENP	**Data Bus Output Enable Request** The Timer asserts this signal to request permission from external logic (usually the BBus controller) to use the BBus Data Bus, TDATAP[31:0].	**Output**

TRDYENP **Data Ready Signal Output Enable Request** **Output**
The Timer asserts this signal to external logic (usually the BBus controller) to request permission to assert $\overline{\text{TBRDYN}}$. In turn the external logic must assert BRDYNODRIVE before the Timer can assert $\overline{\text{TBRDYN}}$. See Section 7.6.2, “Bus Control (Request/Grant),” for more information on use of TRDYENP.

$\overline{\text{TRESETN}}$ **Reset** **Input**
External logic asserts this signal to reset the Timer. Usually the system reset drives this signal.

TWR **BBCC Write/Read Indicator** **Input**
The BBCC drives this signal LOW to inform the Timer that the current BBus transaction is a write. The BBCC drives this signal HIGH to inform the Timer that the current BBus transaction is a read.

$\overline{\text{T0\_INTN}}$ **Timer 0 Interrupt** **Output**
The Timer asserts this signal to external logic when Timer 0 is programmed as an interrupt generator (Interrupt Mode), and Timer 0 hits zero.

$\overline{\text{T0\_OUTN}}$ **Timer 0 Output for General Purpose Counting** **Output**
The Timer asserts this signal to external logic when Timer 0 hits zero in Toggle Mode. The Timer pulses this signal to external logic when Timer 0 hits zero in Pulse Mode.

$\overline{\text{T1\_OUTN}}$ **Timer 1 Output for General Purpose Counting** **Output**
The Timer asserts this signal to external logic when Timer 1 hits zero in Toggle Mode. The Timer pulses this signal to external logic when Timer 1 hits zero in Pulse Mode.

7.5 Registers

All Timer registers are memory-mapped as shown in Table 7.4.

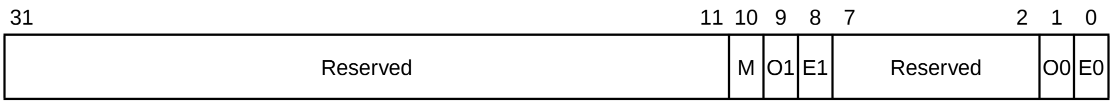
Table 7.4
Timer Register Addresses

Address	Register
0xBFFF0100	Timer 0 Initial Count Register
0xBFFF0104	Timer 0 Current Count Register
0xBFFF0108	Timer 1 Initial Count Register
0xBFFF010C	Timer 1 Current Count Register
0xBFFF0110	Mode Register
0xBFFF0114	Interrupt Status Register (Timer 0 Only)

The Initial and Current Count Registers are all simple 16-bit registers.

Figure 7.3 shows the Mode Register. The Mode Register controls the operation of the Timers.

Figure 7.3
Mode Register



Reserved — **Reserved Bits** **[31:11], [7:2]**
These bits are reserved.

M — **Timer 1 Watch Dog Mode** **10**
Setting this bit to one, puts Timer 1 into Watch Dog Mode. Clearing this bit to zero puts Timer 1 into General Purpose Mode. Note that if Timer 1 is in *Watch Dog Mode* and output is programmed to toggle, Watch Dog Mode forces the output to be a pulse.

O1 — **Timer 1 Output Mode** **9**
Setting this bit to one, puts Timer 1 into Pulse Mode. Clearing this bit to zero puts Timer 1 into Toggle Mode. Note that if Timer 1 is in *Watch Dog Mode* and output is programmed to toggle, Watch Dog Mode forces the output to be a pulse.

E1 **Timer 1 Enable** **8**
Setting this bit to one, enables Timer 1. Clearing this bit disables Timer 1. When disabled, the Counter holds the current value.

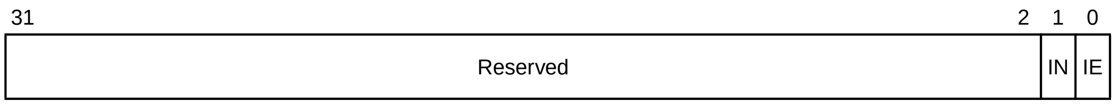
O0 **Timer 0 Output Mode** **1**
Setting this bit to one, puts Timer 0 into Pulse Mode. Clearing this bit to zero puts Timer 0 into Toggle Mode.

E0 **Timer 0 Enable** **0**
Setting this bit to one, enables Timer 0. Clearing this bit to zero, disables Timer 0. When disabled, the Counter holds the current value.

Timer 0 resets its output anytime Mode Register Bit 1 is being written with a new value and there is a change in the output mode. Timer 1 resets its output anytime Mode Register Bit 9 is being written with a new value and there is a change in the output mode. However, if the new mode is the same as the existing one, the Timer ignores the update and does not reset the output. This feature allows the timers to be turned on or off individually.

Figure 7.4 shows the Interrupt Status Register (Timer 0 only).

Figure 7.4
Interrupt Status
Register



Reserved **Reserved Bits** **[31:2]**
These bits are reserved.

IN **Timer 0 Interrupt** **1**
When Timer 0 counts down to zero, it sets this bit to one. The CW400x writes a zero to this bit to clear it to zero (sticky bit).

IE **Timer 0 Interrupt Enable** **0**
Setting this bit to one, enables Timer 0 Interrupt Mode. Clearing this bit to zero, disables Timer 0 Interrupt Mode.

7.6 Operation

This section describes the operation of the Timer.

7.6.1 Reset

A system reset (asserting $\overline{\text{TRESETN}}$) disables both timers, clears all the registers to zero, and drives the outputs, $\overline{\text{T0\_OUTN}}$ and $\overline{\text{T1\_OUTN}}$, HIGH. Reset also puts Timer 1 into General Purpose Mode.

7.6.2 Bus Control (Request/Grant)

Asserting TOUTENP requests permission to drive the BBus Data Bus to external bus control logic. Asserting TRDYENP requests permission to drive the ready signal to external bus control logic. The external bus control logic monitors the BBus Data Bus and ready signals for multiple drivers, and ensures that there is only one driver that can drive the BBus Data Bus and the ready signals in each cycle.

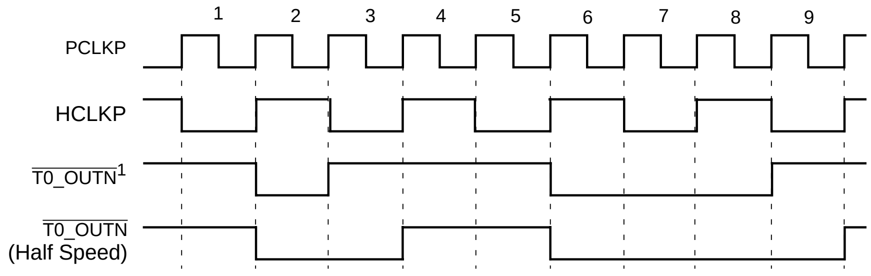
Once the Timer asserts TOUTENP and TRDYENP, the external logic should in turn assert BRDYNODRIVE and DATANODRIVE HIGH, which allows the Timer internal 3-state buffers to drive out the data on to the BBus Data Bus, TDATAP[31:0], and the Ready signal, $\overline{\text{TBRDYN}}$. The current design requires that the external bus control logic assert BRDYNODRIVE and DATANODRIVE in the same cycle that the Timer asserts TOUTENP and TRDYENP.

7.6.3 External Logic Half-Speed Mode

The External Logic Half-speed Mode is implemented specifically for an external memory system that can only run at half the speed of the CW400x, and the Timer Output signal is used for memory refresh. Asserting HALFP informs the Timer that external logic is running at half of the CW400x clock speed. During Timer operation, all internal Timer submodules run at the normal system clock frequency, except the output logic that controls the switching of the primary Timer outputs ($\overline{\text{T0\_OUTN}}$, $\overline{\text{T1\_OUTN}}$, and $\overline{\text{T0\_INTN}}$). The HCLKP input controls the output logic at half the frequency of the system clock. When in Pulse Mode, the Timer generates a pulse that is one HCLKP cycle long. When in Toggle Mode, the LOW-to-HIGH toggle transition occurs one HCLKP cycle later than in Normal-speed Mode.

Figure 7.5 shows a waveform comparing Half-speed Mode with Normal-speed Mode.

Figure 7.5
Half-Speed Mode



1. Cycle 2 - Timer 0 generates a pulse. Cycle 6 - Timer 0 toggles to LOW. Cycle 9 - Timer 0 toggles to HIGH.

7.6.4 Timer 0

Timer 0 is a 16-bit general-purpose down counter that can be configured to either toggle or generate a pulse when it reaches zero. Setting Mode Register Bit 1 causes Timer 0 to pulse $\overline{\text{T0\_OUTN}}$ when it counts down to zero (Pulse Mode). Clearing Mode Register Bit 1 causes Timer 0 to toggle $\overline{\text{T0\_OUTN}}$ when it counts down to zero (Toggle Mode). The CW400x system clock drives the Timer.

Setting the Mode Register Bit 0 to one enables Timer 0. When enabled, Timer 0 loads the value from the Timer 0 Initial Count Register into the Timer 0 Current Count Register, and starts counting down.

Clearing Mode Register Bit 0 to zero disables Timer 0. When disabled, Timer 0 stops counting. The Timer 0 Current Count Register retains the current count value. If the counter is re-enabled, the Timer reloads the Timer 0 Initial Count Value into the Timer 0 Current Count Register and proceeds to count down. It does not resume using the retained current count value.

A system reset (asserting $\overline{\text{TRESETN}}$) disables Timer 0, clears the Timer 0 Initial Count Register to zero, and drives $\overline{\text{T0\_OUTN}}$ HIGH.

Writing to the Timer 0 Initial Count Register causes the Timer to update the Timer 0 Current Count Register value and initiate a countdown from the new value (if the timer is enabled). The Timer also resets $\overline{\text{T0\_OUTN}}$ to the HIGH default state.

Clearing the Timer 0 Initial Count Register to zero disables Timer 0. When disabled, Timer 0 drives $\overline{\text{T0\_OUTN}}$ HIGH and does not count.

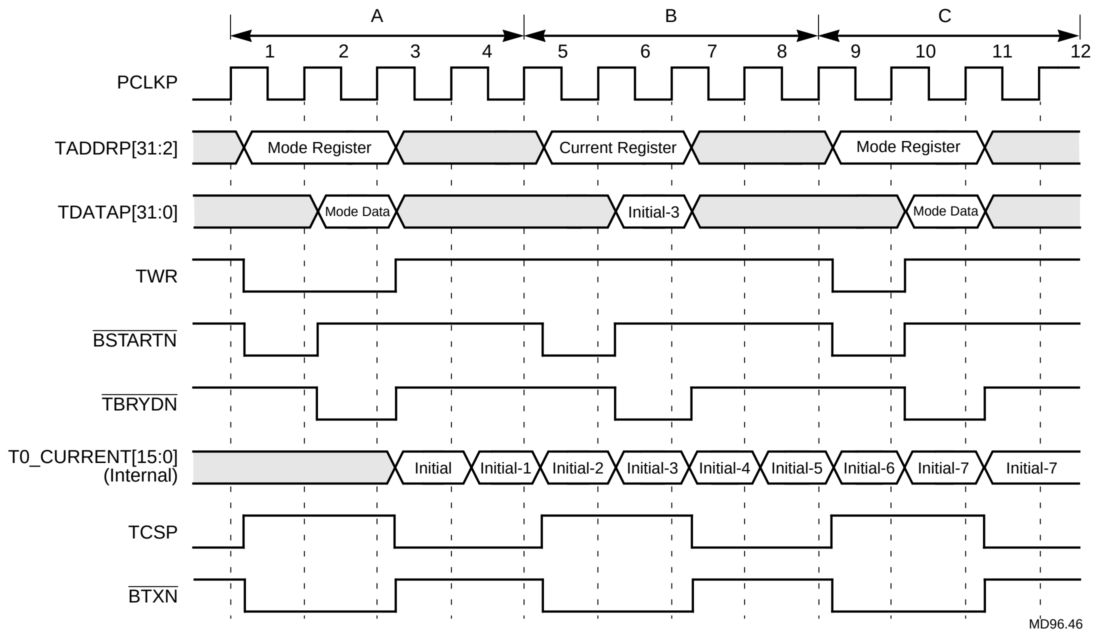
When HALFP is asserted, Timer 0 functions internally the same way it does in Normal Mode (when HALFP is not asserted), except the

$\overline{\text{T0\_OUTN}}$ output lasts one system clock cycle longer. When in Pulse Mode, Timer 0 generates a pulse that is two system clock cycles long. When in Toggle Mode, Timer 0 toggles output one cycle later than in Normal Mode. HALFP allows the Timer 0 $\overline{\text{T0\_OUTN}}$ output to be used for a DRAM refresh when an external memory system is running at half speed.

Timer 0 can generate an interrupt signal to a core output. When the Interrupt Status Register Bit 0 is set to one, Timer 0 drives $\overline{\text{T0\_INTN}}$ LOW and sets the Interrupt Bit in the Interrupt Status Register when Timer 0 reaches zero. Then Timer 0 reloads from the Initial Count Register and counts down again. The $\overline{\text{T0\_INTN}}$ output stays LOW until the Status Bit is cleared by a write to the Interrupt Status Register. Once the Status Bit is cleared, Timer 0 drives $\overline{\text{T0\_INTN}}$ HIGH again. If the Status Bit is not cleared, when the next count down to zero overflow occurs, the $\overline{\text{T0\_INTN}}$ output stays LOW. Note that if the Status Bit is cleared, the Timer does not reload the initial count.

Figure 7.6 shows a waveform of Timer 0 being enabled, read, and then disabled by the CW400x.

Figure 7.6
Timer 0 Enabled, Read, and Disabled



7.6.4.1 Transaction A

Cycle 1: The BBCC asserts TWR and $\overline{\text{BSTARTN}}$ to initiate a write to the Mode Register, which starts Timer 0 counting. The BBCC asserts $\overline{\text{BTXN}}$ to start a bus transaction. Timer 0 decodes the address and asserts TCSP to indicate that the transaction is to the Timer Module.

Cycle 2: Timer 0 decodes the input, loads from the Timer 0 Initial Count Register, and asserts $\overline{\text{TBRDYN}}$ to the BBCC.

Cycle 3: The Timer deasserts $\overline{\text{TBRDYN}}$, and Timer 0 starts counting.

7.6.4.2 Transaction B

Cycle 5: The BBCC asserts $\overline{\text{BSTARTN}}$ to initiate a memory read from the Timer 0 Current Count Register.

Cycle 6: Timer 0 puts the count value onto TDATAP[31:0] and asserts $\overline{\text{TBRDYN}}$ to the BBCC.

Cycle 7: The Timer deasserts $\overline{\text{TBRDYN}}$.

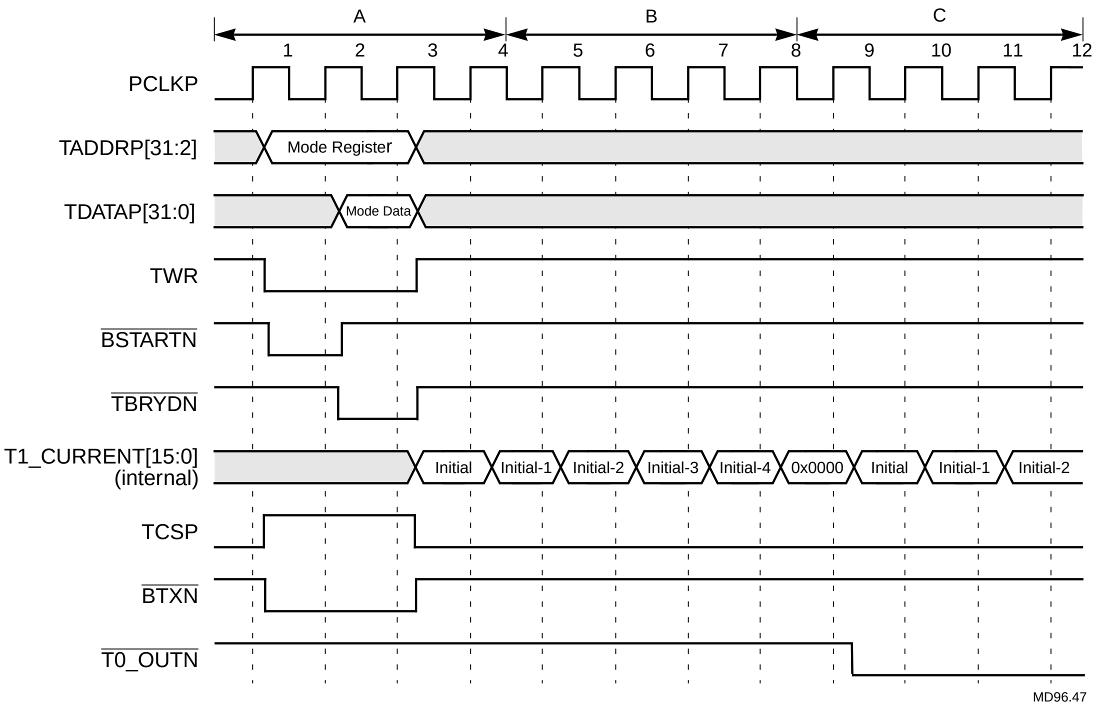
7.6.4.3 Transaction C

Cycle 9: The BBCC asserts signals to the Timer to disable the counting. (The BBCC writes a zero into Bit 0 of the Mode Register to disable Timer 0.)

Cycle 11: Timer 0 stops counting and holds the count value.

Figure 7.7 shows a waveform of the Timer 0 output, $\overline{\text{T0\_OUTN}}$.

Figure 7.7
Timer 0 Output



7.6.4.4 Transaction A

Cycle 1-4: Same as in Figure 7.6 on page 7-11.

Cycle 8: Timer 0 reaches zero.

Cycle 9: Timer 0 toggles $\overline{\text{T0\_OUTN}}$ (or generates a pulse) and reloads the Timer 0 Initial Count Value into the counter again.

Cycle 10: Timer 0 starts counting.

7.6.5 Timer 1

Timer 1 is similar to Timer 0, but with one additional feature. Timer 1 can be programmed as a bus *Watch Dog Timer*. In this mode, Timer 1 loads from the Timer 1 Initial Count Register and starts counting down whenever the CW400x starts a new Bus Transaction Cycle (other than a Timer transaction). The BBCC signals a new Bus Transaction Cycle by asserting $\overline{\text{BTXN}}$ and $\overline{\text{BSTARTN}}$.

Timer 1 is a 16-bit general-purpose down counter that can be configured to either toggle or generate a pulse when it reaches zero. Setting Mode Register Bit 9 causes Timer 1 to pulse $\overline{\text{T1\_OUTN}}$ when it counts down to zero (Pulse Mode). Clearing Mode Register Bit 9 causes Timer 1 to toggle $\overline{\text{T1\_OUTN}}$ when it counts down to zero (Toggle Mode). The CW400x system clock drives the Timer.

Setting the Mode Register Bit 8 to one enables Timer 1. When enabled, Timer 1 loads the value from the Timer 1 Initial Count Register into the Timer 1 Current Count Register and starts counting down.

Clearing Mode Register Bit 8 to zero disables Timer 1. When disabled, Timer 1 stops counting. The Timer 1 Current Count Register retains the current count value. If the counter is re-enabled, the Timer reloads the Timer 1 Initial Count Value into the Timer 1 Current Count Register and proceeds to count down. It does not resume using the retained current count value.

A system reset (asserting $\overline{\text{TRESETN}}$) disables Timer 1, puts it into General Purpose Mode, clears the Timer 1 Initial Count Register to zero, and drives $\overline{\text{T1\_OUTN}}$ HIGH.

Writing to the Timer 1 Initial Count Register causes the Timer to update the Timer 1 Current Count Register value and initiate a count down from the new value (if the timer is enabled). The Timer also resets $\overline{\text{T1\_OUTN}}$ to the HIGH default state.

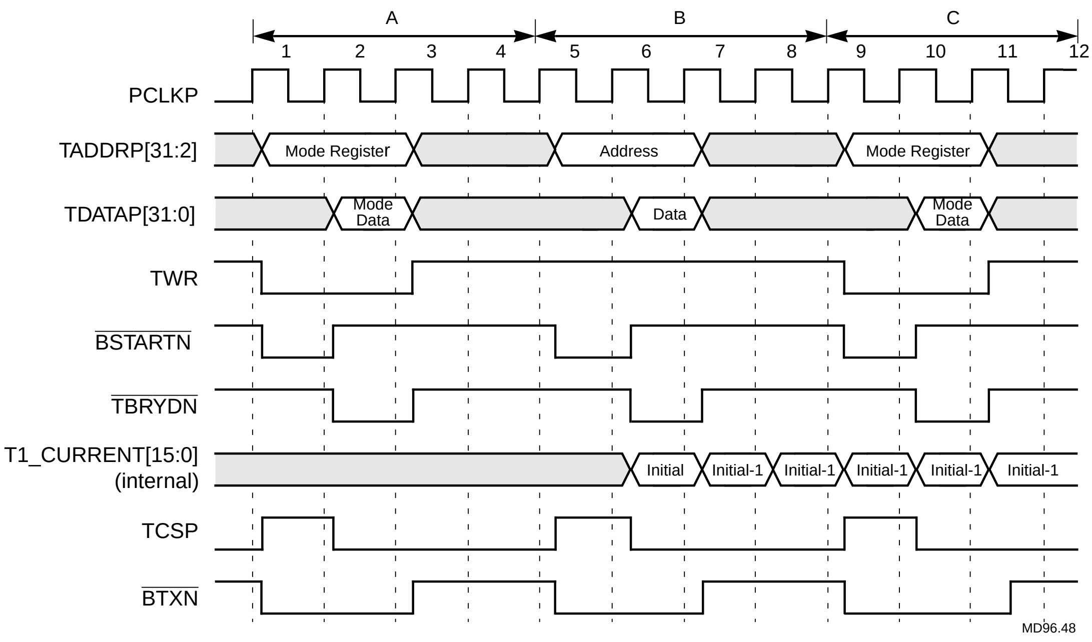
Clearing the Timer 1 Initial Count Register to zero disables Timer 1. When disabled, Timer 1 drives $\overline{\text{T1\_OUTN}}$ HIGH and does not count.

In Pulse Mode, the pulse lasts one system clock cycle, which allows the interrupt controller to register the assertion of $\overline{\text{TBERRORN}}$. Once Timer 1 hits zero, it asserts $\overline{\text{TBERRORN}}$, disables itself, and waits for the next bus transaction. If the bus cycle ends earlier, Timer 1 disables itself and holds the current count value. A new bus transaction causes Timer 1 to reload the Initial Count Value and proceed to count down.

When HALFP is asserted, Timer 1 functions internally exactly the same as it does in Normal Mode (when HALFP is not asserted), except the $\overline{\text{T1\_OUTN}}$ output lasts one system clock cycle longer. When in Pulse Mode, Timer 1 generates a pulse that is two system clock cycles long. When in Toggle Mode, Timer 1 toggles output one cycle later than in Normal Mode. HALFP allows Timer 1 $\overline{\text{T1\_OUTN}}$ output to be used for a DRAM refresh when an external memory system is running at half speed.

Figure 7.8 shows a waveform of Timer 1 being enabled, read, and then disabled by the BBCC.

Figure 7.8
Timer 1 Enabled, Read, and Disabled



7.6.5.1 Transaction A

Cycles 1-4 are the same as in Figure 7.6, page 7-11. The BBCC writes to the Mode Register to enable Timer 1 Watch Dog Mode.

7.6.5.2 Transaction B

Cycle 5:	The BBCC starts a BBus transaction to a device other than Timer 1.
Cycle 6:	Timer 1 loads the Timer 1 Initial Count Value.
Cycle 7:	The transaction ends.
Cycle 8:	Timer 1 latches in the transaction end from $\overline{\text{BTXN}}$ and holds the current cycle count value.

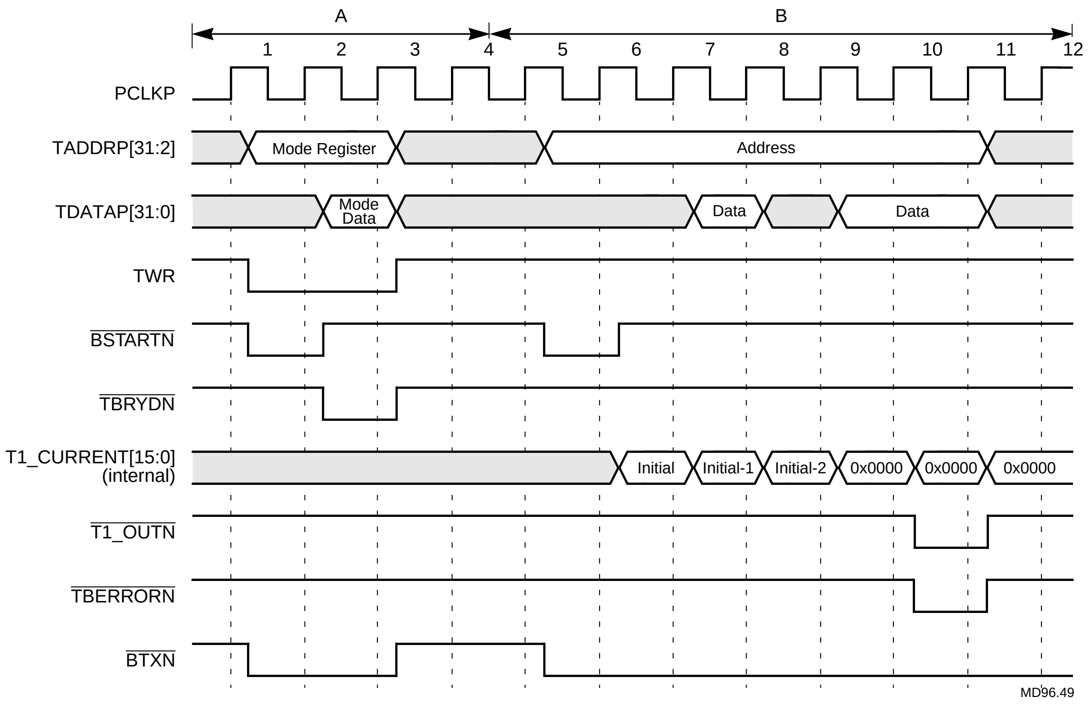
7.6.5.3 Transaction C

Cycle 9: The BBCC writes a zero into Bit 8 of the Mode Register to disable Timer 1 Watch Dog Mode.

Cycle 11: Timer 1 stops counting and holds the Current Count Value.

Figure 7.9 shows waveforms of Timer 1 behavior when Watch Dog Mode triggers $\overline{\text{TBERRORN}}$.

Figure 7.9
Timer 1 Watch Dog
Mode Triggers BERR



7.6.5.4 Transaction A

Cycles 1-4 are the same as in Figure 7.6, page 7-11. The BBCC writes to the Mode Register to enable Timer 1 Watch Dog Mode.

7.6.5.5 Transaction B

Cycle 5: The BBCC asserts $\overline{\text{BTXN}}$ and starts a bus transaction not related to the Timer.

Cycle 6: Timer 1 starts counting down.

Cycle 9: Timer 1 counts to zero.

Cycle 10: Timer 1 pulses $\overline{\text{T1\_OUTN}}$ and $\overline{\text{TBERRORN}}$ to indicate that it reached zero.

Chapter 8
Debugger (DBX)

This chapter describes the Debugger (DBX) building block for the CW400x.

The DBX provides hardware debug support for CW400x systems.

This chapter contains the following sections:

8.1 Overview

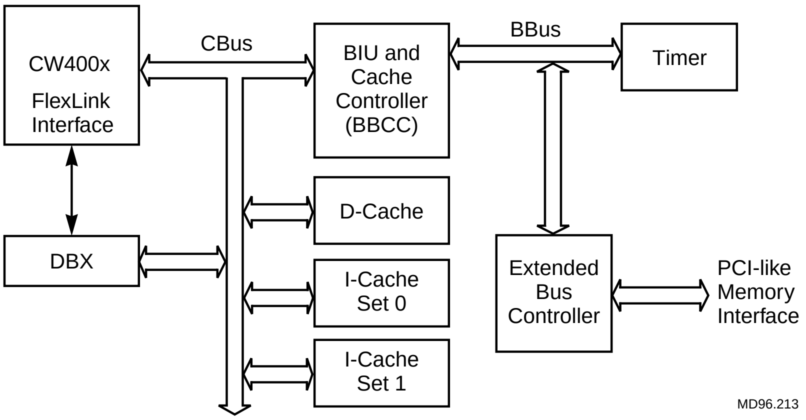
The DBX enables instruction and data access breakpoints as well as trace breakpointing. It also allows customers to use the MiniRISC ScanICE Debug System to control and observe internal registers.

The MiniRISC ScanICE Debug System interfaces with the IEEE1149.1 JTAG pins, so no additional pins are required. LSI Logic developed the ScanICE Debug System for designers using ICE as their debug environment. Because the CW400x is deeply embedded within the ASIC, the CW400x's pins cannot be accessed directly. A standard ICE cannot be plugged into the ASIC's socket, as could be done in a standard-product microprocessor-based design. The ScanICE Debug System accesses the CW400x through its scan chain, thus, providing a *virtual ICE*. From the user's point of view, using a debug environment on a host controller, there is no difference in the debug methodology between a standard product microprocessor ICE and the MiniRISC ScanICE Debug System. ScanICE is also host-independent and nonintrusive.

The DBX attaches to the CW400x FlexLink Interface. The FlexLink Interface allows new instructions to be added to the CW400x default instruction set. For details on the FlexLink Interface, see the *MiniRISC CW400x Microprocessor Core Technical Manual.*

Figure 8.1 shows how the DBX attaches to the CW400x and MiniRISC Building Blocks.

Figure 8.1
DBX Interface to the CW400x and Building Blocks



8.2 Functional Description

The DBX attaches to the CW400x using the FlexLink Interface, which gives programmer access to the CW400x Registers. The DBX allows hardware debugging in real-time using breakpoints; it also allows the scan chain to load the processor state. For details on the FlexLink Interface, see the *MiniRISC CW400x Microprocessor Core Technical Manual.*

To detect data and instruction addresses, the DBX monitors the CBus. See the *MiniRISC CW400x Microprocessor Core Technical Manual* for information on the CBus.

Software controls the DBX through instructions that access the DBX Registers.

8.3 Connection Block Diagram

Figure 8.2 shows how to attach the DBX to the CW400x, the building blocks, and system logic.

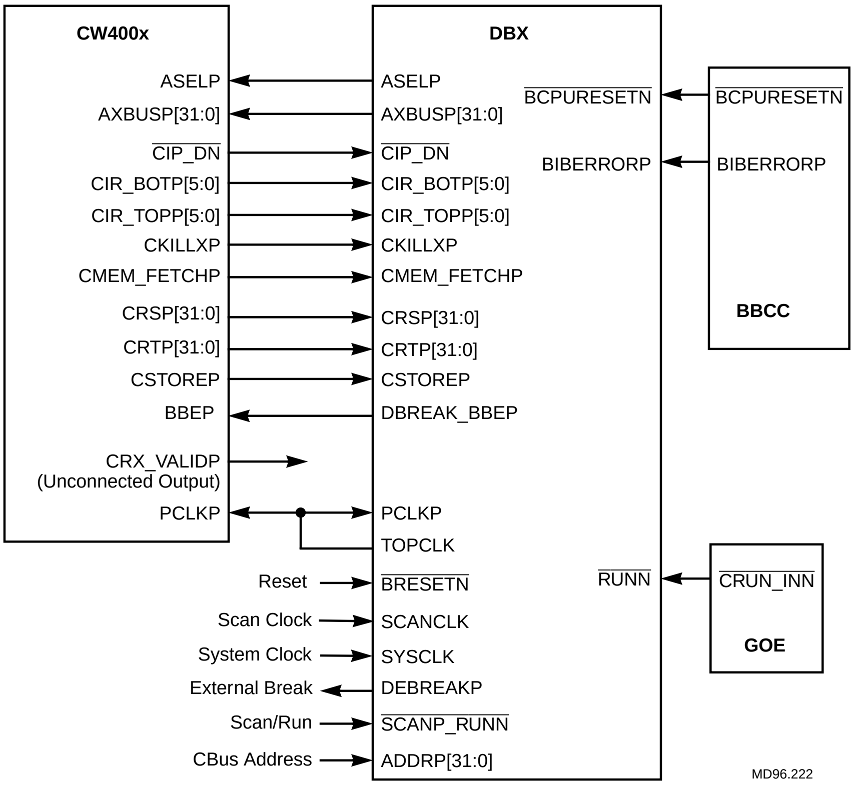


Figure 8.2
DBX in a CW400x System

8.4 Signals

This section contains a description of the signals that comprise bit-level interface of the DBX. Tables 8.1 and 8.2 summarize the DBX signals. Detailed signal descriptions follow the tables.

The signals are described in alphabetical order by mnemonic. Each signal definition contains the mnemonic and the full signal name. The mnemonics for active LOW signals end with an “N” and have an overbar over their names.

In the descriptions that follow, “assert” means to drive TRUE or active and “deassert” means to drive FALSE or inactive.

Table 8.1
DBX Input Signals Summary

Input	**Source**	**Description**
ADDRP[31:0]	CW400x	CBus Address
$\overline{\text{BCPURESETN}}$	BBCC	Reset
BIBERRORP	BBCC	Instruction Bus Error
$\overline{\text{BRESETN}}$	System Logic	Reset
$\overline{\text{CIP\_DN}}$	CW400x	CBus Instruction/Data Flag
CIR_BOTP[5:0]	CW400x	FlexLink Instruction Opcode Bottom Six Bits
CIR_TOPP[5:0]	CW400x	FlexLink Instruction Opcode Top Six Bits
CKILLXP	CW400x	CBus Instruction Killed in Execute Stage
CMEM_FETCHP	CW400x	CBus Memory Fetch
CRSP[31:0]	CW400x	FlexLink Source Register (`rs`) Bus
CRTP[31:0]	CW400x	FlexLink Source Register (`rt`) Bus
CSTOREP	CW400x	CBus Store Indicator
GSCAN_ENABLEP	System Logic	Scan Enable
GSCAN_INP	Scan Chain	Scan Data In
PCLKP	System Logic	System Clock
$\overline{\text{RUNN}}$	GOE	Run Enable/Stall
SCANCLK	System Logic	Scan Clock
$\overline{\text{SCANP\_RUNN}}$	System Logic	Clock Signal Select
SYSCLK	System Logic	Main System Clock
TST_WIO	System Logic	Test Enable

Table 8.2
DBX Output
Signals Summary

Output	Destination	Description
ASELP	CW400x	FlexLink Instruction Select
AXBUSP[31:0]	CW400x	FlexLink Result Bus (rd)
DBREAK_BBEP	CW400x	Internal Break
DEBREAKP	System Logic	External Break
GSCAN_OUTP	Scan Chain	Scan Data Out
TOPCLK	System Logic	Clock Output

ADDRP[31:0] CBus Address **Input**
These signals contain the fetch and store addresses the DBX uses to check for breakpoints.

ASELP FlexLink Instruction Select **Output**
The DBX asserts this signal when the CIR_TOPP[5:0] signals contain a Computational Instruction.

AXBUSP[31:0]
FlexLink Result Bus (rd) **Output**
During Move from Debug (MFD), these signals contain the value of an internal DBX Register for the CW400x.

$\overline{\text{BCPURESETN}}$
Reset **Input**
The BBCC asserts this signal to reset the DBX.

BIBERRORP BBCC Bus Error **Input**
The BBCC asserts this signal to inform the DBX that the current instruction fetch terminated with an error. The DBX combines this signal with its internal break signal to create DBREAK_BBEP.

$\overline{\text{BRESETN}}$ Reset **Input**
Asserting this signal resets the DBX.

$\overline{\text{CIP\_DN}}$ CBus Instruction/Data Flag **Input**
This signal qualifies the type of memory fetch when a memory fetch is indicated by CMEM_FETCHP. The CW400x drives this signal HIGH when it is performing an instruction fetch. The CW400x drives this signal LOW when it is performing a data fetch.

CIR_BOTP[5:0]

FlexLink Instruction Opcode Bottom Six Bits **Input**
These signals from the CW400x contain the bottom six bits of the Instruction Register. They allow the DBX to decode the Computational Instruction.

CIR_TOPP[5:0]

FlexLink Instruction Opcode Top Six Bits **Input**
These signals from the CW400x contain the top six bits of the Instruction Register. They allow the DBX to decode the Computational Instruction.

CKILLXP **CBus Instruction Killed in Execute Stage** **Input**
The CW400x asserts this signal to inform the DBX that the instruction in the Execute Stage is killed.

CMEM_FETCHP

CBus Memory Fetch **Input**
The CW400x asserts this signal to inform the DBX that it is performing a memory fetch. CMEM_FETCHP is valid only during run cycles.

CRSP[31:0] **CW400x Source Register (`rs`) Bus** **Input**
These signals contain the `rs` operand of the current instruction from the CW400x.

CRTP[31:0] **CW400x Source Register (`rt`) Bus** **Input**
These signals contain the `rt` operand of the current instruction from the CW400x.

CSTOREP **Store Indicator** **Input**
The CW400x asserts this signal to indicate a CBus store operation. CSTOREP is valid only during run cycles.

DBREAK_BBEP

Bus Error **Output**
The DBX asserts this signal to cause the CW400x to take a bus error exception.

DEBREAKP **External Break** **Output**
The DBX asserts this signal when an external break condition occurs.

GSCAN_ENABLEP

Scan Enable **Input**
Asserting this signal enables the scan chain.

GSCAN_INP **Scan Data In** **Input**
This signal is the scan data input.

GSCAN_OUTP **Scan Data Out** **Output**
This signal is the scan data output.

PCLKP **System Clock** **Input**
The PCLKP clock is the global clock input.

$\overline{\text{RUNN}}$ **Run Enable/Stall** **Input**
This signal connects to the GOE $\overline{\text{CRUN\_INN}}$ output. The GOE asserts this signal LOW to enable the CW400x to go on to the next run cycle. The GOE deasserts this signal HIGH to stall the CW400x. (For more information on the GOE, see the *MiniRISC CW400x Microprocessor Core Technical Manual*.)

SCANCLK **Scan Clock** **Input**
The DBX generates TOPCLK from either the SCANCLK clock input or the SYSCLK clock input as determined by the $\overline{\text{SCANP\_RUNN}}$ input signal.

$\overline{\text{SCANP\_RUNN}}$
TOPCLK Select **Input**
When external system logic asserts this signal, the DBX selects SCANCLK as the source of the TOPCLK output. When external system logic deasserts $\overline{\text{SCANP\_RUNN}}$, the DBX selects SYSCLK as the source of the TOPCLK output clock.

SYSCLK **Main System Clock** **Input**
The DBX generates TOPCLK from either the SCANCLK clock input or the SYSCLK clock input as determined by the $\overline{\text{SCANP\_RUNN}}$ input signal.

TOPCLK **Clock Output** **Output**
The DBX generates TOPCLK from either the SCANCLK clock input or the SYSCLK clock input as determined by the $\overline{\text{SCANP\_RUNN}}$ input signal. TOPCLK then drives PCLKP.

TST_WIO **Test Enable** **Input**
Asserting this signal puts the DBX in Test Mode for scan.

8.5 Registers

Programmers access the DBX Registers using the MFD and MTD Instructions (see Section 8.6, “Instructions” on page 8-12). All bits are read/write, except for the bits that are hardwired to zero.

8.5.1 DCS Register (7)

The Debug Control and Status (DCS) Register contains the enable and status bits for the System Scan facilities. All status bits are sticky—debug events only set the bits if the enables are set. The bits must be cleared by using an MTD Instruction to update the DCS Register. The UD and KD Bits are always the same—setting either bit sets both bits. Reset clears the DE, IBD, EBE Bits. All other bits are unknown.

MTD Instructions have higher priority than status updates.The DBX updates the DCS Register with MTD data if an MTD Instruction occurs simultaneously with a status update caused by a break event.

Figure 8.3 shows the format of the DCS Register.

Figure 8.3
DCS Register

31	30	29	28	27	26	25	24	23	22	21	20 6	5	4	3	2	1	0
TR	UD	KD	TE	DW	DR	DAE	PCE	DE	IBD	EBE	Reserved	T	W	R	DA	PC	DB

TR **Trap Enable** **31**
Setting TR causes debug events to trap to the debug exception vector. When TR is cleared, trapping is not enabled, but debug status bits are updated with debug event information.

UD **User Mode Debug Event Detection** **30**
Setting UD enables debug event detection in User Mode. UD and KD always contain the same value. Setting either bit sets both bits.

KD **Kernel Mode Debug Event Detection** **29**
Setting KD enables debug event detection in Kernel Mode. UD and KD always contain the same value. Setting either bit sets both bits.

TE **Trace Detection Enable** **28**
Setting TE enables trace (nonsequential fetch) event detection.

DW	**Data Write** If DAE is set, setting DW enables data write event detection.	**27**
DR	**Data Read** If DAE is set, setting DR enables data read event detection.	**26**
DAE	**Data Access Breakpoint Enable** Setting DAE enables data address breakpoint debug events.	**25**
PCE	**Program Counter Breakpoint Enable** Setting PCE enables program counter breakpoint debug events.	**24**
DE	**Debug Enable** Setting DE enables debug breaks. Clearing DE disables debug breaks.	**23**
IBD	**Internal Break Disable** Setting IBD disables internal breaks. Clearing IBD enables internal breaks.	**22**
EBE	**External Break Enable** Setting EBE enables external breaks. Clearing EBE disables external breaks.	**21**
Reserved	**Reserved Bits** These bits are reserved.	**[20:6]**
T	**Trace Event Detected** The DBX sets this bit when it has detected a trace event.	**5**
W	**Write Reference Detected** The DBX sets this bit when it detects a write to the address in the BDA Register.	**4**
R	**Read Reference Detected** The DBX sets this bit when it detects a read from the address in the BDA Register.	**3**
DA	**Data Access Debug Condition Detected** The DBX sets this bit when it has detected a a data access debug condition.	**2**

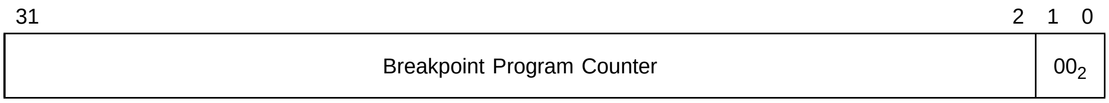
PC	**Program Counter Debug Condition Detected** The DBX sets this bit when it has detected a program counter debug condition.	**1**
DB	**Debug Condition Detected** The DBX sets this bit when it has detected a debug event.	**0**

8.5.2 BPC Register (18)

Software uses the Breakpoint Program Counter (BPC) Register to specify a program counter breakpoint. This register is used in conjunction with the Breakpoint Program Counter Mask Register described below. A breakpoint is detected for any instruction fetch in which all unmasked bits in the BPC Register match the corresponding bits in the program counter value.

Figure 8.4 shows the format of the BPC Register.

Figure 8.4
BPC Register

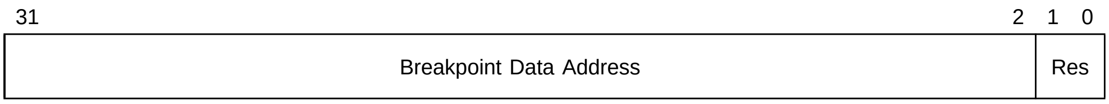


8.5.3 BDA Register (19)

Software uses the Breakpoint Data Address (BDA) Register to specify a data address breakpoint. This register is used in conjunction with the Breakpoint Data Address Mask Register listed below. A breakpoint is detected for any data reference in which all unmasked bits in the BDA Register match the corresponding bits in the data address.

Figure 8.5 shows the format of the BDA Register. The Reserved Bits, Bits [1:0], are read as zeroes.

Figure 8.5
BDA Register

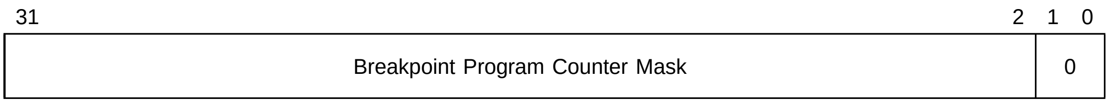


8.5.4 BPCM Register (20)

The Breakpoint Program Counter Mask (BPCM) Register masks bits in the BPC Register. Writing a 1 to any Bit *n* of the BPCM Register unmasks the bitwise comparison of Bit *n* in the BPC Register with Bit *n* in the program counter as the input to the breakpoint detection logic. Conversely, writing a 0 to Bit *n* of the BPCM Register masks the comparison, and forces the breakpoint logic to assume a match between Bit *n* in the BPC and Bit *n* in the program counter regardless of the true result.

Figure 8.6 shows the format of the BPCM Register.

Figure 8.6
BPCM Register

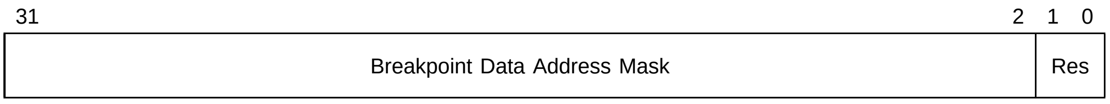


8.5.5 BDAM Register (21)

The Breakpoint Data Address Mask (BDAM) Register masks bits in the BDA Register. Writing a 1 to any Bit *n* of the BDAM Register unmasks the bitwise comparison of Bit *n* in the BDA Register with Bit *n* in the data reference address as the input to the breakpoint detection logic. Conversely, writing a 0 to Bit *n* of the BDAM Register masks the comparison, and forces the breakpoint logic to assume a match between Bit *n* in the BDA and Bit *n* in the data reference address regardless of the true result.

Figure 8.7 shows the format of the BDAM Register. The Reserved Bits, Bits [1:0], are read as zeroes.

Figure 8.7
BDAM Register



8.6 Instructions

The MFD and MTD Instructions access the DBX Registers. These instructions use the FlexLink Interface of the CW400x and are not supported by compilers or assemblers.

8.6.1 MFD Instruction

The Move from Debug (MFD) Instruction loads the contents of DBX Register `rd` into a General Register `rt`. This instruction is only valid if `rd` = 7, 18, 19, 20, or 21. All other `rd` values are undefined. The values for each Debug Register `rd` are indicated in parentheses in the register descriptions (see Section 8.5, “Registers” on page 8-8).

Operation: T: GPR[rt] <- DEBUG[rd]

Exceptions: None

Figure 8.8 shows the format of the MFD Instruction.

Figure 8.8
MFD Instruction

31 26	25 21	20 16	15 11	10 0
MFD = 011111_2	rs = 00000_2	rt	rd	0

8.6.2 MTD Instruction

The Move to Debug (MTD) Instruction loads the contents of General Register `rs` into a DBX Register `rd`. This instruction is only valid if `rd` = 7, 18, 19, 20, or 21. All other `rd` values are undefined. The values for each Debug Register `rd` are indicated in parentheses in the register descriptions (see Section 8.5, “Registers” on page 8-8).

Operation: T: DEBUG[rd] <- GPR[rs]

Exceptions: None

Figure 8.9 shows the format of the MTD Instruction.

Figure 8.9
MTD Instruction

31 26	25 21	20 16	15 11	10 0
MTD = 011110_2	rs	rt = 00000_2	rd	0

8.7 Operation

This section describes the operation of the DBX Building Block in a greater level of detail than the previous sections. It describes breakpoint operation as well as DBX internal blocks operation.

8.7.1 Breakpoints

To enable breakpoints, the system designer must follow three rules:

- ♦ **Rule #1**: SCANCLK must be LOW when debug is enabled.
- ♦ **Rule #2**: SCANCLK must be LOW when switching back to SYSCLK.
- ♦ **Rule #3**: SCANCLK must be LOW when enabling/disabling scan mode for scan debug.

When a breakpoint occurs, the DBX sets the proper status bit. All status bits are sticky—they are never cleared by hardware, and must be cleared by software.

If the EBE Bit of the DCS Register is set, the breakpoint causes DEBREAKP to be asserted for at least one cycle, which causes the system clock, TOPCLK, to be switched from SYSCLK to SCANCLK. After the first cycle, the system clock remains SCANCLK as long as $\overline{\text{SCANP\_RUNN}}$ is HIGH. The system then has access to the internal scan chain of the MR400x through the scan chain input, scan chain output, scan enable, and scan clock signals.

The DCS Status Bits are updated even for instructions that are killed and would have caused a breakpoint. DCS status bits are also updated for instructions in a branch likely delay slot. Breakpoints caused by instructions in delay slots can be misleading because they occur even if the instruction is not executed.

Breaks are signalled according to the order in which the CW400x accesses data. An instruction breakpoint might be signalled before a data breakpoint, if the offending instruction occurs *after* the data access, because the CW400x pipeline requires that the instruction access occur first.

8.7.1.1 Trace Breakpoint

A trace breakpoint occurs when the program counter branches to a nonconsecutive address. A trace breakpoint is not necessarily triggered by a branch instruction. It is only triggered if the instruction stream itself is nonconsecutive.

The DE and TE Bits of the DCS Register enable trace breakpoints. No other user inputs are needed. When the trace breakpoint occurs and TE is enabled, the CW400x signals an IBus Error, and the Exception Program Counter (EPC) Register points to the first out-of-order instruction. This instruction is killed, but all previous instructions complete execution.

8.7.1.2 Data Write Breakpoint

A data write breakpoint occurs when the CW400x writes data to an address that matches all of the bits of the BDA Register that are not masked by the BDAM Register.

The DE, DW, and DAE Bits of the DCS Register enable data write breakpoints. The BDA and BDAM Registers must be properly loaded before the breakpoint is enabled. When the data write breakpoint occurs and the TE Bit of the DCS Register is set, the CW400x signals a DBus Error, and the EPC Register points to the offending write instruction. The instruction is killed, but all previous instructions complete execution.

8.7.1.3 Data Read Breakpoint

A data read breakpoint occurs when the CW400x reads data (not an instruction fetch) from an address that matches all of the bits of the BDA Register that are not masked by the BDAM Register.

The DE, DR, and DAE Bits of the DCS Register enable the data read breakpoint. The BDA and BDAM Registers must be properly loaded before the breakpoint is enabled. When the breakpoint occurs and the TE Bit of the DCS Register is set, the CW400x signals a DBus Error, and the EPC Register points to the offending read instruction. The instruction is killed, but all previous instructions complete execution.

8.7.1.4 Program Counter Breakpoint

A program counter breakpoint occurs when the CW400x fetches an instruction from an address that matches all of the bits of the BPC Register that are not masked by the BPCM Register.

The DE and PCE Bits of the DCS Register enable this breakpoint. The BPC and BPCM Registers must be properly loaded before the breakpoint is enabled. When the Program Counter hits the Program Counter Breakpoint and the TE Bit of the DCS Register is set, the CW400x signals an

IBus Error, and the EPC Register points to the offending instruction. The instruction is killed, but all previous instructions complete execution.

8.7.1.5 BREAK Instruction Breakpoint

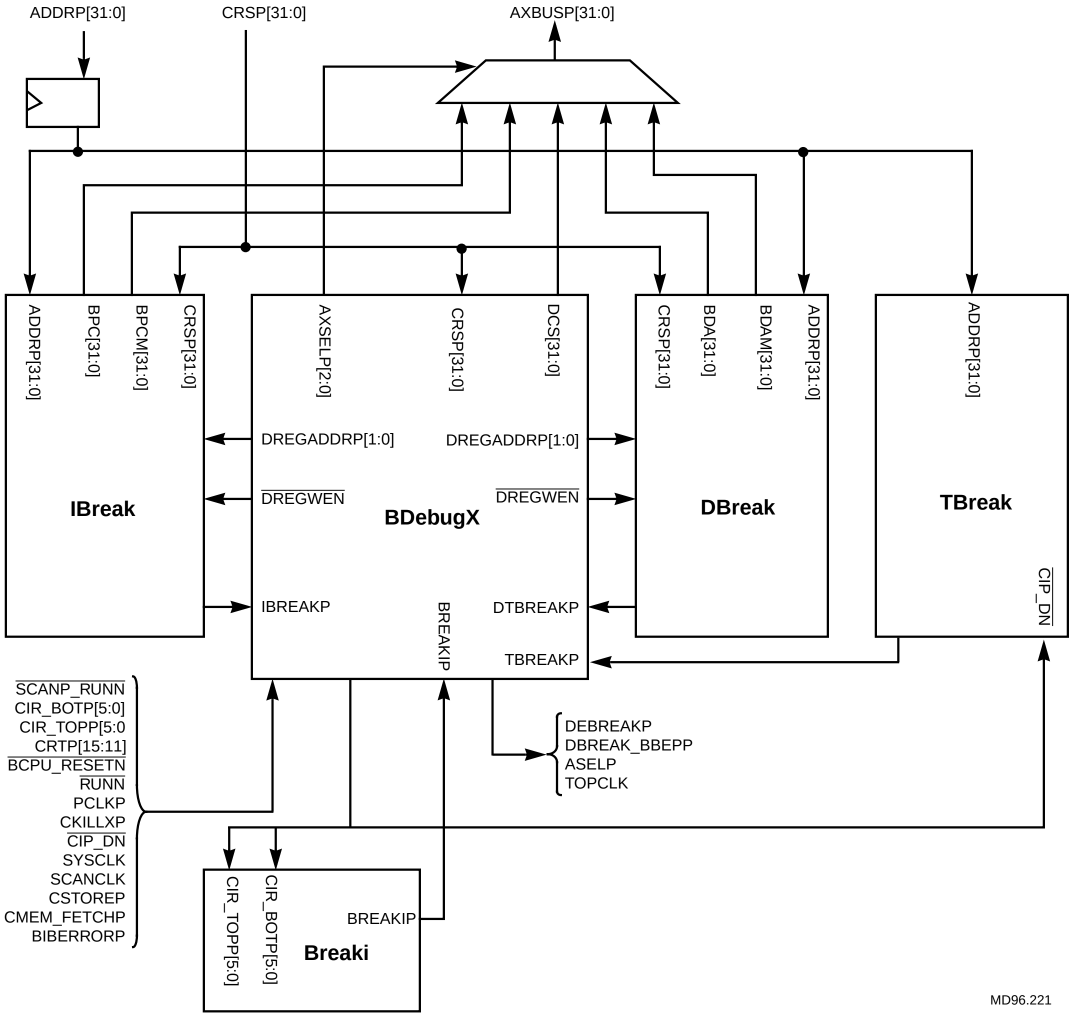
If the DE Bit in the DCS Register is set, the DBX also monitors the instruction stream for BREAK Instructions. BREAK Instructions do not cause an internal break because the CW400x automatically breaks on the instruction. However, if the EBE Bit of the DCS Register is set, an external break occurs. The external break does not update any status bits in the DCS Register.

8.7.2 DBX Module Operation

The DBX Building Block for the MiniRISC CW400x uses the FlexLink Interface, which gives the programmer access to the registers and allows the hardware to be debugged in real-time using breakpoints. It also allows the use of the scan chain to load the state of the processor.

Figure 8.10 shows the DBX internal functional blocks. Subsections following the figure describe each block in detail.

Figure 8.10
DBX Internal Block
Diagram



8.7.2.1 IBreak Module

The IBreak and DBreak Modules check for breakpoint exceptions. The IBreak Module asserts IBREAKP if the content of the BPC Register (BPC[31:2]) matches the content of ADDRP[31:2]. The DBX compares only the bits of the BPC Register to ADDRP[31:0] in which the corresponding bits are set in the BPCM Register. Bits of the BCPM Register that are cleared to zero are not compared. The IBreak module does not compare Bits [1:0] of the BPC Register to the address, because instruction fetches are always on word boundaries.

8.7.2.2 DBreak Module

The DBreak Module works in the same manner as the IBreak Module, but uses the BDA Register and the BDAM Register and compares all 32 bits of the data address.

When the BDebugX Module asserts $\overline{\text{DREGWEN}}$, the CW400x writes the data contained in CRSP[31:0] to a selected register in one of the modules. Table 8.3 shows how the registers are selected.

Table 8.3
Register Selection

DREGADDRP[1:0]	Register Selected
0	BDCM
1	BDAM
2	BPC
3	BDA

8.7.2.3 TBreak Module

The TBreak Module detects trace breakpoints. It contains a 30-bit register and an incrementor. The incremented value of the register is compared against the current Ifetch (ADDRP[31:2] when $\overline{\text{CIP\_DN}}$ is HIGH). The register is always loaded with the new Ifetch address. The TBreak Module asserts TBREAKP for one cycle when the instruction stream branches.

8.7.2.4 Breaki Module

If the DE Bit is set, the Breaki Module also monitors the instruction stream for Break Instructions. The Break Instructions do not cause an internal break, because the CW400x automatically breaks on the instruction. However, if the EBE Bit of the DCS Register is set, an external break occurs. The external break does not update any status bits in the DCS Register.

8.7.2.5 BDebugX Module

The BDebugX Module contains all of the control logic for DBX. The AXSELP[2:0] signals select the appropriate register for the MFD Instruction. DREGADDRP[1:0] select the appropriate register for writing during MTD Instructions.

The BDebugX module asserts $\overline{\text{DREGWEN}}$ at the end of the Execute Stage of an MTD Instruction (during a run cycle), when CKILLXP is deasserted. This action prevents the registers from being altered if the MTD Instruction is killed. The BDebugX Module ignores CKILLXP when it is updating its status registers, if break event detection is enabled, because the BDebugX Module is unable to determine whether it caused the CKILLXP assertion by asserting DBREAK_BBEP or if some other instruction caused CKILLXP to be asserted.

The DBX detects a break according to the enables in the DCS Register and the transaction signals, $\overline{\text{CIP\_DN}}$, CMEM_FETCHP, and CSTOREP. The breaks are reflected in the DCS Register and in the DBREAK_BBEP and DBREAKP signals. DBREAK_BBEP is enabled by default on reset. DBREAKP is disabled by default at reset.

The break signals should be connected to the bus error input of the CW400x. Then, if an instruction break is signalled, it is valid by the end of the Instruction Fetch (IF) Stage. A bus error causes an IBus Error, and the EPC Register points to the offending instruction. All instructions up to this offending one are completed. If a data breakpoint occurs, a bus error is indicated during the X2 Stage, which causes a DBus Error Exception, and the EPC Register again points to the offending instruction. The break signals are pulsed (asserted for only one cycle). The Status Bits are sticky (they hold their value until another breakpoint occurs so that their value is correct for the current breakpoint).

Note also that breaks are signalled according to the order in which the CW400x accesses data. An instruction breakpoint may be signalled before a data breakpoint if the offending instruction occurs *after* the data access. This is because the CW400x pipeline requires that the instruction access happen first.

The DBX does not support break event detection in the branch delay slots. If a break is set in the branch delay slot, the DBX switches the clock to SCANCLK even though the instruction that caused the break is killed.

Normally, the TOPCLK clock output is selected to be the SYSCLK input. The BDebugX subblock switches TOPCLK to SCANCLK if the DBreak module asserts DEBREAKP, or if $\overline{\text{SCANP\_RUNN}}$ is LOW. Then, if DBREAKP is asserted, the processor switches to SCANCLK on a break. The system can control $\overline{\text{SCANP\_RUNN}}$ and SCANCLK to control the scanning of the chain.

8.7.3 Clock Synchronization

The SCANCLK, $\overline{\text{SCANP\_RUNN}}$, and $\overline{\text{BRESETN}}$ signals pass through flip-flops to synchronize them to SYSCLK. SYSCLK is always free-running.

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 address on the following page.

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 F-112
Fax: 408.433.4333

Please tell us how you rate this document: *MiniRISC CW400x Building Blocks 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

Alabama
Huntsville
Hamilton Hallmark
Tel: 800.633.2918

Wyle Electronics
Tel: 800.964.9953

Arizona
Phoenix
Hamilton Hallmark
Tel: 800.528.8471

Wyle Electronics
Tel: 602.804.7000

Tempe
Hamilton Hallmark
Tel: 602.414.7705

California
Culver City
Hamilton Hallmark
Tel: 310.558.2000

Irvine
Hamilton Hallmark
Tel: 714.789.4100

◆ Wyle Electronics
Tel: 714.789.9953

Los Angeles
Wyle Electronics
Tel: 818.880.9000

Rocklin
Hamilton Hallmark
Tel: 916.624.9781

Sacramento
Wyle Electronics
Tel: 916.638.5282

San Diego
Hamilton Hallmark
Tel: 619.571.7540

Wyle Electronics
Tel: 619.565.9171

San Jose
◆ Hamilton Hallmark
Tel: 408.435.3500

Santa Clara
Wyle Electronics
Tel: 408.727.2500

Woodland Hills
Hamilton Hallmark
Tel: 818.594.0404

Colorado
Colorado Springs
Hamilton Hallmark
Tel: 719.637.0055

Denver
◆ Wyle Electronics
Tel: 303.457.9953

Englewood
Hamilton Hallmark
Tel: 303.790.1662

Connecticut
Cheshire
Hamilton Hallmark
Tel: 203.271.2844

Florida
Fort Lauderdale
Hamilton Hallmark
Tel: 305.484.5482

Wyle Electronics
Tel: 305.420.0500

Largo
Hamilton Hallmark
Tel: 800.282.9350

Orlando
Wyle Electronics
Tel: 407.740.7450

Tampa/N. Florida
Wyle Electronics
Tel: 800.395.9953

Winter Park
Hamilton Hallmark
Tel: 407.657.3317

Georgia
Atlanta
Wyle Electronics
Tel: 800.876.9953

Duluth
Hamilton Hallmark
Tel: 800.241.8182

Illinois
Arlington Heights
◆ Hamilton Hallmark
Tel: 708.797.7300

Chicago
Wyle Electronics
Tel: 708.620.0969

Iowa
Carmel
Hamilton Hallmark
Tel: 800.829.0146

Kansas
Overland Park
Hamilton Hallmark
Tel: 800.332.4375

Kentucky
Lexington
Hamilton Hallmark
Tel: 800.235.6039

Maryland
Baltimore
Wyle Electronics
Tel: 410.312.4844

Columbia
Hamilton Hallmark
Tel: 800.638.5988

Massachusetts
Boston
◆ Wyle Electronics
Tel: 800.444.9953

Peabody
◆ Hamilton Hallmark
Tel: 508.532.3701

Michigan
Plymouth
Hamilton Hallmark
Tel: 313.416.5800

Minnesota
Bloomington
Hamilton Hallmark
Tel: 612.881.2600

Minneapolis
Wyle Electronics
Tel: 800.860.9953

Missouri
Earth City
Hamilton Hallmark
Tel: 314.291.5350

New Jersey
Mt. Laurel
Hamilton Hallmark
Tel: 609.222.6400

No. New Jersey
Wyle Electronics
Tel: 201.882.8358

Parsippany
Hamilton Hallmark
Tel: 201.515.1641

New Mexico
Alburquerque
Hamilton Hallmark
Tel: 505293.5119

New York
Hauppauge
Hamilton Hallmark
Tel: 516.737.7400

Long Island
Wyle Electronics
Tel: 516.293.8446

Rochester
Hamilton Hallmark
Tel: 800.462.6440

North Carolina
Raleigh
Hamilton Hallmark
Tel: 919.872.0712

Wyle Electronics
Tel: 919.469.1502

Ohio
Cleveland
Wyle Electronics
Tel: 216.248.9996

Dayton
Hamilton Hallmark
Tel: 800.423.4688

Wyle Electronics
Tel: 513.436.9953

Solon
Hamilton Hallmark
Tel: 216.498.1100

Toledo
Wyle Electronics
Tel: 419.861.2622

Worthington
Hamilton Hallmark
Tel: 614.888.3313

Oklahoma
Tulsa
Hamilton Hallmark
Tel: 918.254.6110

Oregon
Beaverton
Hamilton Hallmark
Tel: 503.526.6200

Portland
Wyle Electronics
Tel: 503.643.7900

Pennsylvania
Philadelphia
Wyle Electronics
Tel: 800.871.9953

Texas
Austin
Hamilton Hallmark
Tel: 512.258.8848

Wyle Electronics
Tel: 800.365.9953

Dallas
Hamilton Hallmark
Tel: 214.553.4302

Wyle Electronics
Tel: 800.955.9953

Houston
Hamilton Hallmark
Tel: 713.787.8300

Wyle Electronics
Tel: 713.784.9953

San Antonio
Wyle Electronics
Tel: 210.697.2816

Utah
Salt Lake City
Hamilton Hallmark
Tel: 801.266.2022

Wyle Electronics
Tel: 801.974.9953

Washington
Redmond
Hamilton Hallmark
Tel: 206.881.6697

Seattle
Wyle Electronics
Tel: 800.248.9953

Wisconsin
Milwaukee
Wyle Electronics
Tel: 800.867.9953

New Berlin
Hamilton Hallmark
Tel: 414.780.7200

◆ Dstributors with Design Resource Centers

Sales Offices and Design Resource Centers

LSI Logic Corporation
Corporate Headquarters
Tel: 408.433.8000
Fax: 408.433.8989

UNITED STATES

California
Irvine
◆ Tel: 714.553.5600
Fax: 714.474.8101

San Diego
Tel: 619.635.1300
Fax: 619.635.1350

Silicon Valley
Sales Office
Tel: 408.433.8000
Fax: 408.433.7783
Design Center
◆ Tel: 408.433.8000
Fax: 408.433.2820

Colorado
Boulder
Tel: 303.447.3800
Fax: 303.541.0641

Florida
Boca Raton
Tel: 407.989.3236
Fax: 407.989.3237

Georgia
Atlanta
Tel: 770.395.3800
Fax: 770.395.3811

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: 908.549.4500
Fax: 908.549.4802

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: 214.788.2966
Fax: 214.233.9234

Houston
Tel: 713.379.7800
Fax: 713.379.7818

Washington
Bellevue
Tel: 206.822.4384
Fax: 206.827.2884

INTERNATIONAL

Australia
Reptechnic Pty Ltd
New South Wales
Tel: 612.9953.9844
Fax: 612.9953.9683

Canada
LSI Logic Corporation of Canada Inc
Ontario
Ottawa
◆ Tel: 613.592.1263
Fax: 613.592.3253

Toronto
◆ Tel: 416.620.7400
Fax: 416.620.5005

Quebec
Pointe Claire
◆ Tel: 514.694.2417
Fax: 514.694.2699

Denmark
LSI Logic Development Centre
Ballerup
Tel: 45.44.86.55.55
Fax: 45.44.86.55.56

France
LSI Logic S.A.
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: 852.2428.00008
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: 82.2.561.2921
Fax: 82.2.554.9327

Singapore
Desner Electronics Pte Ltd
Singapore
Tel: 65.285.1566
Fax: 65.284.9466

Electronic Resources Ltd
Tel: 65.298.0888
Fax: 65.298.1111

Spain
LSI Logic S.A.
Madrid
◆ Tel: 34.1.3672200
Fax: 34.1.3673151

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 Regional Office
Taipei
◆ Tel: 886.2.718.7828
Fax: 886.2.718.8869

Jeilin Technology Corporation
Tel: 886.2.248.4828
Fax: 886.2.248.9765

United Kingdom
LSI Logic Europe plc
Bracknell
◆ Tel: 44.1344.426544
Fax: 44.1344.481039

◆ Sales Offices with Design Resource Centers

ISO 9000 Certified

Printed in USA
996.1K.TP.G

Printed on Recycled Paper